Querying Business Processes*

Catriel Beeri
The Hebrew University
cbeeri@cs.huji.ac.il

Anat Eyal
Tel Aviv University
anat@exanet.com

Simon Kamenkovich
Tel Aviv University
simonkm@cs.tau.ac.il

Tova Milo
Tel Aviv University
milo@cs.tau.ac.il

ABSTRACT

We present in this paper BP-QL , a novel query language for querying business processes. The BP-QL language is based on an intuitive model of business processes, an abstraction of the emerging BPEL (Business Process Execution Language) standard. It allows users to query business processes visually, in a manner very analogous to how such processes are typically specified, and can be employed in a distributed setting, where process components may be provided by distinct providers(peers).

We describe here the query language as well as its underlying formal model. We consider the properties of the various language components and explain how they influenced the language design. In particular we distinguish features that can be efficiently supported, and those that incur a prohibitively high cost, or cannot be computed at all. We also present our implementation which complies with real life standards for business process specifications, XML, and Web services, and is used in the BP-QL system.

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 by the same or other organizations. Consequently, the software implementing such BPs typically operates in a cross-organization, distributed environment.

It is common practice to use XML for data exchange between BPs, and *Web services* for interaction with remote processes [34]. The recent BPEL standard (Business Process Execution Language [7], also identified as BPELWS or BPEL4WS), developed jointly by BEA Systems, IBM, and Microsoft, combines and replaces IBM's WebServices Flow Language (WSFL) [27] and Microsoft's XLANG [35]. It provides an XML-based language to describe not only the interface between the participants in a process, but also the *full operational logic* of the process and its *execution flow*.

Commercial vendors offer systems that allow to design BPEL specification via a visual interface, using a conceptual, intuitive view of the process, as a graph of data and activity nodes, connected by control and data flow edges. Designs are automatically converted to BPEL specifications. These can be automatically compiled into executable code that implements the described BP [30].

Declarative BPEL specifications greatly simplify the task of software development for BPs. More interestingly from an information management perspective, they also provide an important new *mine of information*. Consider for instance a user who tries to understand how a particular travel agency operates. She may want to find answers to questions such as: *Can I get a price quote without giving first my credit card details? What should one do to confirm a purchase? What kind of credit services are used by the agency, directly or indirectly, (i.e. by the other processes it interacts with)?* Obviously, such queries are of great interest to both individual users and to organizations interested in using or analyzing BPs. Answering them is extremely hard (if not impossible) when the BP logic is coded in a complex program. It is potentially much easier given a *declarative specification* like BPEL. For an organization that has access to its own BPEL specifications, as well to those of cooperating organizations, the ability to answer such queries, in a possibly distributed environment, is of great practical potential.

To support such queries, one needs an adequate query language, and an efficient execution engine for it. To address this need, we present in this paper BP-QL , a new query language which allows for an intuitive formulation of queries on BP specifications, and query execution in a distributed cross-organization environment.

Before presenting our results, let us highlight briefly some of the challenges in querying BP specifications in general, and BPEL ones in particular.

Flexible granularity BP specifications may be abstractly viewed as a set of *nested* graphs, possibly with *recursion*: The graphs structure captures the execution flow of the process components; The nesting comes from the fact that the operations/services used in a process are not necessarily atomic and may have a complex internal structure (which may itself be represented by a graph); The recursion is due to the fact that a process may call itself indirectly, through calls it makes to other processes. Users may wish to ask *coarse-grain* queries that consider certain process components as black boxes and allow for high level abstraction, as well as *fine-grained* queries that "zoom-in" on all the process components, possibly recursively. *An adequate query language must thus allow users to query the processes at different, flexible, granularity levels.*

Distribution As mentioned above, BPs typically operates in a cross-organization, distributed environment where each peer holds a set of BPs and may provide (resp. use) services to (of) remote peers. If a service's internal flow has been defined in BPEL, and the service providers make this specification available to their cooperating organizations (say via a web service), *users may wish to zoom-in on*

*The research has been supported by the Israel Science Foundation.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '06, September 12-15, 2006, Seoul, Korea.
Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09

these remote components as well to query the service specification.

Paths extraction When querying BPs, users may be interested in retrieving, as an answer, the qualifying *flow paths* (as for instance in the query "What should I do to confirm my purchase?"). As the number of relevant paths may be large (or even infinite in recursive processes) *a major challenge is to provides the users with a compact finite representation of the (possible infinite) answer.*

Ease of querying As mentioned above, the BPEL standard offers an XML-based language for describing the operational logic of a BP. Since a BPEL specification is essentially an XML document, a natural question is why not query it directly, using XQuery? A key observation is that the BPEL XML format is (1) very complex and (2) was designed with ease of *automatic code generation* in mind; however, it is extremely inconvenient for *querying*. To express even a very simple inquiry about a process execution flow, one needs to write a fairly complex XQuery query that performs an excessive number of joins. Furthermore, even if a more query-friendly XML representation for it had been chosen (as indeed is done internally in our implementation), XQuery, as is, would still not be adequate for the task: XQuery only returns document *elements*, but not *paths*, it does not support querying at *different levels of granularity*, and it does not offer tools for *controlling distributed querying*. Last but not least, querying an XML representation is much more difficult than querying directly a conceptual model. Essentially, ease of querying requires an an intuitive, conceptual, data model, coupled with a matching, equally intuitive, query language.

The BP-QL query language presented in this paper addresses these issues. It is based on an intuitive model of BPs, an abstraction of the BPEL specification, along with a graphical user interface that allows for simple formulation of queries over this model. In a sense, it follows the same design principles that guided commercial vendors in the development of graphical editors for the *specification* of BPEL processes: it hides from the users the tedious BPEL XML details and allows for more natural query formulation. Indeed, we will see that the tight analogy between how BPs are specified in such editors and how they are graphically queried in BP-QL , facilitates intuitive querying. BP-QL also offers facilities for controlling granularity and distribution in query formulation and allows paths in query results.

At the core of the BP-QL language are *BP patterns* that allow users to describe the pattern of activities/data flow that are of interest. BP patterns are similar to the tree- and graph-patterns offered by existing query languages for XML [36] and graph-shaped data [15, 13, 31], but include two novel features designed to address the issues mentioned above. First, BP-QL supports navigation along *two* axis: (1) the standard *path-based axis*, that allows to navigate through, and query, paths in process graphs, and (2) a novel *zoom-in axis*, that allows to navigate (transitively) inside process components (local as well as remote ones) and query them at any depth of nesting. Second, paths are considered first class objects in BP-QL and can be retrieved, and represented compactly, even when involving activities performed on distinct peers.

Together, these features allow for simple formulation of queries on BPs. However, they make the evaluation of queries much more intricate than that of traditional XML/graph patterns. Indeed, some queries that can easily be evaluated on flat graphs/trees may become computationally expensive (or even undecidable) when nested graphs are concerned. To keep the evaluation of queries tractable, we had identified these problematic scenarios and carefully designed the language so that they are avoided, and polynomial-time query evaluation is guaranteed. Our analysis is based on modeling systems of processes and queries as *graph grammars*[21].

Observe that, in general, several modes of querying business processes are possible. One can query the specifications as *data* (e.g. "does the specification include a path from activity A to activity B"). One can also ask about patterns that may occur when the processes are *executed* (e.g. "can there be a run of the system where activity A is followed by activity B"). One can also *monitor* runs as processes execute, or pose queries on *logs* of past runs.

BP-QL is a query language for process *specifications*,[1] not about their *possible runs*. This is for two main reasons. First, querying the possible runs of a system is a *verification problem* [22] and is typically of very high complexity (from NP-hard for very simple specifications to undecidable in the general case [28]). Second, the analysis of runs requires a specification to have a well defined semantics. Unfortunately, BPEL is not based on a formal model [28]. To avoid these obstacles and guaranty complexity that is polynomial in the size of the data, BP-QL 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 this approach offers a reasonable balance between expressibility and complexity. Note that querying of specifications in fact "approximates" the querying of runs (e.g. only specifications that contain two given activities may potentially have runs where both occur). Hence, even when full run verification is desired, BP-QL can be used as an efficient means to narrow the search space for the more costly, interpretation dependent, verification. It can also be used to select the process parts to be monitored at run time[32].

Contributions We now state the contributions of this paper.

1. We present BP-QL , a new graphical query language that allows for intuitive querying of process specifications, by offering a data model and an interface similar to those used for BPs *specification*. It allows to retrieve paths, and offers facilities for querying at different levels of granularity, and for controlling distributed querying.
2. We present a formal model for systems of processes, and for our query language on such systems, based on *graph grammars* [21]. This model allows to distinguish between query features that can be efficiently supported, and those that incur a prohibitively high cost, or cannot be computed at all. Using this model, we explain how to construct a finite and intuitive representation of the (possibly infinite) answers of queries in time polynomial in the size of the specifications.
3. Finally, we describe the system's implementation, highlighting the main challenges faced and the solutions taken.

A first prototype of BP-QL was demonstrated in [5], where only a very high level view of the language was presented. The present paper provides a comprehensive description of the language, of its underlying formal formal and of its implementation. The paper is organized as follows. Section 2 introduces BP-QL informally via a running example. The underlying formal model is presented in Section 3. The system implementation is described in Section 4. We conclude in Section 5, considering related and future work.

2. SYSTEM OVERVIEW

We present here an informal overview of BP-QL via a running example. To illustrate the features of BP-QL , we will consider a set of business processes (BPs) used by a consortium offering travel-related services. These include flight and hotel reservation, car rental, credit and accounting services. The processes, and their BPEL specifications, reside and operate on distinct peers. The specifications include the interactions between the various processes.

[1]A variant for monitoring and querying of logs is planned future research.

We first show how processes are specified, via the system's graphical user interface, and then illustrate how they can be interrogated and queried with BP-QL . The graphical *specification* of BPs that we use is fairly standard, and is similar to those offered by commercial vendors (e.g. [30]). The novelty here is in the BP-QL graphical *query language*, designed especially for querying such specifications. The ease of query formulation is illustrated by comparing the query graphical interface to that used for the processes specification; there is a tight analogy between how processes are specified and how they are queried.

Running example Our running example is along the lines of W3C's travel agent scenario[1]. Alpha-Tours, a fictional travel agency, offers to its potential clients the ability to book complete vacation packages: plane tickets, hotels, car rentals, and so on. The main steps of the reservation process are as follows: The user provides a destination, some dates, possibly some constraints, to the travel agency service. Next, the service obtains information about possible deals from airlines, hotels and car rental agencies and presents them to the user, which selects the ones she is interested in. Those are reserved by the agency. Finally, the user may cancel or confirm the reservation, passing her credit card details. The airline, car rental and hotel services contact a credit card service for payment authorization before they acknowledge the reservation.

We now demonstrate how the services are specified and queried. All screenshots were taken with our BP-QL visual designer and query tool.

2.1 Business Processes

A *system* consists of a set of BPs, possibly residing on distinct peers. A BP specification includes:

1. Some general description of the process properties, including its name, capabilities, the service provider, and so on.
2. The data used in the process, namely the process variables and the input and output parameters for the participating activities/services.
3. The activities of which the process is composed.
4. A description of the process operational and data flow.

Visually, the specification of a BP is represented as a directed labeled graph, with three types of nodes: property nodes (for 1), drawn as hexagons; data nodes (for 2), drawn as ellipses; and activity nodes (for 3), drawn as rectangles. Edges that connect data and activity nodes, called *data flow* edges, describe which data is read or output by which activity. Edges between activity nodes, called *activity flow* edges, describe the operational flow. To capture certain particular aspects of the operational flow of BPs, activity nodes may be identified as *provided operations* or *requested operations*. These describe the services offered by a process to other processes, and the external services that it requests, resp. Activity nodes may also be distinguished as *atomic* or *compound*. The latter represent invocations of composite (possibly remote) processes and are denoted by two little boxes at the top left corner of the activity icon. The interpretation of compound nodes is based on the ideas of *statechart* [23]: a *zoom-in* allows to replace a compound activity by a detailed description of the process that it invokes.

For illustration, consider the BP depicted in Figure 1. It represents the travel agency from our running example. The label under each node is its name. Each node carries some information on the process property/data/activity that it represents, which can be viewed by clicking on it. For instance, the property nodes at the top of the figure describe the process, its provider, and its capabilities. Most attributes of these nodes are references to external

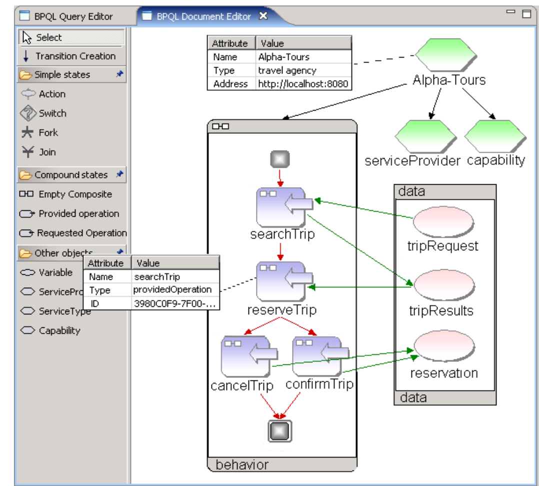

Figure 1: Travel Agency.

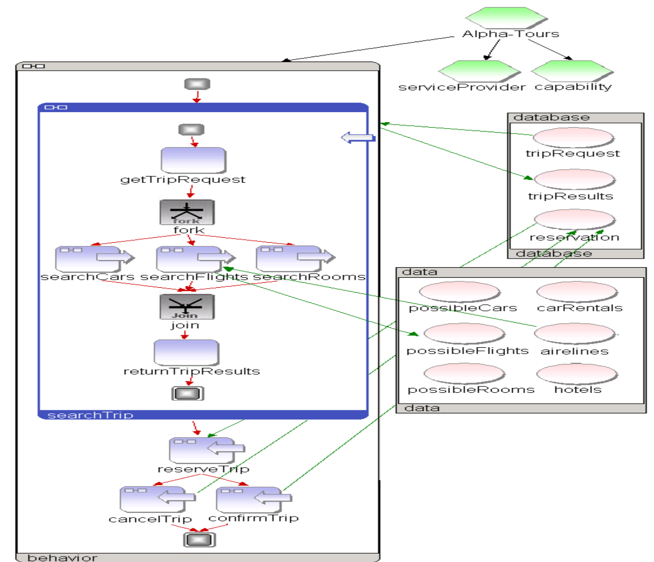

Figure 2: Zooming into *searchTrip*.

taxonomies and ontologies that provide standard definitions of the service domain.[2]

The process flow (on the left) and its data elements (on the right), are displayed in separate boxes. The small rectangles at the top and bottom of the activity flow are its entry and exit points. The BP contains four compound activity nodes, namely `searchTrip`, `reserveTrip`, `confirmTrip` and `cancelTrip`. A short thick incoming arrow indicates a *provided operation*. A client may invoke each of the four provided operations (at the appropriate point in the flow). Edges between data nodes and activity nodes depict the data flow. For example, the client's trip request is imported when the `searchTrip` activity is entered. The results are stored in a `tripResult` variable.

One can zoom into a compound activity node to see what is inside. Figure 2 shows the details of `searchTrip`. We can see that the travel agency interacts with other services to fulfill client requests. The short thick arrows outgoing of the `searchCars`,

[2] Implementation-wise they are stored in a UDDI repository.

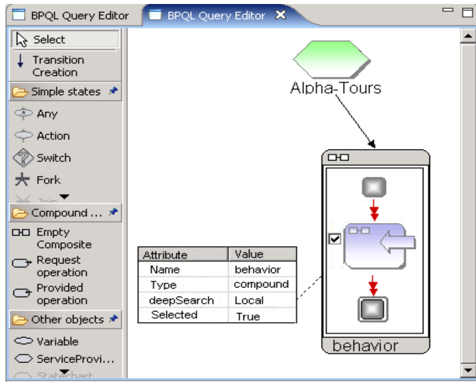

Figure 3: Find provided operations.

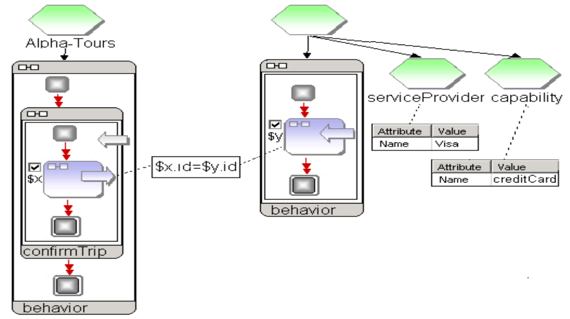

Figure 4: Credit services invoked when searching for trips.

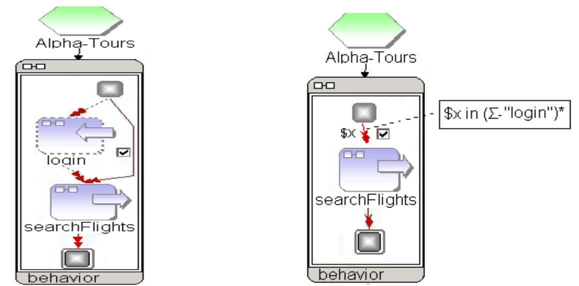

Figure 5: (a) Negation (b) Path constraints.

`searchFlights`, and `searchRooms` icons indicate that those are requested operations. Here, the node attributes (not displayed in the figure) provide the parameters (URL, operation name, ...) that allow one to invoke the relevant Web service. If the service providers make their BPEL specification available, one can zoom in also into these nodes as well to see the service specification.

The figure also shows data flow edges (for clarity some of these edges are omitted). For example, the set of `airlines` that the agency works with is imported when `searchFlights` is entered. The results of searching external airline services are stored in `possibleFlights`.

Before moving on to querying, we highlight two types of cycles that a specification may contain. First, the graph of a given BP may contain cycles, indicating that certain activities may be repeated an unbounded number of times. Second, in a system consisting of several processes that call each other, a BP may call itself indirectly, through calls it makes to other processes. This is another kind of cyclic structure: here one could *zoom into* the corresponding compound operation an unbounded number of times. Note that when querying BPs, users are often interested to retrieve *flow paths* as answers (as for instance in the query "What are the possible ways to purchase a plane ticket?"). In the presence of cycles, the number of qualifying paths may be infinite. One of the contributions of our work is to provide an intuitive, finite (and compact) representation for such possibly infinite answers.

2.2 The BP-QL Query Language

Given that BPs are defined declaratively, we can query the specifications to learn about the processes. In our running example, a user may want to ask questions like: *'Which operations are provided by the travel agency service?'*, *'Which services are called directly or indirectly by the service?'*, *'Does the service allow to make a reservation without first giving credit card details? and if so, what does one need to do for making a reservation?'*. We proceed to explain how BP-QL can be used to express such queries.

BP-QL queries look much like the specifications. For querying BPs, BP-QL offers *BP patterns* which, intuitively, play for BPs a role analogous to that played for XML trees by *tree pattern queries*. They describe the pattern of activity/data flow that is of interest to the user and allow navigation along two axis: path-based and zoom-in based. Following the use of / and // in XPath[36] for denoting single and multiple step navigation, our PB patterns use edges with single and double heads to denote single and multiple edge paths, resp. Similarly, to allow a user to query about flows that are nested at any depth in the zoom-in hierarchy, compound activity nodes may have doubly bounded boxes, to denote an unbounded zoom in into the activities' internal specifications. The nodes and edges of BP patterns can be associated with variables, and these can be used in *selection conditions* on their attributes and data and for *joins*. We also support negation (denoted by *dashed* nodes and edges).

We demonstrate the use of BP-QL via some example queries. Each query describes a process pattern that a user is looking for. The check boxes next to nodes and edges mark selected nodes and paths, resp., that the user wants to to retrieve as the query result.

EXAMPLE 2.1. *The query in Figure 3 searches for operations provided by the* `Alpha-Tours` *BP* and *the services it uses. The double headed edges inside the* behavior *box indicate that activities at any distance from the start/end nodes may qualify; the shape of the node restricts the search to provided operations. The double bounding of the* behavior *box denotes* unbounded zoom-in*; we look for operations provided by the BP and (transitively) the compound activities/services that it invokes. The zoom-in is restricted to activities/services whose specifications reside on the* same peer*, since the* deepSearch *attribute is set to* local*. Setting it to* global *will extend the search to remote services as well.* ⊠

EXAMPLE 2.2. *Figure 4 illustrates a join operation. The query checks which VISA credit card services are called* (directly or indirectly*) by the Alpha Tour's* `confirmTrip` *activity. We use variables to define the join conditions. The join is value based, i.e. the nodes' attributes are checked to have the same values.* ⊠

EXAMPLE 2.3. *The query in Figure 5(a) illustrates the use of negation. It tests whether the users of* `Alpha Tours` *are never required to login when searching for flights. Formally, this is expressed by stating that a path to the* `searchFlights` *activity that passes through a login activity does not exists (dashed edges and nodes denote negation). The existing flow paths leading to* `searchFlights` *are then retrieved (as indicated by the small check box next to the double headed edge).*

A more lenient query, that retrieves, the paths without a login leading to `searchFlights`*, can be expressed by attaching a variable, say x, to the edge, along with the selection condition*

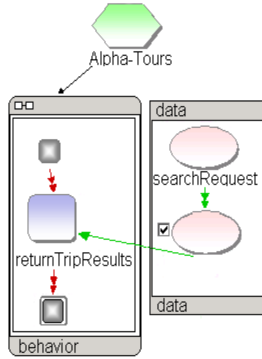

Figure 6: Data flow.

$x \in (\Sigma \setminus$ "login"$)^*$. See Fig. 5(b). Regular path expressions as constraints on paths are discussed in Section 3.4. ⊠

EXAMPLE 2.4. *Finally, Figure 6 illustrates querying the data flow. The query searches for data elements that are (transitively) affected by the `searchRequest`, and serve as input for sending the suggested trips back to the client. By default, a double headed edge between two data (resp. activity) nodes denotes paths consisting only of data (activity) flow edges. To override the default, (e.g. consider paths with all sorts of edges) one can attach, as above, a variable to the edge with an appropriate selection condition.* ⊠

2.3 Query Semantics (informally)

When a query is evaluated, its patterns are matched against the system BPs. Its nodes and edges are assigned activity/data/property nodes and execution/data flow paths, resp. These are then used to construct the query result.

More precisely, the semantics of a query q on system S is defined as follows. An *embedding* is a function from the nodes and edges of q to nodes, edges and paths of S, that satisfies the obvious constraints: Nodes are mapped to nodes of the same type, single/double-head edges are mapped to edges/paths between the corresponding end points. When a compound query node is doubly-bounded, nodes and edges in it may be mapped to nodes and paths in a process obtained by any number of zoom-ins into the activity's specification. For nodes and edges are associated with variables, the query constraints on these variables must be satisfied as well.

Each embedding defines *one result* for the query. The number of qualifying results may be large (possibly infinite in the presence of cycles). However, BP-QL provides a concise, intuitive (and finite) representation for the set. We illustrate this below with an example and provide more details on the construction in Section 3.

EXAMPLE 2.5. *Assume that the `searchFlights` service (invoked by `searchTrip` in Figure 2) has the structure depicted in Figure 7(a). The user can either login and check for the availability of various flights, or call, again, Alpha Tours' `searchTrip` service to start a new search. Now, reconsider the query in Figure 5(b), that retrieves the paths leading to `searchFlights` that do not require a login. Because of the potential cyclic service invocation, `searchTrip` can in fact be reached by an infinite number of paths, as depicted in Figure 7(b). Rather than listing all these paths, the user is provided with a compact representation (see Figure 8) that highlights the recursive structure of the results.* ⊠

3. THE FORMAL MODEL

In this section we briefly present the formal model underlying the BP-QL query language. We discuss the properties of the various language components and explain how these influenced our

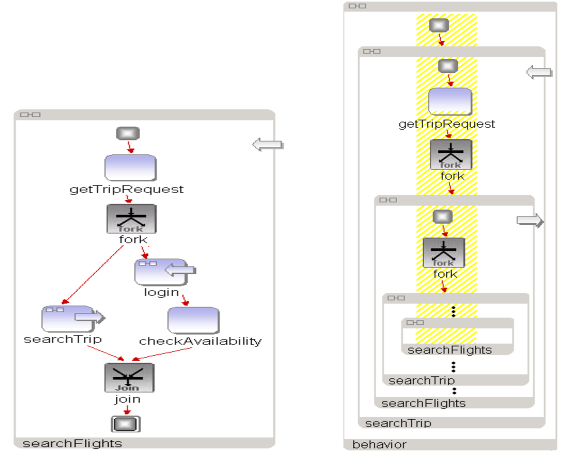

Figure 7: (a) searchFlights. (b) Infinite set of results.

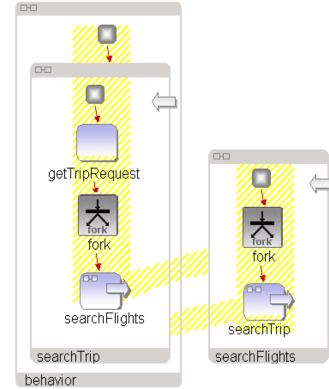

Figure 8: Finite result representation.

system's design. In particular we distinguish features that can be efficiently supported, and those that incur a prohibitively high cost, or cannot be computed at all. To simplify the presentation we first consider a basic data model and query language, then enrich them to obtain the full fledged model.

3.1 Simple Business Processes and Systems

We assume the existence of of two domains $\mathcal{N}$ of nodes and $\mathcal{L}$ of node labels. $\mathcal{L}$ is the disjoint union of several domains including data values, attribute names, data element names, process property names, and atomic and compound activity names. We assume some distinguished property names. These are introduced below, in the appropriate contexts.

Business graphs and processes. We model a (simple) BP as a directed labeled graph with nodes of two types: *concrete* and *compound*. Concrete nodes represent process properties, attributes, data elements, and atomic activities. Compound nodes represent compound activities, namely calls of (possibly remote) operations. Two distinguished nodes of the BP graph represent its start and end activities. Formally,

DEFINITION 3.1. *A (simple)* business graph *is a pair $g = (G, \lambda)$, where $G = (N, E)$ is a directed graph in which $N \subset \mathcal{N}$ is a finite set of nodes, and E is a set of edges with endpoints in N; and $\lambda : N \to \mathcal{L}$ is a labeling function for the nodes. Depending on their label type, we refer to the nodes in g as* activity *nodes,* value *nodes,* property *nodes, etc. Nodes labeled by compound activity names are called* compound *nodes; all other nodes in g are called* concrete.

A (simple) business process *(BP) is a triple $p = (g, \texttt{start}, \texttt{end})$,*

where: g is a business graph; `start`, `end` *are two distinguished activity nodes in g; and each activity node in g resides on some path from* `start` *to* `end`. ⊠

Note that the start and end nodes need not be distinct. For example, a process may consists of just one activity node, which is both its start and its end. Also note that only activity nodes are restricted to be between the start and end nodes. Recall from section 2 that activities can be classified as *requested* or *provided*. This is modeled by assuming two particular property names provided and requested, and attaching to activity nodes appropriate property nodes.

For example, Figure 9 shows several BPs (ignore the "bubbles" for now). As before, we use squares for activity nodes and hexagons for property nodes. The leftmost BP has a single compound activity node, which is both its start and end. The one in the center has two distinct start and end nodes, and four provided operations.

As mentioned above, compound nodes represent calls to composite operations. The internal structure of these operations is not part of the business process graph and is given separately, as we explain next.

Simple systems. A *system* is a collection of business processes (or graphs), along with a mapping between compound nodes and their implementations – the processes they invoke. In the general case, a system may be distributed. This is ignored for now, for simplicity, and is discussed in Section 3.4.

DEFINITION 3.2. *A* system *S of business processes (resp. graphs) is a pair (P, τ), where P is a finite set of business processes (graphs), and τ is a (possibly partial) function, called the* implementation function, *from the compound activity nodes in P to business processes (graphs) in P.*[3] ⊠

This definition can easily be extended to distinguish between *root* processes, that are directly accessible, and *implementation* processes, that are accessible only as implementations of other processes. To simplify the presentation we omit this here.

The implementation function is partial when the internal structure of some compound activities is unknown (for instance when their providers do not wish to expose their specification). Recall from definition 3.1 that the only difference between business *graphs* and business *processes* is that the latter have distinguished start and end nodes. Systems of *processes* are used to model real life applications. Systems of *graphs* will prove useful to model query answers. For brevity, since we will mostly be dealing with systems of *processes*, unless stated otherwise the term *system* should be interpreted as *system of processes*.

Figure 9 shows a partial system. This is a partial description of the Travel Agency system from Figures 1,2 (for simplicity, the data and attribute nodes are omitted). The full system should also contain, for example, the processes of the airline, car reservation, and hotel companies.

System Refinement. Given a system S, some BP p in it, and a compound activity node n in p, a more detailed description of p (and hence of S) can be obtained by zooming-in and replacing the node n by its implementation. We call this a *refinement*.

DEFINITION 3.3. *Given a system $S = (P, \tau)$ and a BP p in P, we say that $p \to p'$ (w.r.t. τ) if p' is obtained from p by replacing some compound activity node n in p by its implementation $\tau(n)$. [Namely, n is deleted from p, and a copy of the BP $\tau(n)$ is plugged in its place, with the incoming/outgoing edges of n now being connected to the start/end nodes of $\tau(n)$, resp. If n was the start/end node of p, the start/end node of $\tau(n)$ now takes this role.]*

If $p \to p_1 \to \ldots \to p_k$ we say that p_k is a refinement *of p.*

We say that $S \to S'$ (w.r.t. τ) if S' is obtained from S by replacing the implementation p of some compound activity node n in S by a refinement p' of p. [Namely, a copy of p' is added to P, the mapping τ for n is updated to point to it, and τ is extended to map compound nodes in p', to the same BPs as in P. Finally, if p is no longer the implementation of any node, it is removed from P.]

If $S \to S_1 \to \ldots \to S_k$ we say that S_k is a refinement *of S.* ⊠

[3] In an actual system, $\tau(n)$ can be represented by attaching to n the peer and process id (Web service URL, operation name, etc.) for the implementation of n.

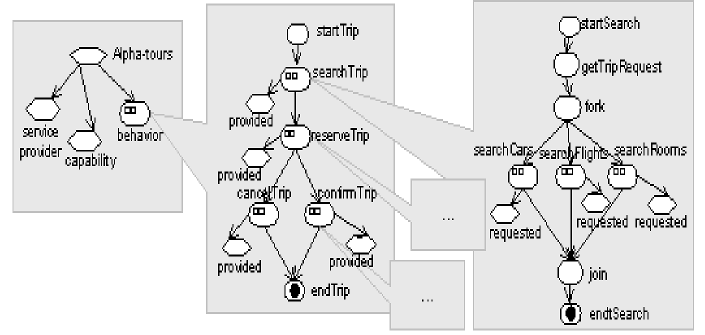

Figure 9: A system of BPs.

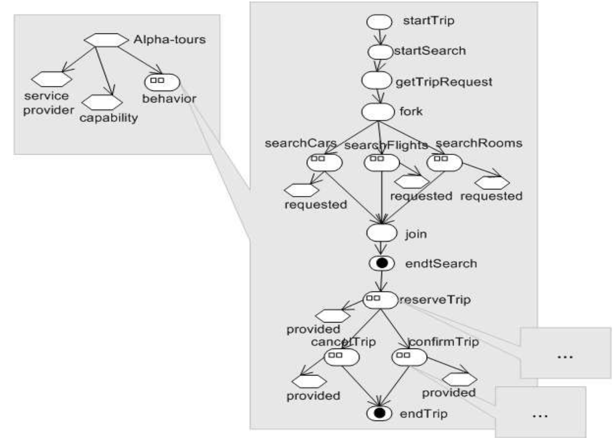

Figure 10: A refined system (after one step).

Note that if S is a system, then each of its refinements is also a system. Figure 10 shows a refinement of the system from Figure 9, after one refinement step, in which the implementation of `behavior` was refined: the node labeled `searchTrip` has been "zoomed into" and replaced by its implementing process.

3.2 Simple 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 in Section 3.4.

Queries. Queries are modeled using *BP patterns*. These generalize BPs similarly to the way tree patterns generalize XML trees. The labels of nodes can be specified, or left open using $*$. Edges in a graph can be either single-headed, in which case they are interpreted over edges, or double-headed, in which case they are interpreted over paths. Similarly, nodes have a single or a double boundary, for searching only in the direct implementation of the node or in all its refinements, resp. We call edges with double head

(resp. nodes with double boundary) *transitive* edges (nodes).

DEFINITION 3.4. *A BP* pattern *is a tuple* (p^*, T, R)*, where*

1. p^* *is a BP where nodes are labeled by elements from* $\mathcal{L} \cup \{*\}$,
2. T *is a distinguished set of edges and compound nodes in* p^* *called the* transitive *edges and nodes, resp.*
3. R *is a distinguished set of edges and nodes in* p^* *called the* result *edges and nodes, resp.*

A simple query q *is a system of* BP patterns (Q, τ)*, where* Q *is a set of BP patterns, and* τ *is an implementation function.* ⊠

To evaluate a query, its patterns are matched to those of (refinements of) the system. A match is called an *embedding*.

DEFINITION 3.5. *Let* $q = (Q, \tau)$ *be a simple query and let* S *be a simple system. An* embedding *of* q *into* S *is a homomorphism* ρ *from the nodes and edges in* q *to nodes edges and paths in some refinement* $S' = (P', \tau')$ *of* S *s.t.*

1. **(nodes)** *each start (resp. end) node in* q *is mapped to a start (resp. end) node in* S'*; each concrete node in* q *is mapped to a concrete node in* S' *of the same kind; and a node with a constant label is mapped to a node having the same label.*
2. **(edges)** *each (transitive) edge from node* m *to node* n *in* q *is mapped to an edge (path) from* $\rho(m)$ *to* $\rho(n)$ *in* S'.
3. **(implementation)** *For each compound activity node* n *in* q*,* ρ *maps the nodes and (transitive) edges in* $\tau(n)$ *to nodes and edges (paths) in* $\tau'(\rho(n))$*. If* n *is not transitive then* $\tau'(\rho(n))$ *must be an original BP of* S *(i.e. not a refinement).*

The query result *defined by* ρ *is the image under* ρ *of* q*, restricted to its output nodes and edges. If the same node/edge occurs several times in the image, distinct copies are used for each occurrence.* ⊠

The result associated with an embedding ρ is, in general, a system of graphs. As the number of such qualifying results may be large (possibly infinite) we provide a concise (and finite) representation for this set.

3.3 Compact representation of query results

To understand the construction of this concise representation, let us first look at the two main factors that contribute to large or infinite answers. We consider first flat graphs, i.e. BPs with no compound activities, and then nested BPs.

Flat BPs When a BP contains cycles, the number of paths that may match a given (transitive) query edge may be infinite. Observe that even when a BP is acyclic, the number of matching paths may be large. For example if the activity flow forks into several paths and then joins back, forks and joins again, and so on, several times, the number of possible paths is exponential in the number of forks. The solution to this problem is easy: We can represent the set of paths between two nodes by a copy of the sub-graph that connects the nodes. One might actually say this is what the user intended: to see the specification of the paths between the two nodes, rather than the individual paths themselves.

Nested BPs Things become more complex in the presence of compound activities. A system may contain recursive activity implementations, hence have an infinite set of refinements. Since the results of a query are constructed from embeddings into all the refinements, there may be infinite number of such possible results as well. The solution here is based on viewing systems and queries as *context free graph grammars*[21], abbreviated CFGG. (Confusingly, these are also called in the literature *regular graph grammars*. We will use only *context free* in this paper.) A CFGG is a finite set of graphs, where graphs may contain non-terminal symbols, and where grammar rules allow to replace a non-terminal by a graph from a given finite collection.

The intuition is that, for a system S, the implementation relationships correspond to grammar rules; the system refinements correspond to the graph language defined by the grammar. Similarly, a query q can also be viewed as CFGG whose graph language consists of all the graphs that satisfy the query constraints (i.e. contain the patterns specified by the query). This intuition can be extended to the query answer: Instead of constructing explicitly the potentially infinite set of results, one may construct a CFGG that represents it. Specifically, the query answer can be viewed as some kind of "intersection" of the languages defined by the system and query grammars (followed by a "projection" that omits the portions that were not requested as output).

In general, the intersection of two CFGG languages may not be a CFGG language [21]. (This generalizes the same property for string CFGs.) In our particular case, however, the query specification is sufficiently simple to guarantee the required closure: one can show that it belongs to a restricted class of CFGGs called *recognizable sets* [14], for which the intersection with another CFGG is known to yield a CFGG. This implies that in principle one could employ the intersection algorithm presented in [14] to construct a finite representation for the query results. The problem, however, with this direct solution is that the algorithm of [14] is of high complexity - exponential in the size of the BPs [4] - hence impractical for query evaluation. An important result of the present work was to detect that BP-QL queries form a subclass of the recognizable sets for which PTIME solution is possible, and to design such an algorithm.

Our algorithm is based on a *modular construction* of a CFGG that describes the query results. It relies on the following two ideas.

1. The first idea is that each query result is a combination of smaller results that describe how one query process, say p^q, is mapped to one system process, say p^S. The combination must satisfy the condition (implementation).
2. The second idea is that many embeddings share the same underlying *node mapping*, and differ only on the assignments to the, possibly transitive, edges. Thus, many results for a pair p^q, p^S, may be represented together by using this shared node mapping (and, as we did above for flat BPs, representing the different possible path assignments for transitive edges between the nodes by a copy of the sub-graph that connects the nodes.) Of course, when a node mapping from p^q to p^S is considered, one must ensure that it satisfies the conditions (nodes) and (edges).

Thus, a CFGG representation for the query results is obtained from a collection of *node mappings* between *distinct pairs of query and system processes*, that satisfy the conditions above.

We next sketch the main lines of this construction. Give a query q and a system S, consider some BP pattern p^q in q and a BP p^S in S. The important observation is that an embedding from p^q to a *refinement of* the system process p^S consists of several parts. The first part maps some of the query nodes and edges of p^q into p^S itself. Subsequent parts map additional nodes and edges of p^q into *implementations* of compound activity nodes of p^S, and so on. The nodes and edges in the query pattern p^q that are *not* mapped to p^S itself, but into implementations of its compound nodes, must have a structure that fits such implementations. In particular, they should form a set J of disjoint sub-graphs of p^q, each with a single entry and exit nodes.

[4] Furthermore, to our knowledge, no PTIME algorithms for this intersection problem are known.

Thus, the construction is recursive. For each such set J it modifies the query pattern p^q, by replacing each sub-graph G in J by a distinct new node labeled $*$ (denoted in the sequel $*_G$), obtaining a modified query graph p^q/J. Then it constructs representations for all results obtained from embeddings from the modified query p^q/J to the BP p^S (grouping together, as explained above, embeddings that agree on their node mappings). Assume the $*_G$ node was mapped to a compound activity node, say A. In the representation, this node is noted as A_G, and it serves as a non-terminal of the grammar. Then, for each G and A_G, it recursively constructs representations, say R_G, for results obtained from embeddings from G to the implementation of A. The grammar rule then allows to replace A_G by R_G.

Special care must be paid in the construction above to transitive edges. These may be mapped to arbitrarily long paths, that may start in one system BP p^S, continue in an implementation of a compound activity node of p^S, and so on, possibly through a cycle of implementations. Such paths are broken into components by introducing special dummy nodes into the query pattern p^q, then employing the construction as described above.

For lack of space, we omit here the presentation of the full algorithm and its correctness proof. (These are available in the full version of the paper[6]). Note that almost all graphs for which representations are constructed are sub-graphs of the query patterns p^q. (The introduction of dummy nodes complicates this argument a bit.) The construction terminates when each mapping from each such graph to each system process has been considered (or before that, if embeddings for some sub-graph are never required).

The important point to observe is the following.

THEOREM 3.6. *The size of the representation for the query results, constructed by the above algorithm, as well as the construction time, are polynomial in the size of the system S (with the exponent determined by the size of q).*

The intuition is that, in the worst case, each sub-graph of each query pattern needs to be mapped into each system BP. The number of query sub-graphs of the appropriate form is a function of the query size alone. Since embeddings that agree on their node mappings are represented together, it suffices to count the distinct node mappings. For each query sub-graph and each system BP, this number is polynomial in the size of the BP (with the exponent determined by the size of the query sub-graph.)

3.4 A Richer model

So far, for the sake of simplicity, we used a very simple data model and query language. We now present some useful extensions that enhance the expressive power, and facilitate the querying of real life business processes.

Negation. In a query *with negation*, the patterns have 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 embedding: Given a query q with negation, the *positive part of* q, denoted $positive(q)$, is the query obtained from q by deleting all the negative edges and nodes, and all the edges incident on these nodes. The embeddings of q into S are those embeddings of $positive(q)$ which cannot be extended to an embedding of any query q' obtained from $positive(q)$ by adding all the negative nodes and edges of some of its BP patterns. The answer of q is defined as before, based on the above embeddings.

A finite representation for the query answer can be constructed essentially as above. The only difference is that only embeddings for the positive portions of the query graphs, which cannot be extended to include the negative portions, are considered.

Label predicates and regular path expressions. The simple queries considered so far only allow nodes with a *particular label* or $*$. But sometimes one may be interested in system nodes that conform to certain conditions. For instance, rather than searching for the `searchFlighs` activity, we may want to retrieve all the activities whose name contains the string "search". This can be achieved by using *label predicates*. In an embedding, a query node labeled by a label predicate must be mapped to system node whose label satisfies the predicate.

Another useful feature are regular path expressions. Transitive edges in the query may be annotated by regular expressions. In an embedding, such edges must be mapped to paths such that their label sequence forms a word in the corresponding regular language.

The construction of a finite representation for the query answer extends naturally to support these two extensions.

Variables and joins. Together with label predicates and regular path expressions, one may also want to use label and path variables and test for (in)equality of the assigned labels and paths. The interpretation is that query nodes labeled by (un)equal label variables are mapped to system nodes with (distinct)identical labels; query edges labeled by (un)equal path variables are mapped to paths whose sequences of labels are (different)equal words. While the use of label variables poses no particular problem, for queries with joins on *path variables*, our construction may fail; the answer to such queries may *no longer be representable as a finite system.*

To understand why, recall that our systems may be viewed as CFGGs. A query that tests for equality of path variables may have for an answer sets of graphs that are not a CFGG language and are inherently harder to compute, as illustrated by the following theorem. The theorem also highlights the difference in *computational complexity* between the querying of flat and nested graphs.

THEOREM 3.7. *For queries with equality conditions on path variables, the problem of testing whether the query answer is empty on a system is undecidable.*

The problem can be solved in exponential time if the system to which the query is applied has no recursive activities. It is PSPACE-hard w.r.t the system size, even if the system BPs also have no cycles.

Finally, for flat BPs, it can be solved in time polynomial in the size of the system (with the exponent determined by the size of q).

Proof:(sketch) The undecidability and hardness proofs are by reduction to the problem of testing whether the intersection of the languages of two *string* context free grammars (CFGs) is empty. Given two CFGs G_1,G_2, we build a system S with a BP that has two branches. The first contains a compound activity node g_1 and the second a compound activity g_2. The implementation of g_i, $i = 1, 2$, (which resembles in spirit the grammar rules of G_i), is defined such that each of its possible refinements has line-shaped structure, representing a word in the context free language of G_i. The implementations are defined such that g_1 and g_2 can be refined to an activity sequence with the same shape iff this sequence represents a word that belongs to both G_1 and G_2. Next, we define query q with two transitive edges e_1,e_2 that match (the refinements of) g_1 and g_2 resp., and have an equality condition on their attached path variables. It is easy to see that the query has a non empty result iff the languages intersection is not empty. This is known to be undecidable in the general case, and was recently proved to be PSPACE-complete for non-recursive context free languages [29].

The polynomial and exponential algorithms work as follows. For

flat BPs, the algorithm considers all possible mappings of query nodes to the BP. Testing join conditions here amounts to testing if the intersection of the regular languages defined by the sub-graph that connects the nodes is empty, which can be done in PTIME. For nested, non-recursive, BPs, the algorithm enumerates all the system refinements (possibly an exponential number) and tests for the existence of a legal embedding in a similar way. ⊠

We have consequently decided to restrict the use of path variables in BP-QL and allow joins only on *label variables*.

Distributed systems and queries. So far, we have ignored distribution. In a distributed setting, each peer holds a set BPs and may provides (resp. use) activities to (of) remote peers. If the service providers make their specification available to their cooperating organizations (say via a web service), users may wish to zoom-in on these remote components as well to query the service specification.

The data model extends naturally to this setting, associating a peer id with each process and each activity node. Queries may then annotate graph patterns and activity nodes by peer ids, restricting the search to the specified peers. In particular, when a (transitive) activity node in a query is annotated by a peer id, the search is restricted to implementations supplied by the specified peer (resp. refinements consisting only of invocations of activities of the specified peer). More generally, queries may use predicates on peer ids to restrict the search to a specific family of peers.

Remark: While the extension of the formal model to a distributed setting is rather immediate, implementation-wise, distribution poses significant challenges in terms of query evaluation. Specifically, we would like to evaluate a query in a "lazy" manner, so that only those peers whose processes and activities are indeed relevant to the query are consulted. Furthermore, it is desirable to "push" parts of the query, when possible, to the peers holding the relevant process information. Our implementation, described in the next section, addresses these issues.

Summary. The design of BP-QL was directed by the special requirement of querying specifications with a zoom-in feature at different levels of granularity and the retrieval of qualifying execution paths. As explained above, this required a careful design of the language to avoid features that might seem to be worthy of inclusion in the language, such as joins on path variables, but incur a prohibitively high computational cost.

The characterization of the exact expressive power of BP-QL is an on-going research. Our initial results indicate that BP-QL can be characterized as a particular subclass of FO(TC)[5]. In particular, for flat BPs BP-QL captures power similar to that of the conjunctive part of XPath and core XQuery, including negation, when considered in the context of graphs. Due to space limitations this is not presented here.

4. IMPLEMENTATION

The query language presented above has been fully implemented and tested in the BP-QL peer-to-peer system. The system provides persistent storage for BPEL specifications, allows users to design new processes, and to query existing specifications.

The visual interface of the system is implemented as an Eclipse [20] plug-in. It allows to: design new business processes and store their specifications in the repository; import existing BPEL documents to the repository; formulate queries, run them and view the results. The rest of the section is devoted to the main component — the query engine.

[5]First Order Logic augmented with Transitive Closure.

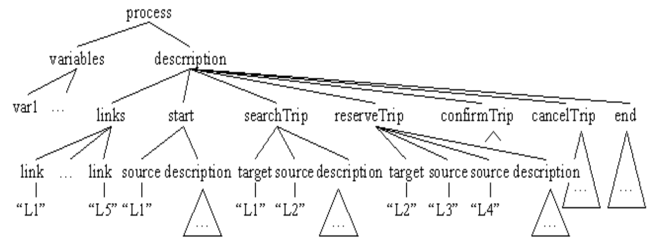

Figure 11: BPEL XML.

4.1 Design Considerations

BP-QL is based on an intuitive, conceptual model of BPs, an abstraction of the BPEL specification, allowing for simple formulation of queries. over this model. When we considered the implementation, the following problem had to be addressed: As mentioned in Section 1, the BPEL XML format was designed with ease of *automatic code generation*, rather than *querying*, in mind. Activities and edges are defined separately, as distinct activity and link elements. The process flow is only recorded by associating with each activity element the ids of its incoming and outgoing edges, represented resp. by *target* and *source* children of the node. This is illustrated in Figure 11, which shows the BPEL XML representation[6] of the Travel Agency business process from Figure 1. Consequently, to check whether flow paths of a given process satisfy the conditions detailed in a BP-QL query, a large, *possibly unbounded*, number of join operations involving edge ids between activity and edge elements needs to be performed. While this is expressible in, say, XQuery, e.g. with the use of recursive functions, the excessive number of joins becomes a performance bottleneck.

To drastically reduce the number of joins, we decided to store a process specification in a structure more similar to its graph view. In XML terms, the parent child relationships in the XML representation of a process should reflect the "followed by" relationship of nodes in the process graph. This would allow the use of XPath's "/" and "//" operators for querying flow paths, avoiding many joins. But, since a typical BP is a *graph*, rather than a *tree*, we also use XML *idrefs* to capture the graph structure.

Another fundamental decision to be made was which of the following two options to choose: (1) to implement a whole new query engine for our model from scratch, or (2) to rely on some existing query engine to perform as much as possible from the computation, and complete the processing of the missing features by an adequate pre and post processing of queries and query results. We opted for the second option. The issues to be considered in selecting an engine were the following:

- Our query language allows to retrieve paths, whereas typical existing XML/graph query languages only retrieve nodes.
- Our query language offers a zoom-in facility.
- Business processes typically operate in a cross-organization, distributed environment. The specifications of the services participating in process may reside on distinct peers. Distributed query processing thus becomes essential.

A natural candidate was to use a standard XQuery engine, enjoying the benefits of indexing and optimization offered by such engines.[7] However, XQuery does not support the retrieval of paths, distribution, or zoom-in queries; nor does it "traverse" idrefs. Necessarily, all of these would have to be implemented by pre and post

[6]For simplicity, the figure provides an abstraction of the actual BPEL XML file structure, with many details omitted.

[7]An alternative viable solution to the graph shape of BPs could be to use a native graph query engine.

processing. Consequently, we decided to base our solution on an extension of XML, called Active XML (AXML for short). AXML is essentially a middleware system that includes an XQuery-like query language, but offers additional facilities which provide better support for addressing some of the above issues. Additional benefits include certain optimization techniques that are implemented in the AXML system, as explained below.

A brief overview of Active XML. Active XML (AXML, for short) is a declarative framework that harnesses Web services for data integration, and works in a peer-to-peer architecture[3]. An AXML document is an XML document where some data is given extensionally, as regular XML elements, while other data is given intensionally, by means of calls to Web services[3], and can be *materialized* by invoking the services. AXML employs the query language XOQL, an XQuery-like query language as its query engine. When a query is evaluated on an AXML document, the service calls whose answer *may be relevant* for the query are identified; only these calls are invoked. Additionally, (sub-)queries are *pushed*, when possible, to the service providers, thus reducing the costs of data materialization and transfer. Recursive calls are tracked, and only the relevant data is materialized (see [2] for details).

In summary, BP-QL uses the AXML system [3] as an implementation platform. The facilities offered by AXML are used to address our needs, as follows: Intentional data, implemented by service calls, are used in our implementation to (1) retrieve, when needed, the specifications of remote processes, thus supporting distributed processing, and (2) account for the graph structure of the specification (service calls play here role similar to XML idrefs, with the advantage that they are traversed automatically in query evaluation). BPEL documents are wrapped and represented as AXML documents; BP-QL queries are pre-processed and compiled into a set of XQuery-like queries over such documents. Post processing is employed to complete the computation, e.g. to validate zoom-in relationships, to extract paths and to construct a compact representation for the result.

From BP-QL to AXML. Here is a brief description of the AXML representation of a BP-QL business process. The representation consists of three parts: Process properties (such as the service provider, the service type and capabilities) are maintained as UDDI entries in a (standard) XML document. The other two, namely the process activities and execution flow, and the data elements and the data flow, are maintained in two AXML documents. The use of two AXML trees, rather than one, allows for efficient evaluation of BP-QL queries with double headed edges: it allows a doubly headed activity (resp. data) flow edge to be mapped to a "//" operator on the corresponding AXML document.

For example, Figure 12 describes (part of) the AXML tree for the Alpha-Tours activities and flow. (Here again, for simplicity, only an abstraction of the actual AXML tree is provided, with many details omitted.) Each activity is represented by an XML element node in the tree. The parent child relationships reflect the flow. Each node representing a compound activity is the root (labeled by *zoom-in*) of a subtree that describes the internal structure of the activity. Nodes with bold labels are special elements that represent calls to Web services. Two types of such calls are embedded in the document:

- A *getActivity* service call plays a role similar to that of an XML *idref*, "pointing" to a certain node in the tree. When a query is evaluated, the relevant calls are detected and invoked. (Cycles are detected and cut by AXML). For each call, the returned data (a copy of the sub-tree "pointed to") is inserted in place of the service call, ready to be accessed. Thus query evaluation can access the returned subtree as if it actually traversed the "pointer".
- A `getOperation` service call retrieves the specification of a remote compound activity and converts it, when needed, from BPEL format to an AXML representation. A zoom-out element is attached to its final state, so that it points to the following activity in the flow.

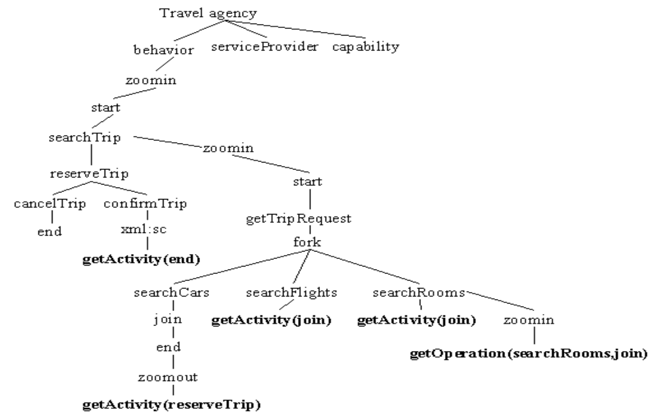

Figure 12: AXML tree.

To illustrate the first type of call, the `getActivity("join")` nodes below the `searchFlights` and `searchRooms`, in the middle of Figure 12, point to the `join` node below `searchCars`. They represent the fact that the three searches are followed by that same `join` operation. [8]

The `getOperation("searchRooms,"join")` in the figure illustrates the second type of call. It retrieves the specification of the *searchRooms* process, and set its zoom-out to the following "join" operation. Here again, AXML invokes `getOperation` calls for the remote activities whose specification is judged to be relevant for query evaluation. As mentioned above, it may also "push" (sub-)queries to capable service providers, such as BP-QL peers that "understand" BP-QL queries.

Data elements and data flow are represented in AXML tree in a similar manner: The tree contains both data and activity element nodes. `getData` and `getActivity` service calls are used as "references" between tree data and activity nodes, resp.

To generate the AXML representation, the BP-QL graph is traversed in a depth-first order, building AXML trees as deep as possible. Local compound activities are then zoomed-in and their graphs are similarly detailed, recursively. Requests to remote operations are represented by `getOperation` service calls. Web services are generated for provided operations, exposing their specification to the requesting peers.

With this representation, both the path-based and the zoom-in axis conditions can be evaluated using XOQL queries on the AXML documents. Some post processing is nevertheless required to match up the components, extract the requested paths (XOQL, like most XQuery engines, returns only document elements not paths), and construct a compact representation of the result. We omit the details for space constraints.

4.2 Trade-offs

As explained above, we have decided to store BPs in a structure close to the BP graph shape, rather than in the BPEL format. Obviously, this reduces the number of join operations required in query

[8] In the implementation, the input to the `getActivity` is a unique identifier for the pointed activity, consisting of BP and activity ids. It is abstracted here, for brevity, by the activity name.

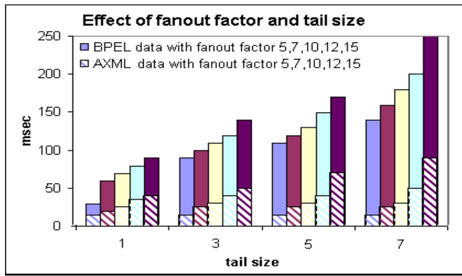

Figure 13: Varying depth and width.

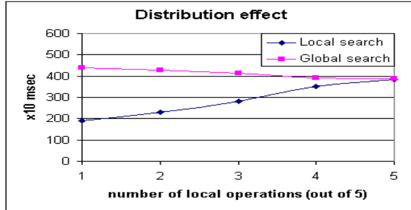

Figure 14: Distribution effect.

evaluation. With this representation, it is still necessary to account for the graph structure of BPs. This can be taken care of by performing joins. Instead, the use of AXML allows to represent "cross edges" by service calls. The price payed for this, performance-wise, is the invocation of service calls: For example, *getActivity* calls are invoked when "pointers" need to be traversed.

To understand the trade-offs, we performed the following experiment. We considered BPs with varying *depth* and *width*, where *depth* is the maximal length of (simple) paths from the start node to the end node of a BP; and *width* is the maximal in-degree of nodes in its graph. They reflect, resp., the number of joins saved by moving from a "flat" BPEL format to the hierarchical representation, and the number of service calls that may be invoked when "traversing pointers" to a given node. We selected as a representative class of path-oriented queries those that search for the occurrence of a given activity, followed (at an arbitrary distance) by another given activity. All the tests were performed on IBM Laptop T43, 1.86 Ghz, 1Gb of RAM with Windows XP, sp2. A representative sample of results is shown in Figures 13. The BP graphs here include a *fork* activity that splits the flow into 5,7,10,12 and 15 different paths that are joined later, and followed by a tail of length 1,3,5 and 7 (on the x-scale). We measured the respective evaluation time of the (translated) BP-QL queries on the AXML and BPEL representations of the BPs. The AXML result columns are presented in front of the BPEL columns. For clarity, the figure shows only the net query running time; the time of Web service calls is excluded from AXML columns. By our measures, an average `getActivity` service call takes about 100msec. AXML performs most calls in parallel, so the typical overall delay due to the materialization of data is also around this number.

As we can seen, the running time of queries (for both BPEL and AXML) grows linearly with the BP width. (For BPEL, this is because more nodes participate in the joins. For AXML, this is because the "//" has more paths to traverse.) For narrow graphs, although the use of our representation reduces the number of joins, the relative overhead of service calls is substantial. The relative benefit of using our representation and AXML over using the BPEL representation for wider graphs grows with the BP depth. For depth greater than 7 (values larger than 7 are omitted from the figure), the gain from the saving of joins outweighs the additional cost of data materialization via service calls.

While the use of Web services brings some (moderate) overhead to query processing, it allows for greater flexibility in distributed data processing. To see if (and how) the distribution of data effects query processing we performed the following experiment. We considered business processes consisting of several compound activities, and varied the number of peers that hold the specifications of activities. At one extreme, the full specification resides on a single peer. At the other extreme, each process activity is provided by a distinct peer. We compared the execution time of queries on these varying configurations, considering both *global* queries (that consult the specifications on all peers) and *local* queries (where the search is restricted to only local specifications.) Figure 14 illustrates a representative sample of the results. It considers the Travel Agency from our running example, and the query from Figure 3 (with the search scope set to local and global, resp.). We varied the number of local compound activities (operations whose specifications reside on the local machine) from one to all (5), moving the remaining specifications to remote peers. We see that the cost of the global queries is practically independent of the distribution level. Not surprisingly, the execution time of the local query increases linearly as more portions of the BP are local, since more data is available for querying.

5. RELATED WORK AND CONCLUSION

We presented BP-QL, a novel graphical Query Language for querying Business Processes. BP-QL allows users to query business processes visually, in a manner very close to how such processes are typically specified, and can be employed in a distributed P2P setting. We described the formal model underlying the BP-QL query language, studied the properties of the language components, and explained how they influenced the language design. We have also described the system implementation, highlighting the main challenges faced and the solutions taken.

The BP-QL language is based on an intuitive model of business processes, an abstraction of the emerging BPEL (Business Process Execution Language) standard [7]. Other previously proposed standards like [18, 9, 16] can similarly be supported, exploiting the abstraction level of our formal model.

There has been a vast amount of previous work in the general area of program analysis and verification (see e.g. [22, 26] for a sample), and more specifically in the analysis of interactions of composite web services and BPEL processes [19, 22, 17]. These works mostly consider logic-based query languages where queries, formulated as logic formulas, test if the runs of the application or program satisfies a certain property; a witness counter example is provided if not. In contrast, we advocate here an intuitive, visual query formulation, where queries are written in essentially the same way as process specifications. BP-QL allows not only to test if a certain pattern occurs, but also displays to the user all the relevant paths. Indeed a major contribution of the present work is the construction of a concise finite representation of the (possibly infinite) set of results.

As mentioned in Section 1, program verification is typically of very high complexity (from NP-hard for very simple specifications to undecidable in the general case [28, 22].) To guaranty complexity that is polynomial in the size of the data, BP-QL queries process *specifications*, rather than possible runs, ignoring the run-time semantics of certain BPEL constructs such as 'choice', parallel execution, and variable values. Identifying semantic constructs that can nevertheless be incorporated without increasing complexity is a challenging future research. It is also interesting to study whether certain data structures (e.g. BDD [26]) that are used to speed up program verification tasks can also be employed in our context to

further accelerate query evaluation.

The design of BP-QL was inspired by previous works on visual query languages for XML, such as XML-GL [12] and XQBE [10]. These languages are descendants of a long line of research on graph based query languages such as G [15], Graphlog [13] and G-Log [31]. The main innovation of BP-QL is in introducing *process patterns* that enrich the standard path-based navigation with (1) a (transitive) zoom-in, that allows to query process components at any depth of nesting, and (2) the retrieval of paths of interest. Together, these features allow for simple formulation of queries on BPs, but also make the evaluation of queries more intricate than that of flat graphs. To keep the evaluation of queries tractable, we had identified the problematic scenarios and carefully designed the language so that they are avoided, and polynomial-time query evaluation is guaranteed. We are currently extending the language to allow also for the construction of new processes based on the retrieved data.

The importance of query languages for business processes has been recognized by BPMI (the Business Process Management Initiative) who started a BPQL (Business Processes Query Language) initiative in 2002 [8]. However, no draft standard was published since. We hope that BP-QL will contribute to such a standard. Complementary to our work is the research performed in the area of Business Process Management (BPM) and Business Process Intelligence (BPI). Both academic (e.g., [32, 11, 33] and commercial tools (e.g., [4, 24, 25]) have been developed to support the definition, execution, and monitoring of BPs, including systems for extracting knowledge from event logs (process mining). We are currently extending BP-QL to serve as a basis for a general query platform, that allows queries that involve process specifications as well as execution data.

6. REFERENCES

[1] W3C Working Group Note 11. Web services architecture usage scenarios, Feb. 2004. http://www.w3.org/.

[2] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy Evaluation of Active XML Queries. In *Proc. of ACM SIGMOD*, 2004.

[3] Active XML. http://activexml.net/.

[4] BEA. Weblogic application server. http://www.bea.com.

[5] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying Business Processes with BP-QL (demo). In *Proc. of VLDB*, 2005.

[6] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying Business Processes. Tech. Report, Tel Aviv University, 2006.

[7] Business Process Execution Language for Web Services. http://www.ibm.com/developerworks/library/ws-bpel/.

[8] BPMI. Business process management initiative: Business process: Business process query language (bpql). http://www.service-architecture.com/web-services/articles/business_process_query_language_bpql.html.

[9] BPMN. Business process modeling notation. http://www.bpmn.org/.

[10] D. Braga, A. Campi, and S. Ceri. *XQBE* (*xq*uery *b*y *e*xample): A visual interface to the standard xml query language. *ACM Trans. Database Syst.*, 30(2):398–443, 2005.

[11] M. Castellanos, F. Casati, M. Shan, and U. Dayal. ibom: A platform for intelligent business operation management. In *ICDE*, pages 1084–1095, 2005.

[12] S. Comai, E. Damiani, and P. Fraternali. Computing graphical queries over xml data. *ACM Trans. Inf. Syst.*, 19(4):371–430, 2001.

[13] M. Consens and A. Mendelzon. The g+/graphlog visual query system. In *Proc. of ACM SIGMOD*, page 388, 1990.

[14] B. Courcelle. The monadic second-order logic of graphs i: Recognizable sets of finite graphs. *Information and Computation*, 85:12–75, 1990.

[15] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In *Proc. of ACM SIGMOD*, pages 323–330, 1987.

[16] Daml services (daml-s/ owl-s). http://www.daml.org/services/owl-s/.

[17] A. Deutsch, M. Marcus, L. Sui, V. Vianu, and D. Zhou. A verifier for interactive, data-driven web applications. In *Proc. of ACM SIGMOD*, 2005.

[18] The ebxml bpss (ebbp). http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ebxml-bp.

[19] E.Clarke, O. Grumberg, and D. Long. Verification Tools for Finite State Concurrent Systems. In *A Decade of Concurrency-Reflections and Perspectives*, volume 803, pages 124–175. Springer-Verlag, 1993.

[20] Eclipse foundation. http://www.eclipse.org.

[21] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[22] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. of the Int. WWW Conf.*, 2004.

[23] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Comp. Programming*, 8:231–274, 1987.

[24] HP. Openview bpi. http://www.hp.com.

[25] Ilog jviews. http://www.ilog.com/products/jviews/.

[26] M. Lam, J., V. B. Livshits, M. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *PODS*, pages 1–12, 2005.

[27] F. Leymann. Web Services Flow Language (WSFL) 1.1, May 2001. http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.

[28] S. Narayanan and S. McIlraith. Analysis and simulation of web services. *Compute Networks*, 42:675–693, 2003.

[29] Mark-Jan Nederhof and Giorgio Satta. The language intersection problem for non-recursive context-free grammars. *Inf. Comput.*, 192(2):172–184, 2004.

[30] Oracle BPEL Process Manager 2.0 Quick Start Tutorial. http://www.oracle.com/technology/products/ias/bpel/index.html.

[31] J. Paredaens, P. Peelman, and L. Tanca. G-log: A graph-based query language. *IEEE Trans. Knowl. Data Eng.*, 7(3):436–453, 1995.

[32] D. M. Sayal, F. Casati, U. Dayal, and M. Shan. Business Process Cockpit. In *Proc. of VLDB*, 2002.

[33] B. van Dongen, A. de Medeiros, H. Verbeek, A. Weijters, and W. van der Aalst. The prom framework: A new era in process mining tool support. In *ICATPN*, pages 444–454, 2005.

[34] The World Wide Web Consortium. `http://www.w3.org/`.

[35] XLANG: Web Services for Business Process Design. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm.

[36] XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath`.