Querying business processes with BP-QL

Catriel Beeri\textsuperscript{a}, Anat Eyal\textsuperscript{b,\ast}, Simon Kamenkovich\textsuperscript{b}, Tova Milo\textsuperscript{b}

\textsuperscript{a}School of Engineering and Computer Science, Hebrew University, Israel
\textsuperscript{b}School of Computer Science, Tel Aviv University, Israel

Received 11 January 2007; received in revised form 7 January 2008; accepted 4 February 2008
Recommended by Dr. P. Loucofoulos

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.

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.

\textsuperscript{\textcopyright} 2008 Elsevier B.V. All rights reserved.

Keywords: Business processes; Query language; XML; BPEL; Distributed query processing

1. Introduction

Background: 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 operate 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 [1]. The recent BPEL standard (business process execution language [2], also identified as BPELWS or BPEL4WS), developed jointly by BEA Systems, IBM, and Microsoft, combines and replaces IBM’s webservices flow language (WSFL) [3] and Microsoft’s XLANG [4]. 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 [5].

Motivation: 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.

Challenges: We briefly highlight 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 (e.g. “Which services are called by the travel agency?” vs. Which services are called directly/indirectly?”). An adequate query language must thus allow users to query the processes at different, flexible, granularity levels.

- Distribution: As mentioned above, BPs typically operate in a cross-organization, distributed environment, where each server holds a set of BPs and may provide services to other organizations, or use services that reside on remote servers. 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 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 intuitive, conceptual, data model, coupled with a matching, equally intuitive, query language.

**BPQL:** 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 facilitate 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 [6] and graph-shaped data [7–9], but include two novel features designed to address the issues mentioned above. First, BP-QL supports navigation along two axes: (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 remote servers.

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** [10].

Observe that, in general, several modes of querying BPs 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* [11] and is typically of very high complexity (from NP-hard for very simple specifications to undecidable in the general case [12]). Second, the analysis of runs requires a specification to have a well-defined semantics. Unfortunately, BPEL is not based on a formal model [12]. To avoid these obstacles and guarantee 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 [13].

The query language presented above has been fully implemented and tested in the BP-QL distributed system. All the examples presented in this paper are screenshots that were taken with our BP-QL visual designer and query tool. A first prototype of BP-QL was demonstrated in [14], where only a very high level view of the language was presented.

**Contribution:** To summarize, the contributions of the BP-QL system are the following:

1. **Query language:** The BP-QL query language is a new graphical language that allows for intuitive querying of process specifications, by offering a data model and an interface similar to those

---

1 A variant for monitoring and querying of logs is planned future research.
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) **Model and algorithms**: BP-QL is based on a formal model based on graph grammars \[10\] for systems of processes, and for our query language on such systems. 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) **Implementation**: The BP-QL system implementation is based on Active XML (AXML) \[15\], a special type of XML with embedded calls to Web services. We describe here the system implementation, highlighting the main challenges faced and the solutions taken.

A preliminary version of this work appeared in \[16\] that presented only a brief high level description of the model and algorithms. The present paper provides a comprehensive description of the formal model and the query evaluation algorithm. It also details useful extensions to the basic model and provides formal proofs for all the supporting correctness and complexity results.

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. We present the algorithm for construction of compact results in Section 5, and discuss some extensions to our model in Section 6. The system implementation is described in Section 7. We conclude in Section 8, 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 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 remote servers. The specifications include the interactions between the various processes.

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. \[5\]). 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 \[17\]. 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, some possible 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 acknowledges 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 servers. 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), data nodes (for 2), and
activity nodes (for 3), drawn as rectangles. Edges that connect data and activity nodes, describe which data are read or output by which activity. Edges between activity nodes, 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 (and described) 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 [18]: 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 Fig. 1. It represents the travel agency from our running example. 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 taxonomies and ontologies that provide standard definitions of the service domain. The process flow (on the left) and its data elements (on the right) are displayed in separate boxes. A provided operation. A client may invoke each of the four provided operations namely searchTrip, reserveTrip, confirmTrip and cancelTrip (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. Fig. 2 shows the details of searchTrip. We can see that the travel agency interacts with other services to fulfill client requests. Here, the node attributes (not displayed in the figure) of the requested operations 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.

Variables are used to share data between activities. Variables are defined in the scope of a compound activity, and are visible to all its contained activities. Data values can also be passed to/from requested (provided) activities. For example,

\[\text{Fig. 1. Travel agency.}\]
the results of searching external airline services are stored in possibleFlights, to be used later by returnTripResults activity (for clarity some of these edges are omitted). 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 services provide travel agency services to Europe’, ‘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[6] 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 query 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 retrieve as the query result.

Example 2.1. The query in Fig. 3 searches for BPs that provide travel agency services and serves a certain geographical area, Europe. To compute such
query, we first search the available UDDI registries (according to specific classifications) to get the list of services that satisfies our requirements. Then, we use this information to retrieve the service specifications and further computing of the query. The behavior box is the default box created automatically by the user interface. The user can draw the desired flow pattern by dragging operations to this box, as explained in the next example.

**Example 2.2.** The query in Fig. 4 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 double bounding of the behavior box denotes unbounded zoom-in; we look for compound activities of type provided operation (as indicated by the thick arrow icon), and (transitively) the compound activities that they invoke. The zoom-in is restricted to activities whose specifications reside on the same server, since the deepSearch attribute is set to local. Setting it to global will extend the search to remote services as well.

**Example 2.3.** Fig. 5 illustrates a join operation. The query checks which VISA credit card services are called (directly or indirectly) by the Alpha Tour's confirmTrip activity by comparing the activities' ids. 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 2.4.** The query in Fig. 6(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 exist (dashed edges and nodes denote negation). The existing flow paths leading to searchFlights are then retrieved.
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 \( x \in (\Sigma - \text{"login"}) \). See Fig. 6(b). Regular path expressions as constraints on paths are discussed in Section 6.

**Example 2.5.** Finally, Fig. 7 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 2.6.** Assume that the `searchFlights` service (invoked by `searchTrip` in Fig. 2) has the structure depicted in Fig. 8(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 Fig. 6(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 Fig. 8(b).
Rather than listing all these paths, the user is provided with a compact representation (see Fig. 9) 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 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 BPs and systems

We assume the existence 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.

3.1.1. 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 nodes in $N$; and $\lambda : N \rightarrow \mathcal{L}$ is a labeling function for the nodes. Depending on their label type, we refer to the nodes in $N$ as activity nodes, value nodes, property nodes, etc. Nodes labeled by compound activity names are called compound nodes; all other nodes in $N$ are called concrete.

A (simple) BP is a triple $p = (g, \text{start}, \text{end})$, where: $g$ is a business graph; $\text{start}, \text{end}$ 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 consist 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, Fig. 10 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 its 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 BP graph and is given separately, as we explain next.

3.1.2. Simple systems

A system is a collection of BPs (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 6.

**Definition 3.2.** A system $S$ of BPs (resp. graphs) is a pair $(P, \tau)$, where $P$ is a finite set of BPs (graphs), and $\tau$ is a (possibly partial) function, called the
implementation function, from the compound activity nodes in $P$ to BPs (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.

Fig. 10 shows a partial system. This is a partial description of the Travel Agency system from Figs. 1 and 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.

3.1.3. 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 $S = (P, \tau)$, a BP $p$ in $P$, and a BP $p'$, we say that $p \rightarrow p'$ (w.r.t. $\tau$) if $p'$ is obtained from $p$ by replacing some compound activity node $n$ in $p$ by its implementation $\tau(n)$, where replacement is defined as below.

If $p \rightarrow p_1 \rightarrow \cdots \rightarrow p_k$ we say that $p_k$ is a refinement of $p$. We say that $S \rightarrow S'$ (w.r.t. $\tau$) if $S'$ is obtained from $S$ by replacing the implementation $p$ of some compound activity node $n$ in $S$ by a refinement $p'$ of $p$. If $S \rightarrow S_1 \rightarrow \cdots \rightarrow S_k$ we say that $S_k$ is a refinement of $S$.

Definition 3.4. Given a system $S = (P, \tau)$, a BP $p$ in $P$, and a BP $p'$, replacement of some compound activity node $n$ in a BP $p$ by its implementation $\tau(n)$ is done as follows: $n$ 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.

Replacement of the implementation $p$ of some compound activity node $n$ in $S$ by a refinement $p'$ of $p$ is done as follows: a copy of $p'$ is added to $P$, the mapping $\tau$ for $n$ is updated to point to it, and $\tau$ 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 $P$.

Note that if $S$ is a system, then each of its refinements is also a system. Fig. 11 shows a refinement of the system from Fig. 10, after one refinement step, in which the implementation of
behavior was refined: the node labeled search-Trip 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 6.

3.2.1. 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 3.5. A BP pattern is a tuple \((p^*, T, R)\), where

1. \(p^*\) is a BP where nodes are labeled by elements from \(L \cup \{*, \}\),
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 \((Q, \tau_q)\), where \(Q\) is a set of BP patterns, and \(\tau_q\) is an implementation function.

3.2.2. Query answers

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

Definition 3.6. Let \(q = (Q, \tau_q)\) be a simple query and let \(S\) be a simple system. An embedding of \(q\) into \(S\) is a homomorphism \(\rho\) from the nodes and edges in \(q\) to nodes edges and paths in some refinement \(S' = (P', \tau')\) of \(S\) s.t.
(1) (component) each BP pattern in \( Q \) is mapped to one BP in \( P \).

(2) (nodes) each node is mapped to a node of the same type (property, value, activity, and so on); a node with a constant label is mapped to a node having the same label; each start (resp. end) node in \( q \) is mapped to a start (resp. end) node in \( S^r \); and, each concrete[compound] activity node in \( q \) is mapped to a concrete[compound] activity node in \( S^r \).

(3) (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 \( S^r \).

(4) (implementation) For each compound activity node \( n \) in \( q \), \( \rho \) maps the nodes and (transitive) edges in \( \tau_q(n) \) 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 result defined by \( \rho \) is the image under \( \rho \) 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 answer of \( q \), denoted \( q(S) \), is the set of all query results.

Note that for activity nodes, the condition component is redundant: since all activity nodes of a BP lie between the start and end nodes, the conditions nodes and edges imply the condition component for such nodes. However, as a query BP pattern may contain isolated nodes of other types, this condition is needed.

4. Issues in answer computation and representation

The result associated with an embedding \( \rho \) is, in general, a system of graphs. As there is no limit on the size of refinements, a result may be large, in terms of the sizes of the system and the query. Further, the number of results may be large, or even infinite. The latter may occur when the implementation function is recursive (i.e. cyclic), since then the number of refinements of a system may be infinite. Thus, we face a double challenge: to compute answers efficiently, and to represent them in a compact manner. We proceed to elaborate on the cases where results and answers may be large or infinite, then explain the main ideas of our solution. The answer construction algorithm is presented in Section 5.

4.1. On the size of results and answers

We consider here the main factors that contribute to large or infinite results and answers, and provide some intuition about how they may be addressed. We consider first flat BPs, i.e. BPs with no compound activities, and then nested BPs, that is BPs with compound activities, and a non-trivial implementation function.

Flat BPs: In this case, each query BP pattern may be considered in isolation. Still, even in this case, answers may be large or infinite. 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. If the BP contains cycles, the number of paths that may match a given (transitive) query edge may be infinite.

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. The details of this idea are presented in Section 5.

Nested BPs: Things become more complex in the presence of compound activities and a non-trivial implementation function. If each BP has several compound activities, then since these may be refined independently of each other, the number of refinements may be large. Additionally, a system may contain a recursive implementation function, hence have an infinite set of refinements, and there is no bound on the size of refinements. Since the results of a query are constructed from embeddings into all the refinements, results may be arbitrarily large, and there may be infinite number of results as well.

The solution here is based on viewing systems and queries as context free graph grammars [10], abbreviated CFGG.\(^4\) 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. The answer is obtained from the language of the query by

---

\(^4\)Confusingly, these are also called in the literature regular graph grammars. We will use only context free in this paper.
homomorphisms (as defined in Definition 3.6) into the language of the system. We represent this language also as a CFGG, which can be efficiently constructed from the query and system CFGGs.

Note that this construction is related to “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 [10]. (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 [19], for which the intersection with another CFGG is known to yield a CFGG.

This implies that in principle we could try to adopt the intersection algorithm presented in [19] to construct a finite representation for the query results. The problem, however, with this solution is that the algorithm of [19] is of high complexity—exponential in the size of the BPs5—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 for our specific problem is possible, and to design such an algorithm.

In the rest of this section, and in the next one, we present and explain our algorithm, and the compact representation of answers. That is, we prove the following.

**Theorem 4.1.** Query answers can be efficiently constructed and compactly represented. The construction time and the representation size are both polynomial in the size of the system $S$ (with the exponent determined by the size of the query).

### 4.2. The key ideas

Our algorithm is based on a modular construction of a CFGG that describes the set of query results. It relies on the several ideas, explained below.

#### 4.2.1. Embedding and result decomposition

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 $p^S$. Specifically, each embedding maps each $p^q \in q$ into a process $p^S \in S$, or to a refinement thereof (cf. the condition implementation). Let us call the restriction of an embedding to one query BP a BP embedding—a simple embedding if it maps the query BP to a system BP; a transitive embedding if it maps it to a refinement of a system BP.

An embedding $\rho$ for the query can therefore be factored into a set of BP embeddings, one for each query BP, that together satisfy the condition (implementation).

We denote the set of all simple BP embeddings from $p^q$ to $p^S$ by $[p^q \rightarrow p^S]$, and the set of transitive BP embeddings by $[p^q \rightarrow p^S]$. Note that the BP embeddings in these two sets need to satisfy only the conditions (nodes and edges) of Definition 3.6.

Recall that each result is the image of the query under an embedding. It follows from the discussion above that a result can be decomposed into results for individual query BP’s. Each such result is called a BP result. It is a simple or a transitive result if it is obtained from a simple or a transitive embedding, respectively.

#### 4.2.2. Representation of BP embeddings by node mappings

The second idea is that many embeddings share the same underlying node mapping, and differ only on the assignments to the, possibly transitive, edges (see for example Fig. 12). Thus, the set of simple or transitive BP embeddings, from a query BP $p^q$ to a system BP $p^S$, can be “factorized” into groups of similar embeddings. The intuition is that these embeddings can be grouped based on their assignments to the nodes of $p^q$. 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).

Once an assignment $\rho$ for the nodes is chosen, the assignments for simple edges are determined, and for each transitive edge the set of paths that are the possible assignments is also determined.

---

5Furthermore, to our knowledge, no PTIME algorithms for this intersection problem are known.
As suggested above, this set of possible path assignments for a transitive edge between two nodes is represented by a copy of the sub-graph that connects the nodes. Formally, this intuition is expressed by the following.

Proposition 4.2. For each embedding \( \rho \in [p^q \rightarrow p^r] \), respectively, \( \rho \in [p^q \rightarrow p^s] \), every embedding \( \rho' \) that is obtained from \( \rho \) by replacing the image of some transitive edge \( e \in p^q \) by some other path with the same endpoints, also belongs to \( [p^q \rightarrow p^r] \), respectively, \( [p^q \rightarrow p^s] \).

Proof. The proposition holds since there are no constraints on the images of edges under embeddings, except that they connect the “right” nodes. \( \square \)

As discussed in Section 6, this proposition holds, with appropriate modifications, even when extensions such as regular path expressions are added to the query language.

In Definition 3.6, the condition (edges) applies to embeddings. It can also apply to node mapping, with the following interpretation: A node mapping to embeddings. It can also apply to node mapping, respectively, every embedding that is obtained from \( \rho \) its members.

Corollary 4.3. For each node mapping \( \rho \) from \( p^q \) to \( p^s \), respectively, to a refinement of \( p^q \), that satisfies (nodes) and (edges), each of its extensions obtained by mapping each simple, respectively, transitive, edge from \( m \) to \( n \) to the edge, respectively, a path, from \( \rho(m) \) to \( \rho(n) \), is a member of \( [p^q \rightarrow p^s] \), respectively, \( [p^q \rightarrow p^s] \). Conversely, each member of \( [p^q \rightarrow p^s] \), respectively, \( [p^q \rightarrow p^s] \), is obtained in this manner from such a node mapping.

Proof. The corollary follows directly from Definition 3.6 and Proposition 4.2. \( \square \)

We say that a node mapping \( \rho \) from \( p^q \) to \( p^s \), respectively, a refinement of \( p^q \), that satisfies nodes and edges represents the set of BP embeddings obtained from it as described in the corollary. By the corollary, each such node mapping represents a maximal subset of embeddings in \( [p^q \rightarrow p^s] \), respectively, \( [p^q \rightarrow p^s] \), that agree on nodes. The sets represented by different mappings are disjoint. A set of such node mappings represents the union of the sets represented by its members.

Let us denote the set of node mappings from \( p^q \) to \( p^s \), respectively, refinements of \( p^q \), that satisfy (nodes) and (edges) by \( N[\rho \rightarrow p^s] \), respectively, \( N[[\rho \rightarrow p^s] \).

Corollary 4.4. The sets of node mappings \( N[\rho \rightarrow p^s] \) and \( N[[\rho \rightarrow p^s] \) represent the sets \( [p^q \rightarrow p^s] \) and \( [[p^q \rightarrow p^s] \), respectively.

Proof. Assume that for each pair \((\rho^q, \rho^s)\) of a query BP and system BP, a node mapping from \( [p^q \rightarrow p^s] \) or from \( [[p^q \rightarrow p^s] \) (as dictated by the query) is given. Select each such node mapping, respectively, embeddings it represents. If this collection of embeddings satisfies the condition (implementation), then so does every other collection of embeddings obtained from this set of node mappings. This follows from the fact that the implementation function \( \tau \) is specified for nodes. That is, it suffices to check for the validity of this condition for combinations of node mappings. \( \square \)

Results representation: Further, since multiple BP embeddings that share the same assignment to nodes have a joint representation, so can the BP results obtained from them. Intuitively, if \( \rho \) is a node mapping, then the corresponding set of results consists of the images of the nodes under \( \rho \), and all edges or paths that connect them. We denote the set of results obtained from \((\rho^q, \rho^s)\), respectively, \( [p^q \rightarrow p^s] \), by \( R[\rho^q \rightarrow p^s] \), respectively, \( R[[p^q \rightarrow p^s] \). Then the observation is essentially that these two sets can be obtained from \( N[\rho^q \rightarrow p^s] \), respectively, \( N[[\rho^q \rightarrow p^s] \).

4.2.3. Embeddings into refinements

The third and last idea concerns the computation and representation of embeddings of a query BP pattern \( p^q \) into refinements of a system BP \( p^s \). Since \( p^s \) may have a large or even an infinite number of refinements, and refinements may be arbitrarily large, it is necessary to also decompose this task as well.

The important observation here 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 nodes and edges of \( p^q \) into \( p^s \) itself. Subsequent parts map additional nodes and edges of \( p^q \) into (refinements of) 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 appropriate structure. Specifically, the nodes and edges mapped into an implementation of a compound activity
must form a sub-graph of \( p^q \), with a single entry and exit nodes.

Now, the same observation applies to the nodes and edges in these sub-graphs. If a sub-graph \( G \) is mapped to a refinement of an implementation \( p^S \) of a compound node of \( p^S \), then parts of \( G \) are mapped directly to \( p^S \), but certain sub-graphs are mapped into refinements of implementations of \( p^S \).

While for a specific mapping, this decomposition into sub-graphs may be arbitrarily deep, the involved sub-graphs are essentially all sub-graphs of \( p^q \). Thus, the set of mappings can be obtained by combining mappings of sub-graphs of \( p^q \) into system processes of \( S \). As explained in Section 5, this combination can be expressed as a \( \text{CFG} \).

Now, the graphs for which representations are constructed are (with a few exceptions) sub-graphs of the query patterns \( p^q \). The construction terminates when mappings from each such graph to each system process have been considered, and combined into a \( \text{CFG} \). While the time required for this construction, and the space required for its outcome are exponential in the query size, they are polynomial in the system size.

5. The answer construction algorithm

This section describes in detail how answers are represented, and the answer construction algorithm, following the ideas presented in the previous section. Following the observation in Section 4.2.1, the construction of answer representations can be structured as follows: First, for each \( p^q \) and \( p^S \), construct the representations \( N[p^q \rightarrow p^S] \) and \( N[p^S \rightarrow p^S] \) for the two sets of BP embeddings \([p^q \rightarrow p^S]\) and \([p^S \rightarrow p^S]\). Then, construct images from these sets to represent \( \mathcal{R}[p^q \rightarrow p^S] \), respectively, \( \mathcal{R}[p^S \rightarrow p^S] \). Finally, select combinations of elements from the sets that satisfy implementation.

Hence, we concentrate on embeddings from one query BP pattern \( p^q \) to one system BP \( p^S \). The algorithm for this construction is presented in several stages. The two main stages are the construction of \( \mathcal{R}[p^q \rightarrow p^S] \) and \( \mathcal{R}[p^S \rightarrow p^S] \). In each of these, we deal first with the easy case that there are no transitive edges, then consider the more difficult case where such edges are present.

5.1. Preliminaries

5.1.1. Context free graph grammars

As explained in Section 3, a system \( S \) can be viewed as a regular graph grammar [10]. We briefly discuss here these grammars.

An \( m \)-order context-free graph grammar (\( \text{CFG} \)) is a quadruple \((N, T, P, I)\), where the components are as follows: \( N \) is a set of non-terminal structures that are polygons of size at most \( m \). That is they can be single nodes, pairs of nodes connected by an edge, triangles, and so on up to polygons with \( m \) vertices. \( T \) is a set of terminal elements. \( P \) is a finite set of rewriting rules of the form \( G \rightarrow H \), where \( G \) is a non-terminal structure and \( H \) a graph containing possibly both terminals and non-terminals. \( I \) is a set of initial graphs.

Rewriting of a graph using this \( \text{CFG} \) corresponds to replacing a sub-graph that is isomorphic to some \( G \) that occurs in the left side of some rule by the graph \( H \) on the right side of the same rule. In this replacement, \( H \) needs to be connected to the rest of the graph in a manner that simulates the connections of \( G \). We skip the details, since we are interested only in a sub-class of this class of grammars. The language of the grammar is the set of graphs that can be obtained from \( I \) by repeated rewritings.

For our needs, 1-order grammars (denoted here by \( \text{CFG} \)) are sufficient. In these, in each rule \( G \rightarrow H \), \( G \) is a single node (a compound activity node), and \( H \) is a graph with a single entry node, and a single exist node. When such a rule is used for a rewriting, each edge into \( G \) is pointed into \( \text{start}(H) \), and each edge from \( G \) becomes an edge from \( \text{end}(H) \).

It is easy to see that a system can be viewed as a \( \text{CFG} \). 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 a \( \text{CFG} \). The query answer consists of the graphs that are images of embeddings from the query BPs to system BPs.

The answer to the query will also be represented as a graph grammar. For conformance of notation with systems and queries, we will represent the set of productions as a multi-valued function \( \varepsilon_A \) on node labels. For a label in its domain, this function associates a set of BP graphs. Thus, each node with this label can be refined (that is, replaced). In the discussion below, we do not limit BP’s to have one

---

6This is not really true in all cases, as explained in Section 5, but the intuition still holds for the construction explained there.
root. This is just for convenience; to obtain a BP with a single root one can add a new root graph consisting of a single node, mapped by \( \tau_1 \) to these multiple previous root graphs.

5.1.2. A requirement from results representations

A basic requirement from a compact representation of a set of results is that one should be able to effectively obtain from it each individual result. A query is a pattern, that specifies a set of nodes and of related paths. Being able to obtain each path of each result is, by itself, insufficient. One needs to be able to identify each set that forms one result, that is a complete image of the query pattern.

We satisfy this requirement as follows. We assume that the nodes of a query are uniquely identifiable by integers, say from 1 to \( n \). When the result representation is constructed, certain nodes in it are associated with indices, which are positive integers. In all parts of the construction, nodes that are images of query nodes are associated with the indices of these nodes. In Section 5.3, additional nodes are associated with indices strictly larger than \( n \).

When results are displayed, only the indices in the range \( [1 \ldots n] \) are displayed. As just explained, the nodes so indexed are the images of query nodes. Thus, a user can clearly see the nodes that are images of query nodes, and the paths that connect them, which are images of query edges. In the construction, when a node with label \( a \) is indexed with integer \( i \), its label becomes \( a_i \).

A BP graph with certain nodes marked as being derived from the query nodes is a representation of BP results from \( p^q \) to \( p^s \), respectively to a refinement of \( p^s \), if every BP graph obtained from it by selecting the marked nodes, adding edges between them for simple edges, and paths for transitive edges, is a simple (transitive) BP result from \( p^q \) to \( p^s \), respectively to a refinement of \( p^s \). A set \( R \) of BP graphs represents \( \mathcal{R}[p^q \rightarrow p^s] \), respectively, \( \mathcal{R}[p^q \rightarrow p^s_1] \), if the union of the sets of results its members represent is equal to the corresponding set.

5.2. The construction of \( N[p^q \rightarrow p^s] \) and \( \mathcal{R}[p^q \rightarrow p^s] \)

In this construction, only simple embeddings from \( p^q \) to \( p^s \) are considered. We first present the case where no transitive edges are present in the query, then we consider the case where they are present.

5.2.1. Without transitive edges

Each node mapping that represents BP embeddings from \( p^q \) to \( p^s \) is essentially a homomorphism from \( p^q \) to \( p^s \)—this is the meaning of the condition (edges). The condition (nodes) filters out some homomorphisms. Finding a homomorphism from a graph \( G_1 \) to a graph \( G_2 \) is NP-complete in the size of \( G_1 \). Heuristics for constructing node mappings that satisfy (nodes) and (edges), that work well for simple graphs, are well known. We therefore assume that a representation \( N[p^q \rightarrow p^s_1] \) of \( [p^q \rightarrow p^s] \) by node mappings has been constructed.

For this simple case, given a representation of BP embeddings as a node mapping \( \rho \), the edges between nodes are uniquely determined. The only issue that needs to be addressed is that several nodes of \( p^q \) may be mapped to the same node \( n \) of \( p^s \). In a result, there should be several copies of \( n \), one per query node mapped to it.

Thus, for each node mapping \( \rho \), the representation of the corresponding set of results is an isomorphic copy of the query pattern, with each * label on a node \( n \) replaced by the label on \( \rho(n) \), and with each node associated with the id of the corresponding query node.

Example 5.1. Fig. 13(a) illustrates a BP \( p^s \) comprised of nodes \( A, B, C \) with a loop on \( B \) and a BP pattern \( p^q \). Then the mapping \( \rho \) maps both \( B \)-nodes of \( p^q \) to the same \( B \)-node of \( p^s \). The answer contains a single result. The edge from \( B_2 \) to \( B_3 \) in the result is a copy of the loop edge from \( B \) to \( B \) in \( p^s \).

Example 5.2. Let \( p^s \) be as in the preceding example, and let \( p^q \) be as described in Fig. 13(b). Then there are two mappings, namely one that maps the node labeled * to the \( C \) labeled node, and another that maps it to the \( B \)-labeled. Thus, the answer now contains two results, as illustrated in the figure.

5.2.2. With transitive edges

Two changes in the algorithm are needed here. First, by the condition (edges), if \( e \) is a transitive

![Fig. 13. (a) Node mapping. (b) * Label.](image-url)
edge that connects node $m$ to node $n$, then there must exist a path from $\rho(m)$ to $\rho(n)$ in $p$. Hence, in the construction of node mappings, whenever $p^\pi$ contains a transitive edge from $m$ to $n$, when images for nodes $m$ and $n$ are considered, it must be checked that there is a path that connects them.

Second, since $p^S$ may contain forks and joins or cycles, a transitive edge may be mapped to a large or an infinite number of paths. To address this issue, when a representation of results is constructed from a mapping $\rho$, if $p^\pi$ contains a transitive edge from $m$ to $n$, then all nodes and edges on all paths from $\rho(m)$ to $\rho(n)$ should be added to the representation. We add new copies of these nodes and edges to the representation of the result. They are not associated with indices, so they are not marked as images of query nodes. Hence, results can still be extracted unambiguously.

**Example 5.3.** Consider the example in Fig. 14. There is only one node mapping from $p^\pi$ to $p^S$, that maps the nodes $A_1$, $B_2$, and $D_3$ of $p^\pi$ to the nodes $A$, $B$, and $D$ of $p^S$. The transitive edge from $B_2$ to $D_3$ in $p^\pi$ can be mapped to any path from $B$ to $D$ in $p^S$. Such a path uses the edges from $B$ to $C$ and from $C$ to $D$, and may also include the edge from $C$ to $B$. The representation of the answer contains a copy of the cycle in $p^S$, as illustrated in the figure.

**Proposition 5.4.** The set $N[p^\pi \rightarrow p^S]$ of node mappings constructed as outlined above, is a representation of $[p^\pi \rightarrow p^S]$, the simple BP mappings from $p^\pi$ to $p^S$.

**Proof.** The proposition follows directly from the construction, and from Corollary 4.4. □

**Corollary 5.5.** The set of BP graphs, constructed as described above from the node mappings in $N[p^\pi \rightarrow p^S]$, is a representation of $\mathcal{R}[p^\pi \rightarrow p^S]$.

**Proof.** The proposition follows directly from the construction, and from Definition 3.6 and Corollary 5.5. □

5.3. Construction of $N[p^\pi \rightarrow p^S]$ and $\mathcal{R}[p^\pi \rightarrow p^S]$

So far, only simple embeddings from $p^\pi$ to $p^S$ were considered. Next, we consider transitive embedding, that is embeddings into refinements of system processes.

In such an embedding, a new kind of cycle may exist: a part of $p^\pi$ is mapped into $p^S$, but some sub-graph of $p^\pi$ may be mapped to an implementation $p^S_1$ of a composite activity node of $p^S$; the same or a smaller sub-graph may be mapped to an implementation $p^S_2$ of a composite activity node in $p^S_1$, and so on, until a process is visited a second time, and possibly more times. Our approach here is to include each process on such a cycle only once, thus representing the possibly infinite set of paths compactly.

As in the case of simple embeddings, we first consider the case without transitive edges, and then generalize to include such edges.

5.3.1. No transitive edges

We first introduce some terminology for describing the parts of embeddings that are into (direct or indirect) implementations of compound activity nodes of $p^S$. Then we describe the construction.

In the construction below, new nodes with label $*$ are generated. To be able to distinguish between the images of these nodes, and between them and images of the original query nodes, these nodes are indexed. We assume a counter is maintained, whose initial value is $n+1$, where $n$ is the number of nodes in the query. As a $*$-labeled node is generated, it is assigned as an index the value $i$ of the counter, which is then incremented. In the discussion below, we refer to these nodes as $*_i$ nodes.

**Single-entry-single-exit sub-graphs:** Note that a sub-graph $g$ of a BP pattern $u$ may be mapped to a compound activity node of a BP $p$ only if it has a single entry and a single exit. That is, all edges incoming into $g$ arrive at a single node of $g$, called its entry; and all edges outgoing from $g$ depart from a single node, its exit. In the mapping, the entry and exit of $g$ must be mapped to $\text{start}(p)$ and $\text{end}(p)$, respectively. We refer to such a sub-graph as a SESE sub-graph of $u$ (SESE for Single-Entry-Single-Exit).

We denote by $\text{SESE}(p)$ the set of single-entry-single-exit sub-graphs of a pattern $p$. Note that $p \in \text{SESE}(p)$ is possible. Also note that for $g \in \text{SESE}(p)$, it holds that $\text{SESE}(g) \subseteq \text{SESE}(p)$. For a $\text{SESE}$ sub-graph $g$, we denote by $\hat{g}$ the
pattern obtained from \( g \) by designating \( g \)'s entry, respectively, exit, as start\((g)\), respectively end\((g)\). These designations ensure that, in an embedding of \( g \) into a BP \( p \), these nodes of \( g \) can only be mapped to start\((p)\) and end\((p)\), respectively.

In a given transitive embedding of a BP pattern \( p \), several disjoint SESE sub-graphs may be mapped to implementations. A subset \( J \) of \( \text{SESE}(p) \) that consists of disjoint sub-graphs is called a \( d \)-subset of \( \text{SESE}(p) \). The set of all \( d \)-subsets of \( \text{SESE}(p) \) will be denoted by \( d\text{-subs}(p) \). For a \( d \)-subset \( J = \{g_1, \ldots, g_k\} \) of \( \text{SESE}(p) \), denote by \( p/J \) the pattern obtained from \( p \) by replacing the \( i \)th element \((1 \leq i \leq k)\) of \( J \) in \( p \) by a new node labeled \(*_i\), and indexed by the current value of the counter. Since the actual value of the index is irrelevant to us, for clarity we denote this node as \(*_i\).

The construction: Given a transitive embedding \( \rho \) of \( p^g \) into \( p^S \), let \( J = \{g_1, \ldots, g_k\} \) be the subset of \( \text{SESE}(p^g) \) of sub-graphs mapped by \( \rho \) into implementations of compound activity nodes of \( p^S \). Then, \( \rho \) can be viewed as a combination of a simple embedding \( p/j \) from \( p^g/J \) to \( p^S \), that maps each \(*_i\) to a compound activity node \( f^j \) of \( p^S \), and additionally, transitive embeddings from each \( g_i \) to \( \tau(f^j) \). These embeddings can be similarly decomposed.

For \( g \in \text{SESE}(p^g) \), a (possibly empty) \( d \)-subset \( J \) of \( g \), and a BP \( p \) from \( S \), denote by \( g/J \) the set of simple embeddings from \( g \) to \( p \) the set of simple embeddings from \( g \) to \( p \), such that each \(*_i\) is mapped to a compound activity node of \( p \). Denote by \( \text{SESE}(g/J \rightarrow p) \) the set of images of these embeddings, in which each image of an \(*_i\) node is indexed by \( i \). That is, if the \(*_i\) node is mapped to a compound activity node \( f^j \), then in the representation of the result, the label of the node will be \( f^j \).

These sets can be constructed as follows: first, construct the set \( N(g/J \rightarrow p) \) of node mappings. To obtain \( (g/J \rightarrow p) \), filter out the mappings that do not satisfy the additional constraint that the images of \(*_i\) nodes are compound activity nodes. From this, \( \text{SESE}(g/J \rightarrow p) \) is obtained as described above, in Section 5.2, with the additional twist of augmenting the labels of images of \(*_i\) nodes.

For \( p^g \) itself, we use the same notation, \( (p^g/J \rightarrow p^S) \), to denote the subset of \( \text{SESE}(p^g/J \rightarrow p^S) \) of embeddings that satisfy the same constraint on \(*_i\) nodes, and \( \text{SESE}(p^g/J \rightarrow p^S) \) for the derived results. However, we do not augment \( p^g \) with designated start and end node.

Recall that the CFGG we want to construct to represent the answer, namely \( \text{SESE}(p^g \rightarrow p^S) \), should consist of a set of graphs, and a set of productions that allow to replace certain nodes in these graphs by graphs. We take the set of graphs of the grammar to be \( K_1 \cup K_2 \), where these two sets are defined as follows: \( K_1 = \{R(p^g/J \rightarrow p^S) | J \in \text{d-sub}(p^S) \} \) and \( K_2 = \{R(g/J \rightarrow p) | g \in \text{SESE}(p^g), J \in \text{d-sub}(g), p \in S \} \). We proceed to describe the productions. Let \( J = \{g_1, \ldots, g_k\} \) be in \( \text{d-sub}(p^g) \), and assume that the result graph \( R(p^g/J \rightarrow p^S) \) was constructed from a mapping in which a node \(*_i\) of \( p^g/J \) was mapped to a compound activity node \( f^j \) of \( p \). Recall that in this result graph, the image of this \(*_i\) node is labeled by \( f^j \). Then, we set \( \tau_A(f^j) \) to be the set \( \{R(g_i/J_i \rightarrow \tau(f^j)) | J_i \in \text{d-sub}(g_i)\} \). A similar construction produces productions for nodes in \( R(g/J \rightarrow p) \), for all \( g \in \text{SESE}(p^g), J \in \text{d-sub}(g) \), and \( p \in S \).

The set \( \text{SESE}(p^g \rightarrow p^S) \) is then the language defined by the grammar.\(^7\) Note that the symbols of the form \( f^j \), that correspond to images of new \(*_i\)-labeled nodes, are non-terminals. The language consists of graphs generated by the grammar that do not contain such non-terminals.

Example 5.6. Fig. 15 illustrates a system with two BPs, \( p^S_1 \) and \( p^S_2 \), in which \( \tau(A) = p^S_2, \tau(B) = p^S_1 \), and a BP pattern \( p^g \). Let use construct \( \text{SESE}(p^g \rightarrow p^S) \).

We start by computing the set \( K_1 = \{R(p^g/J \rightarrow p^S) | J \in \text{d-sub}(p^g)\} \). For that, we need to consider the \( d \)-subsets of \( p^g \). The SESE sub-graphs of \( p^g \) are \( g_1 = \{*_1, g_2 = \{A_2\}, g_3 = \{*_3\}, g_4 = \{*_1, A_2\}, g_5 = \{A_2, *_3\}, g_6 = \{*_1, A_2, *_3\} = p^g \). The \( d \)-subsets are then \( J_0 = \emptyset \), the singleton sets \( J_1 = \{g_i\}, i = 1, \ldots, 6 \), and additionally \( J_7 = \{g_1, g_2\}, J_8 = \{g_1, g_3\}, J_9 = \{g_2, g_3\}, J_{10} = \{g_1, g_2, g_3\}, J_{11} = \{g_1, g_3, g_6\} \). We illustrate the construction for some of these.

For \( J_0 = \emptyset \), two simple solutions are illustrated in Fig. 15.

For the other \( d \)-subsets, the major effort is to construct mappings for \( g_i \), \( 1 \leq i \leq 6 \). These can then be combined to produce results for all other non-empty \( d \)-subsets.

For \( g_1, g_2, \) and \( g_3 \), no mappings into \( p^S_1 \) exist, since in these graphs, the start and end nodes are the same, but not so in \( p^S_2 \). From the remaining graphs, let us consider \( g_6 \) as an example. For this graph, we can map \(*_1 \) and \(*_3 \) to start\( (p^g_1) \) and end\( (p^g_2) \), respectively. A direct mapping of the node \( A_2 \) to the node \( B \) of \( p^S_2 \) is impossible, since the labels are

---
\(^7\)Clearly, unreachable members can be removed from this grammar, by the usual method.
different. Thus, in our search for an embedding, we must factor it as follows: we replace $A_2$ by $*_4$, so we can map $*_4$ to $B$, and we then search for mappings of $\hat{g}_7$ to $p_1^n$, where $g_7 = \{A_2\}$. But, it is clear that such mappings do not exist, since $g_7$ has a single node that serves as its entry and exit, whereas $\text{start}(p_1^n)$ and $\text{end}(p_1^n)$ are distinct nodes.

We conclude that no eligible mappings for $\hat{g}_6$ exist. Similarly, it is easy to see that no mappings exists for $\hat{g}_4$ and $\hat{g}_5$. Thus, only the mappings for $J_0$ produces a result, so Fig. 15 actually shows all the results.

5.3.2. With transitive edges

The new issue to be dealt with here concerns the possibility that a transitive edge is mapped to a path that is both in a process $p$, and in some implementation of a compound activity node of $p$.

Assume the pattern $g$ contains a transitive edge from $m$ to $n$. In constructing transitive embeddings from $g$ to $p$, the following options should be considered. In the discussion below, we use $p'$ as a generic name for some implementation of a compound activity node of $p$.

- The path corresponding to the transitive edge from $m$ to $n$ is fully in $p'$. In this case, there is a SESE sub-graph $g'$ that contains the edge. This includes the case that $m$ is the entry of $g'$, or $n$ is its exit, or both. In this case, the edge will be dealt with when embeddings from $g'$ to $p'$ are considered.

- The transitive edge from $m$ to $n$ is mapped to a path that starts in some $p'$, and ends in $p$. In particular, the image of $m$ is in $p'$ and that of $n$ is in $p$. To construct a suitable transitive sub-graph $g'$, we need to add a new node to serve as the entry of $g'$. We decompose the given transitive edge as follows: We add a new node with a $*$ label, and replace the given edge with two transitive edges, one from $m$ to $*$ and the other from $*$ to $n$. Any graph whose entry is this new node, and also contains $n$ is now also considered as a SESE sub-graph.

- The dual case that the graph contains $n$ but not $m$ is treated similarly.

- For the case that the path starts in $p$, contains $m$, proceeds to some $p'$ and returns to $p$, where it contains $n$ we add two new $*$ nodes, to serve as the entry and exit of a transitive sub-graph $g'$.

With the above modifications in place, d-subsets of transitive sub-graphs can be constructed, and the algorithm outlined in Section 5.2.1 above, modified to deal with transitive edges, as in Section 5.2.2 can be used. One additional point to be considered is that the newly introduced $*$ nodes just described were not present in the original query. Their images, therefore, are not marked as query node images.

**Example 5.7.** Consider the system and query in Fig. 16. $S$ consists of three BP’s, namely $p_1^5$, $p_2^5$, $p_3^5$. The query $q$ consists of $p_1^q$, $p_2^q$. Then $A_1$ in $p_1^q$ is mapped to $A$ in $p_3^5$. We need now to find results derived from embeddings of $p_2^3$ into refinements of $p_3^5$.

We first note that $\text{start}(p_2^3)$ must be mapped to $\text{start}(p_3^5)$. Clearly, a direct mapping of $D_2$ to $B$ is impossible, so we look for a transitive mapping. A mapping of $D_2$ by itself to $p_3^5$ will map part of the transitive edge from $D_2$ to $C_3$ into $p_3^5$, but the tail of the path must come back to $p_3^5$, so that $C_3$ can be mapped to $C$. Hence, we decompose the double headed edge from $D_2$ to $C_3$ by introducing a new $*_4$ node into it. Denote the result by $p_2^*_{5,1}$ (see Fig. 17 on the left). The sub-graph $g$ that consists of $D_2$, $*_4$ and the transitive node between them is a SESE sub-graph, so we can consider mapping it to $p_3^5$. (Note that $p_3^5$ has no refinements.) For that, we replace $g$ in $p_2^*_{5,1}$ by $*_5$ to obtain $p_2^*_{5,2}$ (see Fig. 17 on the right). Now, the nodes $*_5$ and $C$ of $p_2^*_{5,2}$ can be mapped to $B$ and $C$ in $p_3^5$. The transitive edge from $*_5$ to $C_3$ is thus mapped to the edge in $p_3^5$, giving $B_5 \rightarrow C_3$, where $B_5$ is a non-terminal of the grammar.
It remains to find an embedding from $\hat{g}$ to $p^S_1$. This is easy: $D_2$, the start node of $\hat{g}$ is mapped to $D$, and $*_4$, the end node of $\hat{g}$ is mapped to $E$. This gives the graph $D_2 \to E$. Now we set $\tau(B_3)$ to this graph as depicted in the answer in Fig. 18. Note that the image of $*_4$ is not marked as a query node image, nor as a non-terminal. This is the desired result, derived from an embedding of $p^S_2$ into a refinement of $p^S_1$.

Let us denote the grammar constructed as above by $\mathcal{G}[p^q / \to p^S]$.

**Proposition 5.8.** The language of $\mathcal{G}[p^q / \to p^S]$ is a representation of the set of results $\mathcal{R}[p^q \to p^S]$. Specifically, each graph in the language represents a set of members of $\mathcal{R}[p^q \to p^S]$ that are derived from a shared node mapping. Conversely, each set of results derived from a shared node mapping is represented by a graph in the language of $\mathcal{G}[p^q / \to p^S]$.

**Proof.** $\Rightarrow$: This direction is proved using induction on derivation depth. A derivation of a graph in the language starts from $\mathcal{R}(p^q/J \to p^S)$, for some $J$ in $d\text{-}subs(p^q)$. This gives, by Proposition 5.4, a result derived from a node mapping of $p^q/J$ to $p^S$, that maps every new * node to a compound activity node $f$. The graph is obtained by replacing each such $f$ by the language of the same grammar, with roots $\{\mathcal{R}(g/K \to \tau(f))\}$, where $g$ ranges over $J$, and $K$ is a d-subset of $\hat{g}$. By induction on derivation depth, these represent results obtained from transitive embeddings of $\hat{g}$ into $\tau(f)$. The claim follows.

$\Leftarrow$: This direction works by induction on the refinement depth. Let us first define this concept.

Assume $p'$ is a refinement of $p$. The refinement depth of $p'$ with respect to $p$ is defined as follows. The refinement depth of $p$ with respect to itself is zero. If $p \neq p'$, then $p'$ is obtained by replacing some compound nodes of $p$ by implementations. If the maximum refinement depth of these implementations with respect to their original processes is $m$, then the refinement depth of $p'$ with respect to $p$ is $m + 1$.

Assume given a set of results derived from a (maximal) set of transitive embeddings from $p^q$ to $p^S$, where $p^S$ is a refinement of $p^q$ of depth $m$, that share a node mapping $\rho$. The part of $\rho$ in $p^S$ defines a d-subset $J$. The derivation of the grammar starts from $\mathcal{R}(p^q/J \to p^S)$. If $J = \emptyset$, then $p^S = p^q$ and we are done. Otherwise, each remaining part, say $\rho_i$ of the node mapping, that corresponds to a $g_i$ member of $J$, is to a refinement $p^i_S$ of $p^S$, where $p^i_S$ is the implementation of a compound activity node of $p^S$. The refinement depth of $p^i_S$ with respect to $p^q$ is smaller than $m$. Hence, the set of results derived
from \( \rho \), can be represented by the grammar, with the roots replaced by \( \{(g_i/p^S \rightarrow K) \mid K \in d\text{-}subs(G_i)\} \). From this, the claim follows. 

5.3.3. Putting it all together

To summarize, we now have representations for results obtained from simple and transitive embeddings, from each \( p^q \) to each \( p^S \). Note that the representations for simple embeddings may be viewed as degenerate grammars.

Now, consider the combinations of representations, one for each combination of \( p^q \in Q \) and \( p^S \in S \). We discard those that do not satisfy (implementation). In particular, if a representation is obtained from a node mapping \( \rho \) from pattern to some \( p^S \) that maps a transitive query node \( n \) to a compound activity node in \( p^S \), then \( \tau_q(n) \) must be mapped transitively to \( \tau(\rho(n)) \).

**Theorem 5.9.*** The construction outlines above generates a finite representation of the set of results for \( Q \) on \( S \).

**Proof.*** It is easy to see that the construction above generates a finite representation. The correctness follows directly from Proposition 5.8. 

A more efficient version of the algorithm works incrementally. We first construct representations for embeddings from members of \( Q \) to members of \( S \). Then, we construct representations for new pairs on demand, to satisfy implementation. Details are left to the reader.

5.4. Complexity

We are now in position to consider Theorem 4.1. Theorem 5.9 above is the correctness part. It remains to discuss the data complexity part.

Indeed, in the construction above, in the worst case two representations (one regular, one transitive) are constructed for each combination of a transitive sub-graph of a query process \( p^q \) and a system process \( p^S \). The number of transitive sub-graphs of the query processes may be exponential in the query size. Each construction may take time that is exponential in the transitive sub-graph size.

6. A richer model

The query model discussed in Sections 3 and 5 was rather simple. This simplicity allowed us to concentrate in Section 5 on the main ideas of how answers of queries are represented. We now present some useful extensions that enhance the expressive power of the language, and facilitate the querying of real life BPs. Some of these were illustrated in Section 2.

**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:

**Definition 6.1.*** Let \( q = (Q, \tau) \) be a query with negation (i.e. a query whose graph patterns have some nodes and edges that are distinguished as negative), and let \( S \) be a simple system. We denote as \( \text{positive}(q) \), the positive part of \( q \), obtained from \( q \) by deleting all the negative edges and nodes, and all the edges incident on these nodes.

An embedding of \( q \) into \( S \) is a homomorphism \( \rho \) from the nodes and edges in \( \text{positive}(q) \) to nodes and paths in some refinement \( S' = (P', \tau') \) of \( S \) s.t.

1. The conditions nodes, edges and implementation of Definition 3.6 are satisfied.
2. \( \rho \) cannot be extended to an embedding of any query \( q' \) obtained from \( \text{positive}(q) \) by adding all the negative nodes and edges of some of its BP patterns.

A finite representation for the query answer can be constructed essentially as explained in Section 5. Note that both embeddings for the positive part and for the full query can be grouped by shared node mappings. In the construction, each node mapping for the positive part that can be extended to a node mapping for the full query is eliminated.

**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 searchFlights 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.
This can easily be accommodated in our answer construction algorithm.

Another useful feature is regular path expressions on node labels. 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.

**Example 6.2.** Let us reconsider Example 5.3: Fig. 19(a) illustrates the modified example. The edge from $B_2$ to $D_3$ in $p^q$ is labeled with the regular expression $(CB)^2C$. Without this expression, the answer was as illustrated in Fig. 14, and includes a cycle. With it, it is converted to the answer in Fig. 19.

The construction of a finite representation for the query answer extends naturally to support this extension. First, replace the transitive edge by an NFA that represents the regular expression with the edge labels pushed forward to become node labels (in this pushing, epsilon labels are simply absorbed). Then, apply the answer construction algorithm. Note that the nodes of the newly introduced NFA are not considered as query nodes.

**Example 6.3.** Consider the same system BP and query BP as in Example 6.2, except that the regular expression is $(CB)^*C$ (see Fig. 19(b)). After converting the regular expression to an NFA, pushing labels to the nodes, and incorporating the result into the query, one obtains the illustrated result (we use superscripts for new nodes that are not in the original query). The answer is essentially identical to this query.

As for the impact of this construction on the complexity, note that the conversion of a regular expression to an NFA can be done in polynomial time and space. Since the expression is part of the query, it means that the query may grow polynomially. The data complexity is unchanged.

**Example 6.4.** Reconsider a variation on Example 5.7 (see Fig. 20). First, ignore the regular expression, then $A_1$ is mapped to $A$ in $p^S$. The result derived from an embedding of $p^S$ into a refinement of $p^S$ is illustrated in Fig. 21(a). Now, assume the transitive edge from $D_2$ to $C_3$ is labeled with $(E_E)^3E$, that is, three occurrences of $E$ are requested, separated by occurrences of an arbitrary character. Then $p^S$ is replaced as illustrated in Fig. 21(b). Here, the unnumbered new nodes need to be mapped, but their images will be just the image of the original transitive edge.

**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 (unequal)label variables are mapped to system nodes with (distinct)identical labels; query edges labeled by (unequal)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.

![Fig. 19. Regular path expressions (a), (b).](image-url)
Theorem 6.5. 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 even if the system BPs also have no cycles.

Finally, for flat BPs, the problem can be solved in time polynomial in the size of the system.

Proof. 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 shape as in Fig. 22. The implementation of the compound activity $R$ in the figure has two branches. The first contains a compound activity node $R_1$ and the second a compound activity $R_2$. The implementation of $R_i$, $i = 1, 2$ (which resembles in spirit the grammar rules of $G_i$ and is detailed below) is defined such that each of the paths from its start to its end nodes has line-shaped structure, representing a word in the context free language of $G_i$. The implementations are defined such that $R_1$ and $R_2$ 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 the $q$ showed in Fig. 22, with two transitive edges $e_1$, $e_2$ that match (the refinements of) $R_1$ and $R_2$ resp., and have an equality condition on their attached path variables. The query thus 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 [20].

To complete the undecidability and hardness proofs we need to explain how the implementation of $R_i$, $i = 1, 2$ is defined. We first consider the case where $G_i$’s are general context free grammar, and explain later the case of non-recursive ones. For a context free grammar $G_i$, we model each of its non-terminals as compound activity. W.l.o.g. assume that $R_i$ is the root non-terminal of $G_i$. The implementation of each compound activity is defined according to the derivation rules of the corresponding non-terminal. It contains a start and an end nodes, both labeled by some label (letter) $l$ that does not appear in the grammar. The start and end nodes are connected by line-shaped paths, each representing one possible derivation rule of the non-terminal. (If the non-terminal derives the empty word, the word the end
and start nodes are connected by a simple edge.) Additionally we add, between every pair of connected nodes, a “loop” with a node labeled by the new symbol \( l \). To see a simple example, Fig. 23 shows the implementation of the compound activity for the a terminal \( A \) defined by the following derivation rules: \( A \rightarrow cDA, A \rightarrow cA, A \rightarrow \varepsilon \) (where \( A, D \) are non-terminals, \( c \) a terminal and \( \varepsilon \) the empty string). The shaded part corresponds to the derivation rules of \( A \), while the remainder are the added loops.

It is easy to see that a word \( w = a_1, \ldots, a_n \) belongs to the grammar \( G_i \) iff some refinement of \( R_i \) contains a path of the shape \( p = l^*a_1l^* \cdots l^*a_nl^* \), starting (resp. ending) at the start (end) node of \( R_i \)’s implementation. The number of \( l \)’s between the word letters depends on the number of derivations steps performed to obtain \( w \)—each derivation step contributes at most two \( l \)’s between any two consecutive letters (due to the start and end nodes). An unbounded number of additional \( l \)’s can added via loop traversals.

Clearly, when some path \( p \) appears in the refinements of both \( R_1 \) and \( R_2 \) (hence the result of the query \( q \) described above is not empty) then the corresponding word \( w \) belongs to both \( G_1 \) and \( G_2 \). Namely the languages intersection is not empty. To see that the converse also holds, note that if the derivation tree of word \( w = a_1 \ldots a_n \) in \( G_i \) is of depth \( k_i \), then the refinement of \( R_i \) contains, among others, all paths of the form \( l^{k'}a_1l^{k'} \cdots l^{k'}a_nl^{k'} \), for \( k' \geq 2k_i \). This is because, as explained above, each derivation step contributes at most two \( l \)’s between any two consecutive letters, and the additional \( l \)’s need to reach a sequence of length \( k' \), can be added via loop traversal. In particular, if \( w \) belongs to both \( G_1 \) and \( G_2 \) than the refinements of both \( R_2 \) and \( R_2 \) contain a path \( p \) where \( k' = 2 \times \max(k_1, k_2) \), hence the query result is not be empty.

When the grammars \( G_i \) are not recursive, the depth of the derivation trees for words in \( G_i \) is bounded by a constant \( d \). We use here a similar construction as above except that instead of plugging a loop (that allows to generate an unbounded number of \( l \)’s) between the graph nodes, it suffices to connect them with paths of varying length, containing between 1 to \( 2d \) nodes labeled \( l \). This is illustrated in Fig. 24 for \( d = 1 \). Note that the obtained BPs contain no recursion and no loops, and their size is polynomial in the size of \( G_i \).

To conclude the proof we sketch the polynomial and exponential algorithms: 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 server holds a set of BPs and may provide (resp. use) activities to (of) remote servers. 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 server id with each process and each activity node. Queries may then annotate graph patterns and activity nodes by server ids, restricting the search to the specified servers. In particular, when a (transitive) activity node in a query is annotated by a server id, the search is restricted to implementations supplied by the specified server (resp. refinements consisting only of invocations of activities of the specified server). More generally, queries may use predicates on server ids to restrict the search to a specific family of servers.

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 servers 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 servers 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)\(^8\). 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.

7. Implementation

The query language presented above has been fully implemented and tested in the BP-QL distributed setting. 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 [21] plug-in. It allows to: design new BPs 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.

7.1. 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

---

\(^8\)First Order Logic augmented with Transitive Closure.
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.
- BPs typically operate in a cross-organization, distributed environment. The specifications of the services participating in process may reside on distinct servers. 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. 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 processing. Consequently, we decided to base our solution on an extension of XML, called AXML. AXML is essentially a middleware system that includes 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.

9For simplicity, the figure provides an abstraction of the actual BPEL XML file structure, with many details omitted.

10An alternative viable solution to the graph shape of BPs could be to use a native graph query engine.
materialization and transfer. Recursive calls are tracked, and only the relevant data are materialized (see [22] for details).

In summary, BP-QL uses the AXML system [15] 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 BP. 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, Fig. 26 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.

To illustrate the first type of call, the getActivity("join") nodes below the searchFlights and searchRooms, in the middle of Fig. 26, point to the join node below searchCars. They represent the fact that the three searches are followed by that same join operation.

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 servers 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 servers.

With this representation, and with the service calls replaced by the necessary data, 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.

7.2. 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 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 a series of experiments. Different benchmarks were proposed to test BPEL servers’ performance [23,24] and expressive power [25]. However, the BP specifications are targeted to test specific operations, and are too small and simplified in structure to test the performance of our query language. Other samples provided by vendors have the same drawbacks. With the lack of a suitable benchmark, we developed a synthetic benchmark of our own. 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 Fig. 27. 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 100 ms. AXML performs most calls in parallel, so the typical overall delay due to the materialization of data are also around this number.

As we can see, 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 BPs consisting of several compound activities, and varied the number of servers that hold the specifications of activities. At one extreme, the full specification resides on a single server. At the other extreme, each process activity is provided by a distinct server. We compared the execution time of queries on these varying configurations, considering both global queries (that consult the specifications on all servers) and local queries (where the search is restricted to only local specifications). Fig. 28 illustrates a representative sample of the results. It considers the Travel Agency from our running example, and the query from Fig. 4 (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 servers. 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 are available for querying.

8. Related work and conclusion

We presented BP-QL, a novel graphical Query Language for querying BPs. BP-QL allows users to query BPs 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 BPs, an abstraction of the emerging BPEL (business process execution language) standard [2]. Other previously proposed standards like [26–28] can similarly be supported, exploiting the abstraction level of our formal model. Several concrete models have been suggested for describing Web Services behavior including state-machine [29], Pi calculus [4], a variant of Mealy state machine using process algebra for transitions [30], Petri-nets [31].
models to trace-based formalisms like Message Sequence Charts [32]. BPEL4WS language includes artifacts from both XLANG [4] and WSFL [3], each of which took a different approach to workflow. Basically, there are two main approaches; reactive models like State Charts vs. interactive models like Petri-nets. Web services are reactive components, activated by a message arrival via their interface, and as such it is best to describe their behavior using a reactive model. There has been a vast amount of previous work in the general area of program analysis and verification (see e.g. [11,33] for a sample), and more specifically in the analysis of interactions of composite web services and BPEL processes [11,34,35]. 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 [11,12]). To guarantee 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 [33]) that are used to speed up program verification tasks can also be employed in our context to further accelerate query evaluation.

An interesting issue to consider is “unsatisfiable queries”. Since queries on BP graphs can be really complex, it is interesting to study when a query does not have an answer in any kind of graph. Observe that positive BP-QL queries are always satisfiable—it suffices to take an input with shape similar to that of the query (with the transitive edges replaced by regular ones). For queries with negations this is more complex. A query would be unsatisfiable if the negative part is “contained” in the positive part. The study of such query containment is left for future research.

The design of BP-QL was inspired by previous works on visual query languages for XML, such as XML-QL [36] and XQBE [37]. These languages are descendants of a long line of research on graph based query languages such as G [7], Graphlog [8] and G-Log [9]. Querying BP specifications is also related to schema query languages (e.g. [38,39]). Similar to [39] that allow queries to the structure of documents, we also treat paths as first-class citizens.

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 BP-QL (Business Processes Query Language) initiative in 2002 [40]. 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., [13,41,42] and commercial tools (e.g., [43–45]) 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.

References
