Monitoring Business Processes with Queries *

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

Anat Eyal
Tel Aviv University
anate@post.tau.ac.il

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

Alon Pilberg
Tel Aviv University
allonpil@post.tau.ac.il

ABSTRACT

Many enterprises nowadays use business processes, based on the BPEL standard, to achieve their goals. These are complex, often distributed, processes. Monitoring the execution of such processes for interesting patterns is critical for enforcing business policies and meeting efficiency and reliability goals. `BP-Mon` (Business Processes Monitoring) is a novel query language for monitoring business processes, that allows users to visually define monitoring tasks and associated reports, using a simple intuitive interface, similar to those used for designing BPEL processes. We describe here the `BP-Mon` language and its underlying formal model. We also present the language implementation and describe our novel optimization techniques. An important feature of the implementation is that `BP-Mon` queries are translated to BPEL processes that run on the same execution engine as the monitored processes. Our experiments indicate that this approach incurs very minimal overhead, hence is a practical and efficient approach to monitoring.

1. INTRODUCTION

A *Business Process* (BP for short) consists of some business activities undertaken by one or more organizations in pursuit of some particular goal. It often interacts with other BPs of the same or other organizations. *BP Management Systems* are software platforms that facilitate the definition, deployment, execution, and monitoring of BPs. Because of their central role in carrying out business activities, and their complexity, *monitoring* of BPs is a critical activity in modern enterprises.

For some intuition about the type of monitoring that a BP may require, I imagine a manager of a Web-accessible auctioning business. Monitoring of process executions may allow the manager to guarantee fair play, detect frauds, and track services usage and performance. The manager can ask, for instance, to be notified whenever an auctioneer cancels bids too often, or when buyers attempt to confirm bids without first giving their credit details, so that she can block their actions. Similarly, being notified whenever the average response time of the database in a given service passes a certain threshold, allows her to fix the problem or switch to a backup database. In general, BP monitoring encompasses the tracking of particular patterns in the executions of individual processes or in the interaction between different processes, as well as the provision of statistics on the performance of some processes or the system. *Our goal here is to provide intuitive, easy to use, efficient tools to facilitate this critical task.*

Before presenting our results let us briefly highlight some of the main characteristics of existing BP Management Systems and the challenges encountered in monitoring current BPs.

Background. BPs operate in a cross-organization, distributed environment, and the software implementing them is fairly complex. Standards enable the design, development and deployment of such software. The recent BPEL standard (Business Process Execution Language[6]) provides an XML-based language to describe both the *interface* exposed by a process, and its full *operational logic* and *execution flow*. Since the BPEL syntax is quite complex, commercial vendors offer systems that allow to design BPEL specifications via a visual interface, using an intuitive view of the process, as a graph of activity nodes connected by control flow edges. Designs are automatically converted to BPEL specifications. These can be automatically compiled into executable code that implements the described BP and runs on a BPEL application server [27].

An *instance* of a BP specification is an actual running process which includes specific decisions, real actions, and actual data. BP Management Systems allow to trace process instances – the activities they perform, messages sent or received by each activity, variable values, performance metrics – and send this information as events (in XML format) to *monitoring* systems (often called BAM – Business Activity Monitoring – systems). Typical monitoring systems (e.g. [3, 20]) allow users to specify events of interest, and actions to be performed when the events are identified. Events may be atomic or composite (i.e. consist of a group of other atomic or composite event). Detection and processing of (composite) events has been an active research area since the early 90's. Rich event algebras have been proposed for describing composite events (e.g. [16, 29]), and sophisticated evaluation and optimization techniques have been developed for their detection [23] (see Section 7 for details). Nevertheless, existing technology suffers from three main drawbacks when it comes to the monitoring of BPEL BPs.

Abstraction level. In existing systems, the specification of monitoring tasks and, in particular, of the the relevant (composite) events, requires intimate knowledge of both the monitored application and the specific events emitted by activities. This is contradictory to the high level abstraction employed when *defining* BPEL BPs, where implementation details (including the types of run-time events generated by the system) are deliberately hidden. Thus, programmers nowadays use two distinct tools, one for defining BPs at a high level

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

VLDB '07, September 23-28, 2007, Vienna, Austria.

of abstraction, typically via a graphical UI, and another for defining monitoring tasks for the BPs, typically via lower level Event-Condition-Action style rules. The abstraction gap between these tools is akin to the one between assembly and high level languages.

To close this gap, it is desirable that the specifications of monitoring be performed on the same (high) level of abstraction as that of the BPs, possibly even using a similar specification language. Such a monitoring tool would allow (a) simultaneous formulation, by the BP designer, of a BP and its corresponding monitoring tasks, and (b) a faster learning curve of the monitoring language.

Optimization. A variety of methods have been proposed for optimized processing of (composite) events, employing relation and object-oriented database technology [32, 16], petri nets, finite state automata, event graphs, and storage minimization (See [29] for a survey) . The proposed methods are generic, hence can be employed in a variety of application domains, including BP monitoring. A disadvantage, however, of a generic approach is that it does not exploit the particular properties of BPs and available knowledge about them. As a simple example, assume we wish to be notified when a given activity sequence occurs in some process. If, according to its (BPEL) specification, an activity o never co-occurs with such a sequence, monitoring for the sequence can be stopped immediately if an activation of o is detected. While such knowledge about BPs structure is naturally valuable for optimization, to our knowledge it is not exploited by any current BP monitoring tool.

Runtime monitoring that considers the processes structure has been studied, e.g., for models and query languages based on temporal logics as LTL (See Section 7). But what is desirable here are optimizations stated directly in terms of BPEL and a corresponding high-level monitoring language.

Deployment. As mentioned above, BPs are specified in a high level manner and the specifications are automatically compiled into executable code that can, in principle, run on any BPEL application server [27]. Analogously, it is desirable that a monitoring task be defined in a declarative manner, and be compiled, and easily deployed, on whatever BPEL application server chosen for the monitored BP. In existing monitoring tools, however, the monitoring tasks are written in proprietary languages and are not portable[11].

The `BP-Mon` (BP Monitoring) system presented here addresses these three problems, making the following contributions.

Query language We present a high-level intuitive graphical query language that allows for simple description of the execution patterns to be monitored. A tight analogy between the graphical interface used by commercial vendors for the *specification* of BPEL BPs and the graphical query interface that we use for *monitoring* allows natural and intuitive design of monitoring tasks.

Evaluation and optimization We provide a dedicated efficient automata-based algorithm to identify occurrences of monitored patterns. We present a novel optimization technique that speeds up computation, by pruning redundant monitoring, based on an analysis of the process BPEL specification.

Implementation and deployment To support flexible deployment, our system compiles a `BP-Mon` query q into a BPEL process specification S, whose instances perform the monitoring task. As for all standard BPEL specifications, S can now be automatically compiled into an executable code to be run on the same BPEL application server as the monitored BP. We describe experiments that indicate that the resulting monitoring is very efficient and incurs only very minimal overhead.

In summary, `BP-Mon` offers a high-level, intuitive design of monitoring tasks. It compiles these tasks into efficient and standard BPEL processes, thus providing easy deployment, portability, and minimal overhead.

Discussion. In a previous paper [4] we proposed to use a graphical query language for querying BP *specifications*. There, the goal was to be able to retrieve specifications with certain properties (e.g. where an execution path from activity A to activity B is possible), and the solution relied on modeling specifications and queries as graph grammars. In contrast, our work here is concerned with querying the *actual execution* of process instances (e.g. to find when an actual execution path that started at activity A arrives to activity B), and the solution is based on automata construction. The two works are complementary: The query language of [4] can be used to discover parts of BPs that require monitoring, while monitoring can be used to check at runtime properties that cannot be statically determined by querying the specification.

As mentioned above, events are sent to monitoring systems in XML format. A natural question is why not use XQuery, coupled with some XML stream-processing engine [22], to process this stream? A key observation is that the XML elements in this stream describe individual events. To express any non-trivial query about a process execution flow, one needs to write a fairly complex XQuery query, that performs an excessive number of joins, and can hardly (if at all) be handled by existing streaming engines. Furthermore, standard XML stream processing would still be inadequate for the task, even if a more query-friendly nested XML representation, that reflects the flow, had been chosen for the data. XML stream engines manage tree-shaped data, expect to receive the tree elements in document order, and process siblings sequentially, as they arrive. However, a BP execution is essentially a nested set of DAGs. In a DAG, some activities may run in parallel and interleave, hence the events flow in BPs does not necessarily follow document order. Nesting of DAGs in BPs follows from the fact that processes contain composite activities with complex internal execution flow, itself represented by a DAG. Interleaving of events from different DAGs of a BP is another aspect of parallelism. Here, parallel processing, that processes each event according to its position/nesting in the flow is called for. This is provided by `BP-Mon` .

Paper organization. Section 2 provides an overview of `BP-Mon`, and Section 3 briefly describes the underlying formal model. Section 4 deals with query evaluation and optimization. Extensions to the model are considered in Section 5. Section 6 describes our implementation and the experiments preformed to measure the performance of the system. We conclude in section 7.

2. MONITORING BUSINESS PROCESSES

We start by presenting an informal overview of `BP-Mon` via a running example that extends the Web auctioning BP scenario introduced in the Introduction.

2.1 Underlying technology

Let us first briefly describe some of the underlying technology; what BPEL BPs are and what data is available for their monitoring.

BPEL. As mentioned in Section 1, BPEL is essentially a high level specification language with an XML-syntax that allows to describe a process' execution flow and interaction with other processes. A BPEL specification describes a process as a DAG consisting of activities (nodes), and links (edges) between them that detail the execution order of the activities. (Cycles are captured by a particular *while* node, described below.) An activity is either *atomic* or *compound*. The atomic activities that can be used in a BPEL specification include operations such as *invoke*, for invoking an operation of some web service; *receive*, for waiting for a message from an external source; *reply*, for replying to an external source; and *assign*, for copying data from one variable to another. Compound activities are typically composed of several (atomic or compound) activities.

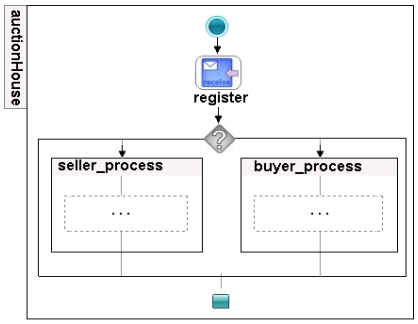

Figure 1: An Auction business process.

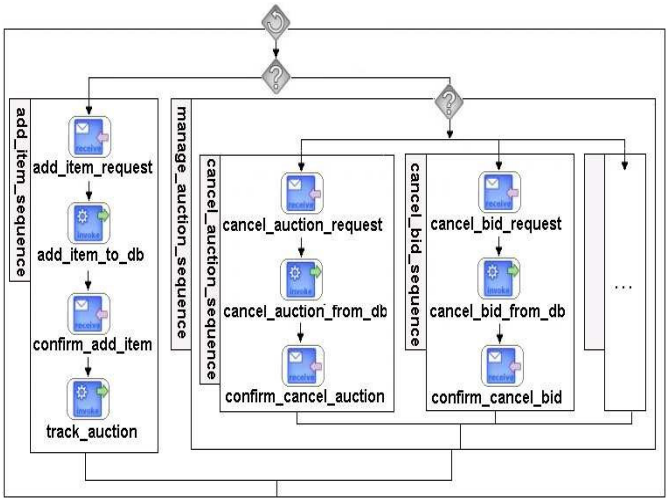

Figure 2: The seller flow.

Their types include *sequence*, where the component activities have sequential execution order; *flow*, where partial order is specified on component activities (possibly with parallelism), *switch*, for conditional execution; *while*, for looping; etc.

The BPEL XML-syntax is complex. Hence, commercial vendors offer systems that allow users to design BPEL specifications via an intuitive graphical interface, (with the graphical design being automatically converted to BPEL syntax). Figures 1 and 2 illustrate such an interface. The circle at the top of a BP (see Figure 1) is its entry point; the square at the bottom is its exit point. In the BP of Figure 1, users register to the system by invoking the *register* activity (whose details are not shown here). As part of this activity, they choose to play a seller or a buyer role. Depending on their choice, they are directed in the following *switch* activity to one of the two *compound activities*: *seller_process* and *buyer_process*. Figure 2 is a zoom-in into the seller process, that shows its internal flow.

Different icons, with activity names attached to them, denote different activity types. Each activity has associated data variables whose values can be tested and/or passed to other activities. For conciseness we omit these from the figures here. Activities that are invoked by (resp. invoke) other activities/users are marked by small incoming (outgoing) arrows. The BPEL *switch*, *while*, and *flow* constructs are represented by diamond shaped nodes that contain a question mark, a circular arrow, and two parallel lines, respectively. The *switch* icon in Figure 1 was explained above. The *while* icon at the top of Figure 2 indicates that the seller can repeat the described activity any number of times. At each round she can either manage her existing auctions (e.g. decide to cancel an auction, or to cancel some specific bid, etc.) or add new items for sale. New items are added to the database by the *add_item_to_db* activity. Once the update is confirmed the *track_auction* process is invoked to wait until the auction ends and declare the winner. The internal structure of this process is depicted in Figure 3. The *flow* construct here allows to handled the winner and the seller *in parallel*. The process notifies them about the auction results and awaits their approval.

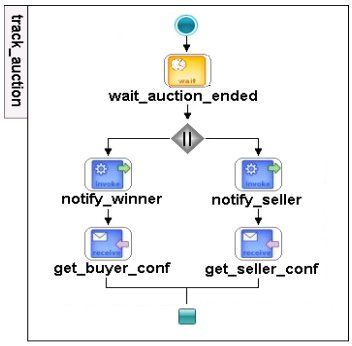

Figure 3: Auction notifier process.

```
⟨actionData⟩
  ⟨header⟩
    ⟨processName⟩ auctionHouse ⟨/processName⟩
    ⟨instanceId⟩ 517 ⟨/instanceId⟩
    ⟨sensor target="add_item_request"/⟩
    ⟨timestamp⟩ 2006-05-31T11:32:46.510+00:00 ⟨/timestamp⟩
  ⟨/header⟩
  ⟨payload⟩
    ⟨activityData⟩
      ⟨activityType⟩ receive ⟨/activityType⟩
      ⟨evalPoint⟩ completion ⟨/evalPoint⟩
      ⟨durationInSeconds⟩ 0.1 ⟨/durationInSeconds⟩
    ⟨/activityData⟩
    ⟨variableData⟩
      ⟨target⟩ $itemVar ⟨/target⟩
      ⟨data⟩ ⟨addItemRequest⟩
               ⟨category⟩ MP3 player ⟨/category⟩
               ⟨description⟩ iPod mini 4GB ⟨/description⟩
               ⟨price⟩ 50 ⟨/price⟩
             ⟨/addItemRequest⟩
      ⟨/data⟩ ⟨/variableData⟩ ⟨/payload⟩
⟨/actionData⟩
```

Figure 4: BPEL event.

BPEL events. An *instance* of a BPEL specification is an actual running process that follows the logic described in the specification. BP Management systems allow to trace instance executions. For each activity issued, two events are generated, at its activation and completion, respectively. Events are reported in XML format. Figure 4 shows a completion event for the *add_item_request* activity of Figure 2 (with some data omitted for brevity). The header includes identification information for the event: the BP name, the instance ID, the activity name, and a time-stamp. The provided data includes the activity type (e.g. *invoke*, *receive*, *sequence* etc.), the reporting point (activation or completion of the activity), the activity duration, and variables information (variable names and values).

For a compound activity, the events corresponding to its internal flow are reported between its activation and completion events. The events stream of an instance can be viewed as a (nested) DAG (see Figure 5). The nodes for an activity represent its activation and completion events, resp. *Flow* edges (represented in the figure by solid arrows) connect activation and completion nodes of the same activity and record causal dependencies between distinct activities of a process. *Zoom-in* edges (represented by dashed arrows) connect the activation (resp. completion) node of each compound activity to the the start (rep. end) nodes of the DAG that describes the activity's internal flow. Note that the edges in the DAG connect nodes with increasing time stamps. Recall that some activities may run in parallel (e.g. *notify_winner* and *notify_seller*). At any given time t, the DAG represents the execution up to point t.

2.2 BP-Mon

In the auction scenario, the system supervisor may want to be notified when a seller cancels bids or auctions too often, or when buyers attempt to confirm bids without first giving their credit details. She may also want to be informed when the average response time

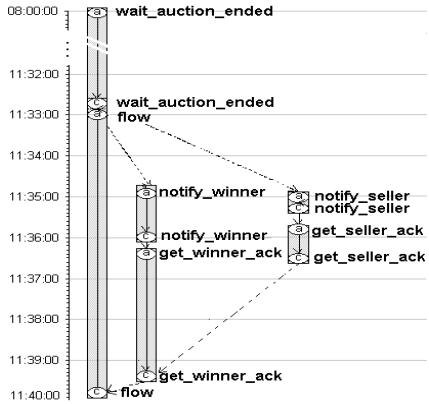

Figure 5: Execution Trace as DAG.

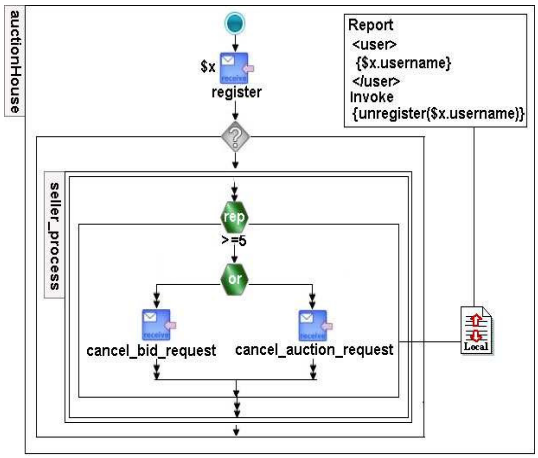

Figure 6: Too many cancels.

of the database server for a given service passes a certain threshold, and to gather statistics about the response time. BP−Mon monitoring queries can be used to accomplish these tasks.

For monitoring process instances, BP−Mon uses *execution patterns* (abbr. EX-patterns). Intuitively, these extend string regular expressions to (nested) DAGs. The patterns look much like the specifications. In addition to standard BPEL constructs, such as *while*, *switch*, etc., they may include two additional new constructs, denoted *or* and *rep*, describing, resp., alternative patterns and repetitions. The patterns also allow to navigate in the activities flow along two axes: path-based and zoom-in-based. Following the use of / and // in XPath to denote single and multiple step navigation, our patterns use edges with single and double heads to denote single and multiple edge paths, resp. Similarly, compound activities may have singly or doubly bounded boxes, the latter denoting an unbounded zoom-in into the activities internal flow.

The activities and edges of EX-patterns can be associated with variables, which can be used in selection conditions on the values of the associated attribues/data variables and in reports. To issue a report, a *reporting icon*, depicted as a page with two small arrows, can be connected to a *reporting point* in the pattern (an atomic or a compound activity). A BP−Mon query may include several such reporting icons/points. Two reporting modes are available: *local*, where an individual report is issued for each process instance, and *global*, that considers all the BP instances. For each report, one can specify when should it be issued (e.g. at the first time that the reporting point is reached, at periodic time interval, or when certain conditions are satisfied) and what should be the structure of the output (in XML format) or the actions triggered at this point. The following examples illustrate the features available for monitoring.

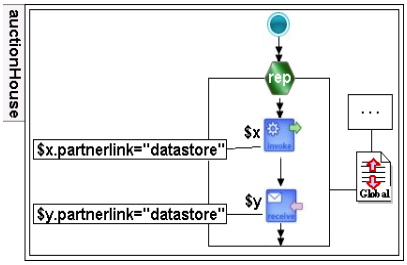

Figure 7: Average response time.

EXAMPLE 2.1. *The query in Fig. 6, monitors auctions to guarantee fair play. It looks for users that register as sellers, and repeatedly cancel bids or auctions. The 'or' here denotes that we are looking for an occurrence of one of the two cancel activities. The 'rep' denotes repetitions of this pattern, with the* ≥ 5 *indicating that at least 5 occurrences are required. The double headed arrows indicate that the activities may occur at any distance from the beginning of the seller process, and also at any distance from each other (in the given instance). The double bounding of the* seller_process *box denotes unbounded zoom-in; we look for cancelation activities in this process and (transitively) the compound activities that it includes/invokes. A report, with the name of the corrupt auctioneer, is issued as soon as the pattern is matched, i.e. when five cancelations are identified. If we want to get re-notified if/when cancelations are further repeated, a* Report* *command can be used instead. Finally, to trigger corrective activity, an* Invoke *command is used.*

EXAMPLE 2.2. *The query in Figure 7 may be used to guarantee service quality. The* datastore *service is in charge of interaction with the database and is used massively in the auctioning process to store and manage items and bids. To monitor its response time, we look for a pattern of an* invoke *activity, immediately followed by a* receive *operation. (The* partnerlink *attribute identifies the target/source service). Note that the single headed arrow here indicates consecutive operations. Also note that we use here a global reporting mode that aggregates the data of all the BP instances. Let us now consider some types of reports. The following is an example for a time-based sliding window report, where we request to get an hourly report of the average response time and standard deviation in the last couple of hours:*

```
Report* Every 1 hrs Range 2 hrs
<response-time>
  <avg>avg($y.startTime-$x.endTime)</avg>
  <std>std($y.startTime-$x.endTime)</std>
</response-time>
```

BP−Mon *also supports match-based windows where the window slides over the previous matches of the pattern in the given instance (if the report is local) or in all the running instances of the given BP (if it is global). For example, to issue a report, every 100 matches, that provides the average response time and standard deviation of the last 200 calls to the database, we could use in the above report*

```
Every 100 entries Range 200 entries
```

Grouping may also be employed. For instance, we can group the database calls according to the type of the requested operation and report response time only for operations with frequency ≥ 10*. For that we add at the bottom of the above command*

```
Group by $x.operation having count()>10
```

EXAMPLE 2.3. *Assume that, to promote sales, we wish to periodically give prizes to our users, for example, to credit the seller and buyer pairs in the 10000th sell, 20000th sell, 30000th sell, etc. The query in Figure 8 reports the names of the winners. Since notifications for each buyer-auctioneer pair are processed in parallel (recall the* flow *construct in figure 3), so is their monitoring.*

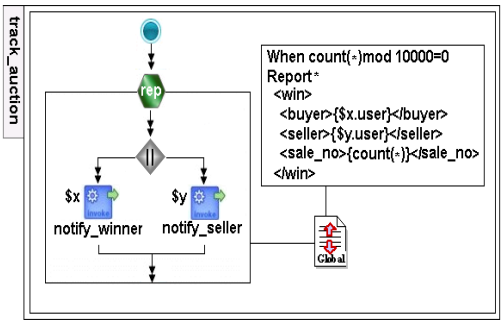

Figure 8: x10,000 buyer.

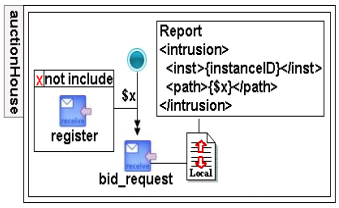

Figure 9: No registration.

EXAMPLE 2.4. *Finally, the query in Figure 9 monitors illegal access. It identifies instances where a user attempts to submit bids without first registering to the system, and reports the instance ID and the corrupt execution path. We use here a* path predicate *(essentially a subquery) that is attached to the transitive edge connecting the start node to the* bid_request *node and restricts the assigned paths to those that do* ***not*** *include registration.*

We may furthermore want to combine run-time monitoring with specification analysis, and identify execution paths that do not comply with the BP specification. The query in Figure 10 compares a bidder's run-time execution flow (the $x on the left) to what is allowed according to the specification (the $y on the right). In this simple example the two query patterns, on the run and the specification, are similar, but in general one can use different patterns, e.g. one pattern on the specification to identify what needs monitoring, and another pattern on the run to perform this monitoring.

The queries so far all have a single reporting point. We also support queries with multiple reports.

EXAMPLE 2.5. *Assume that we wish to obtain weekly statistics about the the average age of the users that register to the system at different times of the day. We can attach to the* register *node of the query in Figure 6 the following report request.*

```
Report Every 1 week Range 1 week
  <age-by-hour>
    <avg>avg($x.age)</avg>
  </age-by-hour>
  Group by $x.startTime.hour()
```

Here is a comment about the semantics of such report points (a full account is given in Sections 3 and 5). As mentioned earlier, a user may request a report to be issued as soon as a match for the pattern is identified. In order not to block reporting, only the nodes and edges in the pattern that *precede* a report point are considered relevant to it. For a report to be conditional on the occurrence of the *full* pattern it needs to be attached to the last node in the pattern or to the outermost box, as in all the previous examples. Thus, the report here will include information about *all registered users*, regardless of whether they had later canceled bids or not. To get the same reports only for corrupt users, the same report should have been connected to the *rep* or the auctionHouse boxes.

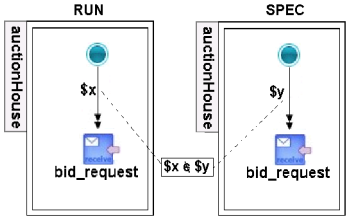

Figure 10: Static and dynamic analysis.

3. THE FORMAL MODEL

BP-Mon queries consist of two main ingredients: (1) EX-patterns that are matched to execution traces and (2) reports generated from these matches. Reports are discussed in Section 5; we focus here on BP-Mon's pattern matching. We first explain how execution traces are modeled and then consider EX-patterns and their semantics. (An efficient algorithm to identify pattern occurrences is presented in section 6). To simplify the presentation we consider in this and the next section a basic data model. We then enrich it in Section 5 to obtain the full fledged model.

Event traces. As mentioned earlier, the execution trace of a process instance can be viewed as a DAG. Each activity is represented by a pair of time-stamped nodes, corresponding to its activation and completion. For a compound activity, the DAG that represents its internal flow appears (time-wise) between its activity activation and completion nodes, and is connected to them by *zoom-in* edges. This is formalized below.

We assume the existence of three domains, $\mathcal{N}$ of nodes, $\mathcal{L}$ of node labels, and an ordered domain $\mathcal{T}$ of time stamps. We first define the auxiliary notion of *activation-completion labeled DAGs*.

DEFINITION 3.1. *An* activation-completion *labeled DAG is a tuple $G = (N, E, \lambda, \tau)$ in which $N \subset \mathcal{N}$ is a finite set of nodes, E is a set of edges with endpoints in N, $\lambda : N \to \mathcal{L}$ is a labeling function on the nodes, and $\tau : N \to \mathcal{T}$ is a time-stamp function on the nodes. We assume G satisfies the following:*

1. *The edges in E are of two types:* flow, *and* zoom-in.
2. *The nodes in N are partitioned into pairs, called* activity pairs. *Each pair n_1, n_2 shares a label, i.e., $\lambda(n_1) = \lambda(n_2)$. In such a pair, one node is designated as an* activation, *the other as a* completion*; they are denoted by* act(l) *and* com(l), *resp., where l is their shared label. There is precisely one flow edge from* act(l) *to* com(l)*; no other flow edges leave* act(l)*, and no other flow edges enter* com(l)*.*
3. *τ assigns distinct time stamps to nodes of G s.t. if there is an edge from n_1 to n_2, then $\tau(n_1) < \tau(n_2)$.*

We assume the graph has a single start *node without incoming edges, and a single* end *node without outgoing edges, denoted by* start(G) *and* end(G)*, resp.*

DEFINITION 3.2. *The set $\mathcal{EX}$ of* execution traces *(abbr.* EX-traces*) is the smallest set of graphs that satisfies the following.*

1. **[flat trace]** *If G is an activation-completion labeled DAG without zoom-in edges, then $G \in \mathcal{EX}$.*
2. **[nested trace]** *If G_1, G_2 are in $\mathcal{EX}$, and (*act(l), com(l)*) is an activity pair of G_1, then the graph G consisting of G_1, G_2, and two new zoom-in edges (*act(l), start(G_2)*) and (*end(G_2), com(l)*), is in $\mathcal{EX}$,* provided *the combined time-stamp function τ on $G_1 \cup G_2$ satisfies constraint 3 of Definition 3.1 above.*

A prefix of an EX-trace is defined in the standard way, as any graph obtained by removing some nodes, all their descendent nodes, and all edges into and out of deleted nodes.

In the sequel, we call a subgraph G_2 that is connected, as in Item 2 above, by zoom-in edges to an activity pair (act(l), com(l)),

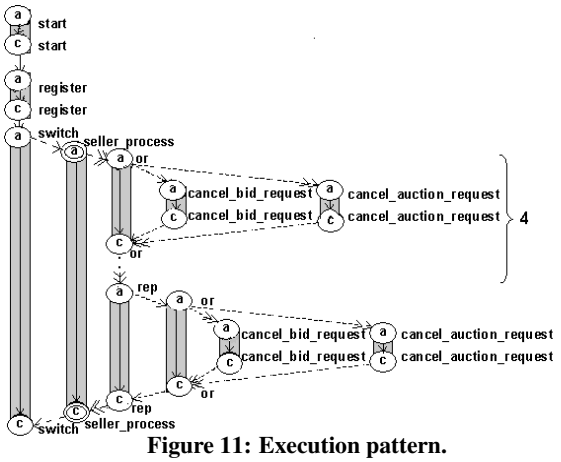

Figure 11: Execution pattern.

an *internal trace* of the pair. Such a subgraph, omitting the internal traces of its own activities, is called a *direct internal trace*. Observe that in general a given activity pair may have several internal traces connected to it. This happens when the activity implementation includes several parallel processes.

Execution patterns. Queries are modeled by execution patterns (abbr. EX-patterns), that generalize EX-traces similarly to the way tree patterns generalize XML trees. EX-patterns are EX-traces without time stamps (since they are not real executions but just patterns) where node labels are either specified, or left open using a special ANY symbol, and where two additional new label symbols can be used: or and rep. or describes alternative patterns and rep describes one or more repetitions of a given pattern. Edges in a graph are either regular edges, interpreted over edges, or transitive, interpreted over paths. Similarly, activity pairs may be regular or transitive, for searching only in their direct internal trace or zoom-in transitively inside it.

DEFINITION 3.3. *An* execution pattern *(EX-pattern) is a pair $p = (\hat{e}, T)$ where $\hat{e}$ is an EX-trace without time stamps*[1]*, whose nodes are labeled by labels from $\mathcal{L} \cup \{any, rep, or\}$, and T is a distinguished set of activity pairs and flow and zoom-in edges in $\hat{e}$, called* transitive *activities and edges, resp. The nodes in p with labels other than rep and or are called* concrete *nodes. We say that p is* concrete *if it contains only concrete nodes.*

Graphical BP-Mon queries are naturally mapped to EX-patterns: Each activity icon labeled l is mapped to a pair of nodes act(l), com(l), that inherit properties like double bounding. Additionally, nested activities are connected by zoom-in edges (simple or double-headed) to these two nodes in the obvious manner. For example, Figure 11 depicts the EX-pattern corresponding to the query of Figure 6. (The reporting part is omitted for now and will be considered in Section 5). The transitive edges are double headed and the transitive activity pairs are double bounded. The ≥ 5 that was attached to the rep node on the query is a shorthand for a sequence of 4 occurrences of the pattern followed by a regular rep node.

Intuitively, EX-patterns with or and rep nodes extend string regular expressions to concrete EX-pattern expressions. Namely, each EX-pattern defines a (possibly infinite) set of concrete EX-patterns, denoted $concrete(p)$, obtained from p by replacing each rep pair by a sequence of one or more copies of its internal trace and each or pair by one of its possible internal traces. (We omit the formal definition for space constraints).

To evaluate a query, the patterns in $concrete(p)$ are matched to a given EX-trace. A match is represented by an *embedding*.

[1]An EX-trace without time stamps is defined as in Definitions 3.1 and 3.2 above, by dropping the time-stamp function τ, and the corresponding constraints.

DEFINITION 3.4. *Let $p = (\hat{e}, T)$ be a concrete EX-pattern and let e be an EX-trace. An embedding of p into e is a homomorphism ψ from the nodes and edges in p to nodes edges and paths in e s.t.*

1. **[nodes]** *an activation (resp. completion) node is mapped to an activation (completion) node. Node labels are preserved; however, a node labeled by any can be mapped to any node.*

2. **[edges]** *each (transitive) edge from node m to node n in p is mapped to an edge (path) from $\psi(m)$ to $\psi(n)$ in e. If the edge $[n, m]$ belongs to a direct internal trace of a transitive activity, the edge(s on the path) from $\psi(m)$ to $\psi(n)$ can be of any type (flow, or zoom-in) and otherwise must have the same type as $[n, m]$.*

The start *and* end *of ψ, denoted by $start(\psi)$ and $end(\psi)$, are the earliest and the latest time stamps of nodes in $\psi(p)$, respectively.*

A pattern may have many matches in a given EX-trace. In some cases, users are satisfied by one match. In other cases, they may want to be informed on all (or some) matches. When one match suffices, it is desirable to find an early one. In the next section, we present an algorithm that is guaranteed to find a match, if one exists, and that can also find all matches, if so desired. The algorithm works in a greedy manner, matching pattern nodes to the earliest possible events. We next formally define the property of the first match it finds. We use the following auxiliary notations. Given a concrete EX-pattern p, and an embedding ψ' on a prefix p' of p, we say that an embedding ψ of p *extends* ψ', if ψ agrees with ψ' on p'. If S is a set of embeddings of p, we denote by $S_{\downarrow \psi'}$ the set of embeddings in S that extend ψ', restricted to $nodes(p) \backslash nodes(p')$. When S is a singleton $\{\psi\}$, we write $\psi_{\downarrow \psi'}$.

DEFINITION 3.5. *Let p be an EX-pattern, e an EX-trace, and S a set of embeddings of patterns in $concrete(p)$ into e. An embedding $\psi \in S$ is* greedy *(in S) if the following holds:*
(1) $start(\psi)$ is minimal in the set $\{start(\phi) \mid \phi \in S\}$.
(2) Let n be the node with minimal time stamp in ψ, i.e. $\tau(\psi(n)) = start(\psi)$, and denote ψ restricted to n by ψ'. Then, inductively, $\psi_{\downarrow \psi'}$ is greedy in $S_{\downarrow \psi'}$.

It is easy to show (by induction on the pattern size) that

PROPOSITION 3.6. *For every EX-pattern p and every EX-trace e, if the set S of embeddings of p into e is not empty, then an embedding that is greedy in S exists.*

Assume given an algorithm that given an EX-pattern p and an Ex-trace e, finds an embedding ϕ that is greedy w.r.t the set of all embeddings of p in e. The following observation, that follows easily from the proposition, implies that the algorithm can easily be extended to find all embeddings of p into e.

OBSERVATION 3.7. *If the set S of embeddings of p contains more than one embedding, and ψ is a greedy embedding in S, then $S \setminus \{\psi\}$ is a non-empty set of embeddings, hence it contains an embedding ϕ that is greedy in it, such that $start(\psi) \leq start(\phi)$.*

Note that since EX-patterns may contain choices (e.g. or) several greedy matches with the same start time may exist. Indeed, even for some concrete EX-patterns more than one greedy match exist, e.g. due to symmetry in the EX-pattern. If several such greedy matches exist, one can be chosen arbitrarily.

4. MATCHING AND OPTIMIZATION

We next explain how pattern matches are detected. We start by describing a simple pattern matching algorithm, then propose an effective optimization technique that exploits the BP specification to speed up computation, by focusing on the relevant parts of the

events trace. It should be noted, however, that already the simple initial algorithm exploits knowledge about the common structure of BP traces, i.e. their nested DAG shape, to optimize the processing. In particular, when searching for an occurrence of a subpattern in the internal trace of a compound activity, if a completion event for the activity occurs, the algorithm immediately infers that the pattern can no longer occur in this internal trace and backtracks. (See details below). While this may remind the reader of XML stream-engines (which, when encountering an end-tag of an element, infer that the matching of a subpattern inside the element failed), there are two important differences which make the processing of BP patterns more intricate. First, BP patterns contain *two* navigation axes: the standard path-based navigation and the novel zoom-in navigation that allow users to query about activity flows that are nested at any depth inside the internal traces of compound activities. Second, unlike XML streams, where tree elements arrive in document order and siblings can be processed sequentially, in BPs the events of parallel sibling activities interleave. Here, a parallel processing of events according to their position in the flow is called for.

4.1 Pattern Matching

We assume that the execution of processes, and matching of patterns, start at time 0. We are given an EX-pattern p and our goal is to find the matches for p in an (incrementally discovered) EX-trace e. At a time t, what is known from an EX-trace e is only a prefix consisting of the nodes with time-stamp $\leq t$ and the edges between them. Each arriving new event is appended to the prefix, with incoming edges of the two kinds described in Section 3.

To simplify the presentation, we first assume that p is a concrete EX-pattern. After presenting the algorithm for this restricted case, we explain how it extends naturally to general EX-patterns.

Concrete patterns. The algorithm works in a greedy manner, trying to incrementally extend a greedy embedding for a prefix of p (initially empty), to a greedy embedding for a larger prefix. On failure it backtracks, refines the prefix embedding and retries to proceed again. Given an EX-pattern p we construct an automaton A whose states are the nodes of p. Its start (resp. end) states correspond to the `start` (`end`) nodes of p. A state can be *active* or *inactive*. Initially, only the start state is active. Other states become active once they get activation messages from *all* their respective parents, or due to the `backtrack` operator described below.

We maintain two data structures for backtracking. The first, an (initially empty) list called the `events-list`, contains trace nodes that may need to be (re)processed. Each new event (node) is appended to its end. The second, called `tested`, is a map from a subset of the states to events in `events-list`, representing the embedding computed thus far for some prefix of the pattern. Initially the mapping of all states is set to null. Each state (pattern node) maintains a `current-event` variable that points to an event in the `events-list` that the state needs to process. If it points to the place after last in the list it means that the state awaits the arrival of a new event. Initially the `current-event` of the start state points to the beginning of `events-list` and the `current-event` of all other states are set to null.

Each active state executes the algorithm depicted in Fig. 12. We assume that every iteration of the algorithm (the body of the while loop), which involves reading and possibly writing in the data structures and (in)activating some states, is executed atomically (our implementation uses for that a simple locking mechanism.)

Each active state s reads iteratively events from `events-list` (line 2) and processes them. This processing stops when s becomes inactive, and restarts when s becomes active again. If an event matches the conditions on the state and on its incoming edges (line 4), it is added to `tested` (line 5). The state is then inactivated (line 6) and we proceed with (i.e. sends activation message to) its children[2] (line 7). The $match?$ predicate is depicted in Figure 13. It tests whether, given a state s, an event n, and the `tested` entries of the parents of s, n is a potential assignment for s.

	Automaton state s
1	While s is active do:
2	n = `current-event`.
3	Advance `current-event` to next event in `events-list`.
4	If $match?(s, n)$
5	(a) Set s's entry in `tested` to point to n.
6	(b) Inactivate s.
7	(c) Send an activation message to the children states of s, setting their `current-event` to that of s.
8	Else % not matched %
9	If n is a completion event, s is a completion state, and the activation event of n's activity is assigned in `tested` to the activation counterpart of s,
10	`backtrack`(s)
11	Else, if n is a completion event for one of s's ancestors in the zoom-in hierarchy, or a completion event for the `end` activity,
12	`backtrack`$'(s)$
13	End While

Figure 12: Processing events.

	match?(state s, event n): boolean
1	If (a) n's labels satisfies the label conditions of s, and
2	(b) for every parent $\hat{s}$ of s, `tested` contains some assignment $\hat{n}$ to $\hat{s}$ and the trace path from $\hat{n}$'to n satisfies the conditions on the edge $(\hat{s}, s)$ in the pattern,
3	return True
4	Else return False

Figure 13: Matching an event.

If a match of an activation node fails, the event is skipped, and the algorithm proceeds to the next event. For a failure of a completion event, we consider two cases: First (line 9), if the state represents activity completion whose activation counterpart was matched in `tested` with the activation counterpart of the given event (but the given state and event nevertheless don't match), this implies a failure for the part of the pattern involving the implementation of the activation-completion pair. The algorithm then backtracks (line 10), trying to find another match for this part. The `backtrack` operator is described in Figure 14. It finds the last point where a decision was made on the matching of a previous relevant activity $\hat{s}$ and retries from there.[3] A second type of failure that needs treatment (line 11) is when the event is a completion for an activity whose activation is assigned in `tested` to an ancestor s' of s (or similarly, if the event is the last event of the trace). This implies a failure to match the part of the pattern from s' to its completion (resp. from the beginning of the pattern to its completion). Here, the algorithm backtracks (lines 12), trying to find another match for this part, using a different version of `backtrack`, denoted `backtrack'`, that considers for backtracking only ancestors *in the same or higher level in the zoom-in hierarchy*.

A match for the pattern is identified if&when the final state of the automaton is inserted into `tested`. On success, `tested` contains the match for the pattern nodes. The mapping for the edges consists of the edges/paths that were used to qualify these assignments in line (2) of the $match?$ procedure. To find consecutive matches, if requested, a $backtrack$ operation is applied to the final state and the matching process continues to find the next match. The matching fails if all active states read the end event of the trace before a successful match is found. An earlier detection of failure is considered below.

[2]A child becomes active after receiving messages from all its parents. It then starts reading events from the last `current-event` it received.

[3]The reactivated $\hat{s}$ now begins to read events starting from its `current-event`.

	backtrack(state s)
1	Choose an ancestor $\hat{s}$ of s, whose event in tested is an activation event with maximal timestamp (among s's ancestors).
2	Clear, in tested, the entries of $\hat{s}$ and its descendant states.
3	Inactivate the descendants of $\hat{s}$ and set their current-event to null,
4	Reactivate $\hat{s}$.

Figure 14: Backtrack.

The correctness of the algorithm is proved by induction on the size of the pattern (omitted here). The worse case time complexity of the algorithm is polynomial in the size of the trace (with the exponent determined by the size of the pattern). The intuition is that the algorithm exhaustively checks all relevant embeddings, and the upper bound on their number is polynomial in the size of the trace (with the exponent determined by the pattern size).

Before extending the algorithm to work with general patterns, let us comment about some of its properties.

Remark 1 The algorithm works greedily, in a deterministic manner, attempting to match events as early as possibly and backtracking on failure. Two possible alternatives could be (1) to use a non-deterministic automaton that checks simultaneously all possible embeddings, thus avoiding backtracking, and (2) to construct some deterministic variant of that non-deterministic automaton. Just like for standard regular expressions, a disadvantage of the first approach is the need to manage simultaneously a large number of active states[18]. A disadvantage of the second approach is the potential exponential growth in the size of the automaton[17]. Our algorithm provides a hybrid solution. We use a small automaton with the same size as the pattern, and since states are inactivated as soon as a matching event is assigned to them, only relatively few states are simultaneously active. The price paid for this is the need for backtracking. An optimization technique that allows to identify failures early and thus to avoid some redundant work and backtracking is presented below. Our experiments, presented in Section 6, show the optimized algorithm to be extremely efficient.

Remark 2 The events of the trace are recorded in events-list for backtracking. It is easy to see that an event n will never be reprocessed if n and its preceding events are not pointed by tested or any of the current-event variables of the states. Such an event can be removed from the list. We show below that the optimization technique mentioned above is also useful for identifying such redundant events.

It is possible to build (rather artificial) scenarios where all events must be retained in events-list "for ever". For example, consider a BP with an activity A that invokes itself (recursively) and may also, arbitrarily later, invoke some other activity B. Assume that our query searches for an A activity that invoked both A and B. If the given BP trace contains a long sequence of A's, we need to keep them all since we do not know in advance which of them (if any) will invoke a B later on. The problem here is that all the A activities remain "alive" for an unbounded time, hence may invoke new children activities arbitrarily late. In practice, in a typical BP, the number of individual activities that are kept alive unboundedly is bounded, so such phenomena are unlikely to occur. Indeed, in all the real life examples we examined, the number of events that needed to be retained was fairly small and proportional to the pattern size. Finding bounds on the number of events needed to be recorded, for various fragments of BP-Mon, is an on-going research.

Handling or and rep. We briefly sketch below the adjustments needed to handle or and rep.

[or] Consider an activity pair (act(or), com(or)) in p. When the automaton state s of act(or) (resp. the state s' of com(or)) is activated it does not read any events but immediately sends activation messages to all its children (with its $current\_event$). For s' to get activated it suffices that it receives an activation message from *one* of the activity internal traces. The children of s (resp. s') check $match$? w.r.t their grandparents rather than their parents (or great grandparents if the grandparents are also or nodes). For the children of s', a more lenient version of $match$? is employed, where condition (b) needs to be satisfied only for the grandparent that activated s'.
Since s' may now be activated several times, due to several branches of the or, and consequently its children may be matched to several events, we maintain in tested a *set* of events for each state, corresponding to the various possible matches. The context of each matching (i.e. to which choices of or branches it corresponds) is recorded with the events, and all consequent tests/operations take into consideration only assignments relevant to the given context. We omit the details for space constraints.

[rep] The processing of rep follows similar lines. It is based on the observation that an activity pair (act(rep), com(rep)) in p, which stands for one or more repetitions of some subpattern p', can be viewed as an or between the pattern p' and the pattern containing one occurrence of p' followed by another rep of p'. This "virtual" or is treated as above, recursively.

4.2 Optimization

So far, our algorithm ignored the BPEL specifications of the monitored processes. Let us now see how to use them to avoid redundant processing and to record only useful history.

As a simple example, consider the query in Figure 6, that monitors corrupt sellers, and the auctionHouse BP in Figure 1. If the process trace reports an invocation of the $buyer\_process$ compound activity, we immediately know that this activity, as well as all the events in its internal trace, are *irrelevant* to the query pattern and can be ignored. Furthermore, since the BP specification indicates that only one of $buyer\_process$ and $seller\_process$ can occur in a given process instance, we can infer that the invocation of $buyer\_process$ is *inconsistent* with the pattern and a match for the pattern is impossible. These notions are now formally defined.

DEFINITION 4.1. *Let S be a BPEL specification and o an activity in S. Given an EX-pattern p and a node n (resp. an edge e) in p, we say that the activation/completion of o is* irrelevant *to n (resp. e) if there is no embedding of p into an EX-trace of S where o's activation/completion event is assigned to n (resp. appears on a path assigned to e).*

We say that o is inconsistent *with p if p cannot be embedded into any EX-trace of S that contains an activation event of o.*

We explain below how irrelevancy and inconsistency are determined. For now, assuming that such a detection algorithm is given, we show how it can be used to refine the algorithm described above.

- When an active state reads events from the events-list, it can ignore the events that are irrelevant to the corresponding pattern node. This prevents event assignments that will for sure be detected later as unfit.
  Events that are irrelevant to all the pattern nodes and edges need not be recorded in the events-list. An event n that is relevant to some of the states/edges may be removed from the events-list as soon as it is not pointed by tested or any of the current-event variables of the states, and is also irrelevant to all states with current-event pointing to a preceding event and their descendent states and edges (as it will never be useful for backtracking).
- When an active state reads an activation event for an activity that is inconsistent with the query pattern it can immediately declare failure and stop the query processing.

- Similarly, if an active state reads an activation event for an activity that is inconsistent with the internal trace of some of the state's ancestors trough the zoom-in relationship (w.r.t the specification of the activity currently assigned to that ancestor), a backtrack operation to the ancestor can be issued. This early backtracking eliminates future redundant matchings.

Testing irrelevance and inconsistency. To conclude the discussion, we need to explain how irrelevance and inconsistency are tested. To check the irrelevance of the activation/completion of an activity o to a node n (edge e) in the pattern p, one needs to test if an instance of the given BP may contain a subtrace of shape similar to p where the activation/completion of o represents n (resp. appears on the path that represents e). If not, the activity is irrelevant to the pattern node (edge). To check inconsistency one needs to check if a BP instance that contains both an activation of o and a subtrace of the shape p may exist. Again, if not, the activity is inconsistent with the pattern.

The key difficulty here is that analyzing the *possible runs* of a BP is essentially a verification problem [15] and is typically of very high complexity (from NP-hard for very simple specifications to undecidable in the general case [26]). To overcome this, and nevertheless provide an algorithm of tractable complexity, we have decided to rely on a *safe*, rather than exact, detection of irrelevance and inconsistency. Namely, our algorithm may miss some cases of these properties, but those that are identified as irerelevant/inconssitent are indeed such. *Optimization-wise this only means that some optimization opportunities may be missed, but the correctness of the matching algorithm is not compromised.*

To detect irrelevance and inconsistency in a safe manner we rely on a query language, called BP-QL, which we have developed in a previous work [4], for analyzing BP specifications. BP-QL is a graphical query language with syntax very similar to that of `BP-Mon` . The key difference between the two languages is in the semantics of the queries: `BP-Mon` is given as input an *execution trace* and checks whether the specified pattern appears in the trace; BP-QL, on the other hand, is given a BP (BPEL) *specification* and checks whether the pattern may appear in some possible instance of the specified BP. If so, it retrieves, for each pattern node (edge), the set of activities relevant to it. To guaranty query evaluation of polynomial time complexity, BP-QL ignores the run-time semantics of certain BPEL constructs such as conditional execution and variable values. So query answers may be a superset of the actual answers (see [4] for details).

The (safe) detection irrelevance and inconsistency works as follows: To find irrelevant activities, we interpret the `BP-Mon` pattern as a BP-QL query over the BP specification. Activities that are not returned for given pattern node (edge) are all irrelevant for it. To check for inconsistency of an activity o with the given `BP-Mon` pattern, we add o to the pattern and interpret the augmented pattern as a BP-QL query. If query the result is empty the activity is inconsistent with the pattern. Observe that since query answers are supersets of the exact answers we may not identify the irrelevance or inconsistency of some activities, but all those that are identified as irerelevant/inconssitent are indeed such.

It is important to note that since we are querying the BP *specifications*, all the decisions regarding the potential inconsistency (irrelevance) of activities with (to) the pattern (pattern nodes) can be made statically, at compile time, *before the monitoring starts*, hence cause no delays in the actual monitoring processes.

5. THE FULL LANGUAGE

For simplicity, we used so far a very simple data model and ignored report generation. We now briefly consider useful extensions that enhance the expressive power, and facilitate the monitoring of real life business processes.

Data values and predicates. In practice, an execution trace carries additional information about the performed activities, such as the names and values of data variables. This is modeled by labeling the nodes in both EX-traces and EX-patterns with this additional data, requiring the embeddings to respect these labels too. EX-patterns may also use label or path predicated. For instance, rather than searching specifically for *cancel_auction_request* and *cancel_bid_request*, one may ask for all the activities whose name contains the string "cancel". One can also use predicates on the activity time-stamps to focus the search on certain time intervals.

Variables and joins. Ex-patterns can be extended by attaching variables to concrete activities and edges, and by (in)equality conditions on variables. Pattern activities attached to (un)equal variables are mapped to trace activities all having (distinct) identical labels, and pattern edges labeled by (un)equal variables are mapped to paths whose sequences of labels are all (different) equal words.

Querying specifications. In some cases we want to relate monitoring to the BP specification. See, e.g., Example 2.4 in Section 2. To query specifications, we rely again on BP-QL [4], whose graphical interface resembles ours. Queries then consist of two parts, that query the specification and the execution trace, respectively. (See, e.g. Figure 10). Join conditions between variables attached to the nodes/paths of the two parts provide the glue between them.

Distributed systems and queries. In a distributed setting, each peer holds a set of BPs and may provide (resp. use) activities to (of) remote peers. Users may wish to monitor these remote components as well, (provided access is allows by the respective organizations). The data model and query language extend naturally to this setting, associating peer ids with activity pairs in execution traces/patterns. When an activity pair s, s' in a query is annotated by a peer id P, the search for its internal EX-pattern is restricted to traces supplied by P. The pattern matching algorithm presented in Section 4 extends naturally to this distributed setting. To avoid shipping events between sites, the (sub)automaton $\hat{A}$, corresponding to the internal EX-pattern of an activity pair s, s' annotated by P is installed on the peer P. When s is matched, it notifies the start node of $\hat{A}$; a matching for $\hat{A}$ is computed on P (as described in Section 4). On success, s' is informed (and is activated). If the matching fails, s in notified (and consequently backtracks).

Reports. We conclude by briefly considering report generation. Assume first that a report is attached to the `end` node of an EX-pattern p. A match (p_c, ψ) for a concrete EX-pattern $p_c \in concrete(p)$ can be viewed as an XML document (tree), that records the ψ assignment for the activities (activation-completion pairs) and the edges in p_c. Each match found by the algorithm of Section 4 generates one such XML entry. The *Report* command is applied to this stream of matches. The syntax and semantics resembles that of previous proposals for such reports [25, 13]; we only mention here the main constructs. By default a report is issued for each entry. To issue a report only when certain conditions are satisfied a *When* *cond* statement can be used, where *cond* is a boolean condition on the value of attributes or aggregate functions (described below). Periodic reports may be generated by the *Every* *time* command, where *time* may be a time interval or the number of entries generated since last report. A sliding window describing the entries relevant for the generation of the report can be defined using the *Range* *time* command. The structure of the report - an XML document - is described in a manner similar to that of the return clause of XQuery and may include grouping of entries and aggre-

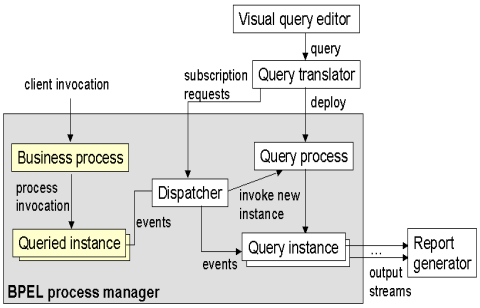

Figure 15: Architecture.

gations like average, max, min, count, sum. Two reporting modes are available: A *local* report is issued for a given process *instance* and uses only entries of that instance. A *global* report is issued per BP and uses entries of *all* the BP instances.

In general, report commands may be attached to any node in the pattern. The portion of the execution pattern relevant to such a report consists of the prefix of the pattern including the report node and its predecessors. The report generated for such a (sub) pattern ignores the rest of the pattern, and is processed in the same way as described above.

Remark: A naive, and very inefficient, approach to process a query with several report nodes is to compute, separately, the matches of each of their respective pattern prefixes. Recall however that our algorithm works in a greedy manner by matching pattern prefixes, then expanding them to matches for larger prefixes. This can naturally be exploited to factorize the common processing, computing matches for reports of "shorter" prefixes and then expending them, when possible, to the reports of larger prefixes.

6. IMPLEMENTATION AND EXPERIMENTS

The query language and algorithms presented above have been implemented and tested in the `BP-Mon` monitoring system. A demo of the system will be given in the upcoming SIGMOD [5]. To support flexible deployment, the system compiles `BP-Mon` queries into BPEL specifications. The specification $S(p)$ generated for a query pattern p describes a process (essentially the automaton described in the previous section) that will perform the monitoring task for p. $S(p)$ is then automatically compiled into an executable code to be run on the same BPEL application server as the monitored BP. The system architecture is depicted in Figure 15. We describe below the various components.

Visual editor. `BP-Mon` queries are written via a visual editor, in one of two modes: The user can draw the patterns from scratch, using a drag-and-drop items palette. Or, starting from a specification of a BP p, use a wizard to create queries to monitor p, as follows: The user marks the nodes of p that she wishes to include in the query. Then by one click a query draft is created, where non selected nodes are omitted and the selected nodes are connected with transitive edges that reflect their flow and zoom-in relationship in p. The user can then add conditions on node values, add report points, make final adjustment, and click a button to finish.

Query translator. The query translator compiles a query on p to a BPEL process - the *Query Process* (QP) in Fig. 15 – that implements the automaton of Section 4. Each state is implemented as a compound activity consisting of two components, one in charge of reading the incoming events, the other in charge of events processing and backtracking. The QP is deployed onto the BPEL server where the instances of p are executed. Several QPs, monitoring the same or different processes, may be deployed on a server.

Dispatcher. For each query, our system generates one QP instance per monitored BP instance. Processes and instances in BPEL servers have id's, and these are used by the *dispatcher* module to dispatch the BP instances events to the right QP instances. It subscribes to relevant events of the queried BPs when a query is deployed, and receives the relevant events generated by instances of these BPs (as described in Section 2). The first event from a new BP instance causes the dispatcher to create a new instance of relevant QPs. Further events are delegated to the running QP instances.

Report generation. The final step is generating the reports. As explained in Section 5, a successful matching for the query pattern associated with a report node generates an XML entry recording the embedding, and the *Report* command is applied to this stream of matches. Observe that from this point and on, since all the special BPEL-related issues have already been treated by the `BP-Mon` engine, we are back to standard XML stream processing, and can use a standard such engine to generate the report. In our implementation we support two alternatives for report generation. The first uses the streaming system of [25].[4] The second uses a lightweight in house reporting tool based on XQuery and XSLT. But in principle any XML streaming tool that supports the needed reporting features can be plugged into our architecture.

6.1 Experiments

The implementation by translation of queries into BPEL processes, then running them on the same server as the queried processes, has two main advantages: Portability of queries between BPEL engines; and a great simplification of the software devolvement, exploiting the infrastructure provided by such engines for parallel and distributed process management, and software composition. The price payed for this is the extra load on the BP server who now needs to also run query instances. To estimate the overhead incurred by running the query on the same server, the performance impact on the queried processes, the scalability of the solution, and the effectiveness of the optimizations, we ran several experiments.

We considered BPs with varying number of activities, where the monitoring involves different percentage of the activities in the BPs. We varied the ratio of processes vs. queries, and also varied the type of the monitored processes, from I/O bounded BPs, to CPU bounded ones.[5] Since the generation of reports is fairly standard, we focused on the parts specific to `BP-Mon`, i.e. the matching of patterns; our measurements do not include report generation time.

In the experiments, we used a family of processes consisting of sequences of nested *while* constructs, with atomic activities that each invokes a given Java class, some run in parallel, and with an optional *wait* activity between them. By configuring the number of *while* iterations and the properties of the Java class we could vary the size of the process, the characteristics of the activities (I/O or CPU bounded), and the percentage of queried process activities (our queries queried activities appearing only in some of the loops). The queries use the Report* option that requires matching of consecutive occurrences of the searched pattern in the EX-trace (as this requires more processing than a single Report). We measured execution time (in seconds) of processes and queries. The tests were performed on Pentium4 3.0GHz, dual core with 1GB RAM memory, running Windows XP Professional, JBoss AS 4.0.4. Oracle BPEL Process Manager 10.1.2. with Oracle 9i database.

A representative sample of results is shown in Figures 16 - 19. Figure 16 demonstrates the very minimal overhead of our solution

[4]This streaming system is actually relational, but the fairly simple structure of the `BP-Mon` XML allows for natural translation to relational format and back to XML.

[5]Since our experiments showed no significant difference between I/O bound and CPU bound BPs, the results discussed below use activities with a uniform mix of I/O and CPU load.

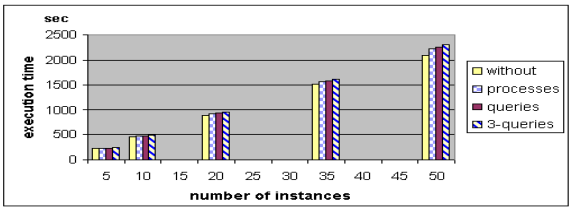

Figure 16: Queries overhead.

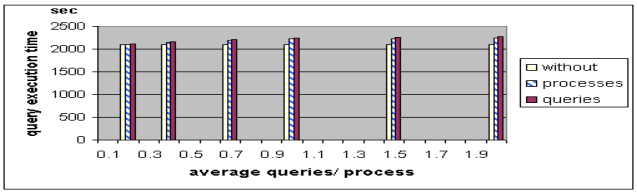

Figure 17: Varying number of queried processes.

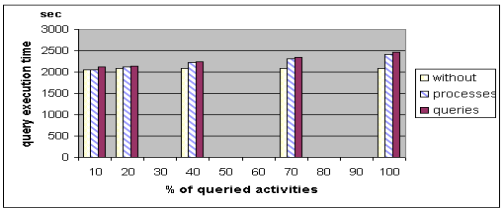

Figure 18: Varying number of queried activities.

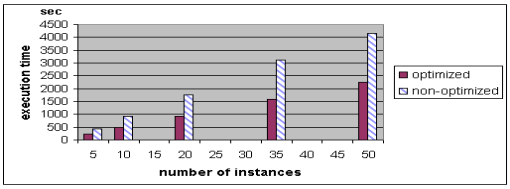

Figure 19: Impact of Optimization.

as well as its scalability. Each BP here consists of 200 activities; the monitored patterns involve about 40% of the activities. The graph shows, for a varying number of BPs, four measurements of total execution time for an entire workload. The first (left-most) column in each set shows the execution time of the BPs, with events generation, but without monitoring. The second column shows the execution time of the BPs when monitored, with one query per process. Clearly, the overhead on process execution due to monitoring is very low. The third column shows the execution time of the queries. As should be expected, their execution time is slightly higher than the processes themselves – a query is invoked *with* the process, but lags behind a bit when processing its events. (Recall that the queries here report consecutive occurrences of the searched pattern in the EX-trace, hence continue the monitoring till the process ends. Queries that report just one occurrence stop as soon at it is detected and thus entail even lesser overhead). Obviously, all the results are affected by the scalability of the BPEL server itself. We can see that the execution time grows linearly with the number of concurrent processes.

The queries that we show here have 3 reporting points. Recall that one of our optimizations is factorizing the common pattern matchings for the reports. To illustrate the reduction in processing time that this achieves, the forth column shows what would be execution time for the three matches if computed separately.

In the above experiment *all* process instances are monitored, each by *one* query. To measure the effect of changing these parameters, we varied the overall number of queries, assigning to each process a subset of random size, with uniform distribution. Figure 17 illustrates representative results, for 50 process instances with parameters the same as above (200 activities, of which 40% occur in the monitored patterns), and the average number of queries per process varying from 0 to 2. We can see that the growing number of queries has only minimal effect on execution time. Indeed as already seen in the previous experiment, the execution time is mostly affected by the running time of the monitored processes and the overhead due to query processing is marginal.

Figure 18 illustrates the effect of monitoring different percentage of the BP activities. We ran the experiment with the same 50 instances as above, and query patterns involving 10% to 100% of the BP activities. The execution time grows moderately with the percentage of monitored activities. In practice the common case is likely to be close to the lower left part of the curve, as typical BP specifications are large with only small part being relevant for a particular monitoring task.

We conclude by considering the effect of our optimization technique of pruning redundant monitoring based on an analysis of the BPEL process specification. Figure 19 illustrates the improvement achieved by applying this method. The scenario here is similar to what we have seen in Example 2.1: the BPEL specification has a *switch* construct, and only one branch is relevant to the query. The process instances choose randomly one of the branches. We measured the execution time of optimized and non-optimized queries, varying the number of process instances. The experiments shows a performance gain of almost 50%, reflecting the 50% of the processing, that involves non interesting branches, that was avoided. Of course, performance improvement in general will depend on the mix of processes, queries, and their properties.

7. CONCLUSION AND RELATED WORK

This paper presents `BP-Mon`, a novel query language and system for monitoring BPs. `BP-Mon` offers a high level intuitive design of monitoring tasks. A novel optimization technique exploits available knowledge on the BP structure to speed up computation. `BP-Mon` queries are compiled into standard BPEL processes, thus providing easy deployment, portability, and minimal overhead. We conclude by discussing some of the language design and implementation challenges with respect to related work.

Visual query languages. The design of `BP-Mon` was inspired by previous works on visual query languages for XML and graph-shaped data, such as XQBE [7] and Graphlog [10]. While all these languages consider only *flat* graphs, `BP-Mon` supports nested graphs and, correspondingly, enriches the standard path-based navigation with a novel (transitive) zoom-in, that allows to query process components at any depth of nesting. `BP-Mon`'s syntax resembles that of the BP-QL language [4] which we have developed for querying BP specifications. The two languages are complementary - BP-QL can be used to focus on parts of BPs that require monitoring, while `BP-Mon` is used to check at runtime properties that cannot be statically determined by querying the specification. Together they provide a uniform framework for static *and* dynamic BP analysis.

Composite events. BP monitoring entails the detection and processing of composite events. Event detection is at the core of several related application domains, such as active databases, publish-subscribe and production systems[32, 16, 31]. A variety of formalisms have been proposed for the specification of composite events, including event algebras, situation calculus, temporal languages, process algebra, transaction logic and computation tree logic (see [28] for an overview), allowing to define composite events based on the time stamps and casual dependencies of individual (or other composite) events. As explained in the Introduction, a key difference of the present work is the higher abstraction level employed here. Following the BPEL philosophy, `BP-Mon` users need not be aware of the underlying implementation details of the monitored

BP and the type of run-time events generated by the system. The specifications of monitoring tasks is performed on the same (high) level of abstraction as that of the BPs specification.

Runtime monitoring. A vast amount of work was performed on verification of concurrent and distributed systems using specification languages such as LTL. Recently, this approach has been applied to Web service composition, using the BPEL framework [21]. Runtime monitoring based on LTL, Statecharts, and related formalisms has also received a lot of attention recently (see [30] and [24]). These works are mainly focused on error detection, e.g. concurrency related bugs. Our approach is different in that it relies on the BPEL model for the visual language used to specify monitoring requests, and on execution on a BPEL engine for the monitoring itself. These points contribute to the ease of deployment and the efficiency of execution of our monitoring tool.

Optimization. A variety of methods have been proposed for optimized processing of (composite) events, employing relation and object-oriented database technology [32, 16], petri nets, finite state automata, event graphs, and storage minimization (See [29] for a survey) . These methods are generic, that is they can be employed in a variety of application domains. To our knowledge the present work is the first to propose a BP-specific optimization that exploits knowledge about the BP structure, and is complementary to the above works. The use of schema knowledge is an important XML query optimization technique [14, 12, 18]. In our case, the "schema" is the BP specification and the optimization goal is to reduce computation time and memory requirements for the BPEL processes implementing a query. The key differences from XML schema-based optimization are the two navigation axes considered here, the inherent higher complexity of BP specifications analysis and the need to resort to safe, rather than exact, analysis. Further optimization to be studied may include pattern simplifications, e.g. replacing non-transitive edges with transitive ones and reducing pattern nesting by eliminating unnecessary compound activities.

DFA vs. NFA. Many XML filtering engines are based on finite automata, either deterministic (DFA) or non-deterministic (NFA). Some works support path sharing, converting large numbers of XPath queries into a single NFA [12, 9]. Other are based on DFA [17, 18]. NFA-based approaches are space efficient, requiring a relatively small number of states to represent complex queries. DFA-based approaches are time efficient since their state transitions are deterministic, but the conversion from an NFA to a DFA increases the number of states exponentially. To avoid this exponential blow up, works like [17] compute the states lazily, at run-time. Following this principle we use a DFA, auto-generated as a BPEL process, which instantiates the required states (activities) as it progresses.

Memory requirements. Lower bounds on the space required for the evaluation of continuous select-project-join queries over relational streams are considered e.g. in [1, 2]. The challenges encountered in our work are similar, but the queries are inherently more complex. We are currently investigating syntactic restrictions on EX-patterns and BP specs to provide bounds for memory needs.

BP management. In the introduction we reviewed BAM (Business Activity Monitoring) systems. Software runtime analysis tools like Purify, Quantify and PureCoverage [19] closely follow the execution of applications and allow to create extensive reports about memory usage, memory leaks, memory and performance bottlenecks, and code coverage. Unlike `BP-Mon` these tools are very low level and are targeted at developers. Complementary to this line of work is the post-analysis of traces that were gathered and stored in databases [8]. We are currently examining the extension of `BP-Mon` with facilities for querying stored logs.

8. REFERENCES

[1] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. *ACM Trans. Database Syst.*, 29:162–194, 2004.

[2] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. Memory requirements of XPath over XML streams. In *PODS*, pages 177–188, 2004.

[3] BEA. Bea AquaLogic BPM suite. http://www.bea.com/bpm/.

[4] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes. In *VLDB*, pages 343–354, 2006.

[5] C. Beeri, A. Eyal, T. Milo, and A. Pilberg. Query-based monitoring of BPEL business processes (demo). In *SIGMOD*, 2007.

[6] Business Process Execution Language for Web Services, 2003. http://www.ibm.com/developerworks/library/ws-bpel/.

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

[8] M. Castellanos, N. Salazar, F. Casati, U. Dayal, and M. Shan. Predictive business operations management. In *DNIS*, 2005.

[9] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *ICDE*, 2002.

[10] M. Consens and A. Mendelzon. The G+/GraphLog visual query system. In *SIGMOD*, page 388, 1990.

[11] FileNet Corporation. Snap in BPM to your existing IT environment: Bruce silver associates industry trend report. http://www.knowledgestorm.com/sol_summary_80244.asp.

[12] Y. Diao and M. J. Franklin. Query processing for high-volume XML message brokering. In *VLDB*, pages 261–272, 2003.

[13] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[14] D. Florescu et al. The BEA streaming XQuery processor. *VLDB J.*, 13(3):294–315, 2004.

[15] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *WWW*, pages 621–630, 2004.

[16] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In *VLDB*, pages 327–338, 1992.

[17] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *ICDT*, pages 173–189, 2003.

[18] A. Kumar Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *SIGMOD*, pages 419–430, 2003.

[19] IBM software - Rational Purity. http://www-306.ibm.com/software/awdtools/purify/.

[20] ILOG JViews. http://www.ilog.com/products/jviews/.

[21] R. Kazhamiakin, M. Pistore, and L. Santuari. Analysis of communication models in web service compositions. In *WWW06*, 2006. http://www06.org.

[22] N. Koudas and D. Srivastava. Data stream query processing. In *ICDE*, page 1145, 2005.

[23] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison Wesley, 2002.

[24] Thierry Massart and Cédric Meuter. Efficient online monitoring of ltl properties for asynchronous distributed systems. Technical report, Université Libre de Bruxelles, 2006.

[25] R. Motwani et al. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

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

[27] Oracle. http://www.oracle.com/appserver/bpel_home.html.

[28] Adrian Paschke. The Reaction RuleML classification of the event /action/state processing and reasoning space. http://ibis.in.tum.de /research/ReactionRuleML/docs/ReactionRuleML_Classification.pdf.

[29] N. W. Paton and Oscar Díaz. Active database systems. *ACM Comput. Surv.*, 31(1):63–103, 1999.

[30] Runtime verification – workshops. http://www.runtime-verification.org/.

[31] D. M. Sayal, F. Casati, U. Dayal, and M. Shan. Business Process Cockpit. In *Proc. of VLDB*, pages 880–883, 2002.

[32] J. Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. In *SIGMOD*, pages 259–270, 1990.