The VLDB Journal (2012) 21:51–68
DOI 10.1007/s00778-011-0234-x

REGULAR PAPER

Type inference and type checking for queries over execution traces

Daniel Deutch · Tova Milo

Received: 20 August 2010 / Revised: 8 February 2011 / Accepted: 18 April 2011 / Published online: 5 May 2011
© Springer-Verlag 2011

Abstract We study here Type Inference and Type Checking for queries over the execution traces of Business Processes. We define formal models for such execution traces, allowing to capture various realistic scenarios of partial information about these traces. We then define corresponding notions of types, and the problems of type inference and type checking in this context. We further provide a comprehensive study of the decidability and complexity of these problems, in various cases, and suggest efficient algorithms where possible.

Keywords Business Processes · Type information · Execution traces

1 Introduction

A Business Process (BP) consists of some business activities undertaken by one or more organizations in pursuit of some particular goal. BP management systems allow to trace (log) the execution course of Business Processes; repositories of execution traces (abbr. EX-traces) obtained in such manner are extremely valuable for companies: their analysis allows to optimize business processes, reduce operational costs, and ultimately increase competitiveness [3]. Since the traces repository is typically very large, the analysis is often done in two steps: the repository is first queried to select portions of the traces that are of particular interest. Then, these serve as input for a finer analysis that further queries the sub-traces to derive critical business information [29].

Type information, i.e., knowledge about the possible structure of the queried (sub-) traces, is valuable for optimization of this querying process [4]. Its role is analogous to that of XML schema for XML query optimization; it allows to eliminate redundant computations and simplify query evaluation. In many cases, such type information is readily available for the original traces (via a specification of the Business Process itself); however, this information is not available for the intermediary sub-traces selected by the first-step queries employed in the above-mentioned analysis. This calls for *Type Inference*, i.e., algorithms that derive the type information of the (sub-)traces that qualify to a given query. Additionally, when the tool used for finer analysis expects data of particular type, one should verify that its input (i.e., the (sub-)traces selected by the first-step queries) conforms to this type. This calls for *Type Checking*.

We present in this paper a thorough study of Type Inference and Type Checking for queries over Execution Traces. Before describing our results, we briefly explain the model used here for Business Processes, their execution traces, and queries over them.

First, the model that we use here for modeling BP specifications is a natural abstraction of the BPEL standard (business process execution language [6]) for BP specifications. A BP is modeled as a nested DAG consisting of activities (nodes), and links (edges) between them, that detail the execution order of the activities. The DAG shape allows to describe parallel computations. Activities may be either atomic or compound. In the latter case, their possible internal structures (called implementations) are also detailed as DAGs, leading to the nested structure. A compound activity

This work has been partially funded by the Israel Science Foundation, by the US-Israel Binational Science Foundation, and by the EU grant MANCOOSI.

D. Deutch (✉)
Ben Gurion University, Beer-Sheva, Israel
e-mail: deutchd@cs.bgu.ac.il

T. Milo
Tel Aviv University, Tel Aviv, Israel
e-mail: milo@post.tau.ac.il

Table 1 Overview of complexity results (data complexity)

	Selective	Masked	Naive
Type Inference	PTIME	EXPTIME, NP-complete	Impossible
Type Checking	Undecidable	EXPTIME, NP-complete	EXPTIME, NP-complete

may have multiple possible implementations, intuitively corresponding to different user choices, variable values, server availability, etc., and exactly one of them will be chosen at run-time, for each activity instance.

An *execution trace* (abbr. EX-trace) of such a Business Process can then be abstractly viewed as a nested DAG that contains nodes representing the activation and completion events of activities and edges that describe their flow order. Tracing may vary in the amount of information that it records on the run. We distinguish three families of execution traces, with decreasing level of information: (1) *naive* traces that "naively" provide a complete and exact record of the activation/completion events of all activities (2) *masked* traces where the activation/completion events are all recorded, but possibly with only partial information ("mask") about their origin activity, and (3) *selective* traces where only a selected subset of the events is recorded (and those that are recorded may be masked).

Different tracing systems may be employed for recording the execution of a given BP specification. A tracing system consists of (1) a renaming function on activities names (whose role is to allow only partial information on the actual activities that took place, e.g., by mapping two distinct activities names to the same name), and (2) a set of activities names that are not recorded in the traces ("deletion set"). This allows to capture naive, masked or selective tracing, with different choices of which activities to disclose (in particular, for naive tracing, the renaming function is the identity function, and the deletion set is empty). A BP specification along with a description of such tracing system is called a *type*.

For example, consider a Business Process describing the operational logic of online travel agency, offering various travel deals, including suggestions of flights, hotels and car rentals. Assume also that the agency offers both luxury and ordinary (i.e., non- luxury) deals but uses different billing systems (represented by distinct billing activities) for the two kinds of deals (e.g., a stricter credit check is done for luxury reservations). An execution trace of the BP details a possible execution of the online travel agency, including offers presented to the user, the choices that the user makes, etc. The owners of the agency may wish to hide the fact that different billing systems are used for the two kinds of reservations, in which case masked tracing may be used, to give a generic name to the two different billing activities. It may further be the case that the treatment of luxury reservations involves the use of other activities that do not have a counterpart in the case of ordinary reservations (e.g., redirection to an external credit check), in which case to hide the different treatment, the additional activities should be omitted from record, using a *selective* tracing.

We then consider *queries* that are used to define EX-traces or parts of them, that are of interest to the analyst. Queries are defined using *execution patterns* (abbr. EX-patterns), generalizing EX-traces similarly to the way tree patterns, used in query languages for XML, generalize XML trees [8]. In more details, an EX-pattern has the structure of an EX-trace, where activities names are either specified, or left open using a special ANY symbol and then may match any node. Edges in a pattern are either regular, interpreted over edges, or transitive, interpreted over paths. Similarly, activities may be regular or *transitive*, for searching only in their direct internal flow or for searching in any nesting depth, respectively.

The role of *transitive edges* is analogous to the role of the "descendant of" operator in XPath [33], in the sense that when a transitive edges between two activities nodes appears in a pattern, a matching trace is required to include these two activities with some path between them (but the path may be of any length and the activities along this path may be any). For instance, in our travel agency BP, analysts may be interested in traces in which a choice of a hotel is followed by payment, but any sequence of activities may occur between these two activities. The need for *transitive nodes* follows from the nested structure of the execution traces (and correspondingly of the patterns); when the pattern describes an implementation of a non-transitive node, it means that the specified implementation must appear in the *direct* implementation of the activity. But when the corresponding node is transitive, then further nested implementations may participate in realizing the corresponding part of the trace. For instance, we may be interested in traces where the payment was associated with a specific credit card, e.g., Visa (and then the payment activity will occur in an implementation of a non-transitive activity corresponding to "Visa"). Alternatively, we may only care if the payment was part of some purchase activity (that invokes, in its implementation, some credit activity that in turn invokes the payment activity); in this case, we will use a transitive "Visa" activity node.

Our results. We study Type Inference and Type Checking for Queries over Execution Traces, for all classes of traces described above. Our results are summarized in Table 1. We show a PTIME (in the size of the input type, with the exponent determined by the query size) Type Inference algorithm when the types used are *selective*. The intuition behind the algorithm comes from the algorithm for intersecting context-free grammars (of which the Business Processes with selective tracing can be thought of as a generalization), with regular expressions (generalized by EX-patterns). However, the algorithm is much more intricate, due to the graph structure of traces (in contrast to the string structure in context-free grammars and regular expressions). For masked types, we show that in general no polynomial size BP with a masked tracing system can capture the type that is to be inferred, but we give an EXPTIME algorithm (whose result includes a BP of exponential size) for inferring the type. Intuitively, this blowup is due to the need to support exponentially many paths, which can not be done compactly with only masked tracing. Consequently, we provide an EXPTIME algorithm. Finally, we show that naive tracing is too weak to capture in general the inferred types, i.e., in general query results cannot be expressed by a type (BP specification) using only naive tracing. For Type Checking, we provide an EXPTIME algorithm for naive or masked trace types and show undecidability of type checking for selective trace types. The undecidability result is based on the observation that a BP with selective tracing is a powerful enough formalism to capture context-free grammars.

An extended abstract of this work had appeared in [12], presenting only a brief high level description of the main results. The present paper provides a comprehensive description of the various algorithms and the formal proofs of all provided theorems.

The paper is organized as follows. Section 2 introduces the definitions for our model and query language. Section 3 defines and details the results on Type Inference, and Sect. 4 defines and details the results on Type Checking. Related work is considered in Sect. 5. We conclude in Sect. 6.

2 Preliminaries

We provide in this section the formal definitions for our basic model of Execution Traces, Business Processes, and queries over such traces and processes. We also explain the notion of types used in this context and explain the roles that type information plays in query optimization. We accompany each definition with an intuitive example.

2.1 Execution traces

An execution trace can be viewed as a nested DAG, containing node-pairs that represent the activation and completion of activities, and edges that represent flow ordering. We assume the existence of two domains, $\mathcal{N}$ of nodes and $\mathcal{A}$ of activity names, and two distinguished symbols `act`, `com`, denoting, respectively, activity activation and completion. We first define the auxiliary notion of *activation-completion labeled DAGs* and then use it to define *execution traces*.

Definition 1 An *activities DAG* is a tuple (N, E, λ) in which $N \subset \mathcal{N}$ is a finite set of nodes, E is a set of directed edges with endpoints in N, and $\lambda : N \rightarrow \mathcal{A}$ is a labeling function, labeling each node by an activity name. The graph is required to be acyclic.

An *activation-completion* (act-comp) DAG g is obtained from an activities DAG by replacing each node n labeled by some label a by a pair of nodes, n', n'', labeled (respectively) by $(a, \texttt{act})$ and $(a, \texttt{com})$. Such pair of nodes is called an *activity pair*. All of the incoming edges of n are directed to n' and all of the outgoing edges of n now outgo n''. A single edge connects n' to n''.

We assume that g has a single `start` node without incoming edges, and a single `end` node without outgoing edges, denoted by $\texttt{start}(g)$ and $\texttt{end}(g)$, respectively.

Example 1 To illustrate, let us consider the act-comp DAG $S2$ in Fig. 1 (ignore for now the "bubbles"). It contains four activities (`Search`, `Hotel`, `Flight`, and `Print`), each represented by an activity pair. In each pair, the node with darker (lighter) background denotes the activity's activation (completion).

Definition 2 The set $\mathcal{EX}$ of *execution traces* (abbr. *EX-traces*) is the smallest set [1] of graphs satisfying the following.

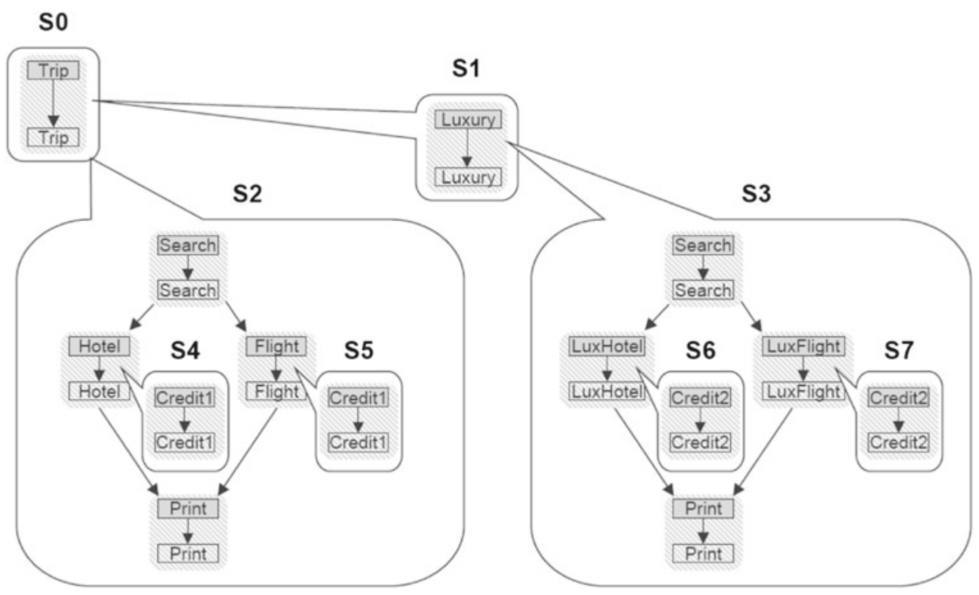
1. **[single activity]** If g is an act-comp DAG consisting of a single activity pair, then $g \in \mathcal{EX}$.
2. **[nested trace]** If g_1 is in $\mathcal{EX}$, (n_1, n_2) is an activity pair of g_1 s.t. n_1 (respectively, n_2) has a single outgoing (incoming) edge, and g_2 is some act-comp DAG, then the graph consisting of g_1, g_2, and two new edges $(n_1, \texttt{start}(g_2))$ and $(\texttt{end}(g_2), n_2)$ is in $\mathcal{EX}$.

The two new edges added in Item 2 above are called *zoom-in* edges. All other edges are called *flow* edges.

In the sequel, a subgraph g_2, connected as in Item 2 of Definition 2, by zoom-in edges, to an activity pair (n_1, n_2), along with the zoom-in edges, is called a *direct internal trace* of the activity. The subgraph consisting of the direct internal trace of an activity, as well as the direct internal traces of its activities, and of their internal activities, etc., is called a *(full) internal trace*.

[1] In the sense that every other set satisfying the constraints is a superset of this set.

Fig. 1 Business process



Example 2 Some example EX-traces are depicted in Fig. 2. Let us focus first on EX-traces (a). It details a possible execution of a travel agency activity `Trip` for reserving trips. Zoom-in edges are marked as dashed arrows, and following them reveals the internal trace of the corresponding compound activities. For each such compound activity, one zoom-in edge originates at its *activation* node and points at the `start` node of another (possibly nested) act-comp DAG g, and another outgoes the `end` node of g, pointing to the activity *completion* node. For example, zooming into `Trip` reveals that a `Search` was performed, after which the corresponding hotel and flight were reserved (in parallel), by the `Hotel` and `Flight` activities, respectively, and a confirmation was printed. Zooming into both reservation activities reveals some credit limit check, namely `Credit1`. The EX-trace in Fig. 2b is another possible EX-trace of the travel agency. Here, the user was looking for a luxury trip, invoking the `Luxury` activity. The internal flow here is similar to that of an ordinary trip reservation, except that luxury hotels and flights are reserved (via the `LuxHotel` and `LuxFlight` activities) and another type of credit check, `Credit2`, (possibly for higher credit limit check), is performed.

2.2 Types

To define EX-trace types, we use an intuitive model of BP specifications as *rewriting systems*, an abstraction of the BPEL standard [6]. A type consists of a BP specification accompanied by a description of the sort of tracing (to be defined in the sequel) employed for the BP. Intuitively, the BP specification and tracing system serve as the *schema* of the corresponding traces.

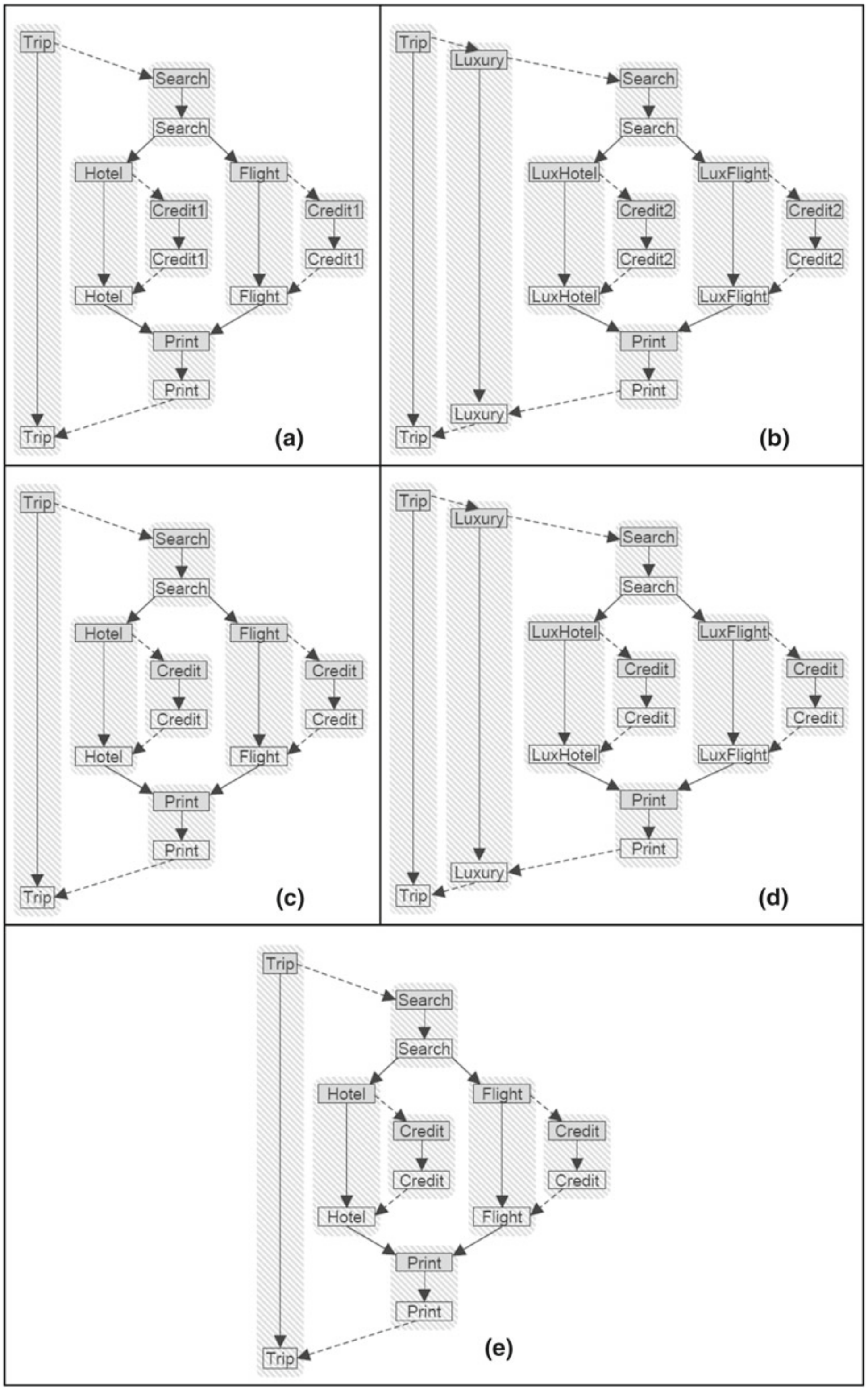
Among the activity names in $\mathcal{A}$, we distinguish two disjoint subsets $\mathcal{A} = \mathcal{A}_{atomic} \cup \mathcal{A}_{compound}$, representing atomic and compound activities, respectively. A BP specification is a collection of activation-completion DAGs, along with a mapping of compound activity names to their implementations.

Definition 3 A *BP specification s* is a triple (S, s_0, τ), where S is a finite set of act-comp DAGs , $s_0 \in S$ is a distinguished DAG consisting of a single activity pair, called the *root*, and $\tau : \mathcal{A}_{compound} \to 2^S$ is a function, called the *implementation function*, mapping compound activity names in S to sets of DAGs in S.

The intuition here is that activities that are not connected via a directed path are assumed to occur *in parallel* (similarly to the BPEL construct *Flow* [6]), while those that are connected via a directed path occur sequentially (corresponding to the use of the BPEL construct *Sequence*). An additional axis is the *implementation* axis: a compound activity may be mapped, through the implementation function, to a set of DAGs. They represent alternative possible implementations for the activity (one of which will be chosen at run-time as the actual implementation), allowing to capture BPEL *Choice* construct. Implementations may be recursive, thereby implementing *loops* (and in particular the BPEL *While* construct). Consequently (see [4]) our formalism is expressive enough to capture the control flow constructs supported in BPEL. A detailed study of the expressive power of our model, compared to other formalisms such as Context-Free (Graph) Grammars and (Recursive) State Machines, can be found in [11].

Example 3 The BP specification s of an online travel agency consists of the act-comp DAGs in Fig. 1. The compound activities here are `Trip`, `Luxury`, `Hotel`, `Flight`, `LuxHotel`, and `LuxFlight`. $S0$ is the root of the BP, and

Fig. 2 Execution traces



the possible implementations of each compound activity are given in bubbles.

The intuitive interpretation of this specification is as follows. When a user starts searching for a trip, she may choose between an ordinary trip (leading to $S2$), or a luxury trip (leading to $S1$, that in turn leads to $S3$). Then, she searches and reserves (luxury) flights and (luxury) hotels, as described above.

We consider three kinds of traces for a given BP specification. The first, called *naive* EX-trace, provides a complete record of the activation/completion events of all BP activities and the internal flow of all compound activities. Given a BP specification s, the set of possible naive EX-traces of s consists of all EX-traces obtained from the root activity of s by attaching, recursively, to each compound activity one of its possible implementations. We call this an *expansion*.

Definition 4 (*Naive EX-traces*) Given a BP specification $s = (S, s_0, \tau)$, an act-comp DAG g, and an activity pair (n_1, n_2) of g labeled by some compound activity name a and having no internal trace, we say that $g \to g'$ (w.r.t. τ) if g' is obtained from g by attaching to the pair some implementation $g_a \in \tau(a)$ through two new zoom-in edges $(n_1, \mathtt{start}(g_a))$ and $(\mathtt{end}(g_a), n_2)$.

If $p \to p_1 \to p_2 \cdots \to p_k$, we say that p_k is an *expansion* of p. p_k is called a *full expansion* if it cannot be expanded further. The set of possible *naive EX-traces* defined by a BP specification s, denoted $Naive(s)$, consists of all the full expansions of s_0.

Intuitively, naive EX-traces of a Business Process contain the full information regarding the process execution. For example, the two EX-traces in Fig. 2a, b are naive EX-traces of the travel agency BP depicted in Fig. 1. They contain a full record of both compound and atomic activities that participated in the execution, as well as the flow and zoom-in edges connecting them. In general, $Naive(s)$ may be infinite, in the case of recursive activity implementations.

While naive EX-traces detail the execution flow fully, there are cases where only partial information about the activities is recorded. For example, for confidentiality reasons, our travel agency may wish not to disclose the fact that the billing system that is invoked in the case of a luxury reservation is different than the one invoked in case of an ordinary reservation.

In terms of the examples above, rather than labeling the EX-trace activity nodes by the exact activity names `Credit1` and `Credit2`, we would like to label them by a generic `Credit` label. This is captured by the following definition of *masked* EX-traces.

Definition 5 (*Masked EX-traces*) Given a BP specification s and a renaming function π from activity names in s to activity names in $\mathcal{A}$, the set of *masked* EX-traces defined by s and π, denoted $Masked(s, \pi)$, consists of all the EX-traces e obtained from the naive EX-traces $e' \in Naive(s)$ by replacing each label a in e' by $\pi(a)$.

Example 4 To continue with our running example, Fig. 2c, d are masked EX-traces of our travel agency BP, for a renaming function π s.t. $\pi(\mathtt{Credit1}) = \pi(\mathtt{Credit2}) = \mathtt{Credit}$, and where π is the identity function for all other activities. These two masked EX-traces are obtained, respectively, from the naive EX-traces in Fig. 2a, b. The trace reveals that some credit checks were issued, but not which ones.

For readers familiar with XML schemas, masked EX-traces can be viewed as the BP analog of XML trees defined by *DTDs with specialization* [28]. In both cases, the nodes' labels give only partial information about the origin of the node (the corresponding BP activity, for EX-traces, or the DTD type, for XML trees).

In some cases, an even more selective tracing is desired, where the occurrence of some activities is not recorded at all. For instance if some activities are completely confidential, we may want to avoid including any memory of them in the trace. Similarly, some activities (e.g., standard input integrity checks) may simply be uninteresting for the business analysts and may be omitted to avoid overloading the logs with redundant information. To model these type of traces, we introduce the notion *selective* EX-traces.

Definition 6 (*Selective EX-traces*) Given a BP specification s, a set A of activity names in s, satisfying condition (*) below, and a renaming function π from activity names in s to activity names in $\mathcal{A}$, the set of *selective* EX-traces defined by s, A, and π, denoted $Selective(s, A, \pi)$, consists of all EX-traces e obtained from the naive EX-traces of s by deleting all activity pairs with labels in A [2] and then replacing each label a of the remaining nodes by $\pi(a)$.

Condition (*): A does not include the root activity of s, and for each activation-completion graph g in s, the graph g' obtained from g by removing the atomic activity pairs with names in A is itself an activation-completion graph (as in Definition 1), or is empty.

The intuition behind condition (*) in Definition 6 is that from a practical point of view, it is reasonable to assume that the graph obtained after each loss of information still bears the shape of a trace, otherwise the loss is easily observable. For instance, removing also the *Search* activity pair will result in a graph in which *two* different zoom-in edges are going out of `Trip`, pointing at two different nodes. This contradicts the definition of an EX-trace. Condition (*) assures that this does not happen.

Example 5 If our travel agency also wishes to keep as a secret the fact that reservations of different types are treated differently, then not only the credit checks need to be

[2] When an atomic activity pair (n_1, n_2) is deleted, the edges incoming to n_1 are now connected to the nodes previously pointed by n_2. For compound activities, the incoming(outgoing) edges of n_1 (n_2) are now being connected to the start/end nodes of the implementation sub-graph.

relabeled, but also the LuxHotel and the LuxFlight, and the record of the Luxury activity should be omitted altogether. Here, $A = \{\texttt{Luxury}\}$, and $\pi(\texttt{Credit1}) = \pi(\texttt{Credit2}) = \texttt{Credit}$, $\pi(\texttt{LuxHotel}) = \texttt{Hotel}$, and $\pi(\texttt{LuxFlight}) = \texttt{Flight}$.

Figure 2e shows the selective EX-trace obtained from the naive EX-traces in Fig. 2b. Note that the same graph (up to node isomorphism) is also the selective EX-trace obtained from the naive EX-traces in Fig. 2a. Thus, given such a selective trace, there is no way to tell from which naive EX-trace (of ordinary or luxury trip reservation) it originated, and the goal of secrecy is achieved.

We denote Naive= $\{E \mid \exists s\ s.t.\ E = Naive(s)\}$, Masked= $\{E \mid \exists s, \pi,\ s.t.\ E = Masked(s, \pi)\}$, Selective= $\{E \mid \exists s, \pi, A,\ s.t.\ E = Selective(s, A, \pi)\}$. We next show that there is a strict inclusion relationship between the classes, and that they do not capture all the possible sets of EX-traces.

Theorem 1 Naive$\subset$Masked$\subset$Selective$\subset 2^{\mathcal{EX}}$.

Proof The inclusion follows naturally from the definitions. We prove next its strictness, using examples that highlight some of the key properties of the various trace classes. These properties will be useful in the sequel.

Naive $\subset$ Masked Consider the BP specification s depicted in Fig. 3a (ignore, for now, the text next to the nodes). The depicted BP includes Trip as its root activity, whose implementation is a sequence of two compound activities labeled Hotel1 and Hotel2. The implementation of Hotel1 (respectively, Hotel2) contains the single atomic activity, Credit1 (Credit2). Now, consider a renaming function π that maps both Hotel1 and Hotel2 to a single activity name, Hotel, and is the identity function for the remaining activity names. Here, the set of masked traces $E = Masked(s, \pi)$ of s contains the single EX-trace e, depicted in Fig. 3b. e contains two Hotel-labeled activities, with the implementation of the first (second) being Credit1 (Credit2).

It is easy to see, however, that no BP specification s' can have e as its *single* naive EX-trace! This is because for e to be in $Naive(s')$, Hotel must have at least two alternative implementations in s', one containing Credit1 and the other containing Credit2. But if this is the case, $Naive(s')$ also contains three additional EX-traces: one where both Hotel occurrences have Credit1 as internal traces, one where they both have Credit2, and one where the first has Credit2 and the second Credit1.

Masked $\subset$ selective Consider the class of BPs where the implementation graphs of all compound activities are chains. For each such BP s, consider the selective traces obtained by deleting all compound activities and keeping the names of the atomic and root activities unchanged. That is, the set A of deleted activities consists of all compound activities appearing in s, and π is the identify function. The resulting EX-traces consist of a root activity whose direct internal trace is a sequence of atomic activities. Viewing the sequence of activity names that appear in the chain as a word, and the set of words represented by the selective traces of a given BP as a language, it is easy to see that for each BP s, $Selective(s, A, \pi)$ defines a *context free-language*. (Its context-free grammar follows naturally from the BP specification). Conversely, for every context-free language L, we can define a BP s (and A, π) s.t. the words in $Selective(s, A, \pi)$ are precisely those of L. (Here again, s's specification resembles L's grammar). In contrast, for every BP s, its naive (or masked) EX-traces that contain only the root activity and atomic activities have a shape that is specified by the direct implementation of the root activity, hence are of bounded length and cannot capture all context-free (or infinite) languages.

selective$\subset 2^{\mathcal{EX}}$ Since $\mathcal{EX}$ is infinite, $2^{\mathcal{EX}}$ is uncountable. In contrast, there are only countably many BPs with selective tracing systems, and there can be no more languages than the number of such encodings, thus only countable many

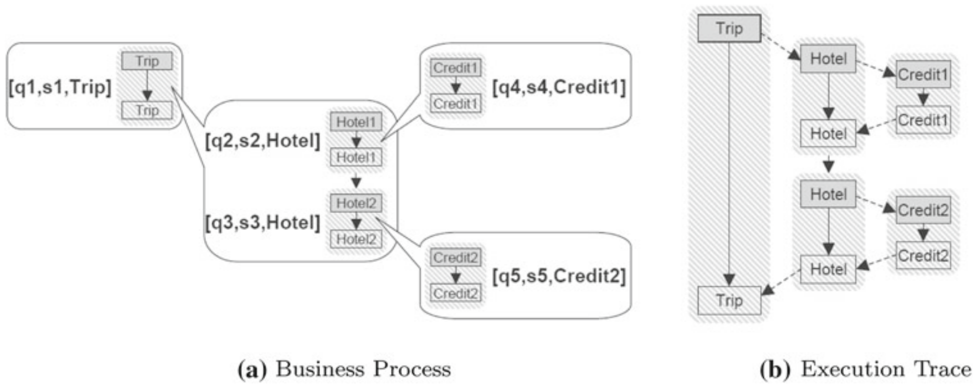


Fig. 3 A Business Process and an Execution Trace

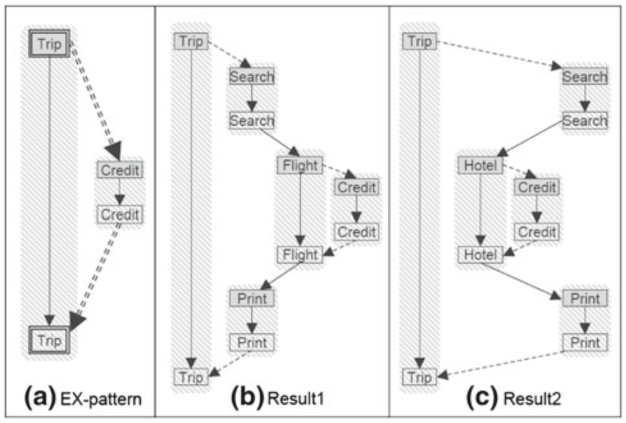


Fig. 4 A query and its results

languages may be captured by BPs with selective tracing systems. □

2.3 Queries

We now consider queries. As mentioned in the Introduction, the query language that we use was originally introduced in [3,4]. Queries are defined using *execution patterns* (abbr. EX-patterns). EX-patterns generalize EX-traces similarly to the way tree patterns generalize XML trees. EX-patterns are EX-traces where activity names are either specified, or left open using a special ANY symbol. Edges in a pattern are either regular, 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 zooming-in transitively inside it. We next give the syntactic definition of an EX-pattern, then define its semantics (including in particular the role of transitive activities and edges).

Definition 7 An *execution pattern*, abbr. EX-pattern, is a pair $p = (\hat{e}, TN, TE)$ where $\hat{e}$ is an EX-trace whose nodes are labeled by labels from $\mathcal{A} \cup \{\text{ANY}\}$, and TN (TE) is a distinguished set of activity pairs (respectively, edges) in $\hat{e}$, called *transitive* activities and edges, respectively.

Example 6 An example EX-pattern is depicted in Fig. 4a. The pattern seeks for all possible EX-traces where the user may make a reservation of a trip and finalize the reservation by performing the credit check. The query looks, visually, very similar to an EX-trace: it has compound activities (such as `Trip`) to which an implementation is attached through implementation edges, and atomic activities (such as `Credit`). Two distinctions are apparent in the figure: first, the `Trip` activity node is *doubly bounded*, signaling that it is *transitive*. Intuitively, this means that we wish to seek for the occurrence of `Credit` in its *indirect* implementation (i.e., in the implementation of `Trip`, or in an implementation of one of the compound nodes appearing in its implementation, etc.). Another difference is that the implementation edge connecting `Trip` and `Credit` is also transitive. This means that it may match any path of activity nodes appearing, in the EX-trace of interest, between the `Trip` and the `Credit` nodes.

If the `Trip` activity node would have been non-transitive then the semantics would imply that the `Credit` activity must appear in its direct implementation (and not in an implementation of activities within its implementation and so on). Similarly, if the edge from `Trip` to `Credit` would have been non-transitive then it would imply that there must appear a direct edge connecting `Trip` and `Credit` (in this particular case, since this edge is also an implementation edge, this would mean that `Credit` must be the root of the direct implementation of `Trip`). In general, transitive edges are useful also as non-implementation edges, and then they simply imply the existence of some path between the two nodes.

To evaluate a query, the EX-pattern is matched against a given EX-trace. A match is represented by an *embedding*.

Definition 8 (*Embedding*) Let $p = (\hat{e}, TN, TE)$ be an 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. **[root]** the root of p is mapped to the root of e.
2. **[nodes]** activity pairs in p are mapped to activity pairs in e, preserving node labels and formulas; a node labeled by ANY may be mapped to nodes with any activity name. For non-transitive compound activity pairs in p, nodes in their direct implementation are mapped to nodes in the direct implementation of the corresponding activity pair in e.
3. **[edges]** each (transitive) edge from node m to n in p is mapped to an edge (path) from $\psi(m)$ to $\psi(n)$ in e.

The result defined by ψ is the image of p in e under ψ. I.e., it is a sub-graph of e induced by restricting to nodes and edges in the range of ψ.

For an EX-pattern p and an EX-trace e, the result of p when applied to e, denoted $p(e)$, is the set of all results of all possible embeddings of p into e. Finally, given a set E of EX-traces, we will also use $p(E)$ to denote the set of all possible outputs of p when applied on EX-traces in E, namely $p(E) = \bigcup_{e \in E} p(e)$.

Example 7 For example, let us now match the EX-pattern in Fig. 4a to the EX-trace in Fig. 2c. Two embeddings are possible here, yielding the results in Fig. 4b, c. In the first embedding, the pattern transitive edge outgoing (incoming) the `Trip` activity is matched to the EX-pattern path passing through the `Flight` activity. In the second embedding, it is matched to the path traversing the `Hotel` activity. It is important to note that the same query (EX-pattern), but with a *non-transitive* `Trip` activity, would yield here an empty result, as it would search for `Credit` activities in the *direct* internal trace (implementation) of `Trip` (while in the given EX-trace it appears only deeper in the implementation hierarchy). Similarly, if the edge connecting `Trip` and `Credit` was non-transitive, again the empty result would have been obtained, as this revised query seeks for a direct *edge* connecting `Trip` and `Credit`, while in the given EX-trace they are connected via a path of length greater than 1.

Finally, since an embedding is a *homomorphism*, multiple query nodes may be mapped to the same EX-trace nodes and edges (or paths).

3 Type inference

We next define the *type inference* problem that we study here. Three variants of the problem correspond to the three families of trace types, as follows. With naive tracing, we are given an EX-pattern p and a set of naive EX-traces defined by some BP s and would like to find a BP specification s' s.t. $Naive(s') = p(Naive(s))$.[3] With masked tracing, we are similarly given a set of masked EX-traces defined by a BP specification s *and a renaming function* π, and wish to infer s', π' such that $Masked(s', \pi') = p(Masked(s, \pi))$. Finally, in the selective tracing setting, we are further given a deletion set A, and wish to infer s', A', π' such that $Selective(s', A', \pi') = p(Selective(s, A, \pi))$.

Section organization. Recall Table 1, indicating that to obtain a Type Inference algorithm, we must allow the *output type* to use (at least) masked tracing. Indeed, we start by suggesting an algorithm that uses masked tracing for its generated output type (and allows masked tracing for its input type); in the worst case, if the query contains transitive edges, then the complexity of the algorithm may be EXPTIME with respect to the size of the input type. Then, we modify the algorithm into a PTIME (with respect to the input type, with the exponent determined by the query size) algorithm, allowing the output type to use selective tracing.

Last, we provide lower bounds. We start by showing that the use of naive tracing is not expressive enough to allow capturing inferred types in general (even when the input type uses naive tracing as well); then we show that when using masked tracing for the output type, the exponential blowup w.r.t. the input type is unavoidable for queries with transitive edges; last, we show that the exponential dependency on the *query size* is unavoidable unless P = NP even when using selective tracing in the input type.

3.1 Basic (EXPTIME) algorithm

We next explain Algorithm `MASKED-TYPE-INFERENCE`, depicted in Algorithm 1. The algorithm assumes that no transitive nodes exist in the pattern (but transitive *edges* may exist); we explain below the extension of the algorithm to handle transitive nodes.

Algorithm 1: `MASKED-TYPE-INFERENCE (no transitive nodes)`

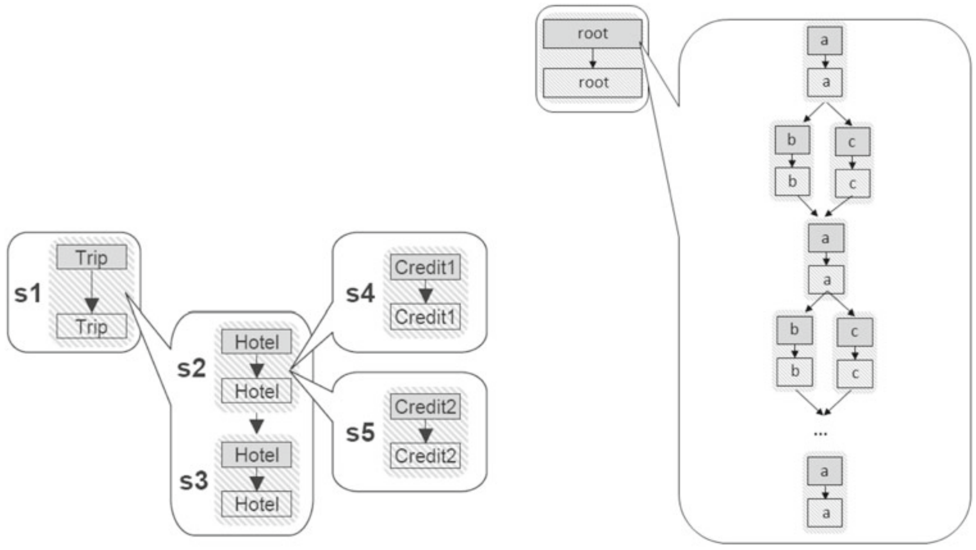
```
   Input: Input type $s, \pi$; EX-pattern $p$;
   Output: Output type $s', \pi'$
1  foreach activity pair $n_s$ in $s$ do
2      foreach activity pair $n_p$ in $p$ with label equal to that of $n_s$, or Any do
3          $a \leftarrow \pi(\lambda(n_s))$ ;
4          Add to $s'$ the activity name $[n_p, n_s, a]$ ;
5          Define $\pi'([n_p, n_s, a]) = a$ ;
6          if $n_s$ is compound then
7              $PatternImp \leftarrow$ the direct internal trace of $n_p$ ;
8              foreach direct implementation $Imp$ of $a$ in $s$ do
9                  $embs \leftarrow ComputeEmbeddings(PatternImp, Imp)$ ;
10                 Add $embs$ as implementations of $[n_p, n_s, a]$ ;
11             end
12             if No embeddings were found then
13                 Mark $[n_p, n_s, a]$ as failure;
14             end
15         end
16     end
17 end
18 $RemoveFailureActivities(s')$ ;
```

The algorithm input is a BP specification s along with a renaming function (i.e., a masked tracing system) π, and an EX-pattern p. Its output is a BP specification s' along with a renaming function π'; s', π' intuitively capture the "intersection" of the set of EX-traces of the original BP s, with the set of EX-traces defined by the pattern p. We next explain how s', π' are constructed by the algorithm.

For every two activity pairs $n_s \in s$ and $n_p \in p$ where n_p is labeled either by the same (compound) activity name as n_s, or by ANY, we use a new activity name $[n_p, n_s, a]$, where $a = \pi(\lambda(n_s))$, to represent the "intersection" of activity pairs (lines 1–5 in Algorithm 1). For compound (non-transitive) activities, the possible implementations in s' (computed in lines 7–10 of the algorithm) of $[n_p, n_s, a]$ consist of a DAG for each possible embedding of the direct internal trace of n_p in p into the possible direct implementations of n_s in s. These embeddings are computed by ComputeEmbeddings, described in the sequel. If no embeddings were found, $[n_p, n_s, a]$ is marked as a failure

[3] For EX-trace equality, we use graph isomorphism up to node identifiers. Equality of sets of EX-traces is defined w.r.t. this equality relation.

Fig. 5 BP specifications



(a) BP Specification (1) **(b)** BP specification (2)

(lines 12–13). As a final step of the algorithm, we perform "garbage collection", by invoking (line 18) the procedure *RemoveFailureActivities* that recursively marks as failure, activities for which all possible implementations contain failure activities, and then removes from s' all implementations that contain such failure activities.

ComputeEmbeddings. We next explain the operation of Algorithm ComputeEmbeddings, used in Line 9 of the type inference algorithm. When the EX-pattern contains no transitive edges (recall that we also assume no transitive nodes), then the embeddings may be found using conventional algorithms for subgraph homomorphism [26]. For each such homomorphism h that we find, the resulting embedding has the shape of the pattern, but each activity pair n_p is assigned an activity name $[n_p, h(n_p), \lambda(h(n_p))]$. When the query pattern contains transitive edges, then the algorithm also defines a new activity name for every transitive edge $e_p \in p$ and activity $n_s \in s$, and maps each transitive edge $e_p \in p$ (that connects two pattern nodes) to a path in the BP (connecting the two corresponding BP nodes).[4] *Each such matching path is represented in a different embedding*. In the resulting graph, a BP node n_s (with label a) that appear on such path is labeled by the triplet $[e_p, n_s, a]$.

Handling transitive nodes. Recall that we have assumed above that the pattern p contains no transitive nodes. With transitive nodes in the pattern, the algorithm becomes somewhat more complex. For a transitive node n_p appearing in a pattern graph G_p, part of the direct internal trace of n_p can be matched with the direct implementation of n_s, while other parts may be matched at deeper levels of the implementation. To account for that, the algorithm considers all possible *splits* of the (internal traces of activities in the) implementation of n_p. For a pattern graph G'_p, $p_0, p_1, \ldots, p_m$ is a split (of size m) of G'_p if each p_i is a sub-graph of G'_p and every node or edge of G'_p appears in p_i. Now, denote as a the label of the specification node n, in which a given transitive node was embedded. Then, for each *direct implementation* G of a, containing m nodes $n_1, \ldots, n_m$ that are labeled with compound activities, we try all possible splits $p_0, p_1, \ldots, p_m$, of size m, of the pattern p. We generate a new implementation graph for a, bearing the same shape as G, but where each node label a_i of n_i is replaced by $[p_i, n_i, a_i]$. Intuitively, implementations of $[p_i, n_i, a_i]$ are "committed" to include an embedding of p_i. We also test for the existence of embedding of p_0 in G. If such an embedding does not exist, we mark the activity as failure. Otherwise, we recursively continue testing for existence of an embedding of p_i in all possible implementations of a_i. We repeat the process for all possible splits. Finally, the last step of "garbage collection" removes all activities marked as failure, as described above.

Example 8 Consider the BP s from Fig. 5a with π being the identity mapping, and the EX-pattern p of Fig. 6. The annotations next to the query and BP activity pairs represent their identifiers. The BP s' constructed by the algorithm is depicted in Fig. 3a. Its activity names are of the form $[q_i, s_j, a]$, where q_i (s_j) is the identifier of a pattern (specification) activity node (ignore now the labels appearing within the nodes; the new activities names appear *next* to the nodes). The renaming function π' maps $[q_i, s_j, a]$ to a. Note that s', π' define a single masked EX-trace, of the shape depicted in Fig. 3b, which is indeed the only answer of the query p when applied on the naive (or masked) traces of s.

[4] Note that the number of such possible paths is possibly exponential in the BP size.

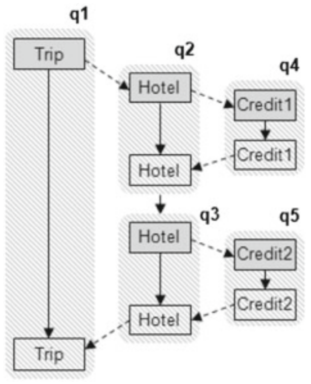


Fig. 6 Query

We next explain the manner in which s' is constructed by the algorithm. The construction of s' begins by matching the query root $q1$ to the specification root $s1$, and forming a new activity name $[q1, s1, Trip]$. Then, the implementation of $q1$ is matched against the single possible implementation of $s1$, thus embedding $q2$ ($q3$) in $s2$ (respectively, $s3$), forming a new activity name $[q2, s2, Hotel]$ ($[q3, s3, Hotel]$). This is a main point of the algorithm: a *unique* activity name `Hotel` that labeled two nodes of the original BP specification s (the nodes identified by $s2$ and $s3$), yielded two *distinct* activity names in s'. Consequently, each of these two activities names can now have distinct implementations that comply with the (different) conditions that the query imposes on their structure. Indeed, we proceed by embedding the implementation of $q2$ in the possible implementations of $Hotel$. There exist two such possible implementations, one of which contains `Credit1` and the second contains `Credit2`. In the latter, there is no embedding of the implementation of $q2$; in the former, there exists an embedding, yielding a node labeled $[q4, s4, Credit1]$. Similarly, we construct the only implementation of $[q3, s3, Hotel]$, containing $[q5, s5, Credit2]$.

Complexity. The time complexity of the algorithm depends on the number of possible sub-patterns that should be considered, for handling transitive nodes (exponential in the size of the EX-pattern p) and the number of possible embeddings of these sub-patterns into the activation-completion DAGs in s. Note that while, for each sub-pattern, the number of possible embeddings for *nodes* is polynomial in the size of the DAGs (with the exponent determined by the size of the sub-pattern), the number of possible embeddings of *transitive edges* may be exponential in the size of the DAGs in s: transitive edges are mapped to *paths* and the number of paths in a DAG may be exponential. The following theorem holds:

Theorem 2 *The complexity of Algorithm* `MASKED-TYPE-INFERENCE` *is exponential in the size of the input BP specification and in the size of the query.*

3.2 Improved (PTIME) algorithm using selective tracing

We have shown above an EXPTIME Type Inference Algorithm for masked tracing. Evidently, when selective tracing may be used in the output type, we can obtain a PTIME algorithm (with respect to the input BP specification size). A first easy step is to extend the algorithm to support selective tracing in the *input type*: let (s, A, π) be the input type, then we may construct an output type (s', A', π'), where s', π' are constructed exactly as in algorithm `MASKED-TYPE-INFERENCE`, and A' is such that $[n_p, n_s, a] \in A'$ if $a \in A$. But this algorithm still incurs EXPTIME. Interestingly, when the *output type* is allowed to use selective tracing systems, we are able to improve the algorithm and achieve an algorithm with PTIME data complexity.

The improved type inference algorithm, namely `SELECTIVE-TYPE-INFERENCE`, is based on the following observation. Consider the manner in which transitive edges are handled by $ComputeEmbeddings$, used in Algorithm `MASKED-TYPE-INFERENCE`. It treats each embedding of the sub-patterns of p into the BP s individually, contributing one graph to the output BP specification s'. For a transitive edge, this means that each of the paths between the nodes matched to its endpoints is treated separately, hence the exponential blowup. To avoid this, we use the added expressive power of selective traces. For every two specification nodes n_1, n_2 to which the end-nodes of a transitive edge (in the query) are mapped, consider all paths in-between n_1 and n_2. For each such path, we view the sequence of activity names in the path as a *string*, and the set of strings obtained from all such paths as a string language L. As we show below, we can define a regular string grammar G describing L. Then, we use G to define a BP s'', along with A'' and π'', s.t. the graphs of $Selective(s'', A'', \pi'')$ are paths, and the word constructed by following the labels along them are precisely those of L. Finally, s'' is then "plugged" into s'. In the sequel, we explain in details these three modifications.

Generating the string grammar G. Algorithm 2 is given as input an implementation graph in a given BP specification s (initially consisting only of the specification root s_0), and two nodes n_1, n_2 and generates a string grammar capturing all paths between n_1 and n_2. The grammar non-terminals correspond to the different nodes (activities pairs) of s while terminals correspond to activities names.

The regular grammar is constructed bottom up, starting with the current node n_{curr} being n_2 and generating a single derivation rule $N_2 \to \lambda(n_2)$ (lines 6–8). For each atomic activity pair n_{prev} such that there exists an edge from n_{prev} to n_{curr} in the specification graph (lines 9–13), we design a rule $N_{prev} \to \lambda(n_{prev})N_{curr}$, and continue recursively (line 13). For a non-terminal N_{prev} corresponding to a *compound activity* node labeled by A with implementations $s_1, \ldots, s_k$ (lines 16–20), we design new derivation rules $N \to start(s_i)$

Algorithm 2: GENERATE-STRING-GRAMMAR

```
   Input: An implementation graph $s$ in a BP specification, activity
          pairs $n_1, n_2$
   Output: A string grammar representing all paths from $n_1$ to $n_2$
1  Generate a grammar $G$ with a non-terminal for every activity pair
   in $s$, and a terminal for every activity name in $s$ ;
2  if $n_1 = n_2$ then
3      Set the initial non-terminal of $G$ to be the non-terminal
       corresponding to $n_1$ ;
4      return ;
5  end
6  Add to $G$ a derivation rule $n_2 \rightarrow \lambda(n_2)$ ;
7  $n_{curr} \leftarrow n_2$ ;
8  $N_{curr} \leftarrow$ the non-terminal corresponding to $n_{curr}$ ;
9  foreach edge $(n_{prev}, n_{curr})$ do
10     if $n_{prev}$ is atomic then
11         $N_{prev} \leftarrow$ the non-terminal corresponding to $n_{prev}$ ;
12         Add to $G$ a derivation rule $N_{prev} \rightarrow \lambda(N_{prev})N_{curr}$;
13         GENERATE-STRING-GRAMMAR$(s, n_1, n_{prev})$ ;
14     end
15     else
16         foreach implementation $s_i$ of $n_{prev}$ do
17             Add to $G$ the derivation rules $N_{prev} \rightarrow \lambda(start(s_i))$
               and $end(s_i) \rightarrow \lambda(end(s_i))N_{curr}$;
18             if a grammar for $s_i$ was not already generated then
19                 GENERATE-STRING-GRAM-
                   MAR$(s_i, start(s_i), end(s_i))$
                   ;
20             end
21         end
22     end
23 end
```

for $i = 1, \ldots, k$ and $end(s_i) \rightarrow \lambda(end(s_i))N_{curr}$, where $start(s_i)$ ($end(s_i)$) is the non-terminal corresponding to the start (end) node of s_i. We then recursively generate a grammar for all paths in-between $start(s_i)$ and $end(s_i)$ (if such grammar was not generated yet). The process terminates when we reach n_1 (where N_1 is its corresponding non-terminal), and we set the grammar root to be N_1, the non-terminal corresponding to n_1 (lines 1–3).

Recall that each non-terminal in G corresponded to a node in s. We use $real(G)$ to denote the set of *non-terminals* that originated from *compound* activity pairs, and $temp(G)$ for all other non-terminals (those generated for atomic activity pairs).

Transforming G into s''. We next explain how to generate a BP s'' along with a selective tracing system A'' and π'', such that each graph in $Selective(s'', A'', \pi'')$ is in fact a path whose labels sequence is a word in the language defined by G. The compound activities of s'' are simply the non-terminals of G. The implementation function follows the derivation rules of G, but the string appearing in the right-hand side of each such derivation rule is replaced by a chain graph where each label of the string is represented by a node bearing the same label as its activity name, and edges connect nodes standing for subsequent such labels. Finally, A'' consists of all activities names corresponding to non-terminals in $temp(G)$. $\pi'' = \pi$ (the renaming function used for the input BP specification).

Plugging s'' into s'. Whenever the algorithm reaches the origin of a transitive edge $N_1 \rightarrow N_2$, we simply add a new compound activity $r_{s''}$ that is the root of s'' (constructed as explained above). Also, we add to the deletion set A' of s', all activity names in A'', and we continue the algorithm operation.

Note that the exponential blow-up incurred in Algorithm MASKED-TYPE-INFERENCE is now avoided, via the use of a string grammar. The following theorem follows:

Theorem 3 *The complexity of Algorithm* SELECTIVE-TYPE-INFERENCE *is $O(\mid s \mid^{|q|})$. where $\mid s \mid$ is the size of the input BP specification, and $\mid q \mid$ is the query size.*

3.3 Lower bounds

We have suggested above an EXPTIME Type Inference algorithm when using masked tracing. We next show that this is the best that can be achieved.

Theorem 4 *There exists an infinite set S of BPs of increasing sizes, and an EX-pattern p (with only three activity pairs), s.t. for each BP $s \in \mathcal{S}$, every s' s.t. $Masked(s', \pi') = p(Naive(s))$ for some π' is of size $\Omega(2^{|s|})$.*

Proof The BPs in the class $\mathcal{S}$ have the form depicted in Fig. 5b, where the root implementation starts with an activity pair a, then splits into two activities pairs b and c, then merges again into another a activity, splits again into b and c, and so forth. The k-th BP in $\mathcal{S}$ contains k repetitions of this form. The EX-pattern p consists of a root activity pair whose internal trace contains a start and end activity pairs both labeled a, and a single transitive edge between them. Each EX-trace in $p(Naive(s))$ has a root activity with an internal trace that is one of the individual paths from the start to the end activity. There is an exponential number of such paths. Thus, each specification generating all of these masked traces must contain each path as an explicit implementation of the root, and hence the result size must be $\Omega(2^{|s|})$.

Next, we show that the use of at least masked (rather than naive) tracing is necessary to obtain a Type Inference algorithm (of any complexity):

Theorem 5 *There exist a BP specification s and an EX-pattern p s.t. there is no BP specification s' where $Naive(s') = p(Naive(s))$.*

Proof Consider the BP specification s whose act-comp DAGs are depicted in Fig. 5a and the EX-pattern p in Fig. 6 (ignoring now the labels next to the nodes). The EX-pattern requires an occurrence of Credit1 and Credit2. Here,

$p(Naive(s))$ contains only the trace in Fig. 3b. However, as explained in the proof of theorem 1, no BP specification s' can have this EX-trace as its single naive EX-trace, because for it to be in $Naive(s')$, `Hotel` must have at least two alternative implementations in s', one containing `Credit1` and the other containing `Credit2`, but this would allow for additional EX-traces in which all other 3 combinations of implementations of the two `Hotel` occurrences, including `Credit1` and `Credit2`.

Last, we note that unless P=NP no algorithm can be polynomial in the size of the given *query*. We define a decision problem, $MATCH?(p, s)$, as the problem of deciding, given a pattern p, a BP s, whether some embedding of p in a naive trace of s exists, i.e., whether $p(Naive(s)) = \emptyset$.

Theorem 6 *Given an EX-pattern p and a BP s, $MATCH?(p, s)$ is NP-hard in the size of p even if the following condition holds.*

– *In the implementations of all activities in p, and in all the act-comp DAGs of s, all nodes besides the* `end` *node have a single parent.*

The problem is NP-hard (in the size of p) even if only naive trace types are considered (i.e., A is empty and π is the identity function).

Proof We construct the following reduction from 3-SAT [15]. Given a Conjunctive Normal Form formula F, with variables $\{X_1, \ldots, X_n\}$, we generate an instance of a BP specification s and an EX-pattern p as shown in Fig. 7. The idea is to create a compound activity node associated with each variable of the formula. Each such node has two possible implementations, i.e., for all i, the implementations of X_i are F_{iTrue} and F_{iFalse}. The former contains a node for each clause that X_i satisfies, and the latter contains a node for each clause that $\neg X_i$ satisfies. The query requires all clauses of the formula F to appear (in an indirect implementation of the root, preceded / followed) by any sequence of activities, hence the use of the transitive node and edges). □

Lemma 1 *There exists an embedding of the query within a naive EX-trace of the constructed BP specification if and only if the formula F is satisfiable.*

Proof Let e be a naive EX-trace of s in which the query is embedded. e was obtained by choosing for some composite nodes their 'true' implementations, and 'false' implementation for the rest. These choices correspond exactly to a satisfying assignment: for every variable whose corresponding compound activity node was implemented by a 'false' ('true') graph, assign 'false' ('true'). Conversely, let A be a satisfying assignment, and let e be the naive EX-trace obtained by choosing, for each composite node, its 'true' ('false) implementation if A assigns 'true' ('false') to the corresponding variable. As A is a satisfying assignment, it holds that for every clause of the formula there exists a corresponding node in one of the chosen implementations in e. Consequently, there exists an embedding of the pattern p in e.

To complete the picture, we show that the $MATCH?$ decision problem is in NP. In fact, we show that an NP algorithm exists even for an extended version of the $MATCH?$ problem, where the tracing system used is selective. We use $MATCH?(p, s, A, \pi)$ to denote the problem of deciding whether $p(selective(s, A, \pi)) = \phi$ (the existence of NP algorithm for the cases of masked and naive tracing follows immediately). □

Theorem 7 *Given an EX-pattern p, a BP s, a set of activities A and a renaming function π, $MATCH?(p, s, A, \pi)$ is in NP (combined complexity).*

Proof The NP algorithm is based on the following Lemma. □

Lemma 2 *If $p(Selective(s, A, \pi)) \neq \phi$, then there exists at least one EX-trace e of s, s.t. the size (i.e., number of nodes and edges) in e is polynomial in s and p and e' satisfies p, where e' is the selective EX-trace obtained from e by removing the activities in A and applying the renaming function π.*

Proof Let $e \in selective(s, A, \pi)$ be a trace in which p may be embedded. The number of nodes and edges of e to which nodes and non-transitive edges of p are mapped is clearly bounded by the size of p. However, the length of paths to which transitive edges are mapped may be arbitrarily long. Still, we may construct a "smaller" trace e' in which p is embedded, by "cutting" the length of such paths as follows. First, if no recursive invocation of activities occurs along the path then its length is linear in $|s|$. Otherwise, if there exists such recursive invocation rooted at an activity pair n and in which the invoked activity pair is n', we create a new EX-trace e' by eliminating from e the entire sub-flow rooted at n and replace it by the sub-flow rooted at n'. Observe that as n and n' share the same activity name, it holds that also $e' \in selective(s, A, \pi)$. Also, there exists an embedding of p in e', mapping the transitive edge to the new path. We repeat for each recursive implementation along the path until obtaining a flow in which no such implementation exists. We further repeat this process for additional transitive edges. Eventually, we get a trace where the length of each path to which a transitive edge of p is mapped is bounded by $|s| * |p|$; the number of such paths is bounded by $|p|$, yielding an overall bound of $O(|s| * |p|^2)$ on the trace size.

The NP algorithm is then simple: guess an expansion sequence of polynomial size starting at the specification root,

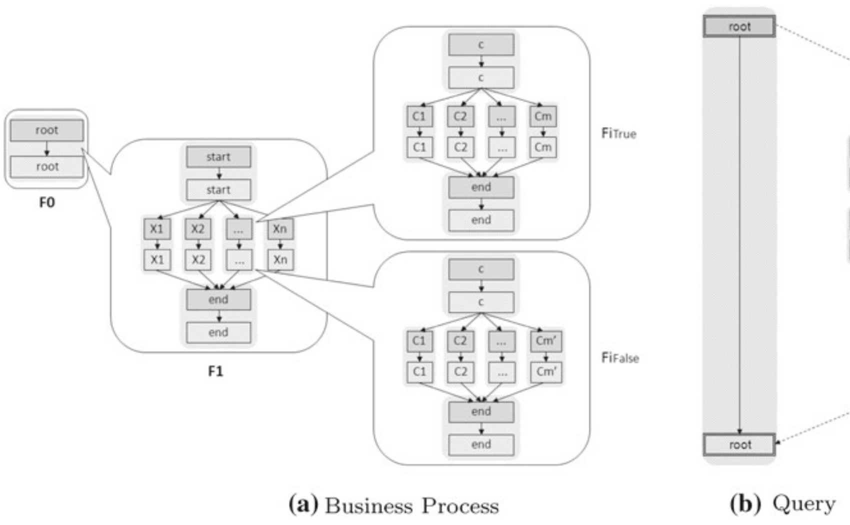


Fig. 7 NP-hardness proof

to obtain a flow e; remove from e all activities in A and apply the renaming function π over the remaining activities, to obtain a trace e'. Finally, guess a mapping of the pattern p to e'.

3.4 Using types for queries optimization

To conclude this section, we illustrate (in a simplified setting) how types, and our results on type inference, allow for query optimization and error detection.

First, we illustrate why types are useful for query optimization in this settings. Consider a simple query that searches for EX-traces containing two activities, A and B, and assume that this query needs to be evaluated on a large repository of execution traces, obtained by logging multiple execution of some process. Type information is very useful in query optimization in this context: for instance, if the traces type (represented by a specification of the Business Process and the tracing system that was used) tells us that all traces contain an A, then we can employ an optimized query evaluation, checking only for the existence of B. Similarly, if the type tells us that no trace contains an A, we can immediately infer inconsistency of the query (in the sense that its output is surely empty) and halt. Indeed, we presented in [4] a type-based optimization technique which eliminates redundant computation, yielding 50% improvement in performance. Note that the optimization algorithm, illustrated in [4] on naive types, works uniformly for naive/masked/selective types, as it builds on general results of [11] that apply to all trace classes.

Now, assume we are given a large database containing EX-traces of a given BP. Consider a scenario where some subset of the traces is selected by a (coarse-grained) query Q_1 whose goal is to identify the EX-traces that may be relevant to the analysis task. The EX-traces selected by Q_1 are then fed as input to another query Q_2 for further analysis. Such two-step processing is typical in EX-traces analysis [29], due to the extensive size of the repositories and the irrelevance of large parts of them to analysis tasks. As seen above, knowing the shape of Q_2 input, or, in other words, inferring the *type* of Q_1 output allows to optimize Q_2. Note that the use of masked, or the more powerful selective, types to capture the output of Q_1 is crucial here; we have shown that inferring a naive type may simply *not be possible*. In contrast, we also showed that type inference that uses an output type with masked / selective is possible (and can be used regardless of whether the *input type* was naive / masked / selective) and may thus be used for the optimization of Q_2. In particular, if we use an output type with selective tracing, then we have shown that such a type can be computed in polynomial time with respect to the original type (for any input type and query), rendering it very useful for query evaluation.

4 Type Checking

The problem of Type Checking is to verify that the query result conforms to a given type. Formally, given a target BP specification s', we want to check if $p(Naive(s)) \subseteq Naive(s')$. Similarly, for masked tracing, given also a renaming function π', we wish to check $p(Masked(s, \pi)) \subseteq Masked(s', \pi')$, and similarly for selective tracing, we wish to verify $p(Selective(s, A, \pi)) \subseteq Selective(s', A, \pi')$.

We next suggest a Type Checking algorithm for naive and masked tracing. We present it in two steps: we start with a restricted case of deterministic trace types, and then explain the general case.

4.1 Deterministic trace types

Let s be some BP specification and π a renaming function for the activities in s. Consider an EX-trace $e \in Naive(s)$ and its image, after activities renaming, denoted by $\Pi(e)$ (note that $\Pi(e) \in Masked(s, \pi)$). Clearly, there is at least one isomorphism from e to $\Pi(e)$ mapping activity pairs labeled a to activity pairs labeled $\pi(a)$. A node n_o in e that is mapped through such isomorphism to a node n in $\Pi(e)$ is called the origin of n. Note that in general, a node may have more than one possible origin, as (a) Π is not one-to-one and (b) even for a specific pair of traces e and $\Pi(e)$, there may be several different isomorphisms between them.

Definition 9 Let s be a BP specification and π be a renaming function over its activities. $Masked(s, \pi)$ is called *deterministic* (w.r.t. s, π) if for every node n in every trace in $Masked(s, \pi)$, all possible origins of n in $Naive(s)$ have the same activity name.

We next provide an algorithm for Type Checking for deterministic sets of traces, in presence of masked Tracing (in both the input and the output types). Note that this means that the algorithm works also for the particular case of naive tracing.

The basic framework of Algorithm DETERMINISTIC-TYPE-CHECKING is depicted in Algorithm 3. The Algorithm first applies Algorithm MASKED-TYPEINFERENCE depicted in the previous section, to infer the type s'', π'' of those traces of s, π that also conform to p. It then computes a type capturing a specific kind of *complement* of the traces in $Masked(s', \pi')$, denoted $\overline{Masked}(s', \pi')$ (the exact notion of this complement is given below); the algorithm intersects $Masked(s'', \pi'')$ with $\overline{Masked}(s', \pi')$ (the complement of the required type), and checks if the intersection result is empty. The intersection result is empty if and only if none of the traces in the obtained type are in the complement of the required type.

To define the exact notion of complement used here, also note that, in fact, it suffices to consider a restricted portion of $\overline{Masked}(s', \pi')$ that includes only the traces of the complement in which the size of each *direct* internal trace is bounded by the size of the largest direct possible internal trace in EX-traces of s''. In other words, we can bound their size by some number k—the size of the largest act-comp graph in s''. This holds because other traces of the complement cannot be isomorphic to some EX-trace of s''; in traces of s'', each activity has a direct implementation chosen out of the act-comp graphs of s, thus its size is bounded by k.

We next explain INTERSECT, COMPLEMENT, and TEST-EMPTINESS algorithms.

Intersection. For ease of presentation, let us first consider naive traces and then extend the discussion to masked traces. Given two BP specifications $s = (S, s_0, \tau)$, $s' = (S', s'_0, \tau')$, we find a BP specification $s'' = (S'', s''_0, \tau'')$ whose naive traces are exactly those in the intersection of the naive traces of s and s'.

Algorithm 3: DETERMINISTIC-TYPE-CHECKING

```
Input: Input type $s, \pi$; EX-pattern $p$; Required type $s', \pi'$
Output: True if and only if $p(Masked(s, \pi)) \subseteq Masked(s', \pi')$
1 $(s'', \pi'') \leftarrow MASKED-TYPE-INFERENCE(s, \pi, p)$ ;
2 $(s''', \pi''') =$
  $INTERSECT(COMPLEMENT(s', \pi'), (s'', \pi''))$ ;
3 $TEST-EMPTINESS(s''', \pi''')$ ;
```

The set of its activity names is the intersection of the activity names sets of s', s''. If the activity names r, r' labeling the root activities of s and s' are different, then clearly $Naive(s) \cap Naive(s') = \emptyset$ and s'' is the empty BP. Otherwise, the root of s'' is labeled by $r = r'$. The construction of s'' proceeds as follows. For every compound activity name a in S'' that was not treated yet, we set $\tau''(a) = \tau(a) \cap \tau'(a)$. This intersection is a regular intersection between sets of graphs, where a graph $g \in \tau(a)$ appears in the intersection if it is isomorphic (up to node identifiers) to some $g' \in \tau'(a)$.

All the act-comp graphs appearing in $\tau''(a)$, for some activity name a, are added to S''. Finally, we perform "cleanup": repeatedly, all the graphs in S'' are checked and the graphs g having compound activities a for which $\tau''(a) = \emptyset$ are removed from S''. τ'' is being adjusted accordingly, removing g from the implementation sets of all activities. Note that this may now make $\tau''(b) = \emptyset$ for some additional activities b and recursively trigger the removal of more graphs from S'', etc.

The algorithm for masked traces is the same up to the following two changes: (1) the graphs of the two BPs are now tested for isomorphism *modulo the activity renaming functions π and π'*, and (2) the nodes in s'' represent pairs of nodes in s and s' and are labeled by pairs of their origin activity names. The implementation of (a, a') is the intersection of the implementation of a in s with the implementation of a' in s, with isomorphisms computed up to π, π'. The result of each such isomorphism is an act-comp graph whose nodes are labeled by pairs of the original activity names from s, s', labeling nodes matched by the isomorphism. The cleanup step remains as above.

Computing the complement. We now consider (restricted) complement. For a natural number k, we say that an EX-trace is k-bounded if the size of any of its direct internal traces is bounded by k. Let k be the size of the largest act-comp graph in s''. Given a BP specification s with activities renaming function π, we use $\overline{Masked_k}(s, \pi)$ to denote the set of all k-bounded EX-traces that do not belong to $Masked(s, \pi)$. We shall construct a BP $\overline{s} = (\overline{S}, \overline{s_0}, \overline{\tau})$ with renaming function $\overline{\pi}$ s.t. $Masked(\overline{s}, \overline{\pi}) = \overline{Masked_k}(s, \pi)$.

For each (compound) activity a in s , let $\overline{a}$ be a new (compound) activity name not in s that will be used to represent the "complement" of a. Let Act be the set of activity names consisting of the activity names in s and their "complements". $\overline{S}$ is the set of all possible act-comp DAGs with activity names in Act and size bounded by k. $\overline{s_0}$ is obtained from the root of s by replacing the root activity name a by $\overline{a}$. The implementation function $\overline{\tau}$ is defined as follows. For compound activities a from s, $\overline{\tau}(a) = \tau(a)$. For the "complement" activities, $\overline{\tau}(\overline{a})$ is a subset of $\overline{S}$ consisting of (1) all graphs $g \in S$ where $\pi(g) \notin \pi(\tau(a))$, (2) the graphs in $\tau(a)$ with one or more or their compound activities a replaced by the corresponding "complement" $\overline{a}$. Last, the renaming function $\overline{\pi}$ maps a and $\overline{a}$ to $\pi(a)$, i.e., $\overline{\pi}(a) = \pi(\overline{a}) = \pi(a)$.

Testing for Emptiness. Testing that the set of traces defined by a given BP specification s is the empty set, is done in a very similar way to the algorithm for testing emptiness of a Context-Free String Grammar, as follows. The algorithm will gradually "mark" all activities that can be the root of some (part of) EX-trace, i.e., that there exists an implementation sequence that starts in them and eventually terminates. This is done iteratively: the algorithm first traverses all non-terminal activities, and "marks" them. The algorithm then repeatedly traverses all compound activity names (in some order), and for each such activity, it looks for a possible implementation in which all activities names were already marked (such implementation would lead to the generation of an EX-trace). We terminate when there are no new activities to be marked; the traces set is not empty if and only if the root activity was marked.

Complexity. The algorithm complexity is dictated by the complexity of its three subroutines. While the complexity of INTERSECT is quadratic in the size of its input specification, that of COMPLEMENT is exponential in the input sizes, and finally the complexity of TEST-EMPTINESS is obviously PTIME. This leads to an overall EXPTIME complexity.

4.2 The general case

Now that we showed decidability of type checking for *deterministic* semi-naive target types, we next show that we can translate *every* BP s and renaming function π to equivalent s' and π' w.r.t. which the set of masked EX-traces is deterministic.

Activities of the new BP First, we group all activities of s that are mapped by π to the same activity, obtaining a set γ of subsets. We say that each such subset is *represented* by the (single) activity to which its members are mapped. As each activity of s is mapped by π to a unique activity (π is a function), we can guarantee that no activity will appear in two different subsets. Furthermore, each subset is represented by a single activity. The set of activities of s is exactly the set of all subsets obtained in the above manner. We overload the notation and use the name of the representing activity to also stand for the subset of activity names that it represents. The root of s' is the activity representing the subset in which the root of s appears.

Implementation function of the new BP In the sequel, we denote $\Pi(g)$ as the graph obtained from g by applying π over all of its activity names. Similarly, $\Pi(G)$ where G is a set of graphs denotes the set obtained by applying Π on each $g \in G$.

We say that a set A' is an equivalence class with respect to a graph g', if for **all** $a \in A'$, there exists a graph g_a, obtained from g' by replacing each activity name B with some $b \in B$, such that $\Pi(g_a) \in \Pi(\tau(a))$. The new implementation function τ' is defined as follows. For an act-comp graph g' labeled by activities of s', and for an activity A' of s', $g' \in \tau'(A')$ if and only if (1) A' is the *maximal* set out of the sets in γ, that is an *equivalence class* with respect to g' and (2) with respect to each specific A' and keeping fixed all other activities in g', each **atomic** activity B in g' represents the *maximal* set of atomic activities out of these in γ.

Renaming function Recall that each activity A' of s' stands for a subset of activities that are mapped by π to a single activity a. The renaming function π' maps A' to its representing activity.

$Masked(s', \pi')$ is deterministic w.r.t. s', π as each activity name is composed as a *maximal* equivalence class of activities, thus for each activity name a appearing in a masked trace there exists a unique activity name A of s' such that $a \in A$, i.e., a unique A such that $\pi'(A) = a$. To conclude, we show that $Masked(s', \pi') = Masked(s, \pi)$.

Lemma 3 $Masked(s', \pi') = Masked(s, \pi)$.

Proof Let $e \in Masked(s', \pi')$ and let e' be a naive EX-trace of s' whose masked trace is e. Consider a compound activity name A' appearing in e' with an implementation g' attached to it. Then A' is an equivalence class w.r.t. g', i.e., We may replace A' with some $a \in A'$ and replace each activity name B in g' by some $b \in B$ to obtain $g \in \tau(a)$. We then assign for all compound activity names appearing in g the implementation appearing in g' for their origin activity and repeat the process for each such activity. The result of this repeated replacements is an EX-trace e'' of s; it holds that $\Pi(e'') = e$ as we only replaced any activity name B by some activity name $b \in B$; for all $b \in B$, $\pi(b) = \pi'(B)$ (by definition). Thus $e \in Masked(s, \pi)$.

Conversely, let $e \in Masked(s, \pi)$. Consider a compound activity name a appearing in e' with implementation g'. Then there exists an equivalence class A such that $a \in A$ and there exists a $g' \in \tau'(a)$ such that g is obtained from g' by replacing each activity B in g' by some $b \in B$. By subsequently making such replacements, we obtain a flow $e'' \in Masked(s', \pi')$ such that $\Pi'(e'') = e$, thus $e \in Masked(s', \pi')$. □

Corollary 1 *Given an EX-pattern p, a BP specification s (with renaming function π), and a target BP specification*

s' (with renaming function π'), testing if $p(Naive(s)) \subseteq Naive(s')$ (respectively, $p(Masked(s,\pi)) \subseteq Masked(s',\pi')$) is in EXPTIME (data complexity).

4.3 Lower bound

Our Type checking algorithm accounted only for masked tracing. To complete the picture, we show undecidability of type checking for selective trace types.

Theorem 8 *Given an EX-pattern p and two BP specifications, renaming functions and deletion sets (s, π, A) and (s', π', A'), testing whether $p(Selective(s, \pi, A)) \subseteq Selective(s', \pi', A')$ is undecidable.*

Proof By reduction from the problem of testing containment of context-free (string) languages. Given two context-free languages L, L', we construct, as in the proof of theorem 1, s, A, π and s', A', π' such that the graphs in $Selective(s, \pi, A)$ and $Selective(s', \pi', A')$ correspond, respectively, to the strings in L and L' (in the same sense defined in the proof of theorem 1). The EX-pattern p consists of a root activity whose implementation contains two activity pairs connected by a transitive edge. When applied to the EX-traces in $Selective(s, A, \pi)$, it retrieves all paths from start to end node of the root internal flow, (hence all the words in L). Hence $p(Selective(s, A, \pi)) = Selective(s, A, \pi)$ and $p(Selective(s, A, \pi)) \subseteq Selective(s', \pi', A')$ iff $L \subseteq L'$. □

5 Related work

The model used here is based on an abstraction of the BPEL (Business Process Execution Language) standard [6]. Commercial vendors offer systems that allow to design BPEL specification via a visual interface, using a conceptual, intuitive view of the process, as a nested graph. Such designs can be automatically compiled into executable code that implements the process described in BPEL [27].

Many works study variations of *Finite State Machines (FSM) [30]*. FSMs can describe "flat" processes, but not nested or recursive processes. StateCharts [18] allow hierarchy of states, but not recursion. A *recursive state machine (RSM)* [2,5] is a further extension of FSM, introducing the definition of a node *implementation*, which may possibly be recursive. A restriction of the model is called *single entry single exit RSM* (SRSM), and we have shown in [11] a syntactic equivalence between SRSMs and BPs. The common approach there is to use a query language based on *temporal logic* [22], e.g., LTL, CTL^* or μ-calculus (these differ from each other, and from our query language, in terms of expressive power, see e.g., [22]).

A database approach for modeling and analysis of processes was also studied in various contexts, as follows. *Active XML (AXML)* extends XML with embedded invocations of Web Services [1], that may be recursive. The main distinction, from a theoretical point of view, between AXML and the BP model is that the first assumes that each individual process graph is a *tree*, rather than a DAG as assumed here. The analysis of interactive data-driven Web applications [7,13] focuses on combination of flow with the *data*; similarly, the analysis of Business Process *artifacts* [19] focuses on the data manipulated by the process. Another branch in which processes have been extensively studied is that of *scientific workflows* (see [10] for an overview); these processes represent activities and computations that arise in scientific problem-solving. Our query language is weaker in terms of expressive power than those studied in these works but allows for better (worst case) performance guarantees. We believe that the model and query language studied here constitute a reasonable tradeoff between expressibility and complexity of query evaluation. We also briefly mention a complementary line of tools that infer process specifications out of a set of run-time generated *traces* (logs). The business process intelligence (BPI) [17] project infers causality relationships between execution attributes using data mining techniques such as classification and association rule mining. Such retrieved relationships may be processed to infer a BP structure along with the tracing system that was used. Such mining techniques were also extensively studied in the context of workflows (see e.g., [32]) and are complementary to our work, in the sense that their output can be used as our input type.

There is a tight connection between the classes of EX-trace types studied here and corresponding classical classes of *string* and *graph* languages. There is an analogy between naive, masked and selective trace types and bracketed [16], parenthesis [23] and context-free string [30] or graph [9,20] languages. Most of the works in this context consider MSO queries that are expressible but incurs very high evaluation time (non-elementary in the size of the query).

Type checking and type inference are well-studied problems in the context of functional programming languages [25]. The complexity there is derived from the interaction of function types, polymorphism, and `let`-bindings; as pointed out in [31], this analysis is valuable for database queries as well. Type inference and type checking were also considered extensively in the context of XML. The XML analogs of the queries studied here are *XML selection queries* [24] that use tree patterns to select subtrees of interest. For such XML queries, Milo and Suciu [24] showed that type checking can be performed in time complexity equal to or lower than type inference (depending on the XML types/queries being considered). Compare to our setting where type checking is harder than type inference, in some cases.

6 Conclusion and future work

We studied in this paper type inference and type checking for queries over BP execution traces. We formally defined and characterized three common classes of EX-trace types. We considered their respective notions of type inference and type checking and studied the complexity of the two problems for query languages of varying expressive power.

Our results for type inference signal the class of selective trace types as an "ideal" type system for BP traces, allowing both flexible description of the BP traces as well as efficient type inference. On the other hand for type *checking*, we showed that it may not be done in PTIME, even for limited trace types. Consequently, we believe that run-time checks of the query result are likely to be a more useful in practice.

The present paper has focused on exact inference of types. For practical cases computing an *approximated* type (e.g., capturing a large fraction of the traces of the "correct" type), may be practically useful when exact inference is impossible or costly. Studying such approximations is an intriguing future research. Last, we have mentioned the study of type information for XML. In this context, *transformation queries* were extensively studied [14, 21]. An interesting future work is to examine the adaptation of corresponding transformation queries to our settings.

References

1. Active XML. http://activexml.net/
2. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T., Yannakakis, M.: Analysis of recursive state machines. ACM Trans. Program. Lang. Syst **27**(4) (2005)
3. Beeri, C., Eyal, A., Kamenkovich, S., Milo, T.: Querying business processes. In: Proceedings of VLDB (2006)
4. Beeri, C., Eyal, A., Milo, T., Pilberg, A.: Monitoring business processes with queries. In: Proceedings of VLDB (2007)
5. Benedikt, M., Godefroid, P., Reps, T.: Model checking of unrestricted hierarchical state machines. In: Proceedings of ICALP (2001)
6. Business Process Execution Language for Web Services. http://www.ibm.com/developerworks/library/ws-bpel/
7. Bultan, T., Su, J., Fu, X.: Analyzing conversations of web services. IEEE Internet Comput. **10**(1) (2006)
8. Chamberlin, D.: XQuery: a query language for XML. In: Proceedings of SIGMOD (2003)
9. Courcelle, B.: The monadic second-order logic of graphs. Inf. Comput. **85**(1) (1990)
10. Davidson, S. B., Freire, J.: Provenance and scientific workflows: challenges and opportunities. In: Proceedings of SIGMOD (2008)
11. Deutch, D., Milo, T.: Querying structural and behavioral properties of business processes. In: Proceedings of DBPL (2007)
12. Deutch, D., Milo, T.: Type inference and type checking for queries on execution traces. In: Proceedings of VLDB (2008)
13. Deutsch, A., Sui, L., Vianu, V., Zhou, D.: Verification of communicating data-driven web services. In: Proceedings of PODS (2006)
14. Engelfriet, J., Hoogeboom, H. J., Samwel, B.: XML transformation by tree-walking transducers with invisible pebbles. In: Proceedings of PODS (2007)
15. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, San Francisco (1979)
16. Ginsburg, S., Harrison, M.: Bracketed context-free languages. J. Comput. Syst. Sci. **1** (1967)
17. Grigori, D., Casati, F., Castellanos, M., Sayal, M., Dayal, U., Shan, M.: Business process intelligence. Comput. Ind. **53** (2004)
18. Harel, D.: Statecharts: A visual formalism for complex systems. Sci. Comput. Program **8**(3) (1987)
19. Hull, R., Su, J.: Tools for composite web services: a short overview. SIGMOD Rec **34**(2) (2005)
20. Janssens, D., Rozenberg, G.: Graph grammars with node-label controlled rewriting and embedding. In: Proceedings of COMPUGRAPH (1983)
21. Maneth, S., Perst, T., Seidl, H.: Exact XML type checking in polynomial time. In: Proceedings of ICDT (2007)
22. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, Berlin (1992)
23. McNaughton, R.: Parenthesis grammars. J. ACM **14**(3) (1967)
24. Milo, T., Suciu, D.: Type inference for queries on semistructured data. In: Proceedings of PODS (1999)
25. Mitchell, J.C.: Foundations for Programming Languages. MIT Press, Cambridge (1996)
26. Nevsetvril, J., de Mendez, P.O.: Tree-depth, subgraph coloring and homomorphism. Eur. J. Comb. **27**(6) (2006)
27. Oracle BPEL Process Manager 2.0 Quick Start Tutorial. http://www.oracle.com/technology/products/ias/bpel/index.html
28. Papakonstantinou, Y., Vianu, V.: DTD inference for views of XML data. In: Proceedings of PODS (2000)
29. Sayal, D. M., Casati, F., Dayal, U., Shan, M.: Business process Cockpit. In: Proceedings of VLDB (2002)
30. Sipser, M.: Introduction to the Theory of Computation. PWS Publishing Company, Boston (1997)
31. van den Bussche, J., van Gucht, D., Vansummeren, S.: A crash course on database queries. In: Proceedings of PODS (2007)
32. van der Aalst, W.M.P., Weijters, T., Maruster, L.: Workflow mining: discovering process models from event logs. IEEE Trans. Knowl. Data Eng. **16**(9), 1128–1142 (2004)
33. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath