Type Inference and Type Checking for Queries on Execution Traces [*]

Daniel Deutch Tova Milo
Tel Aviv University
{danielde,milo}@post.tau.ac.il

ABSTRACT

This paper studies, for the first time, the management of type information for an important class of semi-structured data: nested DAGs (Directed Acyclic Graphs) that describe execution traces of business processes (BPs for short). Specifically, we consider here type inference and type checking for queries over BP execution traces. The queries that we consider select portions of the traces that are of interest to the user; the types describe the possible shape of the execution traces in the input/output of the query.

We formally define and characterize here three common classes of BP execution traces and their respective notions of type inference and type checking. We study the complexity of the two problems for query languages of varying expressive power and present efficient type inference/checking algorithms whenever possible. Our analysis offers a nearly complete picture of which combinations of trace classes and query features lead to PTIME algorithms and which to NP-complete or undecidable problems.

1. INTRODUCTION

Schema and type information has proved to be extremely useful for the management of semi-structured data, in general, and XML, in particular. Knowledge (even partial) about the typical structure of data items allows e.g. for intuitive query formulation, optimized query processing, and minimization of run time errors [20, 4].

This paper studies, for the first time, the management of type information for an important class of semi-structured data: nested DAGs (Directed Acyclic Graphs) that describe execution traces of business processes (BPs for short). We consider here type inference and type checking for queries over such BP execution traces. The queries select portions of the traces that are of interest to the user. The types that we consider describe the possible shape of the execution traces in the input/output of the query.

In this context, we study two problems, as follows. In the *type inference* problem we are given some BP, and a query over its possible execution traces, and want to infer a type describing the shape of the sub-traces selected by the query. In the *type checking* problem we are also given an output type and want to verify that the shapes of all selected sub-traces conform to it. We study here the two problems for type specifications and query languages of varying expressivity.

Before presenting our results, let us briefly consider the practical origin of the problems and its influence on the particular families of graphs, types, and queries studied here.

Business Processes. A BP consists of some business activities undertaken by one or more organizations in pursuit of some particular goal. BPs operate in a cross-organization, distributed environment, and the software implementing them is fairly complex. Standards facilitate the design, development and deployment of such software. In particular, the recent BPEL standard (Business Process Execution Language[5]) allows to describe the full operational logic and execution flow of BPs. A BPEL *specification* describes a process as a nested DAG consisting of activities (nodes), and links (edges) between them, that detail the execution order of the activities. Activities may be either *atomic*, or *compound*. In the latter case their internal structure is also detailed as a DAG, leading to the nested structure. BPEL specifications are compiled into executable code that implements the described BP and runs on a BPEL application server [23].

Execution Traces. 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, its activation and completion events are reported. For a compound activity, the events corresponding to its internal flow are reported between its activation and completion events. An execution trace can be abstractly viewed as a (nested) DAG that contains nodes representing the activation and completion events of activities, and edges that describe their flow. The nesting of DAGs in the trace follows naturally from the nesting of the specification. Traces may vary in the amount of information that they record on the run. In general, one can distinguish three families of execution traces, with increasing flexibility: (i) *naive* traces which provide a complete record of the activation/completion events of all activities (ii) *semi-naive* traces where the activation/completion events are all recorded, but possibly with only partial information about their origin activity, and (iii) *selective* traces where only a selected subset of the events is recorded.

Queries and types. Execution traces are extremely valuable for companies. Their analysis allows to optimize business processes, reduce operational costs, and ultimately increase competitiveness. We next illustrate the notion of types, and intuitively explain its merits for the purposes of query optimization and error detection.

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

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08

Type information, i.e., knowledge about the possible structure of the queried (sub-)traces, is valuable for query optimization [3]. Its role is analogous to that of XML schema for XML query optimization: it allows to eliminate redundant computations and simplify query evaluation. For example, consider a simple query that searches for EX-traces containing two activities, A and B. A traces *type* might tell us that all traces contain an A, and then an optimized query evaluation, checking only for the existence of B, is possible. Similarly, if the type tells us that no trace contains an A, we can immediately infer that the query is inconsistent (in the sense that its output is surely empty) and halt.

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 and mines the sub-traces to derive critical business information [25]. Type information is readily available, as the BPEL specification, for the original traces, and can be used to optimize the first-step queries. However, it is not available for the intermediary sub-traces selected by queries, and fed as input to the second-stage query. *One of the main goals of this paper is to develop efficient algorithms for deriving this type information*, thereby felicitating the optimization of both stages of the querying process. At another level, when the analysis tool expects data of particular type, we would like to verify that the sub-traces selected by the queries are guaranteed to conform to the required type. *Such type checking is the second problem studied here.*

Both type inference and type checking are well studied problems for XML and tree-shaped graphs. We found that the particular nested-DAG shape of BP execution traces, and their corresponding type specifications, pose new challenges that require the development of novel type inference/checking techniques. For instance, while for XML, type checking is in general easier than type inference, we found that for nested DAGs type checking can be harder. (For a detailed comparison to XML type inference and checking see Section 6).

Our results. As mentioned above, we model an execution trace of a BP instance as a nested DAG. A type here consists of a BP specification (modeled abstractly as a rewriting system) accompanied by a description of the sort of tracing (naive, semi-naive, or selective) employed for the BP. The query language that we consider selects sub-traces of interest using *execution patterns*. Execution patterns are an adaptation of the tree- and graph-patterns offered by existing query languages for XML and graph-shaped data. We first consider simple positive queries without joins. We show that the less detailed (and thus less restrictive) the execution traces are, the more efficient type inference can be: it can be done in time polynomial in the size of the input type (with the exponent determined by the size of the query) for selective trace types, but may require time exponential in the size of input type (even for small queries) with semi-naive trace types, and may not be possible at all if all trace types are naive. Type checking, on the other hand, becomes more difficult as the expressivity of the involved types is extended: It is decidable, although expensive, for (semi-)naive trace types but undecidable for selective trace types.

Next, we consider more powerful queries that include patterns with joins (on node and path variables), projection and negation. While the above mentioned complexity results hold for most of these extensions, type inference is no longer possible for queries with joins on path variables. In contrast, type checking is shown to be still possible (for the same class of traces as before).

Our analysis offers a nearly complete picture of combinations of trace types and query features and their corresponding problems complexity. We provide efficient type inference and type checking algorithms whenever possible, and identify the cause of the difficulty when not. Our results have two main important practical implications, the first positive and the second negative.

1. As far as type inference is concerned, our results 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. We thus believe it to be a firm basis for further study of type-based query optimization for BP management systems.

2. On the other hand, we show that *static* type checking is not possible for selective trace types, and even for more limited trace types only EXPTIME algorithms are available. Consequently, we believe that *run-time* checks are likely to be a more useful in practice.

Remark. We conclude with a remark regarding our choice of query language and data model. Already in 2002, the importance of query languages for BPs had been recognized by BPMI (the Business Process Management Initiative)[6]. Yet, unlike the case of XML and tree-shaped data, no draft standard has been published since. The data model and query language that we consider here were introduced in [2, 3]. They are argued to be more intuitive for BP developers than other query formalisms such as temporal logics and process algebras, as they are based on the same graph-based view of processes used by commercial vendors for the specification of BPs [3]. While we state our results in terms of this particular model, they are naturally applicable to (fragments of) other query formalisms sharing similar expressive power. A detailed comparison to related data models and query languages (including temporal logics [18], MSO, equational sets [9], etc.) can be found in [10]. In the Related Work section (Section 6) we further compare known results on these models to our results, highlighting our relative contribution.

The paper is organized as follows. Section 2 introduces the definitions for a simplified version of our model and query language. In Sections 3 and 4 we formally define and analyze Type Inference and Type Checking for this simplified setting. In Section 5 we extend the study to the full-fledged model. Related work is considered in Section 6. We conclude in Section 7.

2. PRELIMINARIES

To simplify the presentation we first consider a basic data model and query language, and then enrich them to obtain the full-fledged model.

2.1 Execution traces

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

DEFINITION 2.1. *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, $\lambda : N \to \mathcal{A}$ *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 labelled by some label a by a pair of nodes, n', n'', labelled (resp.) by $(a, \texttt{act})$ and $(a, \texttt{com})$. 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 $\texttt{start}$ node without incoming edges, and a single $\texttt{end}$ node without outgoing edges, denoted by $\texttt{start}(g)$ and $\texttt{end}(g)$, resp.

EXAMPLE 2.2. *To illustrate, let us consider the act-comp DAG ($d2$) in Figure 2. It contains four activities (*`Search`*,* `Hotel`*,* `Flight` *and* `Print`*), each represented by a pair of nodes. In each pair, the node with darker (lighter) background denotes the activity's activation (completion).*

DEFINITION 2.3. *The set $\mathcal{EX}$ of* execution traces *(abbr.* EX-traces*) is the smallest set 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 (resp. n_2) has a single outgoing (incoming) edge, and g_2 is some act-comp DAG, then the graph g 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 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 2.3, 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*.

EXAMPLE 2.4. *Some example EX-traces are depicted in Figure 1. Let us focus first on EX-traces (a). It details a possible executions 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, resp., and a confirmation was printed. Zooming into both reservation activities reveals some credit limit check, namely* `Credit1`*. The EX-trace in Figure 1(b) is another possible EX-trace of the travel agency. Here we see that the user was looking for a luxury trip, invoking the* `Luxury` *activity. The internal flow here is similar to that of a regular 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 [5]. A type consists of a BP specification accompanied by a description of the sort of tracing (naive, semi-naive, or selective) employed for the BP.

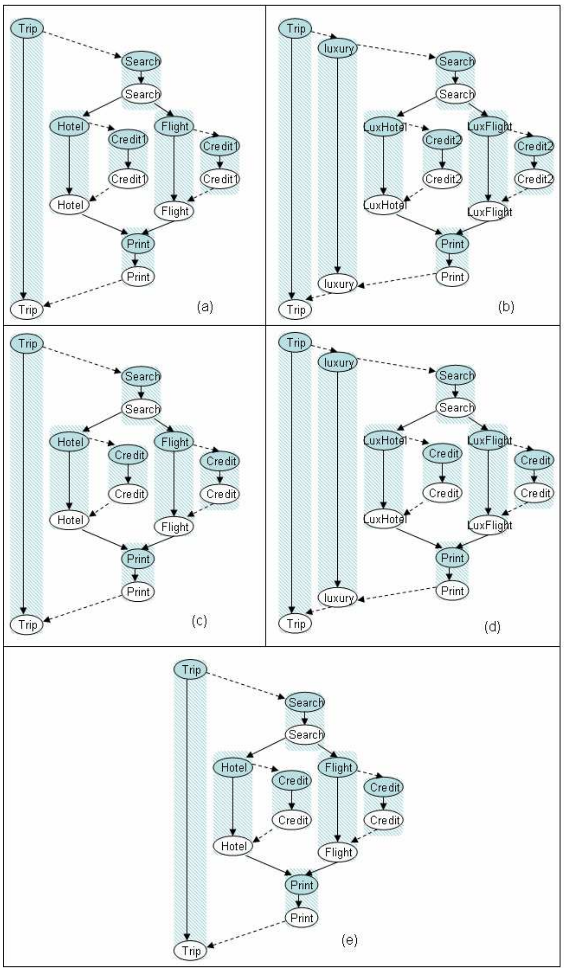

Figure 1: Execution traces.

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, resp. A BP specification is a collection of activation-completion DAGs, along with a mapping of compound activity names to their implementations.

DEFINITION 2.5. *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 $\tau : \mathcal{A}_{compound} \rightarrow 2^S$ is a function, called the* implementation function, *mapping compound activity names in S to sets of DAGs in S.*

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).

EXAMPLE 2.6. *To continue with our travel agency example, its BP specification s consists of the act-comp DAGs in Figure 2. The compound activities here are* `Trip`*,* `Luxury`*,* `Hotel`*,* `Flight`*,* `LuxHotel` *and* `LuxFlight`*. d_1 is the root of the BP, and the implementations of the compound activities are the following :$\tau(\texttt{Trip}) = \{d_2, d_3\}$, $\tau(\texttt{Luxury}) = \{d_4\}$, $\tau(\texttt{Hotel}) = \tau(\texttt{Flight}) = \{d_5\}$, $\tau(\texttt{LuxHotel}) = \tau(\texttt{LuxFlight}) = \{d_6\}$.*

The intuitive interpretation of this specification is as follows. When a user starts searching for a trip, she may choose between a regular trip (leading to d_2), or a luxury trip (leading to d_3, that in turn leads to d_4). Then, she searches and reserves (luxury) flights and (luxury) hotels, as described above.

We consider three kinds of traces. 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 2.7. **[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 $g \rightarrow g'$ (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 $(\mathit{end}(g_a), n_2)$.*

If $p \rightarrow p_1 \rightarrow p_2 \ldots \rightarrow p_k$, 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 its root s_0.*

Intuitively, Naive EX-traces of a process contain the full information regarding the process execution. For example, the two EX-traces in Figure 1(a) and (b) are naive EX-traces of the travel agency BP depicted in Figure 2. 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, however, where only partial information about the activities is recorded. For example, 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 a regular 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 *semi-naive* EX-traces.

DEFINITION 2.8. **[Semi-naive 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* semi-naive *EX-traces defined by s and π, denoted $semiNaive(s, \pi)$, 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 2.9. *To continue with our running example, Figures 1 (c) and (d) are semi-naive 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 semi-naive EX-traces are obtained resp. from the naive EX-traces in Figure 1 (a) and (b). We can see in the trace that some credit checks were issued, but not which ones.*

For readers familiar with XML schemas, semi-naive EX-traces can be viewed as the BP analog of XML trees defined by *DTDs with specialization* [24]. 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).

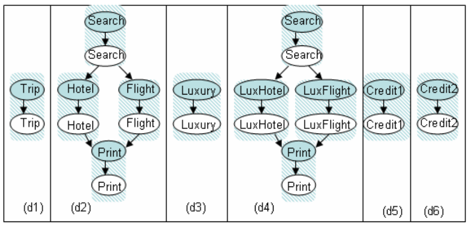

Figure 2: Business Process (1)

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 non- interesting 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 2.10. **[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 [1] 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 2.1), or is empty.*

EXAMPLE 2.11. *For instance, 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 relabeled, but also the* `LuxHotel` *and the* `LuxFlight`*, and the record of the* `Luxury` *activity should be omitted altogether. Here $A = \{\mathtt{Luxury}\}$, and $\pi(\mathtt{Credit1}) = \pi(\mathtt{Credit2}) = \mathtt{Credit}$, $\pi(\mathtt{LuxHotel}) = \mathtt{Hotel}$, and $\pi(\mathtt{LuxFlight}) = \mathtt{Flight}$.*

Figure 1(e) shows the selective EX-trace obtained from the naive EX-traces in Figure 1(b). Note that the same graph (up to node isomorphism) is also the selective EX-trace obtained from the naive EX-traces in Figure 1(a). Thus, given such a selective trace, there is not way to tell from which naive EX-trace (of regular or luxury trip reservation) it originated.

The intuition behind condition (*) in Definition 2.10 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 where *two* zoom-in edges going out of `Trip`, pointing at two different nodes. This contradicts the definition of an EX-trace. Condition (*) assures that this does not happen (proof omitted). The technical implication of this constraint will be discussed in the sequel.

[1] When an atomic activity pair (n_1, n_2) is deleted, the edges incoming 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.

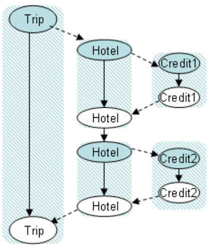

Figure 3: Execution Trace

Each BP specification s, (renaming function π, and activities set A) defines a (possibly infinite) set of EX-traces $E = Naive(s)$ (resp. $E = semiNaive(s, \pi)$, and $E = Selective(s, A, \pi)$). Let `Naive` (resp. `semiNaive` and `Selective`) denote the class of sets of EX-traces that obtained under naive (resp. semi-naive, selective) tracing of some BP specification (a renaming function π, and an activities set A). More formally, `Naive`= $\{E \mid \exists s\ s.t.\ E = Naive(s)\}$, `semiNaive`= $\{E \mid \exists s, \pi,\ s.t.\ E = semiNaive(s, \pi)\}$, `Selective`= $\{E \mid \exists s, \pi, A,\ s.t.\ E = Selective(s, A, \pi)\}$. We can show that there is a strict inclusion relationship between the classes, and that they do not capture all the possible sets of EX-traces.

PROPOSITION 2.12. **`Naive`** $\subset$ **`semiNaive`** $\subset$ **`Selective`** $\subset 2^{\mathcal{EX}}$.

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

`Naive` $\subset$ **`semiNaive`.** Consider the BP specification s depicted in Figure 5 (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` (resp. `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 semi-naive traces $E = semiNaive(s, \pi)$ of s contains the single EX-trace e, depicted in Figure 3. The EX-trace 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`.

`semiNaive` $\subset$ **`selective`.** Consider the class of BPs where the implementation graphs of all compound activities are chains. For each such BP, consider the selective traces obtained by deleting all compound activities and keeping the names of the atomic and root activities unchanged. 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 on 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 (semi-)naive EX-traces that contain only the root activity and atomic activities have a shape which 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` $\subset 2^{\mathcal{EX}}$**.** One might consider each set of EX-traces in `Selective` as a language of graphs. Through this perspective, we can compare our model to common models of graph languages that appear in the literature. A common model, named Recursive State Machines (RSM) [1], naturally extends Finite State Machines (FSM) (bearing entry and exit states), by allowing some states to be `call` states, invoking other FSMs. A call is simulated by *replacing* the `call` state by its implementation. A graph obtained by a sequence of these replacements is called an *unfolding* of the original RSM. The simplest form of RSM is Single Entry Single Exit RSM (SERSM), where each FSM has unique start and exit nodes. We can show that `Selective` captures the same sets of EX-traces as SERSMs.

PROPOSITION 2.13. *A set E of EX-traces is the set of full unfoldings of an SERSM iff it belongs to* `Selective`.

Proof:(Sketch) The inclusion in SERSM follows from condition (*). For the other direction we show that SERSMs which generate EX-traces have an equivalent normal form (like the *Greibach Normal Form* in context-free string grammars[26]) where all FSMs are activation-completion DAGs.

⊠

It is known that SERSMs do not capture all graph languages [1]. The strict inclusion of `Selective` in $2^{\mathcal{EX}}$ follows. This concludes the proof of Proposition 2.12

⊠

We sum up this subsection with a comment on the analogy between the EX-trace types, of the various sorts, and *string grammars*. While the selective trace types form the analog to a *Context Free Language*, semi-naive trace types are the analog to the restricted form of *parenthesis languages*[16] (as we keep track of the activation and completions events, analogous to parenthesis), and naive trace types are the analog of the further restricted *bracketed languages*[14] (that also keep track of the parenthesis 'origin'). This analogy holds in respect to some characteristics of the trace types, as we shall see below.

2.3 Queries

We now consider queries. As mentioned in the Introduction, the query language that we use was originally introduced in [2, 3]. 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. As we will see, this zoom-in navigation axis, particular to nested DAGs, introduces special challenges w.r.t. type inference and checking.

DEFINITION 2.14. *An* execution pattern*, abbr. EX-pattern, is a pair $p = (\hat{e}, T)$ where $\hat{e}$ is an EX-trace whose nodes are labelled by labels from $\mathcal{A} \cup \{$ANY$\}$, and T is a distinguished set of activity pairs and edges in $\hat{e}$, called* transitive *activities and edges, resp.*

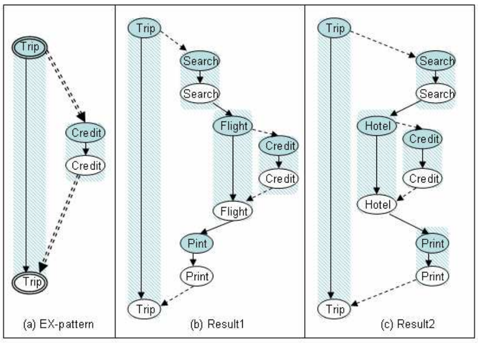

Figure 4: A query and its results

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

DEFINITION 2.15. *Let $p = (\hat{e}, T)$ 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. **[nodes]** *activity pairs in p are mapped to activity pairs in e. Node labels are preserved; however, a node labeled by* ANY *can be mapped to nodes with any activity name.*
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 $[m, n]$ belongs to a direct internal trace of a* transitive *activity, the edge (edges 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 result *defined by ψ is the image of p under ψ. For each activity pair in the image, the edge connecting its activation and completion nodes is also included.*

For an EX-pattern p and an EX-trace e, the result *of p when applied to e, denoted $p(e)$, consists of the 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 $p(E) = \bigcup_{e \in E} p(e)$.*

It is easy to prove that $p(e)$ (resp. $p(E)$) is a set of EX-traces.

.

An example EX-pattern is depicted in Fig. 4 (a). It zooms-in transitively into `Trip`, searching for a `Credit` activity. Double-bounded nodes (resp. double-lined edges) denote transitive nodes (edges). Observe that apart from the transitive nodes and edges, EX-patterns and EX-traces look similar, making the formulation of queries rather intuitive. Let us now evaluate the query over the EX-trace in Figure 1(e). Two embedding are possible here, yielding the results in Figure 4 (b) and (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, with a *non-transitive* `Trip` activity, would yield here an empty result, as it would search for `Credit` activities in the *direct* internal trace of `Trip` (while in the given EX-trace it appears only deeper in the zoom-in hierarchy). Finally, observe that since an embedding is a *homomorphisms*, multiple query nodes may be mapped to the same EX-trace nodes and edges (or paths).

3. TYPE INFERENCE

The type inference problem that we study is defined next. Three variants of the problem correspond to the three families of trace types, as follows.

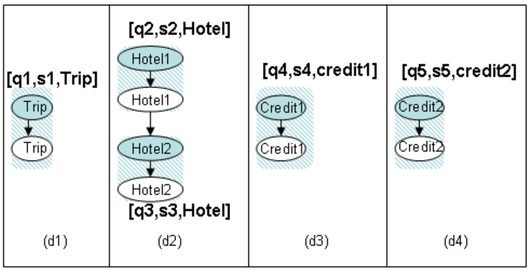

Figure 5: Business Process (2)

We are given an EX-pattern p and a set of naive (resp. semi-naive, selective) EX-traces defined by some BP s (a renaming function π and activities set A), and would like to find a BP specification s' (and π', A') s.t. $Naive(s') = p(Naive(s))$ (resp. $semiNaive(s', \pi') = p(semiNaive(s, \pi)), Selective(s', A', \pi') = p(Selective(s, A, \pi))$)[2].

We first show that type inference may not be possible if only naive trace types are considered. Namely,

THEOREM 3.1. *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: The proof follows lines similar to that of (the first part in the proof of) Proposition 2.12. Consider the BP specification s whose act-comp DAGs are depicted in Figure 6 (ignoring for now the text near the nodes). (d1) is the root of the BP. The implementation $(d2)$ of the root activity `Trip` contains two consecutive compound `Hotel` activities, and where `Hotel` has two possible implementations $(d3)$ and $(d4)$, one containing the activity `Credit1` and the second containing the activity `Credit2`. $Naive(s)$ here contains four EX-traces, e_1 where both `Hotels` have `Credit1` as internal trace, e_2 where they both have `Credit2`, and e_3 (e_4) where the first (second) `Hotel` has `Credit1` and the second (first) has `Credit2`. Now consider the query p in Figure 7 (again ignoring the text near the nodes), which requires an occurrence of `Credit1` and `Credit2`. Here $p(Naive(s))$ contains only e_3, and as explained in the proof of Proposition 2.12, no BP specification s' can have e_3 as its single naive EX-trace.

⊠

In contrast, the greater expressive power of semi-naive trace types allows to capture more EX-trace sets and enables type inference. We show next that type inference is possible with semi-naive trace types, but may be costly.

THEOREM 3.2. *For every EX-pattern p, BP s and renaming function π, there exist s' and π' such that $semiNaive(s', \pi') = p(semiNaive(s, \pi))$. s' and π' can be computed in time exponential in the size of s and p.*

Note that since every naive trace type is also semi-naive type (with π being the identity function), the theorem also implies that type inference is possible for input representing naive traces, with the output type captured by semi-naive representation.

Proof:(Sketch) The BP $s' = (S', s'_0, \tau')$ constructed by the type inference algorithm is intuitively the "intersection" of the original BP $s = (S, s_0, \tau)$ with the pattern p. We next explain how this s' is constructed, then illustrate with an example.

Let us first consider patterns that do not contain transitive nodes an edges. For every two activity pairs $n_s \in s$ and $n_p \in p$ where n_p

[2] 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

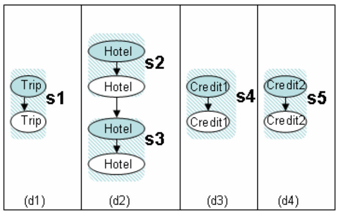

Figure 6: Business Process (3)

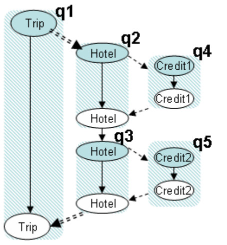

Figure 7: EX-pattern

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 pairs. The renaming function π' maps $[n_p, n_s, a]$ to a. Note that a node n_p (resp. n_s) of the pattern p (BP s) may appear in several such new activities $[n_p, n_s^i, a]$ (resp. $[n_p^i, n_s, a]$).

For compound (non-transitive) activities, the implementation τ' of $[n_p, n_s, a]$ consists of all DAGs that represent possible embeddings of the *direct* internal trace of n_p in p into the possible *direct* implementations of n_s in s. The nodes in the resulting graph (for each of the possible embeddings) are labeled by triplets, as above, recording for every activity pair in p to which activity pair in s it was mapped in the given embedding. If no embedding was found, $[n_p, n_s, a]$ is marked as failure. The embeddings may be found using conventional algorithms for subgraph homomorphism, whose complexity is discussed below.

For efficiency, rather than constructing *all* the triplet activities names and their implementations, the algorithm operates in a top down manner. It starts by matching the pattern outer most activity with the BP root, building the corresponding $[n_p, n_s, a]$ activity. Then it compute its implementations, and the implementations of the activities appearing in them, and so on.

As a final step, we perform "garbage collection". We recursively mark as failure, activities for which all possible implementations contain failure activities, and then remove from s' all DAGs that contain failure such activities.

When the query pattern contains transitive edges, we also define new activity name for every transitive edge $e_p \in p$ and activity $n_s \in s$. When the pattern sub-graphs are embedded into the BP, the transitive edges $e_p \in p$ (that connect two pattern nodes) are mapped to all possible paths in in the BP (connecting the two corresponding BP nodes). In the output graph, a BP node n_s (with label a) that appear on such path is labeled by the triplet $[e_p, n_s, a]$.

Finally, when the pattern p contains transitive activities, the algorithm becomes somewhat more complex. Recall that transitive activities allow to navigate (transitively) inside the compound activities of the EX-trace, and query their internal flow at any depth of nesting. Specifically, 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) pattern into sub-patterns, and the embeddings of those into the DAGs in s.

⊠

Before we consider the complexity of the algorithm, let us see an example that will demonstrate its operation.

EXAMPLE 3.3. *Consider again the EX-pattern p and BP s from the Proof of Theorem 3.1 (the pattern given in Fig. 7, and the BP of Figure 6) with π being the identity mapping. The annotations next to the query and BP activity pairs represent their ids. The BP s' constructed by the algorithm is depicted in Figure 5. Its activity names are of the form $[q_i, s_j, a]$, where q_i (s_j) is the identifier of a query (pattern) activity node (ignore now the labels appearing within the nodes). The implementation of $[q2, s2, Hotel]$ is $d3$ and the implementation of $[q3, s3, Hotel]$) is $d4$. The renaming function π' maps $[q_i, s_j, a]$ to a. Note that this s' has a single semi-naive EX-trace, of the shape depicted in Figure 3, which is indeed the only answer of the query p when applied on the (semi-)naive traces of s.*

We next explain how this 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 implementation of $s1$, thus embedding $q2$ (resp. $q3$) in $s2$ (resp. $q3$), 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 pattern nodes ($s2$ and $s3$), yielded two* distinct *activity names in s'. Consequently, each of these two activities names can now have distinct implementations that comply to 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 exists two such possible implementations, one of which contains* `Credit1` *and the other 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 $[q5, s5, Credit2]$.*

The time complexity of the algorithm depends on the number of possible sub-patterns that should be considered (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 large. The following Proposition shows that the exponential blowup is unavoidable, even for very small queries and naive input traces.

PROPOSITION 3.4. *There exists an infinite set $\mathcal{S}$ of BPs of increasing sizes, and an EX-pattern p (with only six nodes), s.t. for each BP $s \in \mathcal{S}$, every s' s.t. $semiNaive(s', \pi') = p(Naive(s))$ for some π' is of size $\Omega(2^{|s|})$.*

Proof: The BPs in the class $\mathcal{S}$ have a root activity whose implementation has the following form. The start and end activities of the implementation are labeled a. The flow starts with the first a, then splits into two activities 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 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 semi-naive traces must contain each path as an explicit implementation of the root, and hence its size must be $\Omega(2^{|s|})$.

⊠

We have seen above that selective trace types are more flexible and expressive than (semi-)naive trace types. It turns out that adding such expressivity to the output type also allows for a more compact representation of the possible query results, hence a more efficient type inference algorithm.

THEOREM 3.5. *For every EX-pattern p, BP s, and a renaming function π, there exist s',A', π' s.t. $Selective(s', A', \pi') = p(semiNaive(s, \pi))$. s', A', π' can be computed in time polynomial in the size of s (with the exponent determined by p).*

Moreover, the added expressibility allows an efficient algorithm even when the input type has stronger expressibility, i.e. represents selective traces as well.

THEOREM 3.6. *For every EX-pattern p, BP s, set of activities A and renaming function π, there exist s',A' and π' such that $Selective(s', A', \pi') = p(Selective(s, A, \pi))$. s',A' and π' can be computed in time polynomial in the size of s (with the exponent determined by p).*

Note that since every semi-naive trace type is also a selective one (with $A = \emptyset$), the proof of Theorem 3.6 also proves Theorem 3.5.

Proof:(Sketch) The improved type inference algorithm is based on the following observation. Consider the treatment of transitive edges in the previous type inference algorithm, that was described in the proof of Theorem 3.2. In that algorithm, each embedding of the sub-patterns of p into the BP s is treated individually and contributes one graph to 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. We view the sequence of activity names in each path as a word, and the set of words obtained from all paths between two nodes as a language L. It is easy to define, for each pair of nodes, a regular grammar describing L. We then use a similar construction to the one presented in the proof of Proposition 2.12 (when showing that `semiNaive` ⊂ `selective`) to define a BP s'', along with A'' and π'', s.t. the words generated in $Selective(s'', A'', \pi'')$ are precisely those of L. s'' is then incorporated as part of s'.

Aside from the above, the type inference algorithm follows lines similar to that of semi-naive traces. The resulting algorithm is polynomial in the size of s, with the exponent determined by p.

⊠

A first obvious reason for not being able to provide an algorithm of PTIME combined (data and query) complexity is the need to embed the pattern sub-graphs in the DAGs of s. This entails checking for subgraph homomorphism, known to be an NP-complete problem. Interestingly, we can expose an additional type of hardness that comes from the nested shape of the BPs and the use of transitive activities in the EX-patterns to navigate (transitively) inside compound activities and query their internal flow. To explain this, we define a decision problem, $MATCH?(p, s, A, \pi)$, as the problem of deciding, given a pattern p, a BP s, a set of activity names A, and an activities renaming function π, whether some embedding of p in a trace of s exists, i.e. whether $p(Selective(s, A, \pi)) = \emptyset$. We can show the following.

PROPOSITION 3.7. *Given an EX-pattern p, a BP s, a set of activities A and a renaming function π, $MATCH?(p, s, A, \pi)$ is NP-hard in the size of p even if the following condition holds.*

- *In the direct internal traces of all the 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 even if only naive trace types are considered (i.e. A is empty and π is the identity function).

Note that for DAGs with the restricted shape enforced by the condition, sub-graph homomorphism can be solved in PTIME (e.g. using algorithms from [15]). Indeed, the hardness here comes from the need to consider, when treating transitive pattern activities, all the possible splits of the EX-pattern (see the algorithm in the proof of Theorem 3.2). The proof, omitted here, is by reduction from the problem of testing if a 3NF formula is satisfiable, known to be NP-complete as well.

To complete the picture we can show that

PROPOSITION 3.8. *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:(Sketch) The NP algorithm is based on the observation that if $p(Selective(s, A, \pi))$ is not empty then there exists at least one EX-trace e of s, obtained from the root activity by a polynomial number of expansion steps, s.t. $p(e')$ is not empty, where e' is the selective EX-trace obtained from e by removing the activities in A and applying the renaming function π. Though, for space reasons, we do not give here a proof of this, the correctness stems from a "pumping" lemma, analogous to the one existing for context free string grammars [26].

Once this holds, the NP algorithm is simple - for each transitive compound activity in the pattern, guess an expansion sequence for the corresponding activity in the system. Then guess a mapping from the query pattern to (the obtained expansions of) the activities and verify that it satisfies the embedding requirements. ⊠

This also implies that testing for the emptiness of $p(semiNaive(s, \pi))$ and $p(Naive(s))$ is in NP.

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', (a renaming function π' and a activities set A'), we want to check if $p(Naive(s)) \subseteq Naive(s')$ (or, resp., $p(semiNaive(s, \pi)) \subseteq semiNaive(s', \pi'), p(Selective(s, A, \pi)) \subseteq Selective(s', A, \pi')$).

Observe that for a class X of trace types where inclusion is decidable (i.e. one can decide for each two BPs s_1, s_2 if $X(s_1) \subseteq X(s_2)$), the ability to perform type inference implies that type checking is also possible. We will see however that this is just a sufficient condition and in some cases type checking is possible when type inference is not.

First, we show undecidability for selective trace types.

THEOREM 4.1. *Given an EX-pattern p and source and target BP specifications, renaming functions and activity sets s, π, A and s', π', A', the problem of testing whether $p(Selective(s, \pi, A)) \subseteq Selective(s',\pi',A')$ is undecidable.*

Proof: The proof is by reduction from the problem of testing containment of context free (string) languages, known to be undecidable. Given two context free languages L, L', we construct, as in the proof of Proposition 2.12, s, A, π and s', A', π' such that $Selective(s, \pi, A)$ and $Selective(s', \pi', A')$ capture, resp., L and L'. 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 the paths (words) from the start to the end node of the root's internal trace, (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 $L \subseteq L'$. ☒

In contrast, type checking is decidable for naive and semi-naive trace types. To prove this, we start by defining an auxiliary class of semi-naive trace types called *deterministic*. Then we suggest a type checking algorithm for such trace types, and finally we show that we can translate every BP s and renaming function π to equivalent s' and π' w.r.t. which the set of semi-naive EX-traces is deterministic. Note that our proof technique is inspired by [19] that considers inclusion of parenthesis string languages. However, the case is more complex here as our proof must additionally account for the graph structure which is more complex than a string structure, as well as for the activities renaming function.

To define deterministic trace types we use the notion of *nodes origin*. 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, $\Pi(e) \in semiNaive(s, \pi)$. Clearly, there is at least one isomorphism from e to $\Pi(e)$ mapping activity pairs labelled a to activity pairs labelled $\pi(a)$. A node n_o in e that is mapped through such isomorphism to a node n in $\Pi(e)$ is called n's *origin*. 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 4.2. *For a BP specification s and activities renaming function π, the set of semi-naive traces $semiNaive(s, \pi)$ is called* deterministic *(w.r.t. s, π) if for every node n in it, its possible origins in $Naive(s)$ all have the same activity name.*

We can now show the following.

THEOREM 4.3. *Given an EX-pattern p, a BP specification s (with renaming function π), and a target BP specification s' (with renaming function π' s.t. $semiNaive(s', \pi')$ is deterministic), the problem of testing if $p(Naive(s)) \subseteq Naive(s')$ (respectively, $p(semiNaive(s, \pi)) \subseteq semiNaive(s', \pi')$) is decidable, and can be solved in EXPTIME.*

Proof: Note that naive trace types can be viewed as semi-naive trace types where π is the identity function and that such semi-naive traces are naturally deterministic. So it suffices to prove the theorem for semi-naive trace types.

The intuition behind the proof is as follows. We saw above that for every s, π and p we can derive s'' and π'' s.t. $semiNaive(s'', \pi'') = p(semiNaive(s, \pi))$. To prove the theorem we will construct, for every s'', π'' and s', π' where $semiNaive(s', \pi')$ is deterministic, an s''' and π''' such that $semiNaive(s''', \pi''')$ is empty iff $semiNaive(s'', \pi'') - semiNaive(s', \pi')$ is empty. Note that $semiNaive(s'', \pi'') - semiNaive(s', \pi') = semiNaive(s'', \pi'') \bigcap \overline{semiNaive}(s', \pi')$, where $\overline{semiNaive}(s', \pi')$ is the complement of $semiNaive(s', \pi')$, i.e. it contains all EX-traces that are *not* included in $semiNaive(s', \pi')$. Also note that, in fact, it suffices to consider a restricted portion of $\overline{semiNaive}(s', \pi')$ 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 word, 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.

The rest of the proof is dedicated to showing that semi-naive trace types are closed under intersection and (for deterministic semi-naive trace types) also under the above restricted form of complement, and that these may be effectively computed. We start by showing closure under intersection, then move to closure under (restricted) complement.

PROPOSITION 4.4. *For every pair of BPs s, s', we can compute a BP s'' s.t. $Naive(s'') = Naive(s) \cap Naive(s')$.*

Similarly, for every s, π and s', π' there exists s'', π'' such that $semiNaive(s'', \pi'') = semiNaive(s, \pi) \cap semiNaive(s', \pi')$.

Proof: Let us first consider naive traces. Let $s = (S, s_0, \tau)$, $s' = (S', s'_0, \tau')$ be two BP specifications. The required $s'' = (S'', s''_0, \tau'')$ is constructed as follows.

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 hasn't been treated, 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 ids) 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 proof for semi-naive traces follows similar lines. Two main differences are that (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 such activity (a, a') is computed as 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.

☒

We now consider (restricted) complement. We start by defining an auxiliary notion of *k-bounded EX-traces*. A k-bounded EX-trace is defined as a trace in which the size of the direct internal traces of each compound activity is bounded by k. Given a BP specification s with activities renaming function π, we use $\overline{semiNaive}_k(s, \pi)$ to denote the set of all k-bounded EX-traces that do not belong to $semiNaive(s, \pi)$. We shall construct a BP $\overline{s}$ with renaming function $\overline{\pi}$ s.t. $semiNaive(\overline{s}, \overline{\pi}) = \overline{semiNaive}_k(s, \pi)$. This construction may be used for any k.

PROPOSITION 4.5. *For every BP specification s and activity renaming function π, where $semiNaive(s, \pi)$ is deterministic, and for every k, there exists a BP specification $\overline{s}$ and a renaming function $\overline{\pi}$ s.t. $semiNaive(\overline{s}, \overline{\pi}) = \overline{semiNaive}_k(s, \pi)$.*

Observe that every BP specification is deterministic w.r.t to π that is the identity function. Thus, it follows that the same holds for naive EX-traces of any s.

Proof: 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”. The renaming function $\overline{\pi}$ maps a and $\overline{a}$ to $\pi(a)$, i.e. $\overline{\pi}(a) = \pi(\overline{a}) = \pi(a)$.

We construct $\overline{s} = (\overline{S}, \overline{s_0}, \overline{\tau})$ as follows. $\overline{S}$ is the set of all possible act-com 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” $\overline{a}$. We can show that $semiNaive(\overline{s}, \overline{\pi}) = \overline{semiNaive_k}(s, \pi)$, by induction on the nesting depth of traces.

⊠

Applying Theorem 3.2, for inferring type s'' corresponding to the query answer, then combining Proposition 4.4 with Proposition 4.5 (applied for k which is the size of the s''), we obtain an EXPTIME algorithm for type checking, when the target type is deterministic.

⊠

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 semi-naive EX-traces is deterministic.

THEOREM 4.6. *Given a BP s and an activities renaming function π, there exist a BP s' and a renaming function π' such that $semiNaive(s', \pi') = semiNaive(s, \pi)$ and $semiNaive(s', \pi')$ is deterministic w.r.t. s', π'.*

Proof:(Sketch) The idea is to create *equivalence classes* of activity names from s, that have the same set of semi-naive (sub) EX-traces rooted at them. As the trace types here are semi-naive, and not selective, each such equivalence class may only contain activities that are mapped by π to the same activity name. The grouping of activities to equivalence classes will ensure the determinism; to ensure that no new traces are created, we will require that each member of the equivalence class indeed had, in s, an implementation equivalent to *each* implementation of the equivalence class in s'; finally to ensure that no traces are lost we will consider only the *maximal* equivalence classes for which the requirements above holds. The details follow.

We shall use the notation Π over act-comp graphs, and 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 $g \in G$.

First, we group together all activities of s that are mapped by π to the same activity, obtaining a set γ of activity subsets. We say that each activities 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.

Each such subset A' forms an activity of s' and π' maps A' to its representing activity. We exploit the notation of A' to denote both the activity name, as well as the subset of activity names that it represents. 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' (representing a subset of the activities of$s$), $g' \in \tau(A')$ if and only if A' is the *maximal* set out of the sets in γ, that is an *equivalence class* with respect to g'. Also, with respect to each specific A' and keeping fixed all other activities in g', each **atomic** activity in g' is required to represent the *maximal* set of atomic activities out of these in γ.

To conclude we can show that the set of EX-traces stays intact and is deterministic w.r.t. the defined s', π'.

⊠

The immediate corollary of Theorems 4.3 and 4.6 follows.

COROLLARY 4.7. *Given an EX-pattern p, a BP specification s (with renaming function π), and a target BP specification s' (with renaming function π'), the problem of testing if $p(Naive(s)) \subseteq Naive(s')$ (resp. $p(semiNaive(s, \pi)) \subseteq semiNaive(s', \pi')$) is decidable.*

Our proof is constructive, and gives an EXPTIME type checking algorithm. It may be extended to an algorithm that checks equality to the target trace type, rather than just inclusion, in a fairly straightforward manner. For semi-naive trace types, we can show that the problem is CO-NP-hard in the size of the trace types. (The proof works by reduction from the the problem of testing unsatisfiability of 3NF formulas, known to be CO-NP-complete). For naive trace types it is open whether an algorithm polynomial in the size of input and output types exists. We already know by Proposition 3.7 that unless P=NP, an algorithm that is also polynomial in the size of the query does not exist.

5. EXTENSIONS

To conclude we consider some useful extensions to the query language that enhance the expressive power, and facilitate the querying of real life EX-traces.

Variables and joins. The first extension for EX-patterns that we consider is obtained by attaching *variables* to nodes and transitive edges, and introducing equality conditions (called *joins*) on these variables. The notion of *embedding* (from Definition 2.15) extends naturally to account for such variables: For node variables, we require that nodes labeled by equal variables are all mapped to EX-trace nodes having the same activity name; For transitive edges, if e_1 and e_2 are transitive edges to which equal variables are attached, we require that the sequence of node labels on the paths p_1, p_2 to which e_1 and e_2 are mapped (resp.) represent the same *word*. *Inequality* restrictions on variables are similarly defined.

An example for a query with variables and a join is depicted in Figure 8(a). Two of its transitive edges are annotated with the variables $\$x$ and $\$y$, and the join condition is $\$x = \$y$. The intuitive meaning is that the query searches for EX-traces that contain two paths, one from a C activity to an F activity and the other from a D activity to an F activity, s.t. the sequences of labels on the two paths form the same word. In the sequel, we will refer to the variables annotating nodes (transitive edges) as *node* (*path*) variables.

Results. Incorporating *node* variables in our type inference and type checking algorithms is straightforward. Theorems 3.2, 3.6 and Corollary 4.7 still hold for queries with joins on node variables. For queries with joins on *path* variables, however, type inference may no longer be possible, even when the input is of naive trace type. This is because a query that tests for equality of path variables may have for an answer a set of EX-traces that does not belong to `Selective`, as illustrated by the following theorem. The theorem also highlights the difficulty introduced by the use of transitive activities in the query.

THEOREM 5.1. *1. There exist a BP s and an EX-pattern p with join on path variables s.t. there is no BP s', renaming*

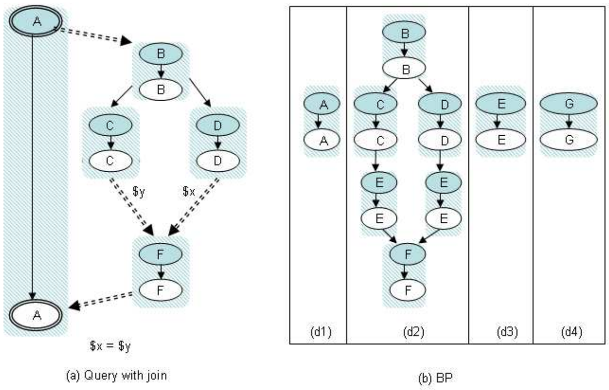

Figure 8: Query with variables.

function π' and set of activities A', where $Selective(s', A', \pi') = p(Naive(s))$.

2. *For EX-patterns with no transitive activities, type inference is possible for semi-naive (resp. selective) input and output types, even with joins on path variables. The time complexity here, in both cases, is exponential.*

Proof:(Sketch) We first consider part 1, then part 2. Consider again the EX-pattern p depicted in Figure 8(a), and the BP s whose act-comp DAGs are depicted in Figure 8(b). Assume that A is the root activity and has a single implementation - the act-comp graph $(d2)$, while E has two possible implementations - $(d3)$ and $(d4)$. Consequently, $Naive(s)$ contains EX-traces where the internal trace of the two Es (that follow C and D) includes an arbitrary number of nested E activities, ending by a G activity. Note that $p(Naive(s))$ includes only traces in which the number of nested E activities on both paths (the one following C and the one following D) is *identical*. It is easy to show that this set of EX-traces may not be captured by any selective trace type (as it requires correlation between activity expansions at distinct parts of the EX-trace).

The pattern p in the above example contains a transitive root node, hence the transitive edges in its internal trace are matched to paths that *zoom into* the nested activities of the EX-trace. We next show that for queries with no transitive nodes, type inference is still possible. The difference is due to the fact that in such queries, the direct internal trace of each compound node in the query is matched against the *direct* internal trace of the corresponding EX-trace node. Thus, the number of possible matchings is *finite*, and we may simply enumerate all matchings, selecting those that conform to the join criteria. However, for transitive edges, the number of possible matches (as the number of paths in a DAG) might be exponential, hence the exponential time complexity. ⊠

While such joins make type inference impossible/harder, type checking is still possible for the same cases as before.

THEOREM 5.2. *Type checking is possible for (semi)-naive input and output types, even with joins on path variables.*

Proof:(Sketch) We first ignore the joins and compute the semi-naive output type t for the BP and the pattern (without the joins). Next we compute (as in the proof of Corollary 4.7) a semi-naive type t' describing the EX-traces that belong to t but not to the required target type. To conclude we show that, for semi-naive trace types, it is possible to check if there exists some EX-trace in t' for which the result of the EX-pattern (with the joins) is not empty. The answer to the type checking problem is positive iff no such EX-trace exists. ⊠

Further extensions. To conclude this section, we briefly and informally discuss two additional extensions to the query language, namely *projection* and *negation*. We first describe them, then consider their effect on the results presented in the previous sections.

Projection. In a query with *projection*, the user is interested in projecting out only the matches to some part of the EX-pattern. These EX-pattern nodes and edges are marked as *projected*. The non-projected EX-pattern parts stand, in this case, as a constraint: the projected parts must co-appear with the non-projected parts in a qualifying trace.

Negation. In a query with *negation*, the patterns have some nodes and edges that are marked as negative. These again stand as constraints: the query searches for all traces that contain occurrences of the positive portions of the patterns, and does not contain *any* of the negative parts co-appear.

We can show that our results on type checking and type inference extend to EX-patterns with negation and projection. That is, Theorems 3.2 and 3.6 and Corollary 4.7 hold for the extended EX-patterns as well. For space constraints the type inference and checking algorithms that support these extensions remain outside the scope of this paper.

6. RELATED WORK

We next consider related work, highlighting our relative contribution.

The query language studied here is used in `BPQ`, a BP management system that allows to analyze BP specifications, monitor BP instances at run time, and query their execution traces [2, 3]. All these tasks are performed via a uniform, intuitive, graphical query interface, whose abstraction (for the case of traces analysis) was presented here. The complexity of query evaluation was studied in [10, 3]. The latter work highlights the value, for query optimization, of knowledge on the structure of the queried (sub-) traces, and motivated the present work.

There is a tight connection between the classes of EX-trace types studied here and corresponding classical classes of *string* and *graph* languages. We already mentioned the analogy between naive trace types and *bracketed* string languages [14], semi-naive trace types and *parenthesis* languages [19], and selective trace types and *context free* languages [26]. There is also a close connection between selective trace types and context free *graph* languages[16, 9]. These are defined using context free graph grammars, an extension of context free string grammars to graphs. They include graphs where some nodes represent non-terminals, and derivation rules allowing to replace them by graphs. A simple variant requires each graph to have single entry and exit nodes. It is easy to show that `Selective` is equivalent to the set of graphs defined by such grammars. However, to our knowledge, no model in this area that is equivalent to `Naive` or `semiNaive` has been studied before.

In terms of query languages, most of the work on context free graph grammars is concerned with formal logic, and specifically First and (Monadic) Second Order Logic (MSO). MSO is very expressive, and in particular may capture our query model. Courcelle [9] has shown that the theory of MSO over context free graph grammars is decidable. However, the complexity of the algorithms suggested in [9] is non-elementary in the size of the query. By choosing a restricted - yet expressive enough for our needs - query language, we are able to obtain more efficient, practical, type inference algorithms. As to type checking, in order to use the decidability results of Courcelle, it should first be shown that the set of semi-naive EX-traces of any given BP and renaming function may be expressed by MSO. (We know that this is not the case for selective trace types or else type checking would be decidable for them). Whether or not this holds is a subject of on-going research, but in any case, even if an algorithm that walks in this path exists, its complexity will be non-elementary in the size of the type specification (whereas the complexity of our algorithm is exponential).

Related models have been studied in the area of *verification*. We have mentioned above that selective trace types can be expressed by SERSM[1]. However, the query languages considered for SERSMs differ in expressive power from ours. For instance, graph homomorphism, which can be expressed in our query language cannot be expressed by the temporal logics often used in these works. Moreover, as the needs and motivations in verification are different, there are (to the best of our knowledge) no equivalences of the particular families of trace types studied here. We note that querying processes using formal languages representation was studied extensively, typically using temporal logics (see e.g. [11, 12, 7]).

Type checking and type inference are well studied problems in functional programming languages [22]. The complexity there is derived from the interaction of function types, polymorphism, and `let`-bindings; as pointed out in [8], this analysis is valuable for database queries as well. In our setting, the complexity is derived from the interaction of nested activities in the types with transitive edges/activities in queries. Type inference and type checking were also considered extensively in the context of XML. The XML analogous of the queries studied here are *XML selection queries* [20] that use tree patterns to select subtrees of interest. For such XML queries, [20] 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 (bearing the obvious distinction of having nested DAGs instead of flat trees), where type checking may be harder than type inference. Another common class of XML queries is *transformation queries* that restructure the input tree. [20] and [21] showed the impossibility of type inference and undecidability of type checking under a general model of transformations. Further work (e.g. [17], [13]) considered restrictive transformation models that allow decidability of type checking, with complexity ranging from PTIME to non-elementary. An interesting future work is to examine the adaptation of corresponding (restricted) transformation queries to our settings. Another aspect to study are extensions of the query language to consider additional information often available in EX-traces, such as the names and values of the activities data variables. It may be interesting to formulate practical restrictions on the use of data values that allow type inference and checking.

7. CONCLUSION

We studied in this paper type inference and type checking for queries over BP execution traces, modeled as nested DAGs. We formally defined and characterized three common classes of EX-trace types - naive, semi-naive, and selective. 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 have two main important practical implications. For type inference they 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. We thus believe it to be a firm basis for further study of type-based query optimization for BP management systems. On the other hand, for type checking, we showed that even for limited trace types, type checking takes EXPTIME (and is impossible for selective trace types). Consequently, we believe that run-time checks of the query result are likely to be a more useful in practice.

8. REFERENCES

[1] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4), 2005.

[2] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes. In *Proc. of VLDB*, 2006.

[3] C. Beeri, A. Eyal, T. Milo, and A. Pilberg. Monitoring business processes with queries. In *Proc. of VLDB*, 2007.

[4] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyen. Type-based xml projection. In *Proc. of VLDB*, 2006.

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

[6] BPMI. Business process management initiative. http://www.service-architecture.com/web-services/articles/business_process_query_language_bpql.html.

[7] T. Bultan, J. Su, and X. Fu. Analyzing conversations of web services. *IEEE Internet Computing*, 10(1), 2006.

[8] J. Bussche, D. Gucht, and S. Vansummeren. A crash course on database queries. In *Proc. of PODS*, 2007.

[9] B. Courcelle. The monadic second-order logic of graphs. *Inf. Comput.*, 85(1), 1990.

[10] D. Deutch and T. Milo. Querying structural and behavioral properties of business processes. In *Proc. of DBPL*, 2007.

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

[12] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. Verification of communicating data-driven web services. In *Proc. of PODS*, 2006.

[13] J. Engelfriet, H. J. Hoogeboom, and B. Samwel. Xml transformation by tree-walking transducers with invisible pebbles. In *Proc. of PODS*, 2007.

[14] S. Ginsburg and M. Harrison. Bracketed context-free languages. *J. Computer and System Sciences*, 1, 1967.

[15] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing xpath queries. *ACM Trans. Database Syst.*, 30(2), 2005.

[16] D. Janssens and G. Rozenberg. Graph grammars with node-label controlled rewriting and embedding. In *Proc. of COMPUGRAPH*, 1983.

[17] S. Maneth, T. Perst, and H. Seidl. Exact xml type checking in polynomial time. In *Proc. of ICDT*, 2007.

[18] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag, 1992.

[19] R. McNaughton. Parenthesis grammars. *J. ACM*, 14(3), 1967.

[20] T. Milo and D. Suciu. Type inference for queries on semistructured data. In *Proc. of PODS*, 1999.

[21] T. Milo, D. Suciu, and V. Vianu. Type checking for xml transformers. In *Proc. of PODS*, 2000.

[22] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

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

[24] Y. Papakonstantinou and V. Vianu. Dtd inference for views of xml data. In *Proc. of PODS*, 2000.

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

[26] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.