Querying Web-Based Applications 
Under Models of Uncertainty

Thesis submitted for the degree 
“Doctor of Philosophy” in Tel Aviv University 
Blavatnik School of Computer Science

By
Daniel Deutch

under the supervision of Prof. Tova Milo

Submitted to the Senate of Tel Aviv University

August 2010
Abstract

Web Applications that suggest some particular service, such as online shopping, e-commerce etc., are extremely popular nowadays. The core logic of these applications is typically captured by Business Processes (BPs). These processes usually operate in a distributed environment and the software implementing them is fairly complex. Thus, effective tools for analysis of such processes are important; these tools can allow to debug and optimize the processes, and to make an optimal use of them.

The goal of this thesis is to develop the foundations for the automated analysis of Business Processes (and thereby of the corresponding Web Applications). In general, this analysis bears two main flavors. First, analysts are interested in analyzing executions that occurred in the past, for instance to identify trends, to make sure that business logic is maintained, etc. We note that executions are usually logged, forming execution traces that are kept in a repository. Thus, analysis of past executions is translated into queries over traces. There are two challenges here: first, the size of a typical execution traces repository is extensively large, calling for query optimization techniques. Second, the traces often contain only partial information on the activities that were performed at run time, due to confidentiality, lack of storage space, etc. Thus query evaluation must be performed under terms of uncertainty.

The second flavor of analysis considers future executions. Namely, given the BP specification, this kind of analysis, again operating under terms of uncertainty, aims at predicting the behavior of future users, e.g. to characterize common behavior of users, or to identify executions where the total induced cost to the customer is the cheapest, etc. This type of analysis is in fact a top-k analysis, as it aims to find the k “best” possible execution flows, under some weight function over pos-
sible flows. The main difficulties here stem from (1) the fact that the number of possible execution flows of a given BP is typically very large, or even infinite in presence of recursion and (2) that the weights (e.g. likelihood, monetary cost, etc.) induced by actions performed during the execution (e.g. product purchase), may be inter-dependent (due to probabilistic dependencies, combined discount deals etc.).

To allow for such an analysis, four ingredients are required: first, one should define models for capturing Business Process specifications, their execution flows and traces. In this context we note that standards such as the recent BPEL [17] (Business Process Execution Language) standard facilitate the design, deployment, and execution of BPs; such standards are helpful as a foundation for our model, but more work is needed to set a full-fledged model that also accounts for partial information and uncertainty of the flavors depicted above. Second, one should design an intuitive query language that allows to easily form queries of interest over BP execution flows and traces. Third, the query language must be supported by efficient query evaluation algorithms, under terms of uncertainty. Finally, these theoretical foundations should be exploited for the development of practical implementations that can analyze real-life processes. This thesis considers each of the above mentioned ingredients, and suggests an integrated solution for them.
Acknowledgements

I had the great privilege of being a student of Prof. Tova Milo and I would like to express my deepest gratitude to her. Her brilliant advice, her priceless support, her dedication and her willingness to work day and night for the well-being and success of her students, were all invaluable for the creation of this thesis. Furthermore, her endless care and support is not limited only to research but extends to all areas of my life. I am also extremely thankful for the long hours that she has spent in reading drafts of this thesis.

The privilege of being a student of Tova has also the benefits of working in the wonderful Databases Lab that she has formed in Tel Aviv University. I would like to thank all (current and past) members of the Lab for making it such a fun place to work in, and for the fruitful cooperation.

I would also like to thank the Israel Ministry of Science for their generous support.

Last but not least, I would like to thank my parents for being such a great source of inspiration and help throughout my entire life, and to thank Yael for her remarkable care, support and assistance.
## Contents

1 **Introduction**  
1.1 Model ................................................. 4  
1.2 Querying Past Executions ............................... 6  
1.3 Querying Future Executions ......................... 8  
1.3.1 Finding Top-K Executions ....................... 8  
1.3.2 Evaluating Top-K Queries ....................... 10  
1.4 Practical Application ................................. 12  
1.5 Thesis Organization ................................. 14  

2 **Preliminaries** ........................................ 15  
2.1 BP Specifications .................................. 15  
2.2 EX-Flows .......................................... 18  
2.3 Queries .......................................... 21  
2.4 Complexity ........................................ 26  
2.5 Related Work ....................................... 26  

3 **Analysis of Execution Traces** ....................... 30  
3.1 Execution Traces .................................... 33  
3.2 Type Inference ...................................... 40  
3.3 Type Checking ...................................... 56  
3.4 Related Work ...................................... 64  

4 **Finding Top-K Future Executions** ................... 66  
4.1 Weighted Model ..................................... 69
Chapter 1

Introduction

Many businesses offer their services via web-based application interfaces. Examples of popular Web Applications include on-line shopping applications (e.g. Yahoo! Shopping [99]), and on-line travel agencies (e.g. Travelocity.com [94]). The business logic of such Web Applications is captured by a Business Process (BP for short) [11, 48]; Business Processes operate in a distributed environment that involve multiple entities, and are typically implemented by a software that is rather complex.

The popularity of Web Applications, along with the typically complex nature of their underlying processes, call for the development of analysis tools. Such tools can allow, for instance, to debug the process, to identify parts of the process that require optimization, or to find optimal ways for using the process.

In general, one can distinguish two categories of possible analysis tasks [34]. The first flavor of analysis considers past executions, and is used, for instance, to identify trends, to make sure that business logic was maintained, etc. Consider for example an on-line travel agency that suggests a variety of possible reservations of flights, hotels and car rental, and combinations thereof. The web-site owners may wish to analyze the navigation choices made by past users, for instance to verify that no such user has made a hotel reservation without relaying her
credit card details. Another example for such analysis seeks for combinations of flights and hotels reservation that were typically made by the same user, serving as a basis for the design of online advertisements. We note that it is a common practice for organizations to maintain a repository of execution logs (also referred to as execution traces); to analyze past executions, one should thus query the execution traces in this repository. The design of such query evaluation mechanism must face two challenges: first, the size of such execution traces repository is typically very large, and query optimization techniques are required to allow feasible evaluation time. Second, the traces often contain only partial information on the activities that were actually performed at run time; this is due to confidentiality, lack of storage space, etc. Thus, query evaluation must be performed under terms of uncertainty.

A second category of analysis tasks relates to future executions. Namely, given a specification of the Business Process, this kind of analysis aims to find the best possible execution flows, where the notion of “best” is determined by some weight function over the possible executions. Continuing with our example of the on-line travel agency, the weights here may correspond, for instance, to monetary cost (e.g., “find a navigation in the web-site that will lead to a purchase of a hotel and flight reservation deal of cheapest overall price”), or to likelihood (for instance, “find the most likely choice of hotel, for a user who makes a reservation of an Air France flight”), etc. This is in fact a top-k [45] problem, as we aim at finding the k best execution flows according to the given weight function. We note that this top-k analysis must also operate under terms of uncertainty, as future executions of the process are dictated by unknown external factors such as user choices, servers response time etc. Two main challenges rise in the analysis of future executions: first, the number of possible execution flows of a given Business Process is typically very large; it may even be infinite, when the Business Process specification is recursive (as a simple example,
users may navigate back and forth, an unbounded number of times, in the travel agency web-site). Second, the weights (e.g. likelihood, monetary cost, etc.) induced by actions made during the flow (e.g. product purchase), may be inter-dependent; this is due to probabilistic dependency for the case of likelihood (such as “users that prefer British Airways flights typically also prefer to stay at Holiday Inn”), combined discount deals (such as “a 10% discount upon a reservation of a British Airways flight along with a night at the Holiday Inn”) etc.

We note that there are two potential classes of “clients” for an analysis of the above flavors. At the side of the web-site owner, she may be interested in verifying that some business logic is kept, e.g. that no customer can make a travel reservation without logging-in with their credit card number; or in identifying the hotel that is most popular among customers choosing a certain airline. At the user side, she may e.g. wish to find the cheapest way to reserve a trip to Paris, or the most common choice of combined (flight and hotel) travel deals.

The goal of this thesis is to develop the foundations for the automated analysis of Business Processes (and thereby of the corresponding Web Applications). Our solution consists of the following ingredients.

1. We first design models for capturing Business Process specifications, their execution flows and the recorded execution traces, as well as an intuitive and generic query language, allowing to form queries over future execution flows as well as over recorded execution traces, in a generic manner. The designed models account for partial information and uncertainty of various flavors.

2. We then focus on the analysis of past process executions, stored as execution traces. We design effective algorithms for such analysis in presence of partial information over these traces.

3. We also design effective algorithms for analyzing future executions, accounting for the inherent uncertainty present in such
analysis. These algorithms are developed in two steps: first, we
design algorithms that identify the top-k (with respect to some
given weight function) executions of a given BP. Then, we consider
queries, that further restrict the results to the top-k execution
flows (or parts of them) that satisfy some user-defined criteria.

4. Finally, we exploit these theoretical foundations for the develop-
ment of a practical system, suggesting an effective analysis of the
Business Processes that underly real-life Web Applications.

We next provide an overview of our main contributions in each of
the above contexts. Each contribution is given here in an informal and
intuitive manner, and their exact descriptions appear in the following
chapters.

1.1 Model

We present a model of BP specifications, their executions, and queries
over which, in a graph-theoretic, abstract manner. This allows to form
a solid theoretical foundation, as well as, where applicable, to borrow
known results and adapt them to our settings. We next informally
explain these models; the formal definitions are given in Chapter 2.

BP specification The basic model that we use here for BP specifica-
tions is a natural abstraction of the BPEL standard (Business Process
Execution Language [17]) for BP specifications. The basic model and
query language, that do not account for partial information and uncer-
tainty, were introduced in [11, 43]; their extensions (described in the
following sub-sections) that do account for these, as well as all of the
subsequent results, are novel.

A BP is modeled as a set of DAGs (Directly Acyclic Graphs), each
consisting of activities (nodes), and links (edges) between them, that
detail the execution order of the activities. Each activity is modeled
by a pair of nodes, standing for its activation (start) point and its completion (end) point. Such node pair is referred to as an activation-completion (abbr. act-comp) node. Each such DAG has a unique activity pair, with no incoming edges, standing as its start activity and a unique activity pair, with no outgoing edges, standing as its end activity. Activities that are not linked via a directed path are assumed to occur in parallel. The DAGs are linked through implementation relationships; the idea is that an activity $a$ in one DAG is realized via the activities in another DAG. We call such an activity compound to differentiate it from atomic activities having no implementations. Compound activities may have multiple possible implementations, corresponding e.g. to different user choices, variable values, servers availability, etc. A unique DAG in the set, consisting of a single act-comp node, serves as the process root, standing for its starting point.

While each individual graph is a DAG, the implementation relation may induce cycles, e.g. for an activity $a$ in some DAG $G$, $G$ may appear as a possible implementation of $a$; alternatively, in some possible implementation $G'$ of the activity $a$, there may appear a compound activity $b$, having $G$ as a possible implementation, etc. In such cases we say that the BP is recursive.

**Execution Flows** Execution Flows (abbr. EX-flows) are actual running instances of the Business Process representing the execution of its activities. An EX-flow may be abstractly viewed as a nested DAG: starting from the BP root, for each compound activity, exactly one of its implementations is chosen at runtime; then, for each compound activity node $n$ in the chosen implementation, one of its implementations is chosen etc. We model this by connecting, via new edges called implementation edges, the start and end nodes of the chosen implementation graph, to the activation and completion nodes of the corresponding compound activity node.
Note that for any given BP specification $s$, the size of a single EX-flow of $s$ may be unbounded if $s$ is recursive; in particular, this means that recursive BPs may have infinitely many possible EX-flows.

Queries Queries are used to define EX-flows or parts of them, that are of interest to the analyst. They are defined using execution patterns (abbr. EX-patterns), generalizing EX-flows similarly to the way tree patterns, used in query languages for XML, generalize XML trees [24, 61]. In more details, EX-patterns bear the structure of an EX-flow, where activity names are either specified, or left open using a special \textit{any} symbol and then may match any node. 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 flow or for searching in any nesting depth, respectively. A match of the query is captured via the notion of an \textit{embedding}, which is a homomorphism from an EX-pattern to an EX-flow, respecting node labels and edge relation.

1.2 Querying Past Executions

The models and results reported in this thesis on analysis of past executions were published in [32], and are given in details in Chapter 3. We provide here a high-level description of the challenges and of our proposed solutions.

BP Management Systems allow to trace execution flows of Business Processes. Execution traces (abbr. EX-traces) obtained in such a manner are extremely valuable for companies, as their analysis allows to optimize business processes, reduce operational costs, and ultimately increase competitiveness.

We model EX-traces as nested DAGs, analogous to the model of EX-flows. In general, EX-traces may record the full and accurate information on the EX-flow. There are cases, however, where only partial
information about the activities is recorded. This may be due to lack of space [62], confidentiality [92] or other reasons. To capture the possible levels of recorded information, we define three classes of tracing systems: (i) naive tracing which accurately records all occurrences of all activities (ii) semi-naive tracing where the activation/completion events are all recorded, but possibly with only partial information about their origin activity, and (iii) selective tracing where only a selected subset of the events is recorded (again, with possibly partial information on the origin of the recorded activities).

We note that analysis of EX-traces repositories 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 [89]. Not surprisingly, type information, i.e. knowledge about the possible structure of the queried (sub-)traces, is valuable for query optimization [12]. Its role is analogous to that of XML schema for XML query optimization: it allows to eliminate redundant computations and simplify query evaluation. Such type information is readily available for the original traces, but not for the intermediary traces selected by queries. This calls for Type Inference. When the analysis tool expects a particular data type, we would also like to verify that the sub-traces selected by queries conform to the required type. Such Type Checking is thus a second challenge.

Results Both type inference and type checking are well studied problems for XML and tree-shaped graphs. We show 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.
In particular, 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. This signals 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. Type checking, on the other hand, incurs exponential data complexity for (semi-)naive trace types, and is undecidable for selective trace types. This indicates that static type checking is probably infeasible in this case, and calls for a different solution, e.g. run-time analysis [12].

1.3 Querying Future Executions

We then study the analysis of future executions. We will start by designing algorithms that identify the top-k executions of a given BP, then utilize them for top-k query evaluation.

1.3.1 Finding Top-K Executions

The results reported in this thesis on identifying top-k executions were published in [33], and are described in details in Chapter 4.

Given a BP specification \( s \), analysts are interested in the possible EX-flows of \( s \) that are “most important”, where the notion of importance is captured by some weighting metric (e.g. likelihood of choice, induced monetary price) that depends on the analysis goal. Following common practice, this analysis is referred to as top-k analysis.

To that end, we first extend the model of BP specifications and EX-flows to a weighted setting. We associate a guarding formula with each possible implementation of every compound activity name. The truth
value of this boolean formula intuitively dictates the choice of implementation at run-time. Furthermore, each such guarding formula (and consequently each implementation choice) is associated with a weight, and the weights along a given EX-flow are aggregated to form the EX-flow weight. Following common practice [45] in top-k algorithms, we assume monotonicity of the aggregation function with respect to the progress of the flow. This captures most practical scenarios, e.g. the total price of a shopping cart subset does not exceed the price of the full cart, even in the presence of discount deals.

We note that weights may be inter-dependent (for instance, due to probabilistic correlations when weights reflect choice likelihood, due to combined discount deals when they reflect monetary cost etc.). Consequently, we define three classes of weight functions: history-independent functions, where the weight of a single choice is independent of those that preceded it along the flow, bounded-history functions, where each such weight may depend on only a bounded (constant) number of preceding choices, and unbounded-history functions where a single weight may depend on unboundedly many previous choices (note that such unbounded dependency is possible only for recursive BPs, where the size of an individual EX-flow of a given BP is unbounded).

**Results**  We consider top-k analysis for weighted BP specifications, for each of the above classes of weight functions. For history-independent functions we provide a PTIME algorithm for top-k analysis. When the weight function is bounded-history, then our analysis algorithm is PTIME in the BP specification size, but with the exponent dependent on the history bound (i.e. “how many” previous choices affect the weight of a given choice). We further show that this is the best that can be done, unless P=NP. Last, we show that for unbounded-history weight functions, top-k analysis is generally impossible. Fortunately, studies [82] of the behavior of Web Applications and their users show
that the dependency of a choice on previously made choices is very “local”; i.e. each choice depends only on a small number of previously made choices. It follows that, in typical cases, the weight function is bounded-history with a small bound, rendering our algorithms efficient in practice.

1.3.2 Evaluating Top-K Queries

So far, we have considered the identification of top-k EX-flows of a given BP specification. In practice, analysts are only interested in a subset of the possible EX-flows, namely those that correspond to a given query; and they wish to restrict the top-k analysis to these EX-flows (or parts of). Our results on top-k query evaluation, for queries of varying expressive power, were published in [33, 35, 30] and are described in details in Chapter 5.

We study top-k query evaluation in a gradual manner. First, we consider a restricted version of our query language, referred to as selection queries. Selection Queries allow to specify EX-flows that are of interest, but does not allow to focus on a sub-flow of these. We then consider projection queries that further allow to focus on some parts of these flows, that are of particular interest.

An important question that rises when considering projection queries is the choice of a ranking metric for the query results. Recall that our model associates a weight with every possible EX-flow of the BP specification; with projection queries, a single projection results may originate from multiple EX-flows. In this case, the weights of all such EX-flows should then be aggregated, to form the result score. Evidently, different choices of such aggregation functions require different query evaluation mechanisms and incur different complexity of query evaluation. We study here the max and sum aggregation functions, and exemplify the applications of top-k query evaluation under both semantics.
Our Results  We start by studying the evaluation of top-k selection queries. We first consider the simple case where the weight function is history-independent, and provide an evaluation algorithm whose time complexity is polynomial in the size of the BP specification, and exponential in the size of the query (no PTIME algorithm with respect also to the query size exists, unless P=NP). When the weight function is bounded-history, then the PTIME algorithm goes through, with an added exponential dependency on the history bound (and again, this is the best that can be done unless P=NP). Our query evaluation technique combines the top-k algorithm from Chapter 4, with a variant of the Type Inference Algorithm given in Chapter 3.

Then we turn to projection queries. We start by considering the max aggregation function for such queries, and see that although there is a tight relationship between the semantics of selection queries and that of projection queries under max semantics (both searching for flows with maximal weight), new techniques need to be developed to provide efficient (PTIME) query evaluation for projection queries. Indeed, we show that the standard, common, use of selection as an intermediate input for projection yields in this case an exponential blow-up. However, we still provide a PTIME (in the BP specification size) algorithm (for the case of history-independent weight functions; the algorithm for bounded-history function follows, similarly to the case of selection queries) that constructs a compact representation of the top-k projections, avoiding materialization of the intermediate (full) qualifying flows.

We then show an application of our algorithm (originally analyzing future possible EX-flows) for the analysis of EX-traces, as follows. Given a partial (semi-naive, selective) EX-trace, analysts often wish to understand what had actually happened at run-time. In other words, they would like to identify the execution flows that are most likely to have been the origin of this log, typically focusing of specific parts of
the log that are of particular interest [49, 14]. We show that our query evaluation algorithms can be utilized for a PTIME (with respect to the BP specification size) algorithm that retrieves the most likely origins of a given (sub)trace; interestingly, we may devise such an efficient algorithm even if the tracing system used is also unknown.

The second aggregation function that we study is \textit{sum} aggregation. The sum semantics is a common semantics for projection queries in \textit{probabilistic} context, i.e. when the weight function reflects choice likelihood. This semantics is employed also for probabilistic XML and probabilistic relational DB [65, 91, 27, 83, 54, 46, 90, 66, 8]. Yet, we will show below that in the BP context it makes query evaluation \textit{computationally much harder}, compared to \textit{max} semantics. In particular, we show that exact computation of the score of a given projection result (defined as the sum of weights of all flow in which the projection result appears as sub-flow) cannot be performed (as this score may be irrational). Under reasonable assumptions, we show that the top-k projection results may still be exactly identified, by approximating scores of results, but their identification may incur EXPTIME (in the BP specification size).

1.4 Practical Application

We have developed a practical application of the theoretical results explained within this thesis, by designing \textbf{ShopIT} (ShoppIng assitanT), a system that assists on-line shoppers by suggesting the most effective navigation flows for their specified criteria and preference. The system was demonstrated in a real-life context of an online computer store [36], and is explained in details in Chapter 6.

Consider an on-line store that allows users to assemble computers from a variety of component parts. The store offers various processors, motherboards, screens etc. Consider a user that is interested in buying
a cheap computer with an Intel processor. Suppose that the user can get a good price by first registering to the store customers club, then passing through some advertisement page that provides members of the club with discount coupons, and finally buying a certain set of components (including a certain Intel processor) that, when purchased together with the above coupons, yields the cheapest overall price. Clearly, the user might be interested in knowing this information if she is looking for the deal with the best price. Alternatively, the user may prefer combinations where the delivery time is minimal, or may want to use the experience of others and view the most popular navigation paths (and purchases thereof), assembling an Intel-based computer.

This is where ShopIT comes into play. We first model, as a weighted BP, the store along with the details of its products. These details include in particular the individual products prices, shipping time etc., as well as details on combined deals, products that may be delivered together, etc. These are compiled into different weight functions.

When the user starts her navigation in the site, she may specify her constraints – compiled into a query – and her weight function of interest (e.g. monetary cost, popularity, products shipping time, etc.) and have the system perform a top-k query evaluation, using our developed algorithm. The output of this algorithm is a set of top-k ranked execution flows of the process, out of these conforming to the constraints. Each such execution flow is presented to the user as a recommendation on how to navigate within the web-site. The user then continues her navigation taking into account the presented recommendations, but may also make choices different than those proposed by the system, in the latter case ShopIT adapts its recommendations to the actual choices made by the user.

To examine its applicability, we have demonstrated the operation of ShopIT over part of Yahoo! Shopping [99] Web-site, and showed that it performs well in practice.
1.5 Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2 we provide some basic definitions that are used throughout the thesis, and accompany each such definition with an intuitive example. In Chapter 3 we study the analysis of past execution traces, and in Chapter 4 we study top-k analysis over possible future executions. We incorporate top-k queries in Chapter 5, and show their application in the context of analyzing future executions as well as past executions. The ShopIT system is depicted in Chapter 6. Towards the end of each Chapter we discuss relevant related work (related work for Chapters 4 and 5 is discussed towards the end of Chapter 5). We conclude the thesis in Chapter 7.
Chapter 2

Preliminaries

We provide in this Chapter the formal definitions for our basic model of Business Processes (BPs) and queries over which. We accompany each definition with an intuitive example.

2.1 BP Specifications

At a high-level, a BP specification encodes a set of activities and the order in which they may occur. The type of activities and their order depend on the business goals that the BP aims to achieve.

We note that in real-life, the operation of activities is not instantaneous, but rather have distinguished activation and completion points. To model this, we first define the auxiliary notion of an activation-completion DAG which represents a group of related activities and their ordering. In the sequel, let $\mathcal{N}$ be a domain of graph nodes and $\mathcal{A}$ be a finite domain of activity names. We also use two distinguished symbols $\text{act, com}$, denoting, resp., activity activation and completion.

Definition 2.1.1 An activities DAG is a tuple $(N, E, \lambda)$ in which $N \subset \mathcal{N}$ is a finite set of nodes, $E$ is a set of directed edges with endpoints in $N$, $\lambda : N \rightarrow \mathcal{A}$ is a labeling function for the nodes, labeling each node by an activity name. The graph is required to be acyclic, and to have a single start node without incoming edges, and a single end node.
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 $(a, \text{act})$ and $(a, \text{com})$. All incoming edges of $n$ are directed to $n'$ and all of the outgoing edges of $n$ now originate at $n''$. A single edge connects $n'$ to $n''$.

**Example 2.1.2** To illustrate, let us consider the act-comp DAG $s_2$ in Figure 2.1 (ignore for now the bubbles depicted in the Figure). It contains four activities (Search, Hotel, Flight and Print), each represented by a pair of nodes. In each pair, the first (second) node denotes the activation (completion) point of the activity.

A BP specification represents a set of act-comp DAGs that are linked through implementation relationships. The idea is that an activity $a$ in one act-comp DAG is realized by the activities in another act-comp DAG. Conceptually, the activities of the latter are inserted in-between the activation and completion nodes of $a$. We call such an activity compound, to distinguish it from an atomic activity which does not have any possible implementation. To that end, among the activity names in $A$ we distinguish two disjoint subsets $A = A_{\text{atomic}} \cup A_{\text{compound}}$, representing atomic and compound activities, resp. Compound activities can have multiple possible implementations, one of which will be chosen at run-time.

**Definition 2.1.3 (BP specification)** A BP specification $s$ is a triple $(S, s_0, \tau)$, 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 $\tau : A_{\text{compound}} \rightarrow 2^S$ is the implementation function, mapping each compound activity name in $S$ to a set of act-comp DAGs from $S$. For a compound activity $a$, $\tau(a)$ is referred to as the set of possible implementations of $a$. 
We note that a BP specification may be recursive, e.g. an implementation of some compound activity in an act-comp DAG $s_i$ may be $s_i$ itself.

**Example 2.1.4** Consider an online travel agency that offers various travel deals, including suggestions of flights, hotels and car rentals. The BP specification $s$ depicting the travel agency business logic consists of the act-comp DAGs in Figure 2.1. The compound activities here are Trip, Luxury, Hotel, Flight, LuxHotel and LuxFlight (and all other activities are atomic). $s_0$ is the root of the BP, and the possible implementations of compound activities are depicted as bubbles; for instance, the possible implementations of the Trip activity are $\{s_1, s_2\}$.

The intuitive interpretation of this specification is as follows. The user starts searching for a trip with the Trip activity, at $s_0$. She may then choose between a regular trip (leading to $s_2$), or a luxury trip (leading to $s_1$, that in turn leads to $s_3$). Then, she searches and reserves (luxury) flights and (luxury) hotels, and is required to pay for each, using one of the credit activities, that perform billing. For regular (i.e. non-luxury) hotels and flights, the user is directed to one billing system,
represented by Credit1. To pay for luxury hotels and flights, which are more expensive, the user is directed to a different billing system, named Credit2 (which may be, for example, more secure).

2.2 EX-Flows

An EX-flow represents the execution of activities from a BP. It may be abstractly viewed as a nested DAG, containing node-pairs that represent the activation and completion of activities, and edges that represent the ordering among activities and any implementation relationships. Of course, the structure of an EX-flow DAG must adhere to the structure of the corresponding BP specification, i.e., activities have to occur in the same ordering and implementation relationships must conform to the function \( \tau \).

Definition 2.2.1 Given a BP specification \( s = (S, s_0, \tau) \), \( e \) is an Execution Flow (EX-flow) of \( s \) if:

1. Base EX-Flow: \( e \) consists only of the root \( s_0 \) of \( s \), or,

2. Expansion Step: \( e' \) is an EX-flow of \( s \), and \( e \) is obtained from \( e' \) by attaching to some activity pair \((n_1, n_2)\) of \( e' \), labeled by some compound activity name \( a \), some implementation \( e_a \) of the activity \( a \), through two new edges, called implementation edges, \((n_1, \text{start}(e_a))\) and \((\text{end}(e_a), n_2)\). We require that this \((n_1, n_2)\) pair does not have any implementation attached to it already in \( e' \), whereas all its ancestor compound activities in \( e' \) do have one. We call \( e \) an expansion of \( e' \), denoted \( e' \rightarrow e \).

We use \( e' \rightarrow^* e \) to denote that \( e \) was obtained from \( e' \) by a sequence of expansions. \( e \) is called a full EX-flow if it cannot be further expanded and a partial EX-flow otherwise. An activity pair in a partial EX-flow \( e \) is called unexpanded if it is not the source of any implementation edge. This implies that such an activity pair may be a candidate
activity for further expanding e according to the inductive rule of the
definition. The set of all full flows defined by a BP specification s is
denoted flows(s).

For a graph e, we say that e is an EX-flow if it is a (partial or full)
flow of some BP specification s.

In the sequel, a subgraph $g_2$, connected as in Item 2 of Definition
2.2.1, by implementation edges, to an activity pair $(n_1, n_2)$, along with
the implementation edges themselves, is called a direct internal flow of
the activity.

We note that, for recursive BPs, an EX-flow of the BP may be
of unbounded size, consisting of an unbounded number of recursive
choices. Consequently, the number of (full) EX-flows of such BP is
infinite.

**Example 2.2.2** Some example EX-flows are depicted in Figure 2.2.
Of these, we focus here on (a) and (b) which are EX-flows of the BP
specification in Fig. 2.1 (we consider the other EX-flows below, in
Section 3). Let us focus first on the EX-flow in Fig. 2.2(a): it de-
tails a possible execution of our online travel agency. Starting from the
root activity Trip of the BP specification, for each compound activity
node, exactly one implementation is chosen. Implementation edges are
marked as dashed arrows, and following them reveals the internal flow
of the chosen implementation of each corresponding compound activity.
For each such compound activity, one implementation edge originates
at its activation node and points at the start node of another (possi-
bly nested) act-comp DAG $g$, and another outgoes the end node of $g$,
pointing to the activity completion node. For example, viewing the im-
plementation of 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.
Viewing the chosen implementations of both reservation activities re-
Figure 2.2: EX-flows.
veals some credit limit check, namely Credit1. The EX-flow in Figure 2.2(b) is another possible EX-flow 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, but luxury hotels and flights are reserved (via the LuxHotel and LuxFlight activities) and a different type of credit check, Credit2, (intuitively providing a higher credit limit check), is performed.

**Sub-Flows** We use the term “An EX-flow rooted at (a compound activity) A” to denote the nested DAG obtained by treating A as the root activity and following expansion steps as defined in Definition 2.2.1. Furthermore, for an EX-flow e and a compound activity node n within e, we say that the sub-graph of e appearing in-between the activation and completion nodes of n (i.e. the sub-graph consisting of all nodes and edges such that there exists a directed path from the activation node of n to them, and there exists a directed path from them to the completion node of n) is the sub-flow of e rooted at n.

2.3 Queries

Queries are defined using execution patterns (abbr. EX-patterns). Such EX-patterns generalize EX-flows similarly to the way tree patterns generalize XML trees. EX-patterns are EX-flows where activity names are either specified, or left open using a special \textit{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 flow or for searching in any nesting depth, respectively.

**Definition 2.3.1 (Execution Patterns)** An execution pattern, abbr. EX-pattern, is a pair \( p = (\hat{e}, T) \) where \( \hat{e} \) is an EX-flow whose nodes are labeled by labels from \( \mathcal{A} \cup \{\text{any}\} \), and \( T \) is a distinguished set of activity
pairs and edges in \( \hat{e} \), called transitive activities and edges, resp.

**Example 2.3.2** An example EX-pattern is depicted in Fig. 2.3 (a). The pattern seeks for all possible EX-flows where the user may make a reservation of a trip, and finalize the reservation by paying. The query looks, visually, very similar to an EX-flow: 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-flow of interest, in-between the Trip and the Credit nodes.

Observe that apart from the transitive nodes and edges, EX-patterns and EX-flows look similar, making the formulation of queries rather intuitive.

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

**Definition 2.3.3 (Embedding)** Let \( p = (\hat{e}, T) \) be an EX-pattern and let \( e \) be an EX-flow. An embedding of \( p \) into \( e \) is a homomorphism \( \psi \) 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. The root activity pair of \( p \) must be mapped to the root activity pair of \( e \).

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 flow of a transitive activity, the edge (edges on the path) from \(\psi(m)\) to \(\psi(n)\) can be of any type (flow, or implementation) and otherwise must have the same type as \([m, n]\).

The result defined by \(\psi\) is the image of \(p\) in \(e\) under \(\psi\). I.e. it consists of all nodes and edges of \(e\) (appearing in some path) to which some node or (transitive, resp.) edge of \(p\) is mapped. 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-flow \(e\), 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-flows, we will also use \(p(E)\) to denote the set of all possible outputs of \(p\) when applied on EX-flows in \(E\), namely \(p(E) = \bigcup_{e \in E} p(e)\).

We can show that \(p(e)\) defined as above (and consequently also \(p(E)\) for a set \(E\) of EX-flows) is a set of EX-flows, as follows.

**Theorem 2.3.4** Given an EX-pattern \(p\) and an EX-flow \(e\), \(p(e)\) is a set of EX-flows.

**Proof.**

Given an EX-pattern \(p\) we define the nesting depth of a node \(n\) in \(p\) as the maximal number of implementation edges on a path from the root of \(p\) to \(n\), and the nesting depth of a pattern \(p\) as the maximal nesting depth of a node, over all nodes in \(p\).

Let \(\psi\) be an embedding of \(p\) in \(e\), and let \(e'\) be the image of \(p\) in \(e\) under \(\psi\). We claim that \(e'\) is an EX-flow, and we show it by induction on the structure of \(p\), as follows. If \(p\) has a nesting depth of 0, i.e. it consists only of a single activity pair (the root), then so does \(e'\) (an activity pair of \(p\) may only be mapped to a single activity pair of \(e\)); this means that \(e'\) is an EX-flow (see the case of base EX-flow in Def. 2.2.1).
Otherwise, let $p$ be a pattern of nesting depth $d + 1$, and let $p'$ be the pattern obtained from $p$ by removing the implementation graphs (denoted $p_1, ..., p_k$) of all compound activity nodes (denoted $n_1, ..., n_k$, correspondingly) of nesting depth $d$. Let $\psi'$ be the restriction of $\psi$ to the nodes and edges of $p'$; it follows that $\psi'$ is an embedding of $p'$ in $e$, and let $e''$ be the image of $p'$ in $e$ under $\psi'$. Following the inductive assumption, $e''$ is an EX-flow; $e'$ consists of all nodes and edges of $e''$, along with all nodes and edges of $e$ to which nodes and edges in $p_1, ..., p_k$ are mapped to by $\psi$. We show that $e'$ is an EX-flow.

Let $i \in \{1, ..., k\}$. First, assume that $n_i$ is non-transitive. Then the implementation edge $I_p (I'_p)$ whose origin (destination) is the activation (completion, resp.) node of $n_i$ must be mapped by $\psi'$ to the implementation edge $I_e (I'_e)$ whose origin (destination) is $\psi'(n_i) = \psi(n_i)$, and the nodes and edges of $p_i$ must be mapped to some nodes and edges in the direct implementation of $\psi(n_i)$. As $p$ has the shape of an EX-flow, $p_i$ bears a unique start node $\text{start}(p_i)$ (which is the destination of the edge $I_p$), and a unique end node $\text{end}(p_i)$ (which is the origin of the edge $I'_p$). $\text{start}(p_i) \ (\text{end}(p_i))$ must be mapped to a unique node which is the
destination (origin) of the edge $I_e (I'_e)$. Consequently, the embedding result corresponds to the addition of an implementation (with a unique start and end nodes, as required) to $\psi(n_i)$, in $e''$, i.e. constitutes an EX-flow.

If $n_i$ is transitive, we claim that $I_p$ must still be mapped to (a path whose first edge is) $I_e$ and $I'_p$ must still be mapped to (a path whose last edge is) $I'_e$; to observe that this is correct, note that the mapping must contain a path from the activation node to the completion node of $n_i$; such path may be empty (in which case $p_i$ must also be empty, and the result of $\psi$ is the same as $\psi'$), but otherwise must start with $I_e$ and end with $I'_e$. Now $\text{start}(p_i)$ must be mapped to an end-point of the edge (or path) to which $I_p$ is mapped. Thus, in $e''$, $\psi'(\text{start}(p_i))$ (or the first node on the path $\psi'(\text{start}(p_i))$, resp.), serves as a unique start node for the implementation of $\psi(n_i)$; and similarly, $\psi'(\text{end}(p_i))$ (or the last node on the path $\psi'(\text{end}(p_i))$, resp.), serves as a unique end node for the implementation of $\psi(n_i)$. Consequently $e''$ is an EX-flow.

Example 2.3.5 For example, let us now match the EX-pattern in Fig. 2.3 (a) to the EX-flow in Figure 2.2(c). Two embeddings are possible here, yielding the results in Figure 2.3 (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 (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 of Trip (while in the given EX-flow it appears only deeper in the implementation hierarchy). Finally, observe that since an embedding is a homomorphism, multiple query nodes may be mapped to the same EX-flow nodes and edges (or paths).
Similarly, queries may be defined over BP specifications. The semantics is that the query result is the set of all results of embedding the query in some possible full EX-flow of the BP specification.

**Definition 2.3.6** Given an execution pattern $p$ and a BP specification $s$, we define the result of evaluating $p$ over $s$, denoted by $p(s)$, as the set of all EX-flows in $p(flows(s))$.

**Observation** If the BP specification $s$ is recursive, the set $p(s)$ may be of infinite cardinality.

### 2.4 Complexity

We briefly recall some common notions employed for the analysis of computational complexity in the context of Database Research.

When analyzing the complexity of query evaluation, one may consider data complexity and combined complexity [97]. Data complexity is the complexity of evaluating a query as a function of the database size, while combined complexity is a function of both the database and the query sizes. In our case, the “data” is typically the given BP specification while the query is the EX-pattern, and the notions of data complexity and combined complexity are adapted correspondingly.

### 2.5 Related Work

We have used nested DAGs as a model for the Business Processes underlying Web Applications, and execution patterns as a basis for a query language over executions of such processes. We first explain the origin of this model, namely the BPEL standard for specification of operational logic of processes; then we discuss additional models for process specification and querying.
The BPEL standard  The model of Business Processes used here is based on an abstraction of the BPEL standard (Business Process Execution Language [17], also identified as BPELWS or BPEL4WS). The BPEL standard was jointly developed by BEA Systems, IBM, and Microsoft, combines and replaces IBM Web Services Flow Language (WSFL) [70] and Microsoft XLANG [98]. It provides an XML-based language to describe not only the interface between the participants in a process, but also the full operational logic of the process and its execution flow. Commercial vendors offer systems that allow to design BPEL specification via a visual interface, using a conceptual, intuitive view of the process, as a nested graph. Designs are automatically converted to BPEL specifications, and can be automatically compiled into executable code that implements the described BP [80]. The analysis of these processes is the missing part in such practical systems: while the importance of query languages for business processes had been recognized by BPMI (the Business Process Management Initiative) [19], no draft standard has been published since. To that end, we complement in this thesis the design of Business Processes with strong algorithmic tools that allow for effective analysis of the process and of its executions, under terms of uncertainty. We note that while a first version of the model and query language presented here was suggested in [10, 11, 43], this version did not account for uncertainty; consequently the refined model that accounts for uncertainty and all results in the following Chapters, are novel.

Additional Common Process Models and Query Languages  We next overview additional models and query languages for processes.

Many works on process analysis studies process models that constitute variations of finite state machines (FSM) [93]; an FSM is simply an edge-labeled directed graph, where the nodes represent states, and the edges represent transitions. The labeling of the edges may represent
conditions for the transition to occur, actions to execute upon transitions, or both. FSMs can be used to describe simple “flat” processes, but lacks the expressive power to capture the nested and possibly recursive nature of some real-life processes. Finite State Machines were extended in [56] to include hierarchy in-between states, in the form resembling non-recursive functions, but again it lacks the expressive power to express the families of Execution Flows defined by a BP in our model. A recursive state machine (RSM) [13, 7] is a further extension of FSM, introducing the definition of a node implementation, which may possibly be recursive. In a way very similar to BPs, an implementation of a label of a RSM node is itself a RSM. An expansion of the state machine is defined as replacing a node with an implementation of its label, and connecting the implementation as a subgraph into the original graph. Some variants exist on the connection pattern, where the simplest version requires each implementation to have a single entry and a single exit nodes. This restriction of the model is called Single Entry Single Exit RSM (SRSM), and we have shown in [31] a syntactic equivalence between SRSMs and BPs.

The common approach in analysis of such processes is to use a query language based on temporal logic [72], e.g. LTL, $CTL^*$ or $\mu$-calculus (these differ from each other in terms of expressive power, see e.g. [72]). Temporal Logics are bisimulation-invariant, intuitively meaning that they care only for the process “behavior” rather than its structure (see [31] for exact notions). In contrast, our query language takes a database approach and allows to further query the structure of the process as well as the structure of its possible executions. Indeed, we have shown in [31] that some queries expressible in our query language are not expressible in temporal logic (and vice versa). Specifically, the database-style analysis (selection, projection, and top-k queries) that we study in the sequel was not studied in this context, nor in the context of the models discussed below (to the best of our knowledge).
Taking a database approach for the modeling and analysis of processes was also studied in various contexts, as follows. Active XML (AXML) extends XML with embedded invocations of Web Services [5], 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 (following the tree structure of XML), rather than a DAG as assumed here. Also, most of the work on analysis of AXML documents has focused on “static analysis”, i.e., the matching of XML tree patterns to the possible states of the AXML document; to the best of our knowledge, an analysis of the sort studied in this thesis (i.e. analysis of execution traces and top-k analysis of future executions) were not studied in the context of AXML. The analysis of interactive data-driven Web applications [37, 21] focuses on the combination of flow with the data that is transferred in-between applications; similarly, the analysis of Business Process artifacts [57] focuses on the data manipulated by the process. As a consequent of these analysis needs, the query language proposed in [37] (LTL-FO) combines temporal logic queries for the flow with First Order Database-like queries for the underlying data. Our query language is weaker in terms of expressive power than LTL-FO 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 over Web Applications, as exemplified in the sequel.
Chapter 3

Analysis of Execution Traces

BP Management Systems allow to trace execution flows of Business Processes. Execution traces (abbr. EX-traces) obtained in such a manner are extremely valuable for companies, as their analysis allows to optimize business processes, reduce operational costs, and ultimately increase competitiveness. In this Chapter we study the analysis of such execution traces. The model and results reported in this Chapter were published in [32].

We start with an informal presentation of the models and tools presented in this Chapter, and then we turn to the formal presentation.

EX-Traces. For each activity issued within an EX-flow, its activation and completion events are reported by BP management systems. 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 EX-flow.

Partial Tracing. Execution Traces may record the full and accurate information on the EX-flow. There are cases, however, where only par-
tial information about the activities is recorded. This may be due to lack of space [62] - when maintaining a repository of all traces accumulated for runs of a large-scale process, the overall size of traces may be excessively large. Another reason is confidentiality [92]; for example, the Web-based travel agency presented in Chapter 2 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.

We model this by distinguishing 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) seminaive 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, again possibly with partial information about their origin activity.

**Type Information.** In typical cases, queries used to analyze EX-traces are issued over a large repository of such traces. When evaluating such a query, type information, i.e. knowledge about the possible structure of the queried (sub-)traces, is valuable for query optimization [12]. 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 in the repository contain an occurrence of A, and then an optimized query evaluation, checking only for the existence of the activity B, is possible. Similarly, if the type tells us that no trace contains an occurrence of 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 [89]. Type information is readily available, as the BP 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-step query. One of our goals, thus, is to develop efficient algorithms for deriving this type information (namely to perform Type Inference, thereby facilitating 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 in this Chapter.

**Our results.** 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 3.4).

In particular, we show that type inference is possible 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 for naive tracing. For type checking, we show that it may incur time exponential in the size of the input type for (semi-)naive trace types, and is undecidable for selective trace types.
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 may only be solved in EXPTIME (and is impossible for selective trace types). Consequently, we believe that run-time checks of the query result (see e.g. [12]) are likely to be a more useful in practice.

We start by formally defining execution traces of various detail level, capturing the output of real-life tracing systems, and finally we consider analysis of such traces.

### 3.1 Execution Traces

As EX-flows, an execution trace (abbr. EX-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. However, as explained above, traces may vary in the amount of information that they record on the EX-flow. To that end, we next introduce three classes of tracing systems: Naive, Semi-Naive, and Selective, as follows. We start by introducing the set $\mathcal{EX}$ of *all* execution traces (regardless of the BP specification they originated from).

**Definition 3.1.1** The set $\mathcal{EX}$ of execution traces is the smallest (in terms of set inclusion) set of graphs $\{g \mid \exists s.g \in \text{flows}(s)\}$.

The simplest form of tracing is *naive tracing*. Such tracing systems record all activities that occurred at run-time, with the correct name of each activity and the exact order in-between activities.
Definition 3.1.1 [Naive EX-traces] Given an execution flow $e$, the naive trace of $e$ is simply $e$. Consequently, the set of naive traces of a BP specification $s$, denoted $\text{naive}(s)$, is defined as $\text{naive}(s) = \text{flows}(s)$.

Example 3.1.2 Figures 2.2 (a) and (b) (in Chapter 2), viewed in Example 2.2.2 as EX-flows, are also the naive traces obtained when tracing the course of these execution flows.

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 Web-based travel agency (introduced in Chapter 2) 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 this case, rather than labeling the EX-trace activity nodes by their actual activity names $\text{Credit1}$ and $\text{Credit2}$, we would like to label them by a generic $\text{Credit}$ label. This is captured by the following definition of semi-naive EX-traces (recall that $\mathcal{A}$ stands for the domain of all possible activities names).

Definition 3.1.3 [Semi-naive EX-traces] Given a BP specification $s$ and a renaming function $\pi$ from activity names in $s$ to activity names in $\mathcal{A}$, the set of semi-naive EX-traces defined by $s$ and $\pi$, denoted $\text{semiNaive}(s, \pi)$, consists of all the EX-traces $e$ obtained from the naive EX-traces $e' \in \text{Naive}(s)$ by replacing each label $a$ in $e'$ by $\pi(a)$.

Example 3.1.4 To continue with our running example, Figures 2.2 (c) and (d) are semi-naive EX-traces of our online travel agency BP, for a renaming function $\pi$ s.t. $\pi(\text{Credit1}) = \pi(\text{Credit2}) = \text{Credit}$, and where $\pi$ is the identity function for all other activities. These two semi-naive EX-traces are obtained resp. from the naive EX-traces in Figure 2.2 (a) and (b). One viewing the EX-trace may identify that some credit checks were issued, but not which ones.
Semi-naive EX-traces can be viewed as the BP analog of XML trees defined by DTDs with specialization [81]. 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 non-interesting for the business analysts and may be omitted to avoid overloading the logs with redundant information [18]. To model this kind of tracing, we introduce the notion selective EX-traces.

**Definition 3.1.5 [Selective EX-traces]** Given a BP specification \( s \), a set \( A \) of activity names in \( s \), satisfying condition (*) below, and a renaming function \( \pi \) from activity names in \( s \) to activity names in \( A \), the set of selective EX-traces defined by \( s \), \( A \) and \( \pi \), 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 2.1.1), or is empty.

The intuition behind condition (*) in definition 3.1.5 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,

\(^1\)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.
otherwise the loss is easily observable. For instance, removing also the Search activity-pair from the EX-trace in Fig. 2.2(c) 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.

**Example 3.1.6** 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 renamed, but also the LuxHotel and the LuxFlight, and the record of the Luxury activity should be omitted altogether. Here $A = \{\text{Luxury}\}$, and $\pi(\text{Credit1}) = \pi(\text{Credit2}) = \text{Credit}$, $\pi(\text{LuxHotel}) = \text{Hotel}$, and $\pi(\text{LuxFlight}) = \text{Flight}$.

Figure 2.2(e) shows the selective EX-trace obtained from the naive EX-trace of Figure 2.2(b). Note that the same graph (up to node isomorphism) is also the selective EX-trace obtained from the naive EX-traces in Figure 2.2(a). Thus, given such a selective trace, there is no way to identify the naive EX-trace (of regular or luxury trip reservation) that is its origin.

Each BP specification $s$, (renaming function $\pi$, and activities set $A$) defines a (possibly infinite) set of EX-traces $E = \text{Naive}(s)$ (resp. $E = \text{semiNaive}(s, \pi)$, and $E = \text{Selective}(s, A, \pi)$). Let $\text{Naive}$ (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 $\pi$, and an activities set $A$). More formally, \( \text{Naive} = \{ E \mid \exists s \text{ s.t. } E = \text{Naive}(s) \} \), \( \text{semiNaive} = \{ E \mid \exists s, \pi, \text{ s.t. } E = \text{semiNaive}(s, \pi) \} \), \( \text{Selective} = \{ E \mid \exists s, \pi, A, \text{ s.t. } E = \text{Selective}(s, A, \pi) \} \). We next show that there is a strict inclusion relationship between the classes.

**Proposition 3.1.7** \( \text{Naive} \subset \text{semiNaive} \subset \text{Selective} \subset 2^{EX} \).

**Proof.**

The inclusion follows naturally from the definitions. We next prove 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 3.1 (ignore, for now, the text appearing in brackets 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 $\pi$ 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 \( E = \text{semiNaive}(s, \pi) \) contains the single EX-trace $e$, depicted in Figure 3.2. $e$ 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 $\text{Naive}(s')$, Hotel must have at least two alternative implementations in $s'$, one containing Credit1 and the other containing Credit2. But if this is the case, $\text{Naive}(s')$ 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 ⊂ selective  Consider the class of BPs where the imple-
mentation 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 un-
changed. That is, the set $A$ of deleted activities consists of all com-
pound activities appearing in $s$, and $\pi$ 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 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 such BP $s$, $Selective(s, A, \pi)$ defines a context free string language.
(Its context free grammar follows naturally from the BP specification).
Then, for every context free language $L$ we can define a BP $s$ (and $A,$
$\pi$) s.t. the words in $Selective(s, A, \pi)$ are precisely those of $L$. (Here
again, the specification of $s$ follows the grammar of $L$). Conversely,
the semi-naive EX-traces of a given BP $s$ can define a string language
according to the above definition only if, in the original EX-flows, the
direct implementation of the root consists only of a sequence of atomic activities. In this case the EX-traces will have a shape which is specified by some of the possible direct implementation graphs of the root activity, in the BP specification. These graphs, and consequently the semi-naive EX-traces are of bounded length for every given BP specification \( s \). Infinite context free string languages can thus not be captured via semi-naive tracing, according to the above definition. In contrast, we have shown that these languages can be captured using selective tracing.

**selective \( \subset 2^{EX} \)** Consider an infinite set of EX-traces \( E = \{t_0, t_1, \ldots\} \) defined as follows. The activation and completion nodes of the root of \( t_i \) are connected, through implementation edges, to an act-comp graph that is in fact a sequence of nodes: there are \( i \) act-comp nodes in this sequence that are labeled by \( a \), followed by \( i \) nodes labeled by \( b \), followed by \( i \) nodes labeled by \( c \). Assume by contradiction the existence of a BP \( s \) along with a deletion set \( A \) and a renaming function \( \pi \) such that \( \text{Selective}(s, A, \pi) = E \); in this case we can construct a context free string grammar defining the language is \( \{a^ib^ic^i \mid i \geq 0\} \), in contradiction. The context free string grammar is obtained from \( s \) in the following manner: its non-terminals are the compound activities of \( s \), its terminals are the atomic activities of \( s \) that are not in the deletion set \( A \), and its derivation rules follow the implementation relation of \( s \), restricting the implementations to atomic activity nodes that are not in \( A \) and to compound activity nodes that bear some possibly indirect implementation in which there exists an atomic activity that is not in \( A \). Observe that such restricted implementations have the shape of a sequence, otherwise \( s \) would have a possible EX-trace that is not a sequence; thus we can treat the sequence as a string, placing it on the right-hand side of a derivation rule.

\( \square \)
Queries  We have defined in Chapter 2 the notion of queries over execution flows. However, it naturally extends to queries over execution traces: the query syntax stays intact, and the definition of embedding is adapted to matching query nodes and edges to nodes and edges of the given EX-trace (rather than EX-flow).

Types  A type of a set $T$ of EX-traces is captured by a BP specification and a naive / semi-naive / selective tracing system (in turn captured by a renaming function $\pi$ and activities set $A$, where the former may be the identity function and the latter may be empty), whose corresponding set of possible traces is exactly $T$.

We sum up the discussion of EX-traces definitions 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 [59] (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 [52] (that also keep track of the parenthesis ‘origin’). This analogy holds with respect to some characteristics of the trace types, as we shall see below.

In the remainder of the chapter we formally define and study the problems of type inference and type checking.

3.2 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.

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 $\pi$ and activities set $A$), and would like to find a BP specification $s'$
We first show that type inference may not be possible if only naive trace types are considered. Namely,

**Theorem 3.2.1** There exist a BP specification $s$ and an EX-pattern $p$ s.t. there is no BP specification $s'$ where $\text{Naive}(s') = p(\text{Naive}(s))$.  

**Proof.** The proof follows lines similar to that of (the first part in the proof of) proposition 3.1.7. Consider the BP specification $s$ whose act-comp DAGs are depicted in Figure 3.3 (ignoring for now the $s_i$ labels next to the nodes). The implementation of the root activity Trip contains two consecutive compound Hotel activities, and where Hotel has two possible implementations, one containing the activity Credit1 and the second containing the activity Credit2. $\text{Naive}(s)$ contains four EX-traces, $e_1$ where both Hotel activities have Credit1 as implementation, $e_2$ 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 EX-pattern $p$ in Figure 3.4 (again ignoring the text near the nodes), which requires an occurrence of Credit1 and Credit2.

\[\text{Naive}(s') = p(\text{Naive}(s)) \quad \text{(resp. } \text{semiNaive}(s', \pi') = p(\text{semiNaive}(s, \pi)), \text{Selective}(s', A', \pi') = p(\text{Selective}(s, A, \pi))\) \]

\[2\]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.
Here $p(Naive(s))$ contains only $e_3$, and as explained in the proof of proposition 3.1.7, 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. We show next that type inference is possible with semi-naive trace types, but may be costly. We start by showing an EXPTIME type inference algorithm, then show that no PTIME algorithm is possible in this setting.

**Theorem 3.2.2** For every EX-pattern $p$, BP specification $s$ and renaming function $\pi$, there exist $s'$ and $\pi'$ such that $semiNaive(s', \pi') = p(semiNaive(s, \pi))$. $s'$ and $\pi'$ can be computed in time exponential in the sizes of $s$ and $p$.

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

**Proof.** We next describe our type inference algorithm for queries over BP specifications with semi-naive tracing, namely Algorithm $\text{SEMI-NAIVE-TYPE-INFEERENCE}$.
The BP specification $s' = (S', s'_0, \tau')$ constructed by our algorithm algorithm is intuitively the “intersection” of the original BP $s = (S, s_0, \tau)$ with the pattern $p$. We next explain how $s'$ is constructed. We start the construction description by considering only “simple” patterns, i.e. patterns that do not include transitive nodes and edges, and then extend the construction to also account for them.

**Simple EX-patterns.** Let us first consider patterns that do not contain transitive nodes and edges. 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 pairs. The renaming function $\pi'$ 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. $[n^i_p, n_s, a]$).

For compound (non-transitive) activities, the implementation $\tau'$ of $[n_p, n_s, a]$ consists 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 $s$. The nodes in this DAG (for each possible embedding) 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 a failure. The embeddings may be found using conventional algorithms for subgraph homomorphism [78], whose time complexity is polynomial in the size of the BP specification.

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 such failure activities.

**Transitive edges.** 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, each transitive edge $e_p \in p$ (that connects two pattern nodes) is mapped to a possible path in the BP (connecting the two corresponding BP nodes). Note that the number of such possible paths is possibly exponential in the BP size. In the output graph, a BP node $n_s$ (with label $a$) that appear on such path is labeled by the triplet $[e_p, n_s, a]$.

**Transitive nodes.** Finally, when the pattern $p$ contains transitive nodes, the algorithm becomes somewhat more complex. Recall that transitive nodes allow to navigate (transitively) inside the compound activities of the EX-trace, and query their internal flow at any depth of nesting. Specifically, 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 first verify for the existence of en embedding of $p_0$ in $p$. 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 contain 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.

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

Example 3.2.3 Consider again the EX-pattern \(p\) and BP \(s\) from the proof of theorem 3.2.1 (the pattern given in Figure 3.4, and the BP of Figure 3.3) with \(\pi\) being the identity mapping. The annotations next to the query and BP activity pairs represent their identifiers. The BP \(s'\) constructed by the algorithm is depicted in Figure 3.1. 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 \(\pi'\) maps \([q_i, s_j, a]\) to \(a\). Note that \(s', \pi'\) define a single semi-naive EX-trace, of the shape depicted in Figure 3.2, which is indeed the only answer of the query \(p\) when applied on the (semi-)naive traces of \(s\).

We next explain the manner in which \(s'\) is constructed by the algorithm. The construction of \(s'\) begins by matching the query root \(q_1\) to the specification root \(s_1\), and forming a new activity name \([q_1, s_1, Trip]\). Then, the implementation of \(q_1\) is matched against the single possible implementation of \(s_1\), thus embedding \(q_2\) (\(q_3\)) in \(s_2\) (resp. \(s_3\)), forming a new activity name \([q_2, s_2, Hotel]\) ([\(q_3, s_3, Hotel]\]). This is a main point of the algorithm: a unique activity name \textit{Hotel} that labeled two nodes of the original BP specification \(s\) (the nodes identified by \(s_2\) and \(s_3\), 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 $q_2$ in the possible implementations of $Hotel$. There exist two such possible implementations, one of which contains $Credit_1$ and the second contains $Credit_2$. In the latter, there is no embedding of the implementation of $q_2$; in the former, there exists an embedding, yielding a node labeled $[q_4, s_4, Credit_1]$. Similarly, we construct the only implementation of $[q_3, s_3, Hotel]$, containing $[q_5, s_5, Credit_2]$.

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.

**Theorem 3.2.4** There exists an infinite set $S$ of BPs of increasing sizes, and an EX-pattern $p$ (with only three activation-completion node pairs), s.t. for each BP $s \in S$, every $s'$ s.t. semiNaive($s'$, $\pi'$) = $p$(Naive($s$)) for some $\pi'$ is of size $\Omega(2^{|s|})$.

**Proof.** The BPs in the class $S$ 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 $S$ 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 semi-naive traces must contain each path as an explicit implementation of the root, and hence the result 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, and consequently a more efficient type inference algorithm.

**Theorem 3.2.5** For every EX-pattern $p$, BP $s$, and a renaming function $\pi$, there exist $s', A', \pi'$ s.t. Selective($s', A', \pi'$) = $p$(semiNaive($s$, $\pi$)). $s'$, $A'$, $\pi'$ 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 a stronger expressive power and represents selective traces as well.

**Theorem 3.2.6** For every EX-pattern $p$, BP $s$, set of activities $A$ and renaming function $\pi$, there exist $s', A'$ and $\pi'$ such that Selective($s', A', \pi'$) = $p$(Selective($s$, $A$, $\pi$)). $s'$, $A'$ and $\pi'$ 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 3.2.6 also proves Theorem 3.2.5.

**Proof.** Recall the EXPTIME (data complexity) algorithm SEMI-NAIVE-TYPE-INFERENCE given in the proof of Theorem 3.2.2 above. It is easy to extend the algorithm to support selective tracing: let $(s, A, \pi)$ be the input type,
then we may construct an output type \((s', A', \pi')\), where \(s', \pi'\) are constructed exactly as in algorithm \textsc{semi-naive-type-inference}, and \(A'\) is such that \([n_p, n_s, a] \in A'\) if \(a \in A\). But this algorithm still incurs \textsc{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 \textsc{PTIME} data complexity.

The improved type inference algorithm, namely \textsc{selective-type-inference}, is based on the following observation. Consider the manner in which transitive edges are handled in Algorithm \textsc{semi-naive-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 \textit{string}, and the set of strings obtained from all such paths as a string language \(L\). As we show below, it is easy to define a regular string grammar \(G\) describing \(L\). Then, we use \(G\) to define a BP \(s''\), along with \(A''\) and \(\pi''\), s.t. the graphs of \(\text{Selective}(s'', A'', \pi'')\) 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 to algorithm \textsc{semi-naive-type-inference} (in addition to the change described above, to support selective tracing in the input type). We show that each of them may be done in time polynomial in the size of \(s\). As, apart from the handling of transitive edges which we replace in the current algorithm, all other constructions in the algorithm incurs time that is polynomial in the size of \(s\) (with the exponent determined by \(p\)), so is the overall complexity of the refined algorithm.
Generating the string grammar $G$. Assume first that no path in-between $n_1$ and $n_2$ contains a compound activity node; in this case the non-terminals of the grammar simply correspond to the nodes along all possible paths between $n_1$ and $n_2$ (note that though the number of such paths may be exponential in the graph size, the number of nodes along the paths is obviously bounded by the overall number of nodes in the graph). Terminals correspond to all (compound and atomic) activity names along these paths. Recall that $\lambda(n)$ denotes the activity name of a specification node $n$; overloading notation, it is also used here as some of the (non-)terminals of the constructed string grammar.

The regular grammar is constructed bottom up, starting with a single non-terminal $N_2$ bearing a single derivation rule $N_2 \rightarrow \lambda(n_2)$. For each node $n'$ such that there exists an edge from $n'$ to $n_2$ in the specification graph, we design a rule $N' \rightarrow \lambda(n')N_2$ where $N'$ is the non-terminal corresponding to the node $n'$. We continue this process where in each step we design a rule of the form $N_{prev} \rightarrow \lambda(n_{prev})N$ where there exists an edge from $n_{prev}$ to $n$, and $N_{prev}$ is the non-terminal assigned for the node $n$. The process terminates when we reach $n_1$ (where $N_1$ is its corresponding non-terminal), and we set the grammar root to be $N_1$. Now assume that there are some compound activities on these paths. The construction operates exactly as above until reaching a compound activity. For a non-terminal $N$ corresponding to a compound activity node labeled by $A$ with implementations $s_1, ..., s_k$, we first assign new non-terminals for the nodes of $s_1, ..., s_k$. We then design new rules $N \rightarrow start(s_i)$ for $i = 1, ..., k$ and $end(s_i) \rightarrow \lambda(end(s_i))N'$, where $start(s_i)$ ($end(s_i)$) is the non-terminal corresponding to the start (end) node of $s_i$, and $N'$ is the node that preceded $N$ in the algorithm traversal. 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). Note that due to the dynamic-programming style algorithm that we employ, recursive implementations do not cause a problem here.
Recall that each non-terminal in $G$ corresponded to a node in $s$. We use $\text{real}(G)$ to denote the set of non-terminals that originated from compound activity nodes, and $\text{temp}(G)$ for all other non-terminals.

**Transforming $G$ into $s''$.** We next explain how to generate a BP $s''$ along with a selective tracing system $A''$ and $\pi''$, such that each graph in $\text{Selective}(s'', A'', \pi'')$ is in fact a path and the labels along each such path constitute a word in the string 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 righthand 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 compound activities names corresponding to non-terminals in $\text{temp}(G)$. $\pi'' = \pi$ (the renaming function used for the input BP specification).

**Plugging $s''$ into $s'$.** To complete the presentation, we explain how our algorithm utilizes a sub-routine that generates such $s''$ specification for every two given nodes $n_1, n_2$. Whenever the algorithm reaches the origin of a transitive edge $N_1 \rightarrow N_2$, we simply add a new compound activity $r_{s''}$ which 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.

The time complexity of the refined algorithm is polynomial in the size of the input BP specification $s$, with the exponent depending on the query size. This concludes the proof of Theorem 3.2.6.

□

A first obvious reason for not being able to provide a type inference algorithm of PTIME combined (data and query) complexity is the need to embed the pattern sub-graphs in the DAGs of $s$. This entails check-
ing for subgraph homomorphism, known to be an NP-complete problem [78] (with respect to the size of the pattern graph, corresponding here to the EX-pattern). 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), 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)) = ∅ (the discussion of semi-naive and selective tracing follow, below). We can show the following.

**Theorem 3.2.7** 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).

**Notes.** Before proving the theorem, we note the following on its implications.

1. First, note that while Theorem 3.2.7 relates to a naive tracing system employed for the input BP specification, it entails NP-hardness also for the cases of semi-naive and selective tracing, of which naive tracing is a special case.

2. Also note that for DAGs with the restricted shape enforced by the condition given in Theorem 3.2.7, sub-graph homomorphism, i.e. finding an embedding of a query in a single finite (“flat”) graph can be solved in PTIME (see e.g. [53]). This shows that the hardness here comes from the nested shape of the BP specification. Specifically, as we show below, it stems from the need to
consider, when treating transitive pattern activities, all the possible splits of the EX-pattern to match within the activity (indirect) implementations.

**Proof.** We construct the following reduction from 3-SAT [51].

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 Figures 3.5, 3.6 respectively. The idea is to create a compound activity node associated with each variable of the formula (Figure 3.5). 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, depicted in Figure 3.6, requires all clauses of the formula $F$ to appear. An embedding thus corresponds to a “correct” choice of either a variable or its negation, i.e. a satisfying assignment to the variables.

To formally prove that the reduction is valid, we show the following lemma.
Lemma 3.2.8 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. Note that we consider naive tracing, so $e$ corresponds to an EX-flow of $s$. This EX-flow was obtained by choosing, for a subset of composite nodes their 'true' implementations, and for another subset their 'false' implementation. These choices correspond exactly to a satisfying assignment, as follows. For every variable whose corresponding compound activity node was implemented by a 'false' ('true') graph, assign 'false' ('true'). This is indeed an assignment, as every compound activity node can only have a single implementation in a given EX-flow, and it is satisfying as for every clause there was at least one composite node where its chosen implementation contains the node corresponding to the clause.
Conversely, let $A$ be a satisfying assignment, and let $e$ be the naive EX-trace corresponding to the EX-flow obtained by choosing, for each composite node, its ‘true’ implementation if $A$ assigns ‘true’ to the corresponding variable, and its ‘false’ implementation otherwise. 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 $e$. Hence the reduction is valid.

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 $p(selective(s, A, \pi)) = \phi$.

**Theorem 3.2.9** Given an EX-pattern $p$, a BP $s$, a set of activities $A$ and a renaming function $\pi$, $MATCH?(p, s, A, \pi)$ is in NP (combined complexity).

**Proof.**

The NP algorithm is based on the following Lemma.

**Lemma 3.2.10** 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 $\pi$.

The correctness of this Lemma stems from a property analogous to the “Pumping Lemma” known for context free string grammars [93].

**Proof.**

Let $e \in selective(s, A, \pi)$ be a trace in which there exists an embedding $h$ of $p$. We first consider all nodes and edges to which all nodes
and all non-transitive edges of $p$ are mapped; clearly their number is bounded by the size of $p$. Now, consider a path $P$ of $e$ to which a transitive edge of $p$ is mapped. The length of $P$, and consequently, the size of $e$, may be arbitrarily long. However, we may construct a “smaller” trace $e'$ s.t. there also exists an embedding of $p$ in $e'$, as follows. Consider the implementations of compound activities appearing on $P$. If there exists no compound activity name $a$ such that a node $n'$ labeled by $a$ appears within an (indirect) implementation of another node $n$ labeled by $a$, then clearly the length of $P$ is linear in $|s|$. Also, If there exists such cases of recursive implementations, but within each such implementation there exists a node (edge) to which a node or a non-transitive edge of $p$ is mapped by the embedding, then the number of such recursive implementations is bounded by $|p|$ and the path size is bounded by $|p| \ast |s|$. Otherwise, if there exists such recursive implementation in $P$ such that no node and no non-transitive edge of $p$ were mapped to any of its nodes and edges, 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 $n'$. Denote by $P'$ the sub-path of $P$ that remains after this omission (i.e. appears in $e'$). Observe that as $n$ and $n'$ share the same activity name, it holds that also $e' \in \mathit{selective}(s, A, \pi)$. Observe also that there still exists an embedding of $p$ in $e'$: this embedding $h'$ is exactly the same as $E$ for all but the transitive edge of $p$ that was mapped by $E$ to $P$. $h'$ maps this edge to $P'$.

We then continue this process for each recursive implementation in $P$ such that no node and non-transitive edge of $p$ was mapped to any of its nodes and edges, until obtaining a flow in which no such implementation exists. We further repeat this process for additional transitive edges in in $p$ (and the paths they are matched to). Eventually, we obtain a trace in which the length of each path to which a transitive edge of $p$ is mapped is bounded by $|s| \ast |p|$; the number of such paths is bounded by $|p|$, yielding an overall bound of $O(|s| \ast |p|^2)$ on the trace.
The NP algorithm that is based on the above Lemma is then simple - we first guess an expansion sequence of polynomial size starting at the specification root, to obtain a flow $e$. Then, we remove from $e$ all activities in $A$ and apply the renaming function $\pi$ over the remaining activities, to obtain a trace $e'$. We then guess a mapping of the pattern $p$ to $e'$. Clearly each such guess may be generated and tested in PTIME (combined complexity).

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

### 3.3 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 $\pi'$ and an 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 3.3.1** Given an EX-pattern $p$ and two BP specifications, renaming functions and activities sets $(s, \pi, A)$ and $(s', \pi', A')$, the problem of testing whether $p(Selective(s, A, \pi)) \subseteq Selective(s', \pi', A')$ 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 3.1.7, $s, A, \pi$ and $s', A', \pi'$ such that the graphs in $Selective(s, \pi, A)$ and $Selective(s', \pi', A')$ correspond, resp., to the strings in $L$ and $L'$ (in the same sense defined in the proof of proposition 3.1.7). 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 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 $L \subseteq L'$. □

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 deterministic trace types, and finally we show that we can translate every BP $s$ and renaming function $\pi$ to equivalent $s'$ and $\pi'$ w.r.t. which the set of semi-naive EX-traces is deterministic. Note that our proof technique is inspired by [73] that considers inclusion of parenthesis string languages. However, the proof is more complicated here as it 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 $\pi$ 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 $n_\omega$ 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 Π(e), there may be several different isomorphisms between them.

**Definition 3.3.2** For a BP specification s and a renaming function π over its activities, the set of semi-naive traces semiNaive(s, π) 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 next show the following.

**Theorem 3.3.3** 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', π') is deterministic), the problem of testing if p(Naive(s)) ⊆ Naive(s') (respectively, p(semiNaive(s, π)) ⊆ semiNaive(s', π')) is decidable, and can be solved in EXPTIME (data complexity).

**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.

We observed above that for every s, π and p we can derive s'' and π'' s.t. semiNaive(s'', π'') = p(semiNaive(s, π)). To prove the theorem we will construct, for every s'', π'' and s', π' where semiNaive(s', π') is deterministic, an s''' and π''' such that semiNaive(s''', π''') is empty iff semiNaive(s'', π'') − semiNaive(s', π') is empty. Note that semiNaive(s'', π'') − semiNaive(s', π') = semiNaive(s'', π'') ∩ semiNaive(s', π'), where semiNaive(s', π') is the complement of semiNaive(s', π'), i.e. it contains all EX-traces that are not included in semiNaive(s', π'). Also note that, in fact, it suffices to consider a restricted portion of semiNaive(s', π') 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 \( k \).

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 3.3.4** For every pair of BPs \( s, s' \), we can compute a BP \( s'' \) such that \( \text{Naive}(s'') = \text{Naive}(s) \cap \text{Naive}(s') \).

Similarly, for every \( s, \pi \) and \( s', \pi' \) there exist \( s'', \pi'' \) such that \( \text{semiNaive}(s'', \pi'') = \text{semiNaive}(s, \pi) \cap \text{semiNaive}(s', \pi') \).

**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 \( \text{Naive}(s) \cap \text{Naive}(s') = \emptyset \) 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 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 \( S'' \). \( \tau'' \) 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 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 $\pi$ and $\pi'$, 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 $\pi$, $\pi'$. 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 $\pi$, we use $\text{semiNaive}_k(s, \pi)$ to denote the set of all $k$-bounded EX-traces that do not belong to $\text{semiNaive}(s, \pi)$. We shall construct a BP $\overline{s}$ with renaming function $\overline{\pi}$ s.t. $\text{semiNaive}(\overline{s}, \overline{\pi}) = \text{semiNaive}_k(s, \pi)$. This construction may be used for any $k$.

**Proposition 3.3.5** For every BP specification $s$ and activity renaming function $\pi$, where $\text{semiNaive}(s, \pi)$ is deterministic, and for every $k$, there exists a BP specification $\overline{s}$ and a renaming function $\overline{\pi}$ s.t. $\text{semiNaive}(\overline{s}, \overline{\pi}) = \text{semiNaive}_k(s, \pi)$.

Observe that every BP specification is deterministic w.r.t to $\pi$ 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 \( \pi \) maps \( a \) and \( \overline{a} \) to \( \pi(a) \), i.e. \( \pi(a) = \pi(\overline{a}) = \pi(a) \).

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 \( semiNaive(\overline{s}, \overline{\pi}) = semiNaive_k(s, \pi) \), by induction on the nesting depth of traces.

\( \square \)

Applying theorem 3.2.2, for inferring type \( s'' \) corresponding to the query answer, then combining proposition 3.3.4 with proposition 3.3.5 (applied for \( k \) which is the size of the \( s'' \)), we obtain an EXPTIME algorithm for type checking, when the target type is deterministic.

\( \square \)

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 \( \pi \) to equivalent \( s' \) and \( \pi' \) w.r.t. which the set of semi-naive EX-traces is deterministic.

**Theorem 3.3.6** Given a BP \( s \) and an activities renaming function \( \pi \), there exist a BP \( s' \) and a renaming function \( \pi' \) such that \( semiNaive(s', \pi') = semiNaive(s, \pi) \) and \( semiNaive(s', \pi') \) is deterministic w.r.t. \( s', \pi' \).

**Proof.** We next explain the construction of \( s' \).
Activities First, we group together all activities of $s$ that are mapped by $\pi$ to the same activity, obtaining a set $\gamma$ of activity subsets. We say that each such activities subset is *represented* by the (single) activity to which its members are mapped. As each activity of $s$ is mapped by $\pi$ to a unique activity ($\pi$ 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 In the sequel, we denote $\Pi(g)$ as the graph obtained from $g$ by applying $\pi$ over all of its activity names. Similarly, $\Pi(G)$ where $G$ is a set of graphs, denotes the set obtained by applying $\Pi$ 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 $\tau'$ 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 $\gamma$, 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 $\gamma$.

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

Clearly, $\text{semiNaive}(s', \pi')$ is deterministic w.r.t. $s', \pi$ as each (com-
pound or atomic) activity name is composed as a \textit{maximal} equivalence class of activities, thus for each activity name \( a \) appearing in a semi-naive 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 \( \text{semiNaive}(s', \pi') = \text{semiNaive}(s, \pi) \).

\textbf{Lemma 3.3.7} \( \text{semiNaive}(s', \pi') = \text{semiNaive}(s, \pi) \)

\textbf{Proof.} Let \( e \in \text{semiNaive}(s', \pi') \) and let \( e' \) be the flow of \( s' \) whose semi-naive 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 a flow \( e'' \) 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 \( e \in \text{semiNaive}(s, \pi) \).

Conversely, let \( e \in \text{semiNaive}(s, \pi) \), and let \( e' \) be the flow of \( s \) whose semi-naive trace is \( e \). Consider a compound activity name \( a \) appearing in \( e' \) with implementation \( g' \). Then it follows from the construction that 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 name \( B \) in \( g' \) by some \( b \in B \). By subsequently making such replacements, we obtain a flow \( e'' \in \text{semiNaive}(s', \pi') \) such that \( \Pi'(e'') = e \), thus \( e \in \text{semiNaive}(s', \pi') \). \( \square \)

This concludes the proof of Theorem 3.3.6.

\( \square \)

The immediate corollary of theorems 3.3.3 and 3.3.6 follows.

\textbf{Corollary 3.3.8} Given an EX-pattern \( p \), a BP specification \( s \) (with renaming function \( \pi \)), and a target BP specification \( s' \) (with renaming
function $\pi'$), the problem of testing if $p(\text{Naive}(s)) \subseteq \text{Naive}(s')$ (resp. $p(\text{semiNaive}(s, \pi)) \subseteq \text{semiNaive}(s', \pi')$) is in EXPTIME (data complexity).

Our proof is constructive, and gives an EXPTIME type checking algorithm.

### 3.4 Related Work

We conclude this Chapter with an overview of related 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 [52], semi-naive trace types and parenthesis languages [73], and selective trace types and context free languages [93]. There is also a close connection between selective trace types and context free graph languages [59, 26]. 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 [26] has shown that the theory of MSO over context free graph grammars is decidable. However, the complexity of the algorithms suggested in [26] 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 for type checking, in order to apply the algorithms of [26], 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).

Type checking and type inference are well studied problems in the context of functional programming languages [76]. The complexity there is derived from the interaction of function types, polymorphism, and let-bindings; as pointed out in [22], 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 [74] that use tree patterns to select subtrees of interest. For such XML queries, [74] 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 is harder than type inference, in some cases. Another common class of XML queries is transformation queries that restructure the input tree. [74] and [75] showed the impossibility of type inference and undecidability of type checking under a general model of transformations. Further work (e.g. [71], [40]) considered restrictive transformation models that allow decidability of type checking, with complexity ranging from PTIME to non-elementary.
Chapter 4

Finding Top-K Future Executions

In the previous Chapter we have focused on the analysis of past executions, given some (possibly partial) information over them, and showed how to exploit knowledge over the BP specification structure to optimize the analysis of these past executions. In this Chapter, as well as in Chapter 5, we focus on another axis of analysis, namely the analysis of future executions. The model and results reported in this Chapter were originally published in [33].

Recall that a BP specification $s$ defines a set of possible execution flows, and that an execution pattern $p$ further restricts the focus to (sub-)flows that are of interest. Since the number of query answers may itself be extensively large (or even infinite), it is important to identify those that are “most important”, where the notion of importance is captured by some weighting metric that depends on the analysis goal. To that end, we study the problem of finding, given a BP specification, a query (given by an EX-pattern), a weighting metric over EX-flows, and a number $k$, the top-$k$ query results.

For some intuition on the kind of analysis of future executions that one may be interested in, and the corresponding queries and weighting metrics, let us consider a simple example, that is slightly different than
the example used in the previous Chapter, and better illustrates the
concepts presented here. Assume that we are given a BP of a Web-
based shopping mall with virtual shops of various vendors. A customer
of the mall may be interested in buying a Toshiba TV and a DVD
player of lowest overall price. Say that we identify that among the pos-
sible EX-flows containing a purchase of two such appliances [query], the
lowest-priced one [weighting] is one where the user first subscribes to
the Toshiba customers club. Such a result may suggest that subscrip-
tion is beneficial even if it entails some registration fee. Alternatively,
suppose that the same user is interested in minimal shipping time [an-
other weighting]. We may identify here that among all possible EX-
flows, those where both products are purchased in the same store are
preferable. As another example, the Web-site owner may be interested
to know which, among all the EX-flows that lead to a TV purchase
[query], are the most profitable/popular ones [weighting]. The answer
here may be used, for instance, to target relevant personalized adver-
sisements to users. Naturally, several weighting functions of interest
may be combined to form a single weight metric.

To formally study top-k analysis, we first propose a generic model
for weighted EX-flows. To weigh EX-flows, we assume that each imple-
mentation choice taken during the EX-flow bears some weight (denoted
$cWeight$, for choice weight), and that $cWeights$ of choices throughout
the EX-flow are aggregated to obtain the EX-flow weight (denoted
$fWeight$, for flow weight). We refer to a BP specification along with
such weight function over its possible EX-flows, as a \textit{weighted BP spec-
ification}.

For example, $cWeight$ may be the price of the product chosen at a
given point of the EX-flow, or the likelihood of a user clicking on a given
store link. In the first case, summation may be used for aggregation;
for likelihoods we may use multiplication. Following common practice
[45], we require the aggregation to be monotonic w.r.t. the progress of
the flow. This captures most practical scenarios, e.g. the total price of a shopping cart subset does not exceed the price of the full cart, even in the presence of discount deals.

It is important to note that the $cWeight$ of a given choice may vary at different points of the EX-flow and may depend on the course of the execution so far and on previous choices. For instance, the price of a given product may be reduced if the user had previously subscribed to a customers club or bought at least two products offered by the same vendor; the likelihood of clicking on a certain store link may depend on stores previously visited. Thus, $cWeight$ is modeled as a function whose input includes not only the choice itself but also information about the history of the EX-flow thus far. We then distinguish 3 classes of such $cWeight$ functions, according to the actual effect that the flow history has on the $cWeight$ of a given choice. Intuitively, a $cWeight$ function is history-independent if the weight of any given choice does not depend on the flow history (i.e. $cWeight$ is in fact a function only of the choice); it is bounded-history if the weight of each choice is dependent only on the $m$ last choices preceding it, for some constant $m$; and is unbounded-history in the general case.

In practice, bounded-history functions occur very naturally. Moreover, studies on the behavior of typical Web Applications (and their users) indicate the history size that actually affects e.g. the likelihood of a choice, the price it incurs etc. to be, in practice, relatively small (approximately 4 [82]). Specifically, unbounded-history functions may occur only when the BP specification is recursive; even for recursive specifications, such functions are extremely rare in practice [82].

After defining the model of weighted BP specifications, we then turn to top-k analysis of such specifications. In this chapter we consider a restricted case, where no query is given; consequently, our algorithm finds the top-k full EX-flows of the given BP specification. Then, in the following chapter, we explain how to utilize this algorithm for effi-
cient top-k query evaluation. Consequently, the related work for top-k analysis is discussed towards the end of the next chapter.

**Our Results**  We show that the complexity of top-k analysis depends on properties of the \( cWeight \) function, as follows.

- When the \( cWeight \) function is history-independent, we provide a PTIME algorithm for retrieval of the top-k EX-flows of a weighted BP specification.

- When the \( cWeight \) function is bounded-history, we provide an algorithm whose time complexity is polynomial in the BP specification size with the exponent dependent on the history size. Recall that this history size, and consequently the exponent, is typically very low in practice. Furthermore, we show that unless \( P = NP \), the exponential dependency on the history size is inevitable.

- In the general case of unbounded-history \( cWeight \) functions, we show that top-k analysis is impossible (i.e. the corresponding decision problem is undecidable). Recall that such functions occur very rarely in practice.

We start by depicting the refined, weighted model, then turn to study query evaluation algorithms.

### 4.1 Weighted Model

We present next the required changes and additions to the model of Business Processes, allowing to support weights.

First, we introduce the notion of *guarding formulas*. These are simply predicate calculus formulas used to annotate each possible implementation of every compound activity. We denote the domain of guarding formulas that are in use by \( \mathcal{F} \). We next explain how to extend
the definitions of Business Processes and Execution Flows to account for guarding formulas. We will then define weight functions over these formulas, and finally use this definition to define weighted EX-flows.

Recall that the definition of Business Processes (Definition 2.1.3) included an implementation function defined as $\tau: A_{\text{compound}} \rightarrow 2^S$; to account for guarding formulas, $\tau$ is now defined as $\tau: A_{\text{compound}} \rightarrow 2^S \times F$, namely $\tau$ maps each compound activity name to a set of pairs, each pair consisting of a guarding formula and a possible implementation graph. Apart for that, the definition of Business Processes stays intact.

**Example 4.1.1** As a running example for this Chapter, we consider here a web-based shopping mall, where users may choose from a variety of stores, products offered by these stores, and various brands for each product. Figure 4.1 shows the BP specification of the Shopping Mall. Note the guarding formulas annotating the different implementations and controlling the choice of implementation. For instance, the Login activity has two possible implementations $S_2$ and $S_3$ corresponding to users logging-in to the system with their Visa or MasterCard credit card, resp. The idea is that exactly one formula is satisfied at run-time, e.g., the user either choose to login with a Visa or a Mastercard credit card, and thus Login is implemented either by $S_2$ or $S_3$ respectively. A similar observation can be made for chooseStore, which has several possible implementations, but exactly one will be chosen at run-time.

Correspondingly, EX-flows are defined similarly to Def. 2.2.1, but each compound activity in the EX-flow is annotated by the guarding formula corresponding to the implementation chosen for the edge.

Note that a given EX-flow may be obtained via different expansion sequences varying in the order of parallel activities expansion. To simplify the presentation, we impose a total order among unexpanded activities of any given partial flow, and assume that at each step, the
first unexpanded activity according to this order is expanded. (This order can be achieved by a topological sort of the EX-flow DAG). Thus, any partial EX-flow corresponds to a unique sequence of expansion steps from the base EX-flow. We stress that this assumption is made solely for presentation considerations– all our results extend to a general context where multiple expansion orders are possible (see Section 4.4).

We next give an example EX-flow of the online shopping mall BP.

**Example 4.1.2** An example EX-flow of the shopping mall BP is given in Figure 4.2. Note that each compound activity is annotated by the formula guarding the implementation chosen for it. The EX-flow describes a sequence of activities that occur during the BP execution. The user logs in with a Visa Credit Card, then chooses to shop at the BestBuy (BB) store. There, she chooses to look for a DVD player and selects
one by Toshiba, then continues shopping at the same store, looking for a TV, and selects one also by Toshiba (this last part is omitted from the figure). Finally she exits and pays.

Equality and containment of EX-flows  Given two EX-flows $e$ and $e'$, we use in the sequel the notation $e = e'$ when there exists an isomorphism between $e$ and $e'$ respecting node labels, edge relation and guarding formula names. We say that $e \subseteq e'$ if there exists such isomorphism between $e$ and some EX-flow $e''$ obtained from $e'$ by removing some of its nodes and edges.

We next turn to defining weight functions over BP specifications and execution flows. To that end, we assume an ordered domain $\mathcal{W}$ of weights. We use three functions: (1) $cWeight$ that describes the weight of each implementation choice, given a preceding sub-flow, (2) $aggr$ that aggregates $cWeight$ values, and (3) $fWeight$ that uses $cWeight$ and $aggr$ to compute the weight of an EX-flow.
The \textit{cWeight function} Given a BP specification $s$, $cWeight$ is a partial function that assigns a weight $w \in W$ to each pair (partial EX-flow $e$ of $s$, guarding formula $f$ in $s$), such that $f$ guards the compound activity node of $e$ that is next to be expanded. Intuitively, the value $cWeight(e, f)$ is the weight of the implementation guarded by $f$, given that $e$ is the flow that preceded it.

Example 4.1.3 Re-consider the BP in Fig. 4.1, and a $cWeight$ function that assigns, to each implementation choice (guarding formula), the additional cost that this choice incurs to the customer. In this case $W$ is the set of all positive numbers. The $cWeight$ of the \$brand = \"Toshiba\" choice, given that the preceding flow indicates that the Toshiba product that is being purchased is a DVD, the store is BestBuy (BB) and the user has identified herself as a Visa card holder, may be the Visa discounted price of a Toshiba DVD at BestBuy. The $cWeight$ of a Toshiba choice, given now that the product is a TV and that the preceding flow already includes a purchase of a Toshiba product (the DVD), may reflect, e.g., a 20\% discount for the second product. The $cWeight$ of other choices (like the store or the product type) may be zero here, as they incur no additional cost to the user.

We distinguish three classes of $cWeight$ functions, called \textit{history-independent}, \textit{bounded-history}, and \textit{unbounded-history} functions, reflecting how much of the preceding flow $e$ actually inflicts on the $cWeight$.

\underline{History-independent:} History-independent $cWeight$ functions compute $cWeight(e, f)$ based solely on the choice $f$ and ignore the history $e$ leading to it. Formally,

\textbf{Definition 4.1.4} A $cWeight$ function is \textit{history-independent} for every guarding formula $f$, $cWeight(e, f) = cWeight(e', f)$ for each two partial EX-flows $e, e'$.

\textbf{Example 4.1.5} To continue with our example, if $cWeight$ represents prices, it is \textit{history-independent} if the prices of all products of all brands
in all stores are the same and no deals are available. If cWeight reflects likelihood of choices, then history-independence means probabilistic independence between the choices.

Bounded-history: Bounded-history cWeight functions capture the more common scenario where the cWeight value does depend on the history e of the EX-flow, but only in a bounded manner. That is, to determine the cWeight of any implementation choice at an activity node n, it suffices to consider only the implementation choices of the last b preceding compound activities, for some bound b. By “choice” we refer to the formula guarding the implementation selected for the activity. By “preceding” we refer here to activity nodes ě that are ancestors of n in e, i.e. nodes n' such that there exists a directed path from n' to n in e (in contrast to activity nodes that occur in parallel and thus in general may or may not precede n).

More precisely, recall that we assumed that given an EX-flow e, the expansion sequence leading to e is well defined. The last choice preceding (the expansion of) a node n in this sequence, denoted PrevChoice(e, n), is the guarding formula of the implementation selected for the last compound activity node ě in this sequence that preceded n (in the above defined sense). Similarly, PrevChoice^2(e, n) are the last two preceding choices, and more generally PrevChoice^i(e, n) is a vector consisting of the i last preceding choices. We are now ready to define bounded-history cWeight functions.

Definition 4.1.6 We say that cWeight is bounded-history, with history bound b, if for every activity name a, every guarding formula f of a, and every two pairs of [EX-flow,next-to-be-expanded-node] [e, n], [e', n'] where λ(n) = λ(n') = a and PrevChoice^b(e, n) = PrevChoice^b(e', n'), it holds that cWeight(e, f) = cWeight(e', f).

Example 4.1.7 In our running example, assume that the price of a given brand depends only on the store, the type of the product being
bought and whether or not the user has identified herself in this pur-
chase as a Visa card holder. Here the bound \( b \) is 3: for nodes \( n \)
with \( \lambda(n) = \text{chooseBrand} \), \( \text{PrevChoice}^3(e, n) \) consists of the choices of
credit card, store and product that preceded the choice of brand, and all
\text{chooseBrand} nodes having the same \( \text{PrevChoice}^3 \) have the same cost
(\( c\text{Weight} \)) for each possible choice of brand (guarding formula). In a
similar way, a greater history bound can be used to take into consider-
ation also (a bounded amount of) information on previously purchased
products.

Unbounded-history: Unbounded-history \( c\text{Weight} \) functions may use an
unbounded portion of the flow history \( e \) to compute the next choice’s
weight.

Example 4.1.8 If the price of a given product depends on the exact full
sequence of search choices that the user performed prior to the purchase,
then the corresponding \( c\text{Weight} \) function is unbounded-history.

Observation. Note that, by definition, for non-recursive BPs, \( c\text{Weight} \)
functions are always bounded-history, as even the size of any given EX-
flow is bounded. For recursive BPs, on the other hand, the \( c\text{Weight} \)
function may be unbounded-history. However, this is extremely rare in
practice [82].

The Aggregation function The weights along the EX-flow are aggre-
gated using an aggregation function. The function \( \text{aggr} : \mathcal{W} \times \mathcal{W} \to \mathcal{W} \)
receives two weights as inputs; the first is intuitively the aggregated
weight computed so far, and the second is the new \( c\text{Weight} \) to be
aggregated with the previous value. For instance, when computing
purchase cost \( \text{aggr} = + \) and \( \mathcal{W} = [0, \infty) \); when computing likelihood,
\( \text{aggr} = \ast \) and \( \mathcal{W} = [0, 1] \). We consider here aggregation functions that
satisfy the following intuitive constraints:
1. \(aggr\) is associative and commutative, namely for each \(x, y, z \in W\), 
\[aggr(aggr(x, y), z) = aggr(x, aggr(y, z)), \text{ and } aggr(x, y) = aggr(y, x)\]

2. \(aggr\) is continuous, that is for each \(x, y, z \in W\), if \(aggr(x, y) < aggr(x, z)\) then there exists \(w \in W\) such that \(aggr(x, y) < aggr(x, w) < aggr(x, z)\)

3. \(aggr\) has a neutral value, denoted \(1_{aggr}\). Namely for each \(x \in W\), 
\[aggr(x, 1_{aggr}) = aggr(1_{aggr}, x) = x\]

4. \(aggr\) is monotonically increasing or decreasing over \(W\). Namely, either \(\forall s, x, y \in W \ x \geq y \implies aggr(s, x) \geq aggr(s, y)\) and \(aggr(s, x) \geq s\), or the same for \(\leq\).

Observe that the above mentioned aggregation functions + and *, for cost and likelihood, satisfy the constraints.

**The fWeight function** Finally, the \(fWeight\) of an EX-flow is obtained by aggregating the \(cWeights\) of all choices made during the flow, and is defined recursively: if \(e\) is an EX-flow consisting only of the root \(s_0\), \(fWeight(e) = 1_{aggr}\). Otherwise, if \(e' \rightarrow e\) for some EX-flow \(e'\) of \(s\), then \(fWeight(e) = aggr(fWeight(e'), cWeight(e', f))\), where \(f\) is the formula guarding the implementation that is added to \(e'\) to form \(e\).

**Example 4.1.9** Assume that a Toshiba TV and DVD, which individually cost 250$ and 150$ resp., are sold together with 20% discount (i.e. for 320$). The \(cWeight\) for the first choice of Toshiba TV (DVD) is 250$ (resp. 150$). The \(cWeight\) for the next choice, of DVD (TV), is 70$ (170$), computed as 320$ minus the cost already incurred for the first product. Using + for aggregation, \(cWeights\) along the flow are summed up, yielding the total 320$ deal price.
**Weighted BP specifications**  We use the term *weighted BP specification* to refer to a BP specification $s$ along with corresponding weight functions ($cWeight$, $aggr$ and the derived $fWeight$). When the weight functions are clear from context, we omit them for brevity.

**Top (bottom)-K flows**  Observe that when the aggregation function $aggr$ is monotonically increasing (resp. decreasing), so is $fWeight$, in the sense that the weight of a flow monotonically increases (resp. decreases) as the execution advances and the flow is expanded. Examples are the “total cost” (resp. the “likelihood”) weights from the previous example, that increase (decrease) as more products are purchased (more choices are made). As in the above examples, for monotonically decreasing (increasing) $fWeights$, and given also a number $k$, we would like to identify the top (bottom)-k flows of the BP specification.\footnote{Note that the opposite problem of finding the top(bottom)-k flows in monotonically increasing (decreasing) functions may not be possible, since the weights may increase (decrease) infinitely in the presence of recursion, and is left for future research.} We exemplify both cases below.

**Example 4.1.10**  Re-consider the Business Process whose specification is depicted in Figure 4.1 and consider first a $cWeight$ function representing the price induced by a purchase. The $aggr$ function of addition considered here is increasing over its domain (the purchase price may only increase as we add more products), and clearly, there is no maximum price of a flow - the price may always increase by resetting and purchasing more items. Rather, in such case we are interested in those flows that minimize the $fWeight$ function, (possibly) restricted to flows satisfying some constraint, such as “How can I get the best price for a deal containing TV and DVD?”.

In contrast, with a monotonically decreasing function, such as one representing likelihood of occurrence, we are interested in the top-$k$ flows, i.e. those that are most likely to occur in practice (note that
analogously, there may be no flow that is least likely).

Given a weighted BP specification \( s \) and a number \( k \), we use top-k(s) to denote the set of \( k \) EX-flows in \( \text{flows}(s) \). We refer to the problem of identifying top-k(s) (given the above input) as \textsc{Top-k-FLOWS}.

### 4.2 Finding Top-K Flows

We start by solving \textsc{Top-k-FLOWS} for BP specifications accompanied by a history-independent \textit{cWeight} function; bounded-history and unbounded-history functions are considered in Section 4.3.

We give next an efficient algorithm for \textsc{Top-k-FLOWS}. As the algorithm is intricate, we present it gradually: we start with a simple variant, explain its operation, then analyze its correctness and show cases where it fails. Consequently, we refine it to obtain the correct algorithm.

#### 4.2.1 First (Unsuccessful) Attempt

The \texttt{FindFlows} procedure given in Algorithm 1 attempts to compute the top-k EX-flows of a given BP. (Ignore for now the boxed lines). It is given as input a BP specification \( s \), a weight function over its EX-flows, represented by \textit{cWeight} and \textit{aggr}, and the number \( k \) of requested results. Its output is an ordered queue \( \text{Out} \) of results. The algorithm uses a function \texttt{AllExps} that, given a partial EX-flow \( e \) returns all EX-flows \( e' \) s.t. \( e \rightarrow e' \), along with their corresponding guarding formulas \( F' \).

The algorithm operates in the spirit of the \textit{A*} [28] search algorithm: it considers, in a greedy manner, the possible EX-flows of \( s \), and computes the \textit{fWeight} of each. The computation may be viewed as a gradual generation of a \textit{search tree}, whose nodes correspond to possible

\(^2\text{Certain EX-flows may have equal weights, which implies that there may be several valid solutions to the problem, in which case we pick one arbitrarily.}\)
(partial or full) EX-flows. The search tree root is an EX-flow consisting of only the BP root; at each step, we choose a leaf $e$ in the search tree, and consider all of its possible next expansions. Each such expansion $e'$ becomes a child of $e$, and the computation continues. We next explain how this search tree is generated.

The algorithm maintains a priority queue $\text{Frontier}$ of (partial) EX-flows, ordered by $f\text{Weight}$, which still need to be considered. Initially it contains a single partial EX-flow, containing only the BP root (line 2). At each step, we pop the highest weighted flow $e$ from $\text{Frontier}$ (line 4). If $e$ is a full flow, we insert it to the output queue $Out$ (line 6). This is justified in Theorem 4.2.1 below. Otherwise, we invoke $\text{HandleExps}$ over $e$. $\text{HandleExps}$, depicted in Algorithm 2, considers all possible direct expansions $e'$ of $e$, obtained by choosing some implementation, guarded by $F'$, at the next-to-be-expanded node (lines 1-2). It computes the weight of each such EX-flow and inserts it to (the global variable) $\text{Frontier}$ (lines 3-4).

**Properties of the Basic Algorithm** We next analyze Algorithm 1. On the positive side, we show that the algorithm inserts EX-flows to $Out$ in a correct order. As a consequence, the algorithm is valid, i.e. if it terminates, $Out$ contains the top-k EX-flows.

**Theorem 4.2.1** At any point of the execution of Algorithm $\text{FindFlows}$, there exist no $e, e' \in \text{flows}(s)$ such that $e \in Out$, $e' \in \text{flows}(s) - Out$, and $f\text{Weight}(e) < f\text{Weight}(e')$.

**Proof.** The proof works by contradiction. Let us assume the existence of such $e, e'$. We say that $e''$ is an ancestor of $e'$ if $e'' \rightarrow^* e'$, and denote by $e''$ the lowest ancestor of $e'$ that was already in $\text{Frontier}$ at the time of moving $e$ to $Out$ (there exists such flow, perhaps consisting only of the root). $f\text{Weight}(e) \geq f\text{Weight}(e'')$, otherwise $e$ would have not been moved to $Out$. But $f\text{Weight}(e'') \geq f\text{Weight}(e')$ by monotonicity, thus $f\text{Weight}(e) \geq f\text{Weight}(e')$. $\Box$
Algorithm 1: FindFlows

**Input:** \( s; c\text{Weight}; \text{aggr}; k \)

**Output:** \( \text{Out} \)

1. Initialize \( \text{Frontier} \) and \( \text{Out} \);
2. Push \((s_0, 1_{\text{aggr}})\) in \( \text{Frontier} \);
3. while \(|\text{Frontier}| > 0 \land |\text{Out}| < K\) do
   4. \((e, w_e) \leftarrow \text{pop}(\text{Frontier})\);
   5. if \(e\) is a full EX-flow then
      6. insert \((e, w_e)\) into \(\text{Out}\)
      7. \text{HandleFullFlow}(e, w_e);
   else
      8. \text{HandleExps}(e)
      9. \text{refinedHandleExps}(e);
   end
4. return \(\text{Out}\);

Algorithm 2: HandleExps

**Input:** \( e \)

1. \( \text{Expansions} \leftarrow \text{AllExps}(e) \);
2. foreach \((e', F') \in \text{Expansions} \) do
   3. \( r_{e'} \leftarrow \text{aggr}(r_e, c\text{Weight}(e, F')) \);
   4. insert \((e', r_{e'})\) into \(\text{Frontier}\);
5. end
On the negative side, the algorithm has two significant drawbacks. First, when the BP contains recursion, the algorithm may repeatedly choose recursive expansions and fail to halt. Second, note that the algorithm explicitly generates EX-flows. As the size of even a single EX-flow may be exponential in the BP size, this may be costly (even for non-recursive BP specifications).

We next show an example for a case where the algorithm does not terminate.

**Example 4.2.2** Consider, for instance, the following recursive BP specification, with cWeight values in the range [0,1], and aggr = *. Its root activity A has two possible implementations: the first, guarded by a formula with cWeight of 0, consists of a single atomic activity a. The second, guarded by a formula with cWeight of 0.5, consists of a recursive invocation of A. The algorithm will keep considering recursive expansions of A, each time obtaining EX-flows with decreasing weight, but nevertheless higher than 0, and will never terminate.

We next present a refined version of the algorithm that ensures both termination and efficiency, while maintaining a compact representation of the EX-flows.

**4.2.2 The Refined Algorithm**

The refined version of the algorithm is based on two observations. Before presenting them, recall that we have defined above the fWeight function only for EX-flows that originate from the BP root activity. However, the definition naturally extends to sub-flows originating from an arbitrary compound activity node n in the EX-flow (treating it as a root). Namely, all cWeight values are defined as before, with fWeight aggregating only cWeight values within the sub-flow rooted at n.

**Observation 1.** We first observe that distinct activity nodes n, n' (appearing in the same or in different EX-flows) sharing the same
tivity name, are in fact equivalent, in the sense that every sub-flow that may originate from \( n \) may also originate from \( n' \), having exactly the same \( fWeight \). An algorithm may exploit this to compute \( fWeight \) values just once for each activity name.

**Observation 2.** Second, observe that the monotonicity of \( fWeight \) facilitates incremental-style computation, i.e. the \( j \)'th ranked sub-flow rooted at an activity node \( n \) can use only better ranked sub-flows rooted at nodes sharing the same activity name as \( n \). More formally (recall that \( \lambda(n) \) stands for the activity name labeling a node \( n \)):

**Lemma 4.2.3**  
1. There exists a best ranked (top-1) EX-flow originating at \( n \) that contains no occurrence of any other node \( n' \) such that \( \lambda(n') = \lambda(n) \).

2. For \( j > 1 \), there exists a \( j \)'th ranked flow originating at \( n \) such that for any occurrence of a node \( n' \) for which \( \lambda(n') = \lambda(n) \) in it, the sub-flow rooted at \( n' \) is one of the top \( (j-1) \) such sub-flows.

**Proof.** Let \( n \) be a compound activity node, and let \( a \) be the compound activity name labeling \( n \).

1. Assume that there exists a best weighted EX-flow \( e \) rooted at \( n \) violating condition (1) above, i.e. \( e \) contains another node \( n' \) labeled by \( a \). Denote the sub-flow rooted at \( n' \) as \( e' \). Note (see observation 1 above) that every EX-flow rooted at \( n' \) may also be treated as an EX-flow rooted at \( n \), and its \( fWeight \) remains intact.

   Further note that \( e' \subseteq e \), and thus (from the monotonicity of \( fWeight \)) it follows that \( fWeight(e') \geq fWeight(e) \), and further \( e' \) contains one less occurrences of nodes labeled by \( a \). If \( e' \) still contains a node labeled by \( a \), we may repeat the process to eliminate it, and so forth. If the original EX-flow \( e \) contains \( m \) nodes

---

3Recall that we are currently assuming that \( cWeight \) is history-independent, and relax this assumption in Section 4.3.
labeled by \( a \) (in addition to the root \( n \)), this process terminates after \( m \) steps, with an EX-flow rooted at \( n \) that contains no nodes labeled by \( a \) (apart for \( n \) itself). The \( \text{fWeight} \) of the obtained EX-flow is greater or equal to the \( \text{fWeight} \) of \( e \).

2. Let \( e \) be a \((j + 1)\)th EX-flow rooted at \( n \) that violates the lemma, i.e. it contains another node \( n' \) labeled by \( a \) such that the sub-flow rooted at \( n' \) is not one of the top-\( j \) such sub-flows. Then we may replace the sub-flow rooted at \( n' \) by one of the top-\( j \) sub-flows. By monotonicity of the weight function, the obtained flow \( e' \) bears at least the same \( \text{fWeight} \) as \( e \); by the inductive assumption there exists a set of top-\( j \) EX-flows such that \( e' \) may not be obtained in the above manner, and thus can serve as the \((j+1)\)-ranked flow in this set. We obtain a set of top-(\( j+1 \)) flows satisfying the lemma constraint.

\( \square \)

Let us illustrate the implications of this Lemma.

**Example 4.2.4** Re-consider example 4.2.2, and recall that while trying to retrieve the top-1 EX-flow rooted at \( A \), \texttt{FindFlows} has encountered a recursive invocation of \( A \), and has examined possible EX-flows of the latter, thus resulting in an infinite loop. Following Lemma 4.2.3, this is redundant: to compute the top-1 EX-flow one may avoid considering flows that contain a recursive calls to \( A \). The top-2 flow may contain a recursive invocation of \( A \), but the only sub-flow that needs to be considered as potential expansion for this occurrence of \( A \) is the (already computed) top-1 flow, and so on.

**Algorithm** Following observation 1, we define an EX-flows table, \( FTable \), which (compactly) maintains the top-k (sub)flows rooted at each compound activity name appearing in the BP specification. It has rows corresponding to compound activity names, and columns ranging
from 1 to $k$. Each entry contains a pointer to the corresponding sub-flow.

The refined TOP-K algorithm, depicted in Algorithm 3, operates in two steps: first it calls a refined version of FindFlows which computes a compact representation of the top-$k$ EX-flows, then it calls EnumerateFlows (depicted below) to explicitly enumerate the EX-flows from this compact representation.

**Algorithm 3: TOP-K**

**Input:** $s$; cWeight; aggr; $k$

**Output:** top-$k$

1. Initialize $FTable$;
2. $tmp ← refined\text{FindFlows}(s, \text{cWeight, aggr, } k)$;
3. $\text{top-k} ← Enumerate\text{Flows}(tmp)$;
4. return $\text{top-k}$;

First, we explain the $\text{refinedFindFlows}$ procedure. The changes, relative to what we have previously seen in Algorithm 1, are depicted in the boxed lines (of this algorithm): The treatment of full EX-flows (in line 6) is replaced by a more refined treatment, performed by the $\text{HandleFullFlow}$ procedure (depicted in Algorithm 5). The treatment of partial EX-flows is also refined: the generation of candidate expansions (line 8), previously performed by $\text{HandleExps}$ is now performed by the refined variant $\text{refinedHandleExps}$ (depicted in Algorithm 4). These two new procedures will be in charge, among others, of the construction of $FTable$. The variable $Out$ (that previously explicitly enumerated the top-$k$ EX-flows) now contains a compact representation of the flows, based on this $FTable$. We next explain these two procedures.

**refinedHandleExps** The refined treatment of candidate expansions is detailed in Algorithm 4. For the next-to-be-expanded node $v$ in $e$ (line 1), we look up its entry in $FTable$ (line 2). If no entry is found, it
Algorithm 4: refinedHandleExps

Input: $e$

1 $v \leftarrow \text{getNext}(e)$ ;

2 $\text{TableRow} = FTable.\text{findEntry}(v)$ ;

3 if $\text{TableRow} = \text{NULL}$ then

4 Create a new frow for $v$ in $FTable$ ;

5 $\text{HandleExps}(e)$ ;

6 end

7 else

8 $\text{UnhandledExp} \leftarrow \{ e'_v \in \text{TableRow} | e'_v$ was not chosen for $v$

when preceded by $e$} ;

9 if $\text{UnhandledExp} = \text{NULL}$ then

10 insert $((e, w_e), v)$ into $\text{OnHold}$ ;

11 end

12 else

13 $e''_v \leftarrow \text{top}(\text{UnhandledExp})$ ;

14 $e' \leftarrow \text{expand e by pointing v to } e''_v$;

15 $w_{e'} \leftarrow \text{aggr}(w_e, f\text{Weight}(e''_v))$ ;

16 insert $(e', w_{e'})$ into $\text{Frontier}$;

17 end

18 end
Algorithm 5: HandleFullFlow

Input: $e, w_e$

1. insert $(e, w_e)$ into Out;

2. foreach node $n \in e$ do
   3. \hspace{1em} $e_{\text{rooted}} \leftarrow$ the sub-flow of $e$ rooted at $n$;
   4. \hspace{1em} $w_{e_{\text{rooted}}} \leftarrow fWeight(f_{\text{rooted}})$;
   5. \hspace{1em} if not ($e_{\text{rooted}} \in FTable$) then
      6. \hspace{2em} FTable.update($n, e_{\text{rooted}}$);
   7. \hspace{1em} end
   8. foreach $(e', n) \in \text{OnHold}$ do
      9. \hspace{2em} insert $e'$ into Frontier;
   10. end

end

means that we haven’t encountered yet an equivalent node (i.e. a node sharing the same compound activity name) during the computation. We thus create a new row in $FTable$ for this activity name (line 4). Entries in this row will be filled later, when corresponding full flows are found. Then, we process $v$’s expansions as before (line 5) using Algorithm 2.

Otherwise, if the appropriate row already exists in the table, we consider the partial EX-flows that appear in this row but were not yet considered for expanding $e$ (line 8). If no such EX-flow exists, (although the table entry itself does exist), it means that $e$ was previously reached when expanding some other node $v'$ (which appears in $e$ as well). Following observation 2, we may compute the next best EX-flow without computing further expansions of $e$. Thus, we put $e$ on hold (line 10). It will be released only later, upon finding a full flow originating in $v'$ (see below). Else (i.e. an unused EX-flow does exist), we take the highest ranked such EX-flow and simply “hang” it on $v$, that is, we make $v$ point to this flow (lines 13-14). Finally, we now compute the weight of
the obtained EX-flow (line 15) and insert it into Frontier, for further expansions (line 16).

**HandleFullFlow** The refined treatment of full EX-flows is detailed in Algorithm 5. As before, full EX-flows are inserted into Out (line 1). Additionally, for every node appearing in the EX-flow, HandleFullFlow fills in the corresponding entry in FTable. The sub-flow rooted at the node is inserted into the table, if it does not appear there already (lines 3-7). Last, all EX-flows that are on hold due to a node participating in the full EX-flow, are returned to Frontier (lines 8-9).

To conclude, let us explain how EnumerateFlows extracts the flows from their compact representation in Out.

**EnumerateFlows** The EX-flows in Out contain activity nodes that point to entries in the FTale, describing the sub-flow rooted at the nodes. The nodes in each such description also possibly point to other table entries describing the structure of the sub-flow rooted at them and so on. EnumerateFlows thus simply follows these pointers to materialize the full EX-flow. This pointer chasing is guaranteed to terminate since, following observation 2, pointers in FTale induce no loops.

To conclude the algorithm presentation, we next show an example of its execution.

**Example 4.2.5** Consider again the BP specification from Example 4.2.2 and assume that we want to compute its top-2 flows. Starting with the root activity A, refinedHandleExps looks for its equivalence class in FTale. Since the table is empty, a new equivalence class containing only A is defined, and a row is generated for it in FTale. Then, the possible expansions of A are examined (Line 5 of refinedHandleExps), and two possible expansions are added to the Frontier: one full EX-flow (denote by f1), corresponding to the implementation choice leading to the atomic activity a, and one partial flow (denote by f2), containing
a recursive invocation of $A$. Next, $f_2$, having the better $cWeight$, is popped out of Frontier. The next activity to be expanded in $f_2$ is $A$, but as it appears in the table, this expansion is removed from Frontier and stays on hold (Line 10). Then $f_1$ is the only EX-flow in Frontier and is popped, then inserted to Out by HandleFullFlow (Indeed, $f_1$ is the top-1 flow). Now, the entry in FTaTable for the top-1 flow rooted at $A$ is updated to be $fWeight(f1)$, and $f_2$ is “released” from OnHold and returned to Frontier (Lines 8-9 of HandleFullFlow), for computation of the 2nd-best EX-flow.

**Correctness** We next show that Algorithm TOP-K is correct.

**Theorem 4.2.6** Algorithm TOP-K halts and returns the top-k weighted EX-flows of the input BP s.

**Proof.** Following the validity of our basic $A^*$-like Algorithm, to prove the theorem it remains to show that (1) the revised version terminates and (2) the order in which partial EX-flows are inserted to Out, in the revised algorithm, does not harm the validity. We next prove both claims.

1. Consider the computation of the top-1 flow rooted at a compound activity name $a$. Assume that during this computation, we get to a flow $e$ whose next node to be expanded $v$ is also labeled by $a$. At this point the FTaTable row corresponding to $a$ is empty. Consequently, at Lines 9-11 of refinedHandleExps, $e$ will be put on hold and is not further expanded throughout the top-1 computation. This will happen for each such $e$. Similarly, if during the computation of the top-1 flow rooted at a compound activity name $a$, we start computing the top-1 flow rooted at $b$, we will not further expand such flows that contain another occurrence of either $a$ or $b$. This means that, in fact, any implementation choice that renders the obtained flow recursive, is not expanded further.
The number of EX-flows rooted at $a$ that contain no (mutual) recursion within them is finite, and the algorithm is thus guaranteed to halt. Similarly whenever we compute the top-$j$ flow rooted at $a$, the only flows rooted at $a$ that we attempt to use as sub-flows are those already computed, and the algorithm is thus guaranteed to halt.

2. We have shown above (Theorem 4.2.1) that the basic $A^*$-like algorithm inserts flows to $Out$ in a correct order. Algorithm $TOP-K$ inserts flows to $Out$ at the same order as the basic algorithm, except for the case where flows are put in the $OnHold$ queue. However, recall that flows put in $OnHold$ are those that violate a constraint of Lemma 4.2.3. Thus, there exists, at each point, a top-$j$ flow that is not one of those put in $OnHold$, and this flow is found by the algorithm following Theorem 4.2.1. Note that the flows that are put in $OnHold$ may be parts of lower-ranked flows, and are thus returned to Frontier for further computation (Line 9 of Algorithm $HandleFullFlow$).

$\square$

**Complexity** We next provide a worst-case complexity analysis of the algorithm, as follows.

**Theorem 4.2.7** Given a BP specification $s$ (with $cWeight$ and $aggr$) and a number $k$, the time complexity of $\text{refinedFindFlows}$ in Algorithm $TOP-K$ is $O(|s|^3 * k)$.

**Proof.**

The number of entries in $FTable$ is $k * |s|$. Now, for each flow node $v$ considered during the course of the algorithm execution, either it already appears in $FTable$, or it does not. The case where the sub-flow requested for $v$ does not appear in the table may only happen
$k \times |s|$ times, while computing the top-$k$ flows rooted at $v$. The cost of computation for such cases is $O(|s|)$ for searching the table, (assuming that we have an index that allows, in $O(1)$ time, to get the worst ranked entry for a given row; otherwise there may be an additional factor of $k$) and then $O(1)$ of additional computation, considering direct expansions of $v$, a total of $O(k \times |s|^2)$.

If the sub-flow considered for $v$ does already appears in $FTable$, we only need to point the implementation of $v$ to the sub-flow that was already computed ($O(1)$). We next consider the number of times that this scenario may occur.

We start by considering the computation of top-1 flows. Now, consider some activity name $a$. Say that we have encountered some node $n$ with activity name $a$, and then, before we are done computing the top-1 flow rooted at $n$, we have encountered, at another point of the search tree, another node $n'$ with label $a$. The course of the algorithm execution follows observation 2 above: it suspends the computation for the top-1 flow of $n'$, until computation of the top-1 flow of $n$ is done (by putting $n'$ “on hold”, line 10 of Algorithm refinedHandleExps. The number of such suspensions, while computing the top-1 flow of $n$, is bounded by the size of the specification $s$, for each such $n$. The same argument holds for computation of the $i$'th highest weighted flow, for each $i = 1, ..., k$, leading to a total bounded by $O(k \times |s|^3)$ for this case. The overall time complexity is thus polynomial in $|s|$ and $k$. □

Similar arguments can be used to show an additional characteristic of the algorithm that may be thought of as the analog of the notion of polynomial delay [85].

**Theorem 4.2.8** The delay between inserting the $i$'th and the $(i+1)$'th result to Out is polynomial in $|s|$.

**Proof.** Note that Algorithm refinedFindFlows computes flows in an incremental manner: to compute the $i+1$'th ranked, we start from the
flow that appears first in *Frontier*, after computing the \(i\)’th ranked flow; for this flow, we try expansions that were not previously used (line 8 of `refinedHandleExps`). The complexity of computation of the \(i + 1\)’th ranked flow, given the \(i\)’th ranked flow (i.e. the delay) is obviously polynomial in \(|s|\) and in \(k\): this follows from the incremental nature of computation, combined with Thm. 4.2.7. It remains to show that the delay is in fact *independent of \(k\)*. Similarly to above, we assume the existence of an index that allows, in \(O(1)\) time, to retrieve (pointer to) the point at the table containing the highest ranked un-handled flows appearing in a given row of *FTable* (such index may be implemented). At each stage, we only need to consider the highest ranked such un-handled flow (line 13), and thus never consider flows computed in steps earlier than the last step for every activity name. This number is thus bounded by \(|s|\) and independent of \(k\). □

The last part of Algorithm **TOP-K** involves the invocation of `EnumerateFlows` that outputs the top-\(k\) flows. Recall that the size of a single flow in this set may be exponential in the BP specification size (even in the absence of recursion), and thus one cannot expect an enumeration algorithm that is polynomial in the BP specification size. However, it is easy to observe that the complexity of `EnumerateFlows` is linear in the *output* size, as it simply follows pointers to materialize each output EX-flow.

### 4.3 History-Dependent Weights

So far we have assumed that the `cWeight` function given as input is history-independent, i.e. the `cWeight` of each implementation choice is the same for every occurrence of the corresponding compound activity. As explained above, this is often not the case in practice. We next discuss the two additional classes of `cWeight` functions defined above, and consider the adaptation of our top-\(k\) algorithm to account for these classes.
First, we may show that if \( cWeight \) may be unbounded-history, then there exists no correct algorithm for \( \text{TOP-K-FLOWS} \). To that end, we define the decision problem of \( \text{BEST-FLOW} \), which tests, given a weighted BP specification \( s \), and a threshold \( t \), whether there exists an EX-flow in \( \text{flows}(s) \) whose \( fWeight \) is greater than \( t \).

**Theorem 4.3.1** If the \( cWeight \) function given as input may be unbounded-history, then \( \text{BEST-FLOW} \) is undecidable.

**Proof.** The proof is by a reduction from the halting problem. Given a Turing Machine \( M \), the idea is to “encode” \( M \) using the BP specification. The states of \( M \) are represented by activity names; implementations model the transitions between states, as well as changes to the tape and to the head location; and the history of flow is utilized to allow “read” operations from the tape.

More formally, given a Turing Machine \( M \) with a set of states \( Q \), an initial state \( q_0 \in Q \), an accepting state \( q_F \in Q \), a tape alphabet \( \Gamma \) and a transition function \( \delta \), we generate a BP specification whose set of compound activity names corresponds to \( Q \), and additionally it contains an atomic activity \( a \). The implementations set of each compound activity corresponding to a state \( a \), contains a single-node implementation for each activity name \( a' \) (possibly \( = a \)) such that there is a transition from \( a \) to \( a' \) according to \( \delta \). The \( cWeight \) value of the formula \( f \) guarding each such implementation is dictated by the preceding flow \( e \), and \( \delta \); \( cWeight(e, f) = 1 \) if and only if, when following, in \( M \), the transitions dictated by the EX-flow \( e \), the symbol under the tape head is \( b \), and \( \delta(a, b) = a' \) (meaning that there exists a transition from state \( a \) to state \( a' \) given that the symbol \( b \) is under the tape head). Otherwise \( cWeight(e, f) = 0 \). As \( cWeight \) function is unbounded-history, it is allowed to determine its value according to the entire preceding flow. For the accepting state, its single implementation consist of the atomic activity \( a \), with \( cWeight \) of 1. We use multiplication for aggregation,
and seek for full flows with cWeight higher than 0. An accepting flow of such cWeight exists if and only if there exists an accepting run of the TM. □

Fortunately, recall that in practical cases the cWeight function is in fact bounded-history. In the sequel we consider such bounded-history cWeight functions, and explain how to adapt our top-k algorithm to weighted BPs with such weight functions. We prove the following theorem:

**Theorem 4.3.2** Given a weighted BP specification s with bounded-history cWeight function (with bound b), and given also a number k, top-k(s) can be computed in time polynomial w.r.t. |s|, k, and the output size, and exponential w.r.t. b.

**Proof.**

We next explain the revised top-k algorithm that accounts for bounded-history cWeight functions.

**Revised algorithm - general framework** Algorithm TOP-K depicted above has assumed that the cWeight function is history-independent. To account for history-dependent functions, we shall utilize the notion of partial tracing as a tool. Our definitions for partial (semi-naive, selective) tracing extend naturally to the context of BP specifications with annotating guarding formulas, by allowing renaming functions (deletion sets) to apply also to (respectively include) guarding formulas.

We then define the notion of a weighted BP specification with semi-Naive tracing capturing a given set of EX-flows. This notion will be useful in the sequel.

**Definition 4.3.3** A weighted BP specification s (and weight functions (cWeight, aggr, fWeight)) along with a renaming function π over its activities captures a (possibly infinite) set of EX-flows T, with respect to a weight function ∆ over T if
1. \( \text{semi-naive}(s, \pi) = T \), i.e., for each \( t \in \text{semi-naive}(s, \pi) \) there exists \( e \in T \) such that \( t = e \), and vice versa, i.e. for each \( e \in T \) there exists \( t \in \text{semi-naive}(s, \pi) \) such that \( t = e \).

2. \( \forall e \in \text{flows}(s) f\text{Weight}(e) = \Delta(\Pi(e)) \) where \( \Pi(e) \in T \) is obtained from \( e \) by replacing each activity name \( a \) in \( e \) by \( \pi(a) \).

The BOUNDED-TOP-K algorithm, computing top-k flows for weighted BP specifications with bounded-history \( c\text{Weight} \) functions, then operates in two steps:

1. The first step of the algorithm, namely Algorithm \text{COMPILE-HISTORY-INDEPENDENT}, compiles, given a weighted BP \( s \) with activities \( a_1, \ldots, a_n \), and weight functions \((c\text{Weight}, \text{aggr}, f\text{Weight})\), such that \( c\text{Weight} \) is bounded-history, denote the history bound of \( c\text{Weight} \) by \( b \). We generate a new weighted BP specification \( s' \), with renaming function \( \pi \) and history-independent \( c\text{Weight}' \), such that \( s', \pi \) captures \( \text{flows}(s) \) (w.r.t. \( f\text{Weight} \)).

2. The second step of the algorithm, namely Algorithm \text{SEMI-NAIVE-TOP-K} is an adaptation of Algorithm TOP-K, applied to find the top-k semi-naive EX-traces of \( s', \pi \).

We next explain both steps.

\textbf{Algorithm COMPILE-HISTORY-INDEPENDENT}  Given a weighted BP specification \( s \) with activities \( a_1, \ldots, a_n \), and weight functions \((c\text{Weight}, \text{aggr}, f\text{Weight})\), s.t. \( c\text{Weight} \) is bounded-history, denote the history bound of \( c\text{Weight} \) by \( b \). We generate a new weighted BP specification \( s' \), with renaming function \( \pi \) and history-independent \( c\text{Weight}' \), such that \( s' \) along with \( \pi \) capture \( \text{flows}(s) \) with correct weights. The different components of \( s', \pi \) are as follows:

\[\text{This equality is an equality between EX-flows, corresponding to a node-label, edge-relation and guarding formulas preserving isomorphism}\]
Activities Names. The activities names in $s'$ are tuples of the form $(a, pre = [pre^1_a, ..., pre^n_a], post = [post^1_a, ..., post^m_a])$ where $a$ is an activity name, $pre^i_j$ denotes a formula guarding the implementation chosen for $a_i$ in its previous $j$ expansions, prior to expanding $a$, and $post^i_j$ denotes a formula guaranteed to guard the implementation chosen for $a_i$ in its next $j$ expansions. The idea is that $pre$ encodes all information required for computing $cWeight$ of $a$’s guarding formulas, and $post$ encodes all information required for activities that follow $a$ in the flow. We use $pre^i_j = \perp$ if $a_i$ was not expanded $j$ steps before the flow reaches $a$, and $post^i_j = \perp$ if $a_i$ will not be expanded $j$ times before the execution of $a$ terminates.

Guarding Formulas and Weight Function. Let $f_1, ..., f_l$ be the guarding formulas appearing in $s$, then guarding formulas in $s'$ are of the form $(f_i, pre)$ where $pre$ is a vector of formulas of size $b$. The $cWeight'$ function is defined as $cWeight'(f_i, pre) = cWeight(f_{pre}, f_i)$ where $f_{pre}$ is some arbitrary partial EX-flow of $s$ for which the next node to be expanded is guarded by $f_i$, and in which the last implementation were guarded by the sequence of formulas in $pre$ $^5$. For aggregation, $aggr' = aggr$, and the obtained flow weight function is called $fWeight'$.

Implementation Function. Next we construct the implementations of $(a, pre, post)$ in $s'$. For each implementation $F_i$ of $a$ in $s$ (guarded by $f_i$), we create a set of new implementations, all guarded by $f_i$. Each implementation is obtained by annotating each activity $b$ in $F_i$ with vectors of pre and post conditions. The construction is as follows: (a) if $r$ is the root of an implementation $F_i$ of $(a, pre_a, post_a)$ guarded by $f_i$ (note that many such activity names are created for any activity name $a$, differing in

\footnote{This weight is uniquely defined, since $cWeight$ has an history bound of $b$}
their pre- and post-condition vectors), then the pre-condition of \( r \) is obtained from \( \text{pre}_a \), shifted by one step, recording \( F_i \) (and possibly deleting some formula from the vector, if reached the bound), (b) if there exists an edge from some node \( n \) to some node \( n' \) in \( F_i \), the post-condition annotating \( n \) complies with the pre-condition annotating \( n' \), and (c) if \( e \) is the end node of \( F_i \), the post-condition annotating it complies with \( \text{post}_a \).

- Renaming Function. We define \( \pi(a, \text{pre}, \text{post}) = a \) for each activity name \((a, \text{pre}, \text{post})\).

We claim that \( s', \pi \) constructed by the algorithm \textit{capture flows}(s) with respect to \( f\text{Weight} \). To observe that this holds, first note that every implementation of \( a' = (a, \text{pre}, \text{post}) \) in \( s' \) was obtained from an implementation of \( a \) in \( s \) by replacing all activity names \( b \) in this implementation by some activity names \( b, \text{pre}', \text{post}' \). Thus, by applying \( \Pi \) over the corresponding implementation of \( a' \) one obtains every possible implementation of \( a \), and only such implementations. I.e. \( \text{semi-naive}(s', \pi) = \text{flows}(s) \). As for weight of flows, let \( e \in \text{flows}(s) \) and let \( e' \in \text{flows}(s') \) such that \( \Pi(e') = e \). We next show, by induction on the size of \( e \), that \( f\text{Weight}(e) = f\text{Weight}'(e') \): for the induction basis, a flow consisting only of the root activity of \( s \) (\( s' \)) has a weight of \( 1_{\text{aggr}} = 1_{\text{aggr}'} \); now, assume that \( f\text{Weight}(e_1) = f\text{Weight}'(e'_1) \) for \( e_1 \) (\( e'_1 \)) that bears exactly the same structure as \( e \) (\( e' \)) except for its last implementation choice. Then this last implementation bears, in \( s \), a weight of \( \text{aggr}(f\text{Weight}(e_1), c\text{Weight}(e_1, f)) \) where \( f \) is its guarding formula. The corresponding implementation in \( s' \) bears a weight of \( \text{aggr}(f\text{Weight}'(e'_1), c\text{Weight}((f, \text{pre}))) \) where \( \text{pre} \) is the \( \text{pre} \) vector encoded within the activity in \( e'_1 \) whose implementation was chosen to form \( e \). But \( c\text{Weight}(e_1, f) = c\text{Weight}((f, \text{pre})) \), because the algorithm construction of the implementation function defines a pre-condition vector that is consistent with the sequence of implementation
choices made in the course of the flow (in this case $e_1$).

This concludes the description of Algorithm \textsc{Compile-History-Independent}.

\textbf{Algorithm \textsc{Semi-Naive-Top-K}} The second part of the algorithm finds the top-k EX-traces defined by $(s', \pi')$. Recall that $s'$ is history-independent, and a naive solution would attempt to apply Algorithm \textsc{Top-K}, as is, over $s'$, to find its top-k EX-flows; then apply $\Pi'$ over each such EX-flow to obtain the top-k EX-traces of $(s', \pi')$ which are also the top-k EX-flows of $s$. The crux here is that distinct EX-flows of $s$ may be mapped by $\Pi'$ to the same EX-trace, thus this algorithm may fail to find $k$ distinct EX-traces.

Instead, we revise the algorithm as follows. Recall that Algorithm \textsc{Top-K} makes use of an \textit{FTable} whose rows correspond to activity names and whose columns are marked 1 to $k$. We maintain an additional “shadow” table \textit{STable}, in which every row corresponds to an activity name $b = \pi(a)$ where $a$ is an activity of $s'$, and its columns are marked 1 to $k^*N(b)$, where $N(b)$ is the number of distinct activity names mapped by $\pi$ to $b$. We now execute Algorithm \textsc{Top-K} on $s'$, but whenever the algorithm adds a (sub-)flow $e'$ to \textit{FTable}, at a row corresponding to an activity $a$ of $s'$, we add $\Pi(e')$ at the appropriate location (according to its weight), in the row of $\pi(a)$ at \textit{STable}. We further change the pointer, set by \textsc{Top-K} to point at the entry in \textit{FTable}, to point to the newly added entry in \textit{STable}. This pointer will be used for checking whether a sub-flow rooted at $a$ was used within a computed EX-flow (Line 9 of Algorithm \textit{refinedHandleExps}) as well as in the flow enumeration (Algorithm \textit{EnumerateFlows}).

\textbf{Complexity} The complexity of Algorithm \textsc{Compile-History-Independent} is polynomial in the BP specification size $|s|$ and exponential in the history bound $b$, as each constructed activity name is composed out of a

\footnote{$\Pi'(e)$ is obtained from $e$ by replacing each activity name $a$ in $e$ by $\pi'(a)$.}
The complexity of Algorithm SEMI-NAIVE-TOP-K is polynomial in $k$, in the size of its input $|s'|$, in the size of its output and in $\max_a N(a)$ where $a$ is an activity name and $N(a)$ is the number of distinct activity names mapped by $\pi$ to $a$. In our case $|s'|$ is polynomial in the BP specification size $|s|$ and exponential in the history bound $b$, and $\max_a N(a)$ is exponential in $b$ (as all activity names of the form $(a, pre, post)$ are mapped to $a$, and their number equals the number of distinct such $pre$ and $post$ vectors, in turn exponential in $b$).

This concludes the proof of Theorem 4.3.2.

\[\square\]

We further show that this exponential dependency on the history size is inevitable, as the following theorem holds (recall that BEST-FLOW is the decision problem corresponding to TOP-K-FLOWS, i.e. is the problem of deciding the existence of an EX-flow of the given BP specification, whose weight is greater than some given bound).

**Theorem 4.3.4**  For bounded-history cWeight functions with bound $b$, BEST-FLOW is NP-hard in $b$.

**Proof.**  We use a reduction from Set Cover. Given an instance of set cover, namely a set $X = \{X_1, ..., X_n\}$ of items, a set of subsets $S = \{S_1, ..., S_m\}$ and a bound $B$, we construct a BP as follows: its activity names are $S_0$ (root), $S_1, ..., S_m$ (compound) and $a$ (atomic). Each $S_i$ ($i = 0, ..., m$) bears $2 \times m + 1$ implementations: for each $j = 1, ..., m$, $S_i$ has two implementations, each consisting of a single node whose activity name is $S_j$: the first is guarded by a formula “$S_i = \text{chosen}$”, and the second by “$S_i = \text{not chosen}$”; the last implementation of $S_i$, guarded by “$S_i = \text{done}$”, consists of a single node whose activity name is $a$. The cWeight of “$S_i = \text{chosen}$” and of “$S_i = \text{not chosen}$” is 1, and that of “$S_i = \text{done}$” depends on the last $B$ choices: it is 1 if and only if the set of $S_i$ sets for which “$S_i = \text{chosen}$”, within these
last \( B \) choices, covers \( X \). Otherwise, its \( cWeight \) is also 0. We use multiplication for \( \text{aggr} \). It is easy to observe that there exists an EX-flow of \( fWeight \) greater than 0.5 if and only if there exists a set cover of size smaller than \( B \). (The EX-flow consists of an implementation choice guarded by “\( S_i = \text{chosen} \)” for each \( S_i \) in the cover, and “\( S_i = \text{not chosen} \)” for each \( S_i \) not in the cover).

\[ \square \]

### 4.4 Multiple expansion sequences

We have assumed in all of the above, for simplicity of presentation, the existence of a total order over the expansion of activities. We next withdraw this assumption, and explain the needed adjustments to our definitions and algorithms.

- The \( cWeight \) function, previously defined as \( cWeight(e, f) \) for an EX-flow \( e \) and a formula \( f \), should now be defined as \( cWeight(e, n, f) \) with \( n \) being a node of \( e \) and \( f \) being a guarding formula of an implementation of \( \lambda(n) \). \( cWeight(e, n, f) \) is the weight of \( f \), given that \( n \) was chosen for expansion in \( e \).

- \( fWeight \) is now defined as
  \[
  fWeight(e) = \max_{e' | e \rightarrow e} \text{aggr}(fWeight(e'), cWeight(e', n', f))
  \]
  where \( n' \) is the node of \( e' \) expanded to form \( e \) and \( f \) is the guarding formula on the corresponding expansion of \( n' \).

- Given a partial flow \( e \), Algorithm \text{TOD-P-K} now examines the expansions of all nodes of \( e \), rather than just those of a single node that is the next-to-be-expanded in \( e \), and selects the best weighted one of these.
Chapter 5

Evaluating Top-k Queries

In the previous chapters we have focused on identifying the top-k EX-flows of a given BP specification. We next extend the algorithms to account also for queries, that restrict the focus to (sub-)flows that are of interest.

First, we note that the queries defined in Chapter 2 are projection queries in the sense that their output consists of the sub-flows to which EX-pattern nodes and edges are matched. One may also consider the evaluation of top-k selection queries, namely queries that given an EX-pattern and a BP specification, retrieve the (top-k) full (start-to-end) possible execution flows in which an embedding of the EX-pattern exists. We will start by considering query evaluation for such queries. We then turn to study top-k projection queries, that allow users to select portions of the EX-flows they are interested in, and that bear highest score. There are (at least) two reasonable semantics according to which the score of answers (sub-flows) can be defined. We refer to them as max and sum scores. For the first, the score of a sub-flow is the maximal weight of a full EX-flow in which it occurs. For the second, it is the sum of weights of such EX-flows.

To illustrate the difference between the two semantics, consider an online travel agency that allows its users to choose from a variety of flights, hotels and car rental reservations. Further consider a weight
function that accounts for popularity of choices and consider a case where one particular deal $D$, consisting of a specific flight and car rental reservation, is very popular, but where packages consisting of flight and hotel reservations are overall more common (even though each given offer is individually less popular than the specific flight+car deal $D$).

Now, consider a query that seeks to identify how a package that includes a flight reservation is typically booked (i.e. projecting over the part related to flight reservations). With sum semantics, the flight+hotel option is ranked highest, as it appears in most EX-flows. But with max semantics, flight+car will be ranked highest, as there exists one very popular EX-flows in which it appears.

**Our Results** We next provide an overview of the results provided in this chapter.

- We start by considering top-k query evaluation for selection queries, and show that an efficient (PTIME data complexity) such query evaluation algorithm may be obtained by utilizing a combination of a Type Inference algorithm to capture all qualifying EX-flows, along with Algorithm $\text{TOP-K}$ given in the previous chapter for computing the top-k out of these EX-flows.

- Then, we turn to top-k projection queries. We start here with the simpler max semantics and show that although there is a tight relationship between selection and projection queries under such semantics (both searching for flows with maximal weight), new techniques need to be developed to provide efficient (PTIME) query evaluation algorithms for projection queries. Indeed, we show that the standard, common, use of selection as an intermediate input for projection yields in this case an exponential blow-up. Instead, our novel query evaluation algorithm constructs a direct, compact representation of the top-k projections, avoiding materialization of the intermediate (full) qualifying flows.
• Interestingly, we then show that our algorithm for projection queries evaluation (with max semantics) is also valuable for the analysis of EX-traces. Given a partial (semi-naive, selective) EX-trace, users often wish to understand what had actually happened at run-time. In other words, they would like to identify the execution flows that are most likely to have been the source of this EX-trace, typically focusing of specific parts of the trace that are of particular interest [49, 14]. Intuitively, the recorded sub-flow may be viewed as a query, with the query result being the set of (relevant parts of) flows that are most likely to have occurred in practice. We show that our query evaluation algorithms can be adapted to retrieve the most likely sources of the given (sub)trace. We first consider the case where the names of the BP activities that are masked/omitted from the trace are known in advance, then the more difficult case where this information is not given. A challenge here is to avoid considering all possible sets of potential omitted/masked operation names (whose number is exponential in the size of the BP). To overcome this we prove a small world theorem, showing that only a polynomial number of representative options need to be tested.

• Finally, we study evaluation of projection queries under the sum semantics, and show that these are inherently harder than projection queries with max semantics (and than selection queries). Intuitively, this is because we need to sum up infinite number of weights just to compute the weight of a single projection result (and there may be infinitely many such results). Indeed, we show that under the sum semantics, even computing the score of a single projection result may be impossible (as scores may be irrational, even when starting from rational weights for EX-flows); however, we study plausible restrictions that allow for EXPTIME top-k query evaluation algorithms, and show that this is the best
that can be achieved (unless P=NP).

The results reported in this chapter for evaluation of top-k selection queries were published in [33]. Those related to projection queries with \textit{max} semantics were published in [35], and projection queries with \textit{sum} semantics were studied in [30].

5.1 Definitions

We start by formally defining the notions of selection and projection queries studied in this Chapter. We revisit the definition of queries through EX-patterns (Def. 2.3.1) and of an embedding of such EX-pattern within an EX-flow (Def. 2.3.3), and utilize these for defining (top-k) selection queries.

Definition 5.1.1 (Selection Queries) A selection query \( q \) is represented by an EX-pattern \( p \). When evaluated over a weighted BP specification \( s \), the result of \( q \) (denoted \( q(s) \)) is defined as the set of all flows of \( s \) in which there exists an embedding of \( p \). The top-\( k \) results of such query, denoted \( \text{top-k}(q,s) \), are the \( k \) EX-flows in \( q(s) \) having the highest weight.

For Projection queries we recall that the semantics dictated by Def. 2.3.3, defines the query result as all nodes and edges to which some nodes or edges of the EX-pattern are mapped. We generalize the definition of such queries and allow a specific part of the pattern to be specified as the \textit{projected part}. The rest of the pattern serves for selecting EX-flows of interest. Namely, only EX-flows in which an embedding of the entire pattern are considered; within these flows, only nodes and edges matching the projected part are in fact projected out and appear in the query result.

Formally,
Definition 5.1.2 (Projection Queries) A projection query \( q = (p, P) \) consists of an execution pattern \( p \) accompanied by a sub-graph \( P \) of the pattern, itself forming an execution pattern, called the projected part of the pattern.

The result of a projection query is defined as follows:

- Given an embedding \( h \) of \( p \) in an EX-flow \( e \), we use \( h \downarrow P \) to denote the graph consisting of the nodes and edges of \( e \) to which \( h \) maps the nodes and edges of the projected part \( P \). We say that two embeddings \( h, h' \) of \( p \) in two (possibly isomorphic) EX-flows \( e, e' \) are equivalent if \( h \downarrow P \) is isomorphic to \( h' \downarrow P \). We further denote \( \Psi \) as a set consisting of a single representative from each such equivalence class of embeddings.

- For an EX-flow \( e \) and an embedding \( h \) of \( p \) in \( e \), the result of \( q \) on \( e \), w.r.t. \( h \), is denoted \( q_\downarrow(e, h) \) and contains all nodes and edges of \( P \), with Any-labels of nodes in \( P \) replaced by activity names of the nodes in \( e \) they are mapped to by \( h \), and transitive edges in \( P \) replaced by the paths they are mapped to by \( h \). For each activity pair appearing in the result, the edge connecting its activation and completion nodes is also included.

- For an EX-flow \( e \), the result of \( q \) on \( e \), denoted \( q_\downarrow(e) \), consists of the results of all possible embeddings of \( q \) into \( e \), i.e. \( q_\downarrow(e) = \bigcup_{h \in \Psi} q_\downarrow(e, h) \).

- Finally, for a BP \( s \), the result of \( q \) on \( s \), denoted \( q_\downarrow(s) \) is the set of all possible results of \( q \) when applied on the EX-flows of \( s \). Namely \( q_\downarrow(s) = \bigcup_{e \in \text{flows}(s)} q_\downarrow(e) \).

Note that an EX-flow \( e' \in q_\downarrow(s) \) may originate from several EX-flows of \( s \), namely there may be several \( e \in \text{flows}(s) \) s.t. \( e' \in q_\downarrow(e) \). Before defining the top-k projection results, we should first decide on
how to aggregate the weights of these individual EX-flows, to form the score of projection result.

We consider here two possible aggregation functions (and consequently, semantics of queries), \( \text{max} \) and \( \text{sum} \). Under \( \text{max} \) semantics, the score of \( e' \) is defined as \( \text{score}(e') = \max\{ fWeight(e) \mid e \in \text{flows}(s) \land e' \in q_1(e) \} \). Under \( \text{sum} \) semantics, \( \text{score}(e') = \sum_{e \in \text{flows}(s) \land e' \in q_1(e)} fWeight(e) \).

**Definition 5.1.3** The top-\( k \) results of a projection query \( q \) over a BP specification \( s \), with respect to \( \text{max} \) (\( \text{sum} \)) semantics, denoted top-\( k_{\text{max}}(q_1, s) \) (top-\( k_{\text{sum}}(q_1, s) \)), is a set of EX-flows in \( q_1(s) \) having highest score values according to \( \text{max} \) (\( \text{sum} \)) semantics, respectively. When the semantics used is clear from context, we omit it from notation and simply write top-\( k(q_1, s) \).

### 5.2 Evaluation of Selection Queries

We start the discussion of query evaluation by considering evaluation of top-\( k \) selection queries. We assume first that the given weighted BP specification is accompanied by a history-independent weight function, then explain how to extend the results to history-bounded weight functions.

To evaluate a top-\( k \) selection query represented by an EX-pattern \( p \) over a given weighted BP specification \( s \), we employ a two-steps algorithm, that bears the same structure as Algorithm **BOUNDED-\( K \)**, depicted in Chapter 4. We refer to the following Algorithm as Algorithm **TOP-\( K \)-SELECTION**.

1. The first step, namely Algorithm **INTERSECT-QUERY**, computes a new weighted BP specification \( s' \) (along with a renaming function \( \pi' \)) that captures (see Def. 4.3.3) \( p(s) \).

2. The second step then employs Algorithm **SEMI-NAIVE-\( K \)** from Chapter 4 over \( s', \pi' \), to retrieve the top-\( k \) EX-traces of \( s' \) w.r.t.
the semi-naive tracing \( \pi' \); these are also the top-k EX-flows in \( p(s) \).

**Algorithm INTERSECT-SELECTION-QUERY**  Recall the proof of Theorem 3.2.2 suggesting an EXPTIME algorithm for capturing the results of projection queries. The exponential blow-up there was caused by the treatment of transitive edges, and the need to represent exponentially many possible paths to which each such edge matches. In contrast, with selection queries, we only need to verify the existence of some path in-between the nodes to which the transitive edge end-nodes are matched. Consequently, we are able to provide a PTIME algorithm for computing \( s' \). We next provide the required changes to the Algorithm SEMI-NAIVE-TYPE-INFERENCE given in the proof of Theorem 3.2.2.

- **Selecting full EX-flows** Recall that Algorithm SEMI-NAIVE-TYPE-INFERENCE gradually “intersected” the EX-pattern with the BP specification, generating new activity names of the form \([p', n_s, a]\) where \( p' \) is a query part, \( n_s \) is a node of \( s \) and \( a \) is an activity name from \( s \) whose implementations are parts of implementations of corresponding activities of the original BP specifications. The implementations considered are those in which an embedding of the query (part) exists; the parts of implementations that are kept consisted of the nodes and edges to which the embedding indeed matched nodes and edges of the query. To support selection queries, we perform essentially the same procedure, but whenever we identify an implementation in which an embedding exists, we use the entire implementation graph as an implementation of the new activity.

- **Considering guarding formulas.** Algorithm SEMI-NAIVE-TYPE-INFERENCE did not account for guarding formulas and weight functions, but the extension required to support these is rather straightforward. We use here a slight generalization of the BP specification definition: recall that the original definition required that only a single
guarding formula is satisfied at any given time. We relax this restriction, and note that this relaxation does not affect our results.

Now, for queries with no transitive nodes, each implementation in which an embedding was found is simply carried along with its original guarding formula, to form an implementation of the corresponding new compound activity. As for queries with transitive nodes, multiple matchings may be found for several sub-graphs of the same implementation. In this case, multiple implementations for the same activity name may be created by the algorithm, where each of these newly created implementations refer to the same original implementation. Thus, multiple instances of the original guarding formula are created, each guarding a different implementation of the newly created activity. These instances are treated as distinct formulas, though sharing the same values assigned by the weight function. Apart from this change, cWeight and aggr stay intact.

The renaming function \( \pi' \) over activity names in \( s' \), maps each activity name \([p', n_s, a]\) to \( a \).

**Finding Top-K flows** We then apply Algorithm `SEMI-NAIVE-TOP-K`, as is, given \( s', \pi' \) as input. The algorithm outputs the top-k EX-traces of \( s', \pi' \). These are exactly the top-k EX-flows of the set \( p(s) \).

This concludes the presentation of Algorithm `TOP-K-SELECTION`.

**Complexity Analysis** We next analyze the algorithm complexity. The complexity of the first step, namely computation of \( s' \), is polynomial in the size of \( s \) and exponential in the query size \( q \). As for the second step, recall that the complexity of Algorithm `SEMI-NAIVE-TOP-K` is polynomial (1) in the size of its input BP specification \( s' \), (2) in \( \max_a N(a) \) where \( a \) is an activity name and \( N(a) \) is the number of distinct activity names mapped by \( \pi' \) to \( a \), (3) in \( k \) and (4) in the output.
size. The sizes of (1) and (2) are polynomial in the size of the input BP specification \( s \) and exponential in the EX-pattern size \( p \), leading to the following Theorem.

**Theorem 5.2.1** Given a weighted BP specification \( s \) with an history-independent cWeight function, a selection query \( p \) and given also a number \( k \), \( \text{top-k}(p,s) \) can be computed in time polynomial w.r.t. \(|s|, k, \) and the output size, and exponential in \(|p|\).

As a direct corollary from Theorem 3.2.7, no algorithm polynomial in \(|p|\) is possible unless \( P = NP \).

**Bounded-History cWeight function** So far we have assumed that the cWeight function accompanying the input BP specification is history-independent. When it is in fact history-bounded, we employ a 3-steps algorithm:

1. First, apply Algorithm \texttt{COMPILE-HISTORY-INDEPENDENT} (given in Chapter 4) over \( s \) to obtain \( s', \pi' \) capturing the flows of \( s \), such that the cWeight function used for \( s' \) is history-independent.

2. Second, apply a slightly generalized version of Algorithm \texttt{INTERSECT-SELECTION-QUERY}, given above, over \( p, s' \) to obtain \( s'', \pi'' \) capturing the flows of \( p(s') \). The only change to the algorithm accounts for the changed activity names in \( s' \): upon matching query nodes to specification nodes, the revised algorithm matches a query node with activity name \( l \) to a specification node with activity name \([a, pre, post]\) if \( l \) matches \( a \) (i.e. \( l = a \), or \( l \) is \text{Any}-label.)

3. Third, apply Algorithm \texttt{SEMI-NAIVE-TOP-K} over the BP specification \( s'' \) and \( \pi' \circ \pi'' \) (the composition of the two renaming functions \( \pi' \) and \( \pi'' \)) as a renaming function.
The first step generates a BP specification whose size is exponential in the history bound, and the second step further causes an exponential blowup in the size of the query, but both steps incur time that is polynomial in the size of $s$. The following theorem thus holds:

**Theorem 5.2.2** Given a weighted BP specification $s$ with a bounded-history $c$Weight function (with bound $b$), a selection query $p$ and given also a number $k$, top-$k(p,s)$ can be computed in time polynomial w.r.t. $|s|$, $k$, and the output size, and exponential in $|p|$ and in $b$.

### 5.3 Evaluation of Projection Queries Under max semantics

We next discuss evaluation algorithms for top-$k$ projection queries, where the results are ranked based on the max semantics. We first show that direct adaptations of the selection queries evaluation algorithm) to projection queries fail at achieving a PTIME query evaluation algorithm, and finally describe an alternative algorithm that overcomes this.

Again, for ease of presentation, we assume in the sequel that the $c$Weight function is history-independent; otherwise we first use Algorithm COMPILER-HISTORY-INDEPENDENT depicted in Chapter 4 to generate a BP specification with an history-independent $c$Weight function, then employ the relevant algorithm depicted here.

**First attempt - explicit enumeration of EX-flows.** The first simple approach for evaluating projection queries is to first treat the query as a selection query, and use Algorithm TOP-K-SELECTION to explicitly enumerate the flows of $s$ satisfying the query, along with their $f$Weight values (If multiple flows lead to the same projection result, one may need to generate the top-$k'$ flows for some $k' > k$, leading to $k$ distinct projection results.) Next, for each obtained flow, we may compute
the result of projecting it on the EX-pattern projection part. Each projection result is ranked based on the maximal fWeight of a flow leading to it.

Note, however, that this naive algorithm may incur time that is exponential in the **BP specification** size, as well as in the **output size**, as the following theorem holds:

**Theorem 5.3.1** There exists a fixed-sized query $q$ such that for every natural number $n$ there exists a BP specification $s_n$ whose size is linear in $n$, such that even the smallest size of any EX-flow of $q(s_n)$ is exponential in $n$, while the size of the largest EX-flow in $q_\downarrow(s_n)$ is bounded by a constant.

**Proof.**
Consider a BP specification in which each activity has a single implementation, as follows: the implementation of each activity $a_i$ ($i = 1, ..., n-1$) consists of two nodes labeled $a_{i+1}$, and the implementation of $a_n$ consists of a single atomic activity. Consider also a query that seeks for the root activity node, whose (indirect) implementation consists of two any-labeled nodes, connected by a transitive edge. The query then projects over the root node. It is easy to observe that $q(s_n) = s_n$, and it has only a single possible EX-flow. The size of this flow (containing two instances of $a_2$, 4 instances of $a_3$, etc.) is exponential in $n$. Note that in contrast, $q_\downarrow(s_n)$ consists of a single activity pair. □

**Second attempt - Computing a compact representation of all projections.** An alternative approach for adapting the selection queries evaluation algorithm follows the idea of generating a BP specification $s'$ along with a renaming function $\pi'$, that captures all results, namely $q_\downarrow(s)$, and then retrieving the top-k out of these. Unfortunately, this approach would also lead to an infeasible algorithm, as Theorem 3.2.4
indicates that there exist no such \( s', \pi' \) s.t. \( s' \) is of size polynomial in \( s \). In contrast, one may replace the first step by generating a BP specification with a selective tracing system, such that its EX-traces correspond to \( q_i(s) \); however it is open whether top-k analysis over BP specifications with selective tracing systems may be done in PTIME.

Next, we show an alternative, PTIME, algorithm for evaluating top-k projection queries. As shown above, it is infeasible to compute a compact representation that captures all projection results (in the sense of Definition 4.3.3) and then retrieve the top-k results out of them. However, we may still perform a two-steps algorithm, similar to TOP-K-SELECTIONS. However, instead of a first step that generates a BP specification capturing the entire set of results \( q_i(s) \), the first step of the refined evaluation algorithm generates a specification that captures only a subset of \( q_i(s) \), including in particular the top-k weighted projections. We say that such BP k-captures \( q_i(s) \). Formally,

**Definition 5.3.2** Given two weighted BP specifications \( s', \ s \), a renaming function \( \pi' \) mapping activities names in \( s' \) to activities names in \( s \), a query \( q \), and a number \( k \), we say that \( s' \) k-captures \( q_i(s) \) if \( \text{top}-k(q_i, s) \subseteq \pi'(\text{flows}(s')) \subseteq q_i(s) \) (Overloading notations, \( \pi'(E) = \{\pi'(e)|e \in E\} \)), and the score of each projection is the same as the \( f\text{Weight} \) of the corresponding flow of \( s' \).

The following theorem holds.

**Theorem 5.3.3** Given a weighted BP specification \( s \) and a projection query \( q = (p, P) \), we may compute a weighted BP specification \( s' \) (and a renaming function \( \pi' \)) that k-captures \( q_i(s) \) in time polynomial in \( s \), with the exponent determined by the query size.

**Proof.**

We present an evaluation algorithm, named Algorithm K-CAPTURES, whose output is a BP specification \( s' \), weight functions for the flows of
s′ (denoted cWeight′, aggr′ and fWeight′), and a renaming function π′ over the activity names of s′, that k-captures q↓(s). Similarly to Algorithm INTERSECTION-SELECTION-QUERY (depicted in Section 5.2), Algorithm K-CAPTURES builds upon Algorithm SEMI-NAIVE-TYPE-INFERENCE (depicted in Chapter 3). There are two changes w.r.t. Algorithm SEMI-NAIVE-TYPE-INFERENCE: the first change is designed to account for guarding formulas, and is exactly as explained in Algorithm INTERSECTION-SELECTION-QUERY; the second change accounts for transitive edges, and improves upon the exponential time complexity incurred by Algorithm SEMI-NAIVE-TYPE-INFERENCE, as we next explain.

The end-nodes of each transitive edge are matched by the algorithm to specification nodes (denote these as n for the start node of the edge and m for the end node), and it remains to consider the paths in-between them. If the transitive edge does not appear in the projection part, we only need to verify that a path exists, as in Algorithm INTERSECTION-SELECTION-QUERY.

Otherwise, when the transitive edge does appear in the projection part, then each projection result contains a path, to which the edge is matched. It is infeasible to create an implementation for each distinct projection (i.e. each distinct path), as the number of paths between two nodes of the specification may be exponential in the specification size. Fortunately, we are only interested in the top-k projections, in which only the top-k paths may appear. We thus design a Dynamic Programming algorithm, namely TOP-K-PATHS (explained next), returning a BP specification whose possible EX-flows are exactly the top-k paths from n to m (along with their correct weights). TOP-K-PATHS is then employed whenever the algorithm encounters a transitive edge.
**Note** We assume in the sequel that the transitive edge appears within an implementation of a transitive node. If not, then any \( k^1 \) paths may be chosen, as they all co-appear in the same implementation, thus share the same weight.

**TOP-K-PATHS** First, we assign a unique identifier to each specification node, then generate a new set of nodes \([n_j, i]\) for each \( n_j \) of the specification and for each \( i = 1, \ldots, k \), and initialize a table which keeps track of the \( i \)'th most likely path from each \( n_j \) to \( m \), along with its weight. When the table is full, it contains in particular the top-k paths from \( n \) to \( m \), which are explicitly generated.

To fill in the table, we use the auxiliary notion of *children*, as follows. We say that \( n_2 \) is a child of \( n_1 \) if there is an edge from \( n_1 \) to \( n_2 \), or if \( n_1 \) is a activation node of a compound activity and \( n_2 \) is a start node of its implementation, or if \( n_2 \) is an completion node of a compound activity and \( n_1 \) is an end-node of its implementation.

The computation of values in the table proceeds as follows, computing the \( i \)'th paths for increasing values of \( i \). When computing the top-\( i \) path originating at some node \( l \), we consider all *children* of \( l \) in the specification. For each such child \( u \), say that \( j \) is the greatest index of a path originating in \( u \) that was used as a sub-path in some \( r < i \) ranked path of \( l \) that was previously computed. Then we compute the \( j + 1 \) path originated in \( u \) (if it was not already computed), and obtain \( \text{score}(u) \), the score of \( u \) as a candidate for the path being generated for \( l \). In case the edge between \( l \) and \( u \) is a zoom-in edge, \( \text{score}(u) \) is aggregated with the *cWeight* of \( f \), where \( f \) is the formula guarding the implementation rooted at \( u \). The child of \( u \) with the maximal score, along with its corresponding sub-path, is chosen for the top-\( i \) path originating at \( l \).

Each obtained path serves as a separate implementation of the cor-

\[1\] If there are only \( k' < k \) paths, then all are matched.
responding compound activity.

The renaming function $\pi'$ The renaming function $\pi'$ is simply designed as $\pi'([a, q']) = a$ for each sub-query $q'$, and $\pi'([a, i]) = a$ for each $i = 1, \ldots, k$.

The following lemma holds.

**Lemma 5.3.4** The BP specification generated by Algorithm $\text{TOP-K-PATHS}$ captures the $k$ highest weighted paths from $n$ to $m$.

**Proof.**

The proof is by induction on $i$. For $i = 1$, the lemma trivially holds (the best path consists of the best implementation for the start node, along with the best path originating at any of its children). Now assume correctness for $i = k$. That is, the $k$th ranked path contains only the $p_1 \leq k$ best implementation of $l$, and the $p_2 \leq k$ best implementation of some child. The $k + 1$ ranked path obviously uses, at worst, the $p_1 + 1$ best implementation of its origin. As for the children, if it has the same son as the $k$th ranked path then it obviously uses its $p_2 + 1$ best ranked path; if it uses the $p'_2$ ranked path of some other child, then assume by contradiction that $p'_2 > k + 1$. Then there are at least $k + 1$ better paths obtained by improving the path originated at the chosen child, in contradiction to the path being ranked $k + 1$. $\square$

**Complexity** Note first that for each activity node occurring in $s$ there are at most $2^{|q|}$ activities mapped to it by $\pi'$. Further note (Observation 2 of Chapter 4) that an EX-flow that is one of the top-$k$ EX-flows of $s$, contains at most $k$ recursive invocations of each activity. This means that in total the number of EX-flows of $s'$ mapped to the same EX-flow is bounded by $|s| \times 2^{|q|} \times k$. The complexity of the above algorithm is thus polynomial in the BP size $s$ and in $k$, with the exponent determined by the query size $q$. The exponential dependency on the query size is due to the algorithm for transitive nodes, considering all query splits, as well
as due to the embedding algorithm whose complexity is exponential in the size of the sub-query matched.

\[ \square \]

**Corollary 5.3.5** We may compute \( \text{top} - k(q_1, s) \) in time polynomial in the size of the BP and in the output size, and exponential in the query size.

**Proof.**

The algorithm, namely \texttt{TOP-K-PROJECTIONS}, first applies Algorithm \texttt{K-CAPTURES} given in the proof of Theorem 5.3.3, to obtain \( s', \pi' \) that k-capture \( q_1(s) \). The flows of \( s' \) capture all results of top-k(\( q_1, s \)), but possibly some additional flows. Then, \texttt{TOP-K-PROJECTIONS} applies Algorithm \texttt{SEMI-NAIVE-TOP-K} to find the \( k \)-highest weighted semi-naive EX-traces of \( s', \pi' \). The time complexity of both steps is polynomial in the size of the input BP specification and exponential in the query size, and so is the complexity of Algorithm \texttt{TOP-K-PROJECTIONS}. \( \square \)

### 5.4 Querying Origins of Selective Traces

We have studied above projection queries with \textit{max} semantics as a tool for analysis of future executions. In this section we consider another application of our results, as follows. Given a partial trace, obtained via a selective tracing system, the user may wish to identify flows that are most likely to have occurred in practice, and moreover, focus on parts of these flows that are of interest. We next formally define this problem.

#### 5.4.1 Definitions

We start by defining the notion of \textit{source} flows, which are flows that may have occurred given an observed trace, as follows:
Definition 5.4.1 An execution flow \( e \) is a possible source of an execution trace \( t \), w.r.t. a selective tracing system \( \sigma = (A, F, \pi) \) if deleting from \( e \) all occurrences of activity names in \( A \) and of formula names in \( F \), and renaming all other activities according to \( \pi \), results in \( t \). In this case we denote \( \sigma(e) = t \).

Given an execution trace \( t \) we wish to identify the top-k flows that are most likely to have occurred in practice, or relevant parts thereof. Thus, we use here a specific kind of weight function modeling the likelihood of implementation choices and EX-flows; meaning \( cWeight \) values for implementation choices are in the range \([0, 1]\), and \( aggr = \ast \). We use the notations \( c\)-likelihood and \( f\)-likelihood to denote the \( cWeight \) and \( fWeight \) functions capturing the likelihoods of implementation choices and EX-flows (respectively).

We further consider a more generalized setting, where the input consists of a query, and we seek for (relevant parts of) the EX-flows that are most likely to occur, out of the possible sources of any trace conforming to the query. (This setting is indeed more general: an observed trace \( t \) may be represented by the pattern, with the projection part being the part of interest).

We next define top-k projections of source EX-flows, under selective tracing; recall that we denote \( e' = e \ (e' \subseteq e) \) when \( e' \) is isomorphic to (a sub-graph of) \( e \).

Definition 5.4.2 Given a BP specification \( s \), a selective tracing system \( \sigma = (A, F, \pi) \) and a projection query \( q \), we define \( \text{sources}(q_1, s, \sigma) = \bigcup_{e \in \text{flows}(s)} \{ e' \subseteq e \mid \sigma(e') = q_1(\sigma(e)) \} \), and say that \( e \) leads to \( e' \). The score of each projection result \( e' \) is defined as the maximal \( f\)-likelihood of a flow \( e \) leading to it, and \( \text{top-k}(q_1, s, \sigma) \) consists of the \( k \) projections in \( \text{sources}(q_1, s, \sigma) \) having the best scores.

Intuitively, the results set \( \text{sources}(q_1, s, \sigma) \) contains all sub-flows \( e' \) of some flow \( e \) of \( s \) (before renaming), such that after renaming, \( \sigma(e') \)
is the projection of $\sigma(e)$ over $q$. It thus captures the relevant parts of conforming flows. As before, each sub-flow $e'$ is ranked by the maximal ranking of a corresponding flow $e$.

In the remainder of the discussion we consider computation of $\text{top-k}(q, s, \sigma)$, in two different settings. First, we assume knowledge of $\sigma$, i.e. we know exactly which activities are deleted / renamed. Then, we consider a more intricate scenario where this information is unknown.

### 5.4.2 Known Selective Tracing

We start by showing that, given $q, s, \sigma$, we may efficiently capture $\text{top-k}(q, s, \sigma)$.

**Theorem 5.4.3** Given a BP specification $s$, a selective tracing system $\sigma$, and a query $q$, we can compute $\text{top-k}(q, s, \sigma)$ in time polynomial in the size of $s$ and in the output size, and exponential in $q$.

**Proof.** Re-visiting Algorithm $\text{TOP-K-PROJECTIONS}$ (corollary 5.3.5), we adapt the first step ($\text{K-CAPTURES}$), given in the proof of Theorem 5.3.3, to the setting of selective tracing, as follows:

1. Upon matching of nodes, we may match a query node $n'$ to a specification node $n$ if the label of $n$, after renaming, is the same as the label of $n'$.

2. Upon matching of edges, we may also match a non-transitive edge to a specification path, given that the path includes only nodes whose labels are in the deletion set of the tracing system, and all zoom-in edges along the path are guarded by formulas in the deletion set. $\text{TOP-K-PATHS}$ is adapted to support these newly created edges, verifying that nodes and edges (in any nesting level) along the matched path indeed conform to the above requirement.

3. Each composite node is treated as transitive, and its corresponding zoom-in edges are treated as transitive, as in item (2).
This concludes the proof of Theorem 5.4.3. □

5.4.3 Unknown Selective Tracing

In the previous discussion we have assumed knowledge of the information lost in the tracing process. Such knowledge may generally be absent, for instance if we have no access to the tracing system itself, but only to some samples of EX-traces.

Obviously, assuming no information at all on the tracing system, we have no knowledge of the EX-flows that occurred in practice given an EX-trace. We thus assume the tracing system follows some selective tracing, that is, all occurrences of some subset of activity names are deleted, and all occurrences of another subset are renamed.

We then say that an EX-flow is a source of an EX-trace if it is its source under some selective tracing system, and we correspondingly define top-k-uSelective(q₁, s), as follows:

**Definition 5.4.4** Given a BP specification s and a query q, let Sel be the set of all possible selective tracing systems over s, namely all pairs of a renaming function from activity names in s to A, and a deletion set which is a subset of the activities names in s.

We define sources(q₁, s, uSelective) = ∪_{σ∈Sel} sources(q₁, s, σ), and score(e', uSelective) = max_{σ} score(e', σ), and we denote top-k-uSelective(q₁, s) as the k results with highest scores out of sources(q₁, s, uSelective).

A simple solution for top-k query evaluation under unknown selective tracing would be to generate all possible selective tracing systems, find the top-k results for each, then combine the results. The number of such possible systems, however, is exponential in the BP specification size.

However, we claim that our results still hold for such setting. The idea behind the evaluation algorithm is to generate a sufficient set of possible tracing systems, then use Theorem 5.4.3 to find the top-
k projections given each tracing system. Finally, the top-k lists are combined to find the actual top-k projection.

The following theorem holds:

**Theorem 5.4.5** Given a BP specification $s$, and a query $q$, we may compute $\text{top-k-uSelective}(s, q)$, in time polynomial w.r.t. the size of $s$ and the output size, and exponential w.r.t. the query size.

**Proof.**

We first give a naive, exponential algorithm that explicitly enumerates all possible tracing systems, then optimize it by considering only a subset of the possible tracing systems. Finally, we prove that it is indeed sufficient to consider only this subset.

**Naive Algorithm** Given a BP specification $s$ and a query $q$, a naive algorithm generates all possible systems $\sigma = (A, F, \pi)$ such that $A$ contains activities names appearing in $s$, $F$ contains guarding formulas appearing in $s$ and $\pi$ is a mapping from activity names in $s$ to activity names in $q$. Then, for each such $\sigma$, we apply the algorithm given in the proof of Theorem 5.4.3, but modify the underlying Algorithm $\text{TOP-K-PROJECTIONS}$, such that it does not explicitly enumerate the computed results but rather just compute the FTable compact representation of them (i.e. does not apply the EnumerateFlows method); another modification is that throughout the computation we propagate the weights of ranked sub-flows to the compound activity in which they serve as its implementation. Now, we have in hand the scores for the top-k results obtained for each $\sigma$, to find the overall top-k results we employ the following simple algorithm: the top-1 EX-flow is the best ranked out of all the top-1 EX-flows obtained for all tracing systems (denote this tracing system by $\sigma$); the overall second ranked EX-flow is the best ranked among the set including the remaining top-1 flows of the individual tracing systems, along with the second ranked EX-flow.
obtained for \( \sigma \), and so forth (note that the weights of these flows are given thus one does not have to materialize all of them). Finally, the top-k EX-flows that were found are materialized.

As the number of tracing systems tested is exponential in the size of \( s \), such algorithm is infeasible.

**Optimized Version** We avoid explicit generation of all possible tracing system, and generate only a subset of these. We first generate all possible mappings from query nodes (formulas) to BP activities names (formula names), ignoring their original activities (formulas). Now, for each such mapping \( L \), we first check \( L \) for *consistency*. That is, verify that no two query nodes whose original activities (formula names) were identical, are mapped to different labels (formula names). The algorithm generates a selective tracing system, denoted \( \sigma_L \), with its deletion set \( A_L (F_L) \) including all activity names (formula names) except for those assigned by \( L \) to some query node (formula name). The renaming function \( \pi_L \) matches each BP activity name to the original activity name of a query node mapped by \( L \) to it (uniquely defined, as due to the consistency of \( L \), all such nodes bear the same activity name). For each consistent mapping \( L \), \( \sigma_L \) is added to a set of tracing systems, denoted \( \Lambda \). The size of \( \Lambda \) is polynomial in that of \( s \). We claim that \( \Lambda \) contains all tracing systems that need to be checked, namely:

**Lemma 5.4.6** Given a BP specification \( s \) and a query \( q \), and for any \( e \), there exist \( \sigma, t \) s.t. \( e \in \text{top-k}(q_L, s) \) if and only if there exist such \( \sigma', t' \), with \( \sigma' \in \Lambda \).

**Proof.**

The first direction is immediate. We thus assume that there exists such \( \sigma, t \), and denote by \( L \) the node embedding of \( q \) in \( t \), that is the assignment of labels to query nodes, according to the embedding \( E \) of \( q \) in \( t \). Denoting the result of applying \( \sigma_L \) on \( e \) by \( t' \), we construct the embedding \( E' \) of \( q \) in \( t' \): it maps nodes to nodes exactly as \( E \) does.
(note that no node of $t$ to which $E$ maps a node of $q$ was removed by $\sigma_L$). Edges are mapped to edges connecting corresponding nodes; as for transitive edges, there still exists a path (possibly containing one edge) in $t'$ connecting the corresponding end-nodes, as edges are not removed unless their end-nodes are removed.

$\square$

As the number of tracing systems generated is polynomial in the specification size, and so is the evaluation complexity for each tracing system, we obtain a PTIME data complexity algorithm. $\square$

5.5 Evaluating Boolean Queries

We studied above a particular $\text{max}$ function for aggregating the weights of EX-flows into a single rank of a projection result. In the context of querying future likely sub-flows, a more suitable aggregation function is $\text{sum}$. Before we study projection with the $\text{sum}$ aggregation function, we start by defining boolean query, whose evaluation result is the sum of all weights of EX-flows that satisfy the query. We then show how to use utilize such boolean queries in evaluation of projection queries with the $\text{sum}$ aggregation function.

We first note that such sum semantics for queries was used in the context of two related models, namely probabilistic XML [4, 65, 64, 25] and Recursive Markov Chains [42]; since we combine in the sequel some (extensions of) the techniques developed for these two models, we start the discussion by describing these models and comparing them to our model; further related work is considered in Section 5.7.

Probabilistic XML extends XML to a probabilistic setting, by introducing distributional nodes. For such node, a subset of its children is randomly chosen to appear in the concrete XML documents. In [65] the authors study top-k query evaluation over such probabilistic XML documents. Again, probabilistic XML introduce only finitely many
possible worlds (XML documents). Note that in contrast to relational data and XML, our model for BP flows allows representation of an infinite number of items, out of which the top-k are retrieved; this of course leads to further difficulties in query evaluation. In [65] the authors show that evaluation of projection queries over probabilistic XML is possible in polynomial time (data complexity). This result by itself, however, does not imply PTIME algorithm for our setting, which is more complex: first, our model allows representation of nested DAG structures, rather than trees. For projection queries, this entails another level of complexity in finding matches serving as implementations, and projecting them over relevant nodes; second, an infinite number of flows may be represented by our probabilistic BP (serving as schema), due to possible recursive calls. Upon projection, the results set may be infinite as well, posing a further challenge.

Recursive Markov Chains (RMCs) are syntactically equivalent to probabilistic BPs; however, a major semantic difference lies in the definition an execution, being a single sequential path in [42] vs. a DAG in our context. DAGs are important to capture parallelism in BPs and to compactly describe scenarios where the execution order of a given set of activities is irrelevant. In turn we shall see that they entail higher computational complexity, notably for projection queries with sum semantics. Another distinction from [42] is the query language, where only a limited query language (including in particular the problems of reachability and termination) was studied there. To our knowledge, evaluation of projection queries over Recursive Markov Chains was not studied.

Consequently, while most hardness results from the context of Recursive Markov Chains and probabilistic XML go through to our context, their positive results may not be adopted as is.

In the sequel we consider the special case where the weight functions given as part of the input reflect likelihood of choices / EX-flows. In
this context, we introduce the notion of boolean queries. A boolean query is simply an EX-pattern, but its semantics here is “compute the likelihood that the given pattern appear in a random EX-flow” of the given BP specification.

More precisely, given an EX-pattern $p$ referred to as a boolean query, we say that an EX-flow $e$ satisfies a boolean query $p$ (denoted $p(e) = true$), if there exists an embedding of $p$ in $e$. Given a BP specification $s$, the probability that a random EX-flow of $s$ satisfies $p$ is the sum of likelihoods of EX-flows satisfying $p$, namely

$$likelihood^b(p, s) = \sum\{\Delta(e) \mid e \in flows(s) \land p(e) = true\}$$

where $\Delta$ is the $f$–likelihood function for $s$. When $s$ is clear from context we omit it and use $likelihood^b(p)$.

**Example 5.5.1** Consider the BP specification of the online travel agency depicted in Figure 5.1. Further consider Table 5.1 that depicts the likelihoods of value assignments for different variables, and consequently the $c$–likelihood function for the guarding formulas.
Consider also the query in Figure 5.2(a) (ignore for now the rectangle surrounding a sub-graph of the pattern) describing EX-flows that contain a choice of some British Airways (abbr. “BA”) flight followed by a confirmation. Assume first, for simplicity, that the edge between Flights and Confirm is a regular one, rather than transitive. Observe that, still, this query has an infinite number of possible satisfying EX-flows, differing e.g. in what the user had done prior to the reservation. The infinite number of options is due to the fact that the user may make some selections and reset, then make other selections and reset again, an unbounded number of times before making her final (confirmed) selection. The query likelihood is the sum of likelihoods of all these flows; to compute it, we define a variable $x$ reflecting the likelihood of a query.

### Table 5.1: c-likelihood function

<table>
<thead>
<tr>
<th>$\text{searchType}$</th>
<th>$P(\text{searchType})$</th>
<th>$\text{airline}$</th>
<th>$P(\text{airline})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;flights only&quot;</td>
<td>0.5</td>
<td>&quot;BA&quot;</td>
<td>0.7</td>
</tr>
<tr>
<td>&quot;flights+hotels&quot;</td>
<td>0.25</td>
<td>&quot;AF&quot;</td>
<td>0.2</td>
</tr>
<tr>
<td>&quot;flights+hotels+cars&quot;</td>
<td>0.25</td>
<td>&quot;AL&quot;</td>
<td>0.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\text{hotel}$</th>
<th>$P(\text{hotel})$</th>
<th>$\text{choice}$</th>
<th>$P(\text{choice})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Marriott&quot;</td>
<td>0.6</td>
<td>&quot;reset&quot;</td>
<td>0.6</td>
</tr>
<tr>
<td>&quot;HolidayInn&quot;</td>
<td>0.3</td>
<td>&quot;confirm&quot;</td>
<td>0.2</td>
</tr>
<tr>
<td>&quot;CrownePlaza&quot;</td>
<td>0.1</td>
<td>&quot;cancel&quot;</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Figure 5.2: Query
match. Intuitively, $x$ is the probability that the query is matched in the first reservation choice of the user, or that the user resets and a match is obtained in a following reservation choice. Thus

$$x = c\text{-likelihood} (\text{searchType} = “flightsOnly”) \times$$

$$c\text{-likelihood} (\text{airline} = “BA”) \times c\text{-likelihood} (\text{choice} = “confirm”)$$

$$+ c\text{-likelihood} (\text{choice} = “reset”) \times x.$$ 

Namely, $x = 0.5 \times 0.7 \times 0.2 + 0.6 \times x$. Thus $x = 0.175$.

We have assumed above that the edge connecting Flights and Confirm is non-transitive. If the edge is transitive, the computation becomes more complex, accounting for all possible paths in-between Flights and Confirm. We discuss such computations below.

We then refer to the problem of computing $\text{likelihood}^* (p, s)$, given a boolean query $p$ and a BP specification $s$, as $\text{BOOL-EVAL}$, and to the problem of approximating it up to any given $\epsilon$ as $\text{APPROX-BOOL-EVAL}$.

First, we note that, using the hardness result of [42], it follows that in the general case $\text{BOOL-EVAL}$ may not be solved exactly.

**Proposition 5.5.2** $\text{likelihood}^* (p, s)$ may be irrational, even if all $c$-likelihood values in $s$ are rational and $p$ contains no transitive nodes and edges.

**Proof.** The proof follows directly from Theorem 6 in [42] that presents a Recursive Markov Chain (RMC) $m$, with rational probabilities, where the probability for execution termination is irrational. We construct a BP specification $s$ with structure similar to $m$ and a boolean query $p$ that is satisfied by all terminating execution.

The BP specification $s$ has a root DAG with a single activity $r$, having possible 3 implementations: the first, with $c$-likelihood of $\frac{1}{6}$, is a chain, of length 5, with all activities labeled by $r$. The second contains a single atomic activity node, labeled by $a$, and its $c$-likelihood is $\frac{1}{3}$, and the third contains a single compound activity node $A$. $A$ has a single recursive implementation, i.e. an implementation of $c$-likelihood 1.
consisting of a single $A$-labeled node. The EX-pattern $p$ seeks for EX-flows with root activity $r$ (namely $p$ consists of a single activity pair, labeled $r$). Is thus satisfied by all the full EX-flows of $s$, or in other words, by all terminating executions. $\text{likelihood}^p(p, s)$ is the solution of the equation $x = \frac{1}{6} \times x^5 + \frac{1}{3}$ which is known not to have any rational roots. \hfill \Box

Results from [42] on the hardness of approximating the termination probability in Recursive Markov Chains apply here as well, indicating that a PTIME approximation algorithm is unlikely to exist also in our context. Namely,

**Proposition 5.5.3** The existence of a PTIME algorithm for APPROX-BOOL-EVAL, even for queries with no transitive nodes and edges, implies that $\text{SQRT-SUM} \in \text{PTIME}$.

$\text{SQRT-SUM}$ is the problem of deciding, given natural numbers $(d_1, ..., d_n)$ and a natural number $k$, whether $\sum_{i=1,...,n} \sqrt{d_i} \leq k$, strongly believed not be solvable in PTIME under Turing Computation Model [50]. The proof follows immediately from Theorem 12 in [42].

We next show, however, that an EXPTIME approximation is possible.

**Theorem 5.5.4** We may solve APPROX-BOOL-EVAL, approximating the probability up to $j$ bits of precision, in time exponential in the size of $s$ and linear in $j$.

We next provide an EXPTIME approximation algorithm, proving Theorem 5.5.4. Our algorithm combines techniques from query evaluation over probabilistic XML [65] and termination analysis of RMCs [42]. We note that the problem here is more complex than the case of XML due to (1) the recursive nature of BPs and (2) the fact that the same specification DAG may be used in multiple places in the execution. These will be reflected by the higher computational complexity.
The problem is also more complex than the case of RMCs since we need to consider general boolean queries (as opposed to just termination/reachability queries). This will be reflected by the intricacy of the algorithm.

We present the algorithm in two steps. We first consider a restricted setting where the BP specification is non-recursive, and show that exact probability computation is possible here (though may be expensive). Then, we explain why this algorithm does not apply to the general case, and show the changes required to obtain an approximation algorithm for this setting.

5.5.1 The non-recursive case

Even in the absence of recursion, still, an obstacle here, absent from the XML setting, is that the same specification DAG may be used in multiple places of the execution (i.e. as an implementation of multiple occurrences of the same compound activity). This increases the algorithm complexity, relative to the XML case.

Our algorithm is based on the following two principles.

1. Decomposition into smaller problems: Recall that when a pattern \( p \) is embedded into an EX-flow \( e \), parts of the pattern are matched to parts of the EX-flow. Our algorithm will correspondingly compute the likelihood of the full boolean query (w.r.t. the given BP) out of the likelihoods of the query parts (w.r.t. parts of the BP). To make this precise we next extend the notions of queries and likelihood.

   1. Given an EX-pattern \( p \), we denote by \( \text{Parts}(p) \) the set of all EX-patterns (related to as boolean queries) that are obtained from \( p \) by removing one or more nodes and edges, and all the conjunctions of such queries. The semantics of a conjunction is defined in a natural manner. Fig. 5.3 depicts an EX-pattern \( p \) and its possible splits (denoted \( p1 - p7 \)).
2. Recall that an EX-pattern may include simple and transitive activities. Implementations of simple pattern activities are matched to direct implementations of the corresponding EX-flow activities; implementations of transitive pattern activities may be matched anywhere (deeper) in the implementation subgraph. We extend, correspondingly, our definition of Parts(p): each EX-pattern q appears in it in two forms: as $q^{\text{direct}}$ and as $q^{\text{indirect}}$.

3. We have defined $\text{likelihood}^h(p)$ for the likelihood that an EX-flow starting from the BP root satisfies the EX-pattern p. In an analogous way, for a compound activity a and some $q^{\text{direct}}$ ($q^{\text{indirect}}$) in Parts(p), we can define $\text{likelihood}^h(q^{\text{direct}}, a)$ (resp. $\text{likelihood}^h(q^{\text{indirect}}, a)$) for the likelihood that a sub-flow starting from an implementation of a satisfies q (in)directly. For that, we extend the definition of $f$-likelihood to sub-flows rooted at any compound activity a.

2. The Principle of Inclusion and Exclusion: As explained above, our algorithm will compute the likelihood of the full query p out of the likelihoods of other “smaller” sub-queries in Parts(p). In various points of the computation we shall utilize the principle of inclusion and exclusion (in its probabilistic form) [16]. This principle allows to compute the likelihood of a disjunction of query parts from the likelihoods of conjunctions of query parts, namely from the likelihood of other sub-queries in Parts(p).

In general, this principle may be captured by the following equation:

\[
\text{likelihood}^h(\bigvee_{i=1,\ldots,n} p_i, a) = \sum_{k=1,\ldots,n} (-1)^{k-1} \sum_{I \subseteq \{1,\ldots,n\}, |I| = k} \text{likelihood}^h(\bigwedge_{i \in I} p_i, a) \quad (5.1)
\]
Observe, however, that applying this principle yields an exponential blow-up of the expression size. Thus, we shall take caution in applying it only over disjunctions whose size is independent of the BP specification size (and depends only on the query size).

Algorithm for non-recursive BPs  We next describe our algorithm, called EVAL-BOOL-QUERY. W.l.o.g., we assume below that the query root activity is labeled either by the name of the BP root activity or by any. (Otherwise likelihood\(_b(p) = 0\).) We further start by assuming that compound query activities are not annotated by formulas, and treat formulas afterwards.

Given a query \(p\) and a BP \(s\), EVAL-BOOL-QUERY (Algorithm 6) computes likelihood\(_b(p)\) via Dynamic Programming. Observe that the non-recursive nature of the BP specification induces a partial order \(>_{s}\) over its compound activities, such that \(a_1 >_{s} a_2\) if \(a_2\) may appear in an EX-flow originating from \(a_1\). The algorithm first completes this partial order to a full one (line 1) and processes the compound activities of \(s\), in reversed order, from the most internal activities to the root activity (lines 2-3). EVAL-BOOL-QUERY (gradually) fills in a table \(T\) of likelihoods whose rows and columns correspond to sub-queries (direct and indirect) and compound activities, resp. For each compound activity \(a\) and for all direct (resp. indirect) queries \(q^{\text{direct}}(q^{\text{indirect}})\) in Parts\((p)\), the algorithm computes likelihood\(_b(q^{\text{direct}}, a)\) (likelihood\(_b(q^{\text{indirect}}, a)\)), using the likelihoods computed for the preceding activities (Lines 5 and 8, resp.). The indirect likelihoods are computed only as auxiliaries, as will be explained below.

Let \(\hat{p}\) denote the query pattern \(p\) with its root activity removed, and annotated as indirect, if \(p\)'s root activity is transitive, and otherwise as direct. For example, for the query \(p\) in Figure 5.3(a), \(\hat{p} = p_2^{\text{indirect}}\).

Note that with the notations introduced above, likelihood\(_b(p) = \text{likelihood}_b(\hat{p}, r)\), with \(r\) being the root activity of the BP \(s\). At the last iteration of
EVAL-BOOL-QUERY the root activity $r$ is reached, and (among others) $\text{likelihood}^\#(\hat{p}, r)$ is computed. Then $T[\hat{p}, r]$, which contains this value, is returned (line 11).

---

**Algorithm 6: EVAL-BOOL-QUERY**

- **Input:** $p$, $s$ (with root activity $r$)
- **Output:** $\text{likelihood}^\#(p)$

```plaintext
activities ← OrderActsBottomUp(s) ;
while activities ≠ φ do
    a ← pop(activities) ;
    foreach $q^{\text{direct}} \in \text{Parts}(p)$ do
        $T[q^{\text{direct}}, a] ← \text{ComputeDirectLikelihood}(q^{\text{direct}}, a)$ ;
    end
    foreach $q^{\text{indirect}} \in \text{Parts}(p)$ do
        $T[q^{\text{indirect}}, a] ← \text{ComputeIndirectLikelihood}(q^{\text{indirect}}, a)$ ;
    end
end
return $T[\hat{p}, r]$ ;
```

---

We next explain the two functions responsible for the computation of likelihoods, namely \text{ComputeDirectLikelihood} and \text{ComputeIndirectLikelihood} invoked in Lines 5 and 8 (resp.) of Algorithm 6.

**ComputeDirectLikelihood**  Given $q^{\text{direct}} \in \text{Parts}(p)$ and a compound activity $a$ of $s$, \text{ComputeDirectLikelihood} computes $\text{likelihood}^\#(q^{\text{direct}}, a)$. Recall that $q$ has a nested-DAG shape. The “upper level” of $q$ refers to the outer most nodes and edges of $q$, reachable by paths that do not include implementation edges. A matching of $q^{\text{direct}}$ corresponds to (1) matching its upper level to some direct implementation of $a$, and then (2) matching the implementations of the compound activities nodes $N_1, ..., N_k$ appearing in the upper level to implementations of the corresponding BP activities (a direct/indirect match for the simple/transitive compound activities).
Ignore for now the matching of the upper level, and consider the compound activities nodes \( N_1, ..., N_k \). We denote the sub-queries rooted at these nodes by \( q_1, ..., q_k \). If \( N_i \) is transitive, then \( q_i \) appears in an “indirect” form, otherwise in a “direct” form. We then consider the matching of all sub-queries, i.e. \( \bigwedge_{i=1, ..., k} q_i \).

The following identity holds:

\[
\text{likelihood}^b(\bigwedge_{i=1, ..., k} q_i, a) = 1 - \text{likelihood}^b(\bigvee_{i=1, ..., k} \neg q_i, a) \quad (5.2)
\]

We thus focus on computing \( \text{likelihood}^b(\bigvee_{i=1, ..., k} \neg q_i, a) \). Using the principle of inclusion and exclusion, this term can be represented as sum of terms, all having the form \( \text{likelihood}^b(\bigwedge_{i \in J} \neg q_i, a) \) for some subsets \( J \) of \( \{1, ..., n\} \). Note that the exponential blow-up here is only in the size of the query, and not in that of the BP specification.

Now, re-consider the possible embeddings of the query’s upper level, denoted \( \text{embs}(q, \text{imp}) \) for each implementation \( \text{imp} \) of \( a \). Each such conjunction must hold (1) in the chosen implementation of \( a \) and (2) for all embeddings of the query within the chosen implementation. As we require that for all embeddings, all nodes are \textit{not} matched, we may simply consider a single set of specification nodes, that contain all nodes that participated in any embedding. Moreover, the fulfillment of the negated expression is independent in-between nodes. Thus we conclude (recall that for a node \( n \) of a BP specification, \( \lambda(n) \) is the activity name labeling \( n \)):

\[
\text{likelihood}^b(\bigwedge_{i \in J} \neg q_i, a) = \sum_{\text{imp} \in \tau(a)} \text{c-likelihood} \ (\text{imp})^* \prod_{n \in \text{embs}(q, \text{imp})} \text{likelihood}^b(\bigwedge_{i \in J} \neg q_i, \lambda(n)) \quad (5.3)
\]

Note that now each of the expressions \( \text{likelihood}^b(\bigwedge_{i \in J} \neg q_i, \lambda(n)) \) satisfies \( \lambda(n) <_s a \), as all of these nodes appeared in implementations.
However, $\bigwedge_{i \in J} \neg q_i$ does not belong to $Parts(q)$, so they do not appear in $T$. We thus apply the following manipulation over it. First, we apply negation:

$$likelihood^b(\bigwedge_{i \in J} \neg q_i, \lambda(n)) = 1 - likelihood^b(\bigvee_{i \in J} q_i, \lambda(n))) \quad (5.4)$$

Now we apply again the principle of inclusion and exclusion over $\bigvee_{i \in J} q_i$ (again, only dependent on the query size) and obtain expressions of the form $likelihood^b(\bigwedge_{i \in J} \neg q_i, \lambda(n))$. The expressions of the sort $\bigwedge_{i \in J} q_i$ are conjunctions of sub-queries, hence belong to $Parts(p)$, and thus the required likelihood values already appear in the Dynamic Programming table $T$ and can be used.

**ComputeIndirectLikelihood** The computation here is similar, but slightly more complicated due to the possibly indirect matches. Recall that when embedding a query indirectly, query parts may be matched to different levels of the implementation nesting. Thus, instead of dividing the query simply by its compound nodes, we define the notion of *query splits*. $\{q_1, ..., q_m\}$ is a split of $q_{\text{indirect}}$ if each $q_i$ is a sub-graph of $q_{\text{indirect}}$, and every node or edge of $q_{\text{indirect}}$ appear in exactly one of the $q_i$’s. We denote the set of all splits of $q_{\text{indirect}}$ by $\text{splits}(q_{\text{indirect}})$, and consider the likelihood of $\bigvee_{sp \in \text{splits}(q_{\text{indirect}})} \bigwedge_{q_i \in sp} q_i$.

By applying the principle of inclusion and exclusion, we obtain expressions of the form $\bigwedge_{sp \in SP} \bigwedge_{q_i \in sp} q_i$ for some subsets $SP$ of $\text{splits}$. (Note that the number of splits is, once again, only a function of the query size). We can now unite the two $\bigwedge$ expressions and obtain $\bigwedge_{q_i \in sp'} q_i$ for some $sp'$. From this point on the computation proceeds exactly as for **ComputeDirectLikelihood** (with the only difference being that the $q_i$’s are now not necessarily partial flows rooted at compound activities).
Note. We have assumed above, for simplicity, that compound activities in the query are not annotated by formulas. In the presence of such formulas, the sub-queries $q_i$ in Eq. 5.2 are annotated by the formula of their respective compound activity $N_i$. Then, only implementations guarded by the same formula are considered for embedding (see Eq. 5.3).

Example 5.5.5 To illustrate the operation of the algorithm, let us consider the BP specification $s$ given in Fig. 5.1 and the boolean query $p$ of Fig. 5.3(a). The BP in Fig. 5.1 is recursive, whereas our current algorithm handles only non-recursive BPs. So, just for this example, let us modify the BP, and obtain a non-recursive one, by removing the implementation guarded by $\text{choice} = \text{“reset”}$. To compensate for its likelihood, assign a likelihood of 0.8 to the formula $\text{choice} = \text{“confirm”}$. The activities may now be partially ordered according to the implementation relation, e.g. $\text{chooseTravel} \succ_s \text{Login}$, $\text{chooseTravel} \succ_s \text{Flights}$, $\text{chooseTravel} \succ_s \text{Advertise}$, etc., and this partial order can be completed to a full one. We next explain the manner in which our algorithm computes the likelihood $b(p, s)$. The
computation starts at the “smallest” (according to the above order) activities of \( s \), e.g. Hotels, Flights and Confirm. For each such activity, all Parts(\( p \)) are matched. Here, most matches yield a match likelihood of 0, apart from those matching the choice of “BA”, “Marriott”, and “cancel” to the corresponding implementation of Flights, Hotels and Confirm, resp. (yielding likelihood values of 0.7, 0.6 and 0.2 resp.).

Proceeding to chooseTravel, we again consider the matching of all elements of Parts(\( p \)). In particular, let us focus on the matching of \( p^2_{\text{indirect}} \) to chooseTravel (recall that its likelihood = \( \text{likelihood}^b(p) \)).

As we are dealing with an indirect match, we consider all the possible splits of \( p^2 \). One possible split consists of \( p^5 \), \( p^6 \) and \( p^7 \), another consists of \( p^5 \) and \( p^4 \), etc. We disjunct over all splits, and in each split conjunct over all its parts. Observe that the existence of matchings to the splits are dependent: for instance, if there exists a match in a given EX-flow to \( p^5 \) and \( p^4 \), then obviously matches to \( p^5 \), \( p^6 \) and \( p^7 \) exist as well. This is where the Principle of Inclusion and Exclusion (Equation (2)) over all possible splits (along with the likelihood of match for each split) comes into play, for each possible implementation of chooseTravel. At the end of the computation we obtain the following equation.

\[
\text{likelihood}^b(p^2_{\text{indirect}}, \text{chooseTravel}) = \\
ce^{-\text{likelihood} (\$\text{searchType} = \text{“flights + hotels”})} \\
\times [ \text{likelihood}^b(\$\text{airline} = \text{“BA”}_{\text{indirect}}, \text{Flights}) \\
\times \text{likelihood}^b(\$\text{hotel} = \text{“Marriott”}_{\text{indirect}}, \text{Hotels}) \\
\times \text{likelihood}^b(\$\text{choice} = \text{“cancel”}_{\text{indirect}}, \text{Confirm})] \\
\quad + ce^{-\text{likelihood} (\$\text{searchType} = \text{“flights + hotels + cars”})} \\
\times [ \text{same term as above } ]
\]

(5.5)

Additional summands, summing up to 0 due to absence of embedding,
are omitted for brevity. ²

Recall that the likelihood values for the terms that appear in the right-hand side of the above equation were already computed in previous steps, due to the bottom-up computation. We thus obtain

\[
\text{likelihood}^b(p_{2\text{ indirect}}, \text{chooseTravel}) = 0.25 \times 0.7 \times 0.6 \times 0.2 + 0.25 \times 0.7 \times 0.6 \times 0.2 = 0.042
\]

**Complexity**  The number of arithmetic operations performed by the algorithm is polynomial in the BP size, with the exponent depending on the query size. The number of bits of the computed likelihood values may become, however, exponential in the size of \( s \), as shown by the following example.

**Example 5.5.6** The following is an example for a specification \( s \) and a query (EX-pattern) \( p \) such that \( \text{likelihood}^b(p, s) \) requires exponentially many bits. \( s \) has a root DAG with a single compound activity \( A_0 \), having a single implementation (with \( c \)-likelihood value of 1) containing two compound \( A_1 \) activities. \( A_1 \), again has a single implementation (with \( c \)-likelihood value of 1) containing two compound \( A_2 \) activities, the single implementation of \( A_2 \) contains two \( A_3 \) activities, and so on, up to \( A_n \). \( A_n \) has 2 implementations, each with \( c \)-likelihood of 0.5. The first (second) implementation consists of a single atomic activity \( a \) (b).

Now, consider a boolean query \( p \) that tests if an activity \( a \) occurred somewhere in the execution. (\( p \) has a transitive root activity whose implementation consists of an activity labeled by \( a \) and connected to the root activity by transitive implementation edges.) \( p \) is satisfied by all the EX-flows of \( s \), except the one where the implementation chosen for all the \( A_n \) activities is \( b \). Thus \( \text{likelihood}^b(p, s) = 1 - 0.5^{2n} \), requiring number of bits that is exponential in the size of \( s \)

²When a sub-query in this equation is a guarding formula, it represents the empty sub-query annotated by the corresponding formula as its only constraint.
Nevertheless, we note that if we have a unit-cost RAM model with exact rational arithmetic (i.e., algebraic operations on arbitrary rationals can be done in unit time) [15], we do not have to worry about the size of the numbers. Consequently,

**Theorem 5.5.7** For non-recursive BP specifications, BOOL-EVAL is in EXPTIME under Turing computation model and in PTIME (data complexity) under unit-cost model with exact rational arithmetic.

### 5.5.2 The general case

To see that Algorithm EVAL-BOOL-QUERY cannot be directly applied over recursive BPs, observe that it assumed a total order over the BP activity names. Likelihood of queries with respect to a given activity $a$ were computed out of previously computed likelihoods for “smaller” activities. Obviously, such an order does not exist for recursive BPs. Thus, instead of using simple arithmetic, our refined algorithm generates an equations set whose solution corresponds to the query likelihood. We explain this in more details next.

**Refined Algorithm** Given a BP specification $s$ and an EX-pattern $p$, recall that EVAL-BOOL-QUERY gradually computed likelihoods for each [sub-query $q \in Parts(p)$, activity name $a$]. We create a variable $X_{q,a}$, whose value will reflect $likelihood^b(q,a)$, for each such pair $[q,a]$. We then choose some arbitrary order over the BP activities, and using this order, we follow the computation of EVAL-BOOL-QUERY, attempting to gradually compute likelihood values. However, in contrast to the non-recursive case, the computation of likelihood for some $[q,a]$ (i.e. computation of value for $X_{q,a}$) may require some value $likelihood^b(q',a')$ that was not computed yet (possibly $[q',a']$ is $[q,a]$ itself). To account for that, we create an equation with $X_{q,a}$ on its left-hand side. The right-hand side will contain an arithmetic expression similar to that obtained in the non-recursive case, but with $likelihood^b(q',a')$ replaced.
by $X_{q',a'}$, and so on. For each pair $[q,a]$, this process results in a single equation; repeating the computation for all pairs of activities names and sub-queries, the result is a set of polynomial equations. We denote the obtained equations system by $ES[p]$ and show the following.

**Proposition 5.5.8** The solution of $ES[p]$, with all variables in $[0,1]$, restricted to $X_{p,r}$ ($r$ is the root activity of $s$), is exactly $\text{likelihood}^b(p,s)$ (If more than one such solutions exist, we use the Least Fixed Point solution).

**Proof.** To prove the proposition we need to show that a Least Fixed Point (LFP) solution exists and captures the correct likelihood values.

We first note that [42] also uses a set of equations to describe the termination probability of Recursive Markov Chains. An important property of the equations in [42] is that all the coefficients in the equations are positive. The consequent monotonicity of the polynomials is then used to prove the existence of an LFP.

In contrast, in our case, the equations may have negative coefficients (due to the use of the inclusion-exclusion principle). Thus, the proof of [42] cannot be directly applied here. Nevertheless, the system is “piece-wise” monotone, in the following sense: consider a “part-of” partial order over the sub queries $q$ of $p$ (including $p$ itself). For each such $q$ and an activity $a$, the computation of $\text{likelihood}^b(q,a)$ uses either likelihood values computed for queries that are “smaller” than $q$, or values of the form $\text{likelihood}^b(q,b)$ for some activity $b$ (possibly $b = a$). We thus solve the equations for $\text{likelihood}^b(q,a)$ in an increasing order of such $q$ (and for all compound activities $a$). Now, terms in the polynomial that correspond to queries smaller than $q$ may be simply replaced by constants (computed in previous iterations). After substitution, the formula contains only variables for some $\text{likelihood}^b(q,b)$. It follows from the construction that these variables appear with positive coefficients, as follows: recall the com-
putation performed by Algorithm EVAL-BOOL-QUERY and note that 
likelihood^b(q, b) = likelihood^b(\bigwedge_{i=1, \ldots, k} q_i, b) (where k is the number 
of query parts considered in a split) may only appear in our com-
putation as is (i.e. as likelihood^b(\bigwedge_{i=1, \ldots, k} q_i, b)) or as the “evolve-
ment” (via mathematical transformations) of an expression of the form 
likelihood^b(\bigvee_{i=1, \ldots, k} q_i, b) (i.e., using the algorithm notation, J = \{1, \ldots, k\}). 
Furthermore, note that in each application of the inclusion and ex-
clusion transformation over the above expression, likelihood^b(q, b) is 
multiplied by (-1)^{k+1}. Now let us track back the creation of this ex-
pression by the algorithm: at the last step (see text following equation 
5.4), it was obtained from likelihood^b(\bigvee_{i=1, \ldots, k} q_i, b) and bears a coef-
ficient of (-1)^{k+1}. likelihood^b(\bigvee_{i=1, \ldots, k} q_i, b) itself appears in equation 
5.4 in a negative form, thus we have to multiply by (-1) and obtain a 
coefficient of (-1)^{k+2}. 
Next, equation 5.3. multiplies expressions by 
c-likelihood values of guarding formulas, which are positive and do not 
change the sign. Next, the text below equation 5.2 uses again the prin-
ciple of inclusion and exclusion, leading to the added factor of (-1)^{k+1} 
and yielding a current total of (-1)^{2k+3} (multiplied by some positive 
value) as a coefficient. Last, note that equation 5.2 yields a multi-
plication by (-1), leading to a total coefficient of (-1)^{2k+4} multiplied 
by some positive value; (-1)^{2k+4} is positive, hence the coefficient of 
likelihood^b(q, b) in the obtained equation is also positive. The least 
fixed point solution for the system may thus be computed in a bottom-
up fashion (dictated by the order over query parts) using in each step 
the Algorithm of [42]; this least fixed point solution constitutes the 
correct probabilities. □

The LFP solution of ES[p] may consist of irrational values and 
cannot be computed exactly (recall Theorem 5.5.2). However, it may 
be approximated.

Lemma 5.5.9 Given an equations system ES[p] as above, its LFP so-
lution may be approximated up to j bits of precision, in time exponential
in the number of variables in $ES[p]$ and linear in $j$.

**Proof.** We follow here the proof of Thm. 4.2 of [42] that uses the existential theory of reals [23] to approximate the LFP solution of an equations set. We note that here (unlike in Prop. 5.5.8), the proof of [42] does not use the coefficients positiveness, so we can use the same proof.

$\square$

Combined with Proposition 5.5.8, this concludes the Proof of Theorem 5.5.4.

### 5.6 Projection Queries With sum semantics

We are now ready to study top-k projection queries with sum aggregation function. Recall our notation of $q_1(s)$ for the set of projection results (given a projection query $q$ and a BP specification $s$). For an answer $e' \in q_1(s)$ we say that $e \in \text{flows}(s)$ is a source of $\alpha$ if there exists an embedding of $q$ in $e$ whose restriction to the projected part of $q$ is isomorphic to $e'$. Under sum semantics, the likelihood of an answer $e' \in q_1(s)$ is the sum of likelihood of its sources, namely

$$\text{likelihood}(e', q, s) = \sum \{\Delta(e) \mid e \in \text{flows}(s) \land e \text{ is a source of } e'\},$$

where $\Delta$ is the f-likelihood function for $s$. When $s$ is clear from the context we will omit it and simply use $\text{likelihood}(\alpha, q)$.

We define $\text{TOP-K-PROJ-SUM}$ as the problem of retrieving, given a BP specification $s$, a projection query $q$ and a number $k$, the set of $k$ answers having the highest likelihood out of these in $q(s)$ (if there is more than one such set, we may choose one arbitrarily).

In the remainder of this Chapter we consider the complexity of $\text{TOP-K-PROJ-SUM}$, and in particular show it can be harder than $\text{BOOL-EVAL}$. Then, we show restricted cases that allow for a more efficient query evaluation. We start with an example.
Example 5.6.1 Re-consider the query in Figure 5.2(a), with the rectangle denoting its projected part. The projection focuses on the sub-flows that may occur between the choice of a “BritishAirways” flight and the final confirmation. There are infinitely many answers to this query, (varying in the path assigned to the projected transitive edge), as in-between these two choices there may appear paths of arbitrary length. Two possible such paths appear in Fig. 5.2 (b) and (c). The first corresponds to users that choose at some point a “flights only” search, pick a “BA” flight and then immediately confirm. The second corresponds to users that choose at some point a “flights+hotels” search, pick “BA” as airline and Mariott as hotel, and confirm. The respective query answers each have an infinite number of origins, corresponding to any number of selections made and reset by the user, prior to making her final reservation. To compute the score of a projection result, one must (implicitly) sum the likelihood values of infinitely many qualifying EX-flows.

General Framework Given a projection query \( q = (p, P) \) and a BP specification \( s \), a candidate answer is obtained by some (restricted) embedding \( \alpha \) that assigns (1) activity names from \( s \) to the any labels of \( P \), and (2) a sequence of labeled nodes to each transitive edge of \( P \). (Obviously, some of these candidate answers are not valid, i.e. have no origin flow in \( s \), and will have zero likelihood). Consider such an embedding \( \alpha \) leading to a query answer \( e' \). Let \( \alpha(p) \) denote the pattern obtained from \( p \) (including its non-projected part) by instantiating the any nodes and the transitive edges in \( P \) according to \( \alpha \). The score of \( e' \) is precisely the likelihood of \( \alpha(p) \), treated as a boolean query.

Thus, in principle one may think of the following algorithm: generate all possible assignments \( \alpha \), then use Algorithm \textsc{Eval-Boolean-Query} from the previous section, to compute \( \text{likelihood}^b(\alpha(p)) \) and (consequently \( \text{likelihood}(e', q) \) for the corresponding \( e' \)) for each \( \alpha \), and finally
return the top-k out of these. This is very similar to the framework employed for probabilistic XML query evaluation in [65]. However, there are two obstacles here, absent from the XML settings: first, the number of possible answers may be infinite, and second, there are cases where \( \text{likelihood}^b \) values may not be computed exactly. We next consider these two obstacles and explain how we tackle them.

**Obstacle 1: Possibly infinite number of answers** We re-consider the embeddings \( \alpha \) used, in our general framework, to generate possible query answers. Recall that the assignments are over (1) any-labeled nodes and (2) transitive edges. While the number of combinations for the first case (any labels) is bounded by \( |s|^{|q|} \), the number of possible paths in a recursive BPs, and consequently the number of path assignments to transitive edges, may be infinite. This leads to infinitely many possible projection answers. To bound the number of assignments considered for transitive edges, we use a “small world” property, similar to the one used for \textit{max} semantics in Algorithm \textsc{Top-K-Paths} above; this time the proof has to account to the \textit{sum} semantics, effecting which of the answers appear in the top-k set. The following Lemma holds:

**Lemma 5.6.2** Given a BP \( s \) and a projection query \( q = (p, P) \), there exists a set of top-k answers of \( q \) w.r.t. \( s \) where in each answer all the paths assigned to transitive edges in \( P \) are of length bounded by \( |s| \ast k \).

**Proof.** The Lemma may be shown by induction on \( k \). Consider first \( k = 1 \). Assume that \( q \) is embedded by some embedding \( \alpha \) in some EX-flow \( e \), and let \( P \) be a path in \( e \) such that a transitive edge \( T \) of \( q \) is matched to \( P \). Further assume that \( |P| > |s| \). In particular, this means that \( P \) contains a recursive invocation of at least one activity \( a \in S \), otherwise the length of \( P \) may not exceed the total number of nodes in implementation graphs of \( s \). We construct an EX-flow \( e' \) that is obtained from \( e \) by subsequently omitting sub-flow of \( e \) that are rooted at recursive invocation of activities: first, we omit the sub-flow
rooted at the recursive invocation of $a$. If there still exists a recursive invocation of some activity $a'$ within $P$, we then omit the sub-flow rooted at $a'$, and so forth, until any activity name appears at most once among the remaining nodes of the path $P$. We denote the remaining nodes and edges that originally were in $P$ as $P'$, and observe that $P'$ is still a path (as we only removed connected sub-paths of $P$). Clearly, $e'$ is a flow of $S$, and $f$-likelihood ($e'$) $\geq$ $f$-likelihood ($e$) due to the monotonicity of $f$-likelihood. There exists an embedding $\alpha'$ of $q$ in $e'$, obtained from $\alpha$ by replacing the path $P$ assigned to $T$ by the new path $P'$. The same construction may be employed for each such path $P$, and consequently we obtain the existence of a top-1 answer where each path assigned to $T$ is of length bounded by $|s|$.

For $k > 1$, assume that there exists $k$-1 results with transitive edges mapped to paths of length bounded by $|s| \times (k - 1)$. Consider an embedding that assigns a path $P$ of length greater than $|s| \times k$. Then in particular, it contains more than $k$ recursive invocations of compound activities. We may employ the same technique as above to shorten the path into a path $P'$ that contains at most $k$ such recursive invocations. $P'$ does not appear as one of the $k$-1 results as its length is greater than $|s| \times (k - 1)$, but it is better weighed than $P$ due to the monotonicity of $f$-likelihood, and may thus be used as the $k$'th-best result. $\square$

**Obstacle 2: No exact computation for boolean queries** We have shown in Section 5.5 cases where an exact algorithm for $\textsc{Bool-Eval}$ does not exist. In such cases, we must settle for an algorithm for $\textsc{Approx-Bool-Eval}$ that approximates the likelihood values up to a given $\epsilon$. However, we can still utilize such an approximation algorithm for exact top-k query evaluation, if the answers have *discrete likelihoods*, as defined next.

**Definition 5.6.3** Given a projection query $q$ and a BP specification $s$, we say that the answers of $q$ (w.r.t $s$) have discrete likelihoods if
there exists some $\epsilon$ such that for each two answers $e, e' \in q(s)$ where $\text{likelihood}(e, q) \neq \text{likelihood}(e', q)$, we have that $| \text{likelihood}(e, q) - \text{likelihood}(e', q) | > \epsilon$. $\epsilon$ is called the separation factor of $q$ with respect to $s$.

We assume in the sequel that the query answers have discrete likelihoods, and that the separation factor is known. We consider below the implications of withdrawing these assumptions.

The following Lemma holds.

**Lemma 5.6.4** For a BP $s$ and a query $q$ s.t. the answers of $q$ w.r.t. $s$ have discrete likelihoods with separation factor $\epsilon$, a set of top-k answers according to approximated likelihood$^b$ values, up to $\epsilon/2$ precision, is also a set of top-k answers for to the exact likelihood$^b$ values.

**Proof.** Denote the approximated score of a projection result according to our algorithm by approx-likelihood, and the set of $k$ results having the highest approx-likelihood by APPROX-TOP-K. Assume, by contradiction, that there exists two results $e, e'$ such that $e \in \text{APPROX-TOP-K}$ and $e'$ is not, but $\text{likelihood}^b(e') > \text{likelihood}^b(e)$. Since $e'$ is not in APPROX-TOP-K, it holds that approx-likelihood($e'$) < approx-likelihood($e$), i.e. approx-likelihood($e$) - approx-likelihood($e'$) > 0. But $\text{likelihood}^b(e) - \epsilon/2 < \text{approx}\text{-likelihood}(e) < \text{likelihood}^b(e) + \epsilon/2$, and similarly for $e'$. Thus $0 < \text{approx-likelihood}(e) - \text{approx-likelihood}(e') < \text{likelihood}^b(e) - \text{likelihood}^b(e') + \epsilon$. I.e, $\text{likelihood}^b(e') - \text{likelihood}^b(e) < \epsilon$. But we have assumed $\text{likelihood}^b(e') \neq \text{likelihood}^b(e)$. This contradicts the assumption of the $\epsilon$ separation factor. \qed

We now have an exact EXPTIME algorithm for TOP-K-PROJ-SUM: generate the (exponentially large) set of candidate answers based upon Lemma 5.6.2, then use an algorithm for APPROX-BOOL-EVAL to approximate likelihood$^b$ for each such answer up to $\epsilon/2$, where $\epsilon$ is the separation factor, and declare the top-k answers.

143
Following common practice, let $F^L$ be the class of problems solvable in time complexity $F$, when given an oracle solving a problem $L$. We obtain:

**Theorem 5.6.5**  
1. If there exists an oracle for \textsc{bool-eval}, then \textsc{top-k-answers} $\subseteq \text{EXPTIME}^{\text{bool-eval}}$  
2. If discrete likelihoods are guaranteed with respect to the input BP and query, then \textsc{top-k-answers} $\subseteq \text{EXPTIME}^{\text{approx-bool-eval}}$

In particular, using Algorithm \textsc{eval-bool-query} as an oracle, we obtain the following corollary:

**Corollary 5.6.6** \textsc{top-k-answers} may be solved in \text{EXPTIME}, for: (1) non-recursive BP specifications, and (2) recursive BP specifications, when the query has discrete likelihoods w.r.t. the specification.

We can also show that this is the best that can be achieved, in the general case. Define the decision problem \textsc{top-k-proj-sum} as the problem of deciding, given a BP specification, a projection query, and a bound $b$ whether there exists a projection result of likelihood greater than $b$.

**Theorem 5.6.7** \textsc{top-k-proj-sum} is NP-hard (under Turing computation model) in the size of the BP specification $s$, even for non-recursive specifications and for queries with no transitive nodes.

**Proof.** The proof works by reduction from 3-SAT. Given a 3NF formula with $C$ clauses and $n$ variables, $x_1, \ldots, x_n$, we generate a BP specification $s$ and a query $q$ as follows: The BP root activity $r$ has $C$ implementations, each with $c$-likelihood of $1/C$. Each implementation represents a clause, and has a start activity $S$, end activity $E$, and three sub-graphs connecting $S$ and $E$, each corresponding to one literal of the clauses. The first activity in each such subgraph is $X_1$. If $x_1$ appears in the corresponding literal positively (negatively), $X_1$ has one child activity labeled $T$ ($F$). Otherwise, if $x_1$ does not appear in
the literal, $X_1$ has two children labeled $T$ and $F$. These children are parents to an activity $X_2$ that again has one or two children, depending if the variable $x_2$ appear in the given literal or not, and so on. (Thus all the $X_i$’s in the subgraph, besides the one of the literal, have two children). The last $T/F$-labeled activities leads to $E$. The EX-pattern of the query $q$ consists of a root activity labeled $r$ with internal flow containing two activities, $S$ and $E$, connected by a transitive edge. The projected part of $q$ consists of all nodes and edges of the EX-pattern.

Note that an answer of $q$ with respect to $s$ is a path, and this path uniquely defines a truth assignment $A$ for the variables $x_1, ..., x_n$: $A(x_i) = true (= false)$ if the $X_i$-labeled node is followed in the query answer by a node labeled by $T$ ($F$). We claim that the 3NF formula is satisfiable iff the top-1 answer of $q$ defines a satisfying assignment. To see this, note that if a satisfying assignment exists, every paths that describes such an assignment appears in all the implementations of the root (since each includes all paths that are consistent with the corresponding claus), hence its likelihood is 1. On the other hand, for every path that describes a non-satisfying assignment, there is at least one implementation that does not contain it (corresponding to a non-satisfied clause), thus the likelihood value of such path is less than 1. It thus suffices to check the top-1 path in order to determine the formula satisfiability.

We note that 3-SAT is a problem not known to be solvable in PTIME under the unit-cost rational arithmetic model. Hence, unlike for BOOL-EVAL, the existence of a PTIME solution for TOP-K-ANSWERS in this model is questionable.

**Restricted Cases.** The exponential overhead in the time complexity of our algorithm was only due to the large number of assignments to transitive edges, that had to be considered. For queries that do not project over such edges, it only remains to consider assignments
to *Any*-labeled nodes, whose number is only exponential in the query size. Consequently,

**Theorem 5.6.8** For queries that do not project over transitive edges, Theorem 5.6.5 holds, with the EXPTIME complexity class replaced by PTIME (data complexity).

In particular, when we combine this restriction over queries with restriction over BP specifications to be non-recursive, we obtain (from Thm. 5.5.7) the following.

**Theorem 5.6.9** For non-recursive BP specifications, and queries that do not project over transitive edges, \textsc{Top-K-Answers} may be solved in PTIME (data complexity) with unit-cost rational arithmetic.

**Withdrawing the assumptions.** Recall that we have assumed above that the projection results bear discrete likelihoods, and that we are given some bound on the separation in-between answers. For practical needs, if this does not hold, one may easily adapt our algorithm to find the top-k results up to an error of $\mu$, for any given $\mu$. I.e, if an answer $e$ is ranked by the algorithm above $e'$ (w.r.t. a BP specification $s$ and a query $q$), it is guaranteed that $\text{likelihood}(e, q, s) > \text{likelihood}(e', q, s) - \mu$. This is done by approximating the likelihood of every boolean answer up to $\frac{\mu}{2}$, then declaring the top-k results ordered by their approximated likelihood.

### 5.7 Related Work

We conclude this Chapter with an overview of related work, complementary to the related work studied in the body of this Chapter (in particular, refer to Section 5.5 for a more detailed comparison to Probabilistic XML and Recursive Markov Chains).
Top-k queries were studied extensively for relational and XML data [58]. Notably, [45] presented an algorithm for top-k queries that aggregate individual scores given to joining tuples, and had many follow-ups [6, 9, 44]. In our context, one may think of $cWeight$ as the equivalent of an individual score, and of $fWeight$ as the aggregation of $cWeight$ values along a given EX-flow. Difficulties specific to our settings are that (1) the size of a given flow, thus the number of aggregated scores, is unbounded (2) the particular properties of the $cWeight$ functions are unique to EX-flows and (3) the number of items (EX-flows) that are ranked is infinite. The key challenge is to find a “small world” whose examination suffices for identifying the top-k EX-flows. Note that while an infinite setting also appears in top-k queries over streamed data [67], such works aggregate over a bounded size sliding window, whereas we consider aggregation over flows of unbounded size.

A particular weight metric that we have studied weigh execution flows by their likelihood of occurring in practice. Ranking by likelihood was also studied in several other settings, specifically for probabilistic XML and for probabilistic (relational) databases. We next provide an overview of both models.

We have mentioned above the model of probabilistic XML [4, 65, 64, 25], and highlighted the differences between this model and our model, in the context of ranking query results using the sum semantics. We further note that in an XML setting, evaluation of projection queries with the maximum aggregation function may be done by a straightforward adaptation of selection queries evaluation algorithms. Thus, studies focused on the sum aggregation function. In contrast, we have shown that, in the settings of probabilistic BP, there are inherent difficulties in adaptation of algorithms for support of projection queries, even for maximum aggregation.

Probabilistic Databases (PDBs) [27, 90] and Probabilistic Relational Models (PRMs) allow representation of uncertain relational informa-
tion. They extend relational algebra to a probabilistic setting, and analyze the complexity of query evaluation in this setting. They show that finding the exact probabilities for query results is \#P-hard (data complexity) [27], though top-k query evaluation is in PTIME, by approximating results probabilities [84]. The hardness, however, stems from combination of projections with joins, absent in our context, and thus does not hold in our settings. In contrast, different difficulties rise in our setting, with the representation of the dynamic nature of flow and the possibly unbounded number of recursive (possibly dependent) activity invocations. In terms of the possible worlds semantics [27], the number of worlds in our model is infinite (rather than large, yet finite, in PDBs). We note that extensions of PRMs to a dynamic setting, called Dynamic PRMs [87], do not allow for practically efficient algorithms.

The sum semantics is a common semantics for projection queries, employed also for probabilistic XML and PDBs [65, 91, 27, 83, 54, 46, 90, 66, 8]. Yet, we have shown that in the BP context it makes query evaluation computationally much harder, compared to max semantics (EXPTIME vs. PTIME). Intuitively, this is because the computation now has to implicitly compute sum-of-probabilities over all relevant flows (and there may be infinitely many).

A variety of formalisms for probabilistic process specifications exist in the literature; in fact, a probabilistic counterpart appears for virtually all models discussed above. For example, Markov Chains [63] extend Finite State Machines to a probabilistic settings, Probabilistic Recursive State Machines (PRSMs) [47] and Recursive Markov Chains (RMCs) [41, 42], discussed above, constitute probabilistic versions of Recursive State Machines, and Stochastic Context Free (Graph) Grammars (SCFG, SCFGG) [79] extend the corresponding notions of grammars to a probabilistic setting. As such, SCFGGs are very expressible, and consequently incur extremely high (non-elementary in the size of
the query [26]) complexity of query evaluation. As explained for the non-probabilistic case, Finite State Machines (and consequently here, Markov Chains) lack the expressive power to capture the possibly recursive processes studied here. For PRSMs, evaluation of strongly expressive query languages such as \textit{PCTL}^* [86] is undecidable in general, and EXPTIME-hard [20] for some fragment. Other works consider linear-time formulas [41], allowing to express only string-like properties. To our knowledge, evaluation of projection queries over probabilistic process models known in the literature was not studied.
Chapter 6

Practical Application

We have utilized the theoretical foundations explained in the previous Chapters for the development of a practical system, named ShopIT (ShoppIng assitanT). ShopIT is a recommendation system for online shoppers. We have implemented the system and demonstrated [36] its assistance to users, in the context of a real-life web-site, namely a fragment of Yahoo! Shopping [99]. We next depict the contributions that ShopIT suggests to users, the system design and the required steps for its application over a real-life web-site. We further present the system technical details, highlighting the models and algorithms that come into play here, and last we present an experimental study of the algorithms performance.

6.1 Online Shopping

On-line shopping is extremely popular nowadays, with many users purchasing products via Web interfaces provided by various vendors. It is common for on-line shops to offer a vast number of products and combinations thereof (e.g. [99, 39]). This is very useful but, at the same time, makes shopping rather confusing. Indeed, it is often very difficult to find the specific navigation flow in the site (i.e. a flow of user clicks / choices / actions) that will lead to an optimal result, best suiting
the needs and preferences of the given user. Consider for example an on-line store (whose example screen appears in Figure 6.1) that allows users to assemble computers from a variety of component parts. The store offers various processors, motherboards, screens etc. Consider a user that is interested in buying a cheap computer with Intel processor inside. Suppose that the user can get a good price by first registering to the store customers club, then passing through some advertisement page that provides such members with discount coupons, and finally buying a certain set of components (including a certain Intel processor) that, when purchased together with the above coupons, yields the cheapest overall price. Clearly, the user might be interested in knowing this information if she seeks for the deal with the best price. Alternatively, the user may prefer combinations where the delivery time is minimal, or may want to use the experience of others and view the most popular navigation flows (and purchases incurred by such flows), assembling an Intel-based computer. If, for instance, the above proposed cheap deal is not among them, the user would more carefully check the quality of the deal components.

6.2 ShopIT

To that end, the ShopIT system comes into play. The system assists on-line shoppers by recommending them on the most effective navigation flows for their specified criteria (e.g. Intel processor inside) and preferences (e.g. low price, delivery time, popularity). Finding these most effective flows is not an easy task: first, the number of possible navigation flows in a given web-site is not only large but infinite, as users may navigate back and forth between pages. Hence, enumerating and ranking all relevant flows is clearly not an option. Second, it is critical to maintain a fast response time in order to provide a pleasant user experience. Finally, the computation must be flexible and adap-
tive: users may wish not to follow a given recommendation as is, but rather navigate on their own, possibly making choices different than those recommended by the system. In this case, the system should adapt its recommendations to comply with the actual choices made by the user.

To face these challenges, we employ the models and algorithms depicted throughout this thesis and use them as a basis for the development of ShopIT. Here the web-site logic is represented via a Business Process, the criteria specified by the user are compiled into a query, and the preferences serve as weight function over the possible choices in the web-site. We note that this weight function is bounded-history, where the weight of choices (e.g. product prices) occasionally depend on prior choices (e.g. in the case of combined discount deals), but this bound is typically low (see discussion in Chapter 4, and also in Section 6.4 below). Then, ShopIT utilizes the top-k query evaluation algorithm described in Section 5, namely Algorithm TOP-K-SELECTIONS given in Chapter 5.2 (in its variant accounting for bounded-history weight functions), to find a set of top-k ranked execution flows. Each such execution flow corresponds to a navigation within the site, and the top-k execution flows are thus presented as recommendations to the user on how to navigate within the site. For simplicity to users, the flow is represented as a sequence of activities to be performed within the website. The user then continues her navigation taking into account the presented recommendations, but may also make choices different than those proposed by the system. For instance, ShopIT might have proposed the purchase of a DDR motherboard and a Pentium 4 processor, but the user might have nevertheless chosen an RDRAM motherboard. In this case, new recommendations, consistent with the actual choices made by the user, are automatically computed. Namely, the system dynamically proposes new top-k continuations that are up to date with the user current navigation, by executing the appropriate top-k algo-
rithm with a revised BP specification, in which the current user position stands for the specification root.

6.2.1 System Overview

ShopIT is implemented in C++, uses the SQLite database and php GUI, and runs on Windows XP. Figure 6.2 depicts the system architecture. We give here a brief overview of the main components and their interaction.

Store Model. We demonstrated [36] the application of the system over a part the Yahoo! Shopping [99] web-site, namely its online Computer Store. To that end, we have designed a Web shop that simulates the on-line computers store of Yahoo! Shopping. The shop is based on real-life data obtained from the original web-site interface, and the only reason that we use a simulated version is technical: avoiding to commit actual purchases in the site. To that end, we have generated a weighted BP modeling the web-site, as follows.

Example 6.2.1 A schematic (partial) description of the model of our on-line computers store is depicted in Fig. 6.3. A customer starts by
logging into the system, authenticating and giving her credit card details. Then, she may choose a product (e.g. Motherboard, CPU, etc.) out of a list. In parallel (i.e. by opening a new tab) she may review suggestions for hot deals or buy discount coupons. Upon a choice of product type, she is directed to a page where the different brands that the store offers for this product type are listed, and the corresponding products can be selected. The user can next choose to cancel her selection, to search for more products, or alternatively to exit and pay. Note the recursive nature of this application: users can re-start the search (i.e. call S1) an unbounded number of times, each time purchasing another item.

The BP structure constituting the application abstract model, was manually configured following the logical flow structure of Yahoo! Shopping application. We note however that, in general, many Web-based applications are specified in declarative languages such as BPEL [17], and then an automated extraction of their abstract model structure may be possible [11].

The products information, including compatibility relation in-between products, as well as additional parameters such as products cost, dis-
count deals, shipment time etc. were automatically retrieved via a standard Web interface provided by the Yahoo! website. The \textit{cWeight} function was automatically derived to reflect this data, for each of the above ranking metrics. Note that due to the existence of discount deals, the \textit{cWeight} standing e.g. for monetary cost, is history-bounded.

The aggregation function \textit{aggr} was then easily configured for the various proposed weight functions (e.g. summation for monetary cost, multiplication for products popularity.)

\textbf{Query Engine.} The query engine is composed of two components. The first is the \textit{top-k queries evaluator}, implementing the top-k algorithms of the algorithm given in Section 4. The query engine receives, as input, from the user, her search criteria and chosen ranking metric, and computes the initial suggestion of top-k qualifying navigation paths. shopIT supplies a Graphical User Interface that allows users to specify their criteria for search, as depicted in Figure 6.4. The specified criteria are compiled into an EX-pattern, which in turn is fed to the
The second component is the adaptive recommendation engine, that is continuously informed about the user actual navigation choices (or changes to her search criteria and ranking choice) and adapts the presented top-k suggestions accordingly. Namely, at each point of the navigation, Algorithm $\text{TOP-K}$ is executed, with the current user location serving as the new BP root (optimizations of this repeated computation are subject of current research).

We have designed the query engine so that it is accessible through an API that allows the placement of user queries and preferences, and the retrieval of the corresponding recommendations. This general API can be used to incorporate ShopIT within a given website.

**The ShopIT virtual store.** Users interact with a virtual store that wraps the original store. User actions are passed, through the API, to the ShopIT engine and to the (original) store application. The obtained recommendations are then presented to the user alongside the resulting store screens (see Fig. 6.5, Fig. 6.6 for the recommendations provided by ShopIT for the minimum price and maximum popularity weight functions, respectively). Each recommendation consists of a sequence of proposed actions, such as “click on button X”, “choose option Y"
Figure 6.5: ShopIT - Ranking by price

Figure 6.6: ShopIT - Ranking by popularity
at box Z', etc., and is accompanied by its corresponding weight (e.g. total price, likelihood, etc.). For “in-house” applications that allow interference with their graphical interface (like our simulated Yahoo! Shopping store), ShopIT further reorders the items on the screen to reflect their relative “precedence” (i.e. the rank of the best navigation flow in which they participate).

6.3 Experimental Study

We have provided throughout the thesis a detailed theoretical analysis of our algorithms performance. Using our top-k algorithms within the framework of the ShopIT system, we have obtained indications of its good performance in practical cases. To further assess the applicability of these algorithms to real-life cases we conducted an experimental study of their performance.

Recall that our top-k query evaluation algorithms consists of two steps: the first step is the “intersection” of the query and the BP specification, and the second part is a top-k analysis over the obtained “intersected” BP specification. The performance of the algorithm for the first part always reaches the worst case bound, and the experimental study of its execution time thus yields no interesting results. In contrast, the top-k analysis algorithms use an “early stop” condition; the goal of our experimental study was thus to identify the improvement gained by the early stop condition in practical cases. To that end, we generated a simple benchmark based on the worst case performance of our algorithms: recall that Algorithm TOP-K (see Chapter 4) serves as the basis for all top-k algorithms suggested in this thesis. Note that the worst-case complexity of TOP-K is reached when all entries of FTable are filled. To assess how much time this would take we implemented a variant of TOP-K, called WC (for Worst-Case), that

\footnote{To our knowledge, there are no other published benchmarks for top-k algorithms in the context of Web Applications.}
ignores the early stop condition and continues the processing until the table is full. We compare below the performance of Algorithm $\text{TOP-K}$ and Algorithm $\text{WC}$.

We ran our experiments on a Lenovo T400 laptop, with Intel Core2 Duo P8600 processor and 2GB RAM. We ran two series of experiments. First, we used synthetic data to vary the main parameters that may affect the relative performance of the algorithm. Second, we used real data, in the context of the Yahoo! Shopping Web-site BP to evaluate how the performance compares in a real life setting.

**Experiments with Synthetic Data** We generated our synthetic input by varying a number of different parameters that affect the complexity of the $\text{TOP-K-FLOWS}$ problem. The ranges of parameter values were chosen based upon surveys on the structure of typical Web Applications [60, 82]; to examine the scalability of our techniques, we favored values that are on the higher end of the spectrum. In what follows, we describe the parameters and the chosen value ranges.

**BP Specifications size.** We varied the total number of activities in our BP specifications from 1000 to 40000; we note that [60] states that a typical number of activities in a given Web-site is 4000. To demonstrate scalability we have studied here BP specifications whose size is up to 10 times larger than that of a typical application.

**History Bound.** We considered bounded-history $cWeight$ functions with bounds ranging from 0 to 10. Note that a previous study [82] on the behavior of Web surfers concluded that a typical history bound is 4.

**Equivalence classes.** We have experimented with the BP specification size (number of activity nodes) as well as with the history bound $b$ used for the $cWeight$ function. Following the algorithm construction we found that the dominant factor dictating its performance is in fact the combination of the BP specification size and the history bound,
namely the number of *equivalence classes* of activity nodes, according to the following equivalence relation: two activity nodes within two partial EX-flows are said to be equivalent if they may be the root of exactly the same set of possible sub-flows, bearing exactly the same weight. We have experimented with BPs that have 1K-260K equivalence classes.

**Monotonicity “strength”**. We varied the percentage of *cWeights* that are equal to the neutral value of the aggregation function. This percentage determines how strongly/weakly monotone is the *fWeight* function, and in turn affects the algorithm performance. To further study the effect of monotonicity strength on the algorithm performance, we varied the *standard deviation* of the distribution of *cWeight* values (we considered uniform and normal distributions); high standard deviation implies greater monotonicity strength. We used addition and multiplication for aggregation functions, and since the choice of aggregation function did not affect the results we show below only the results for multiplication.

**Number of results** Finally, we varied the number *k* of requested results from 1 to 500.

**Additional Parameters**. As for the BP structure (i.e. number of activities in each implementation graph and number of possible implementations for each activity), we tried parameter values that are up to 5 times greater than observed in the real-life case of Yahoo! Shopping Web-site: we varied the number of implementation choices for each activity ranging from 2 to 1000, and the number of activities in each implementation from 100 to 1000. But given a fixed number of equivalence classes, we noted no significant effect of the BP structure on the algorithm performance, and consequently, we show the results for a fixed number (namely 50) of possible implementation choices (the number of activities in each implementation is dictated by the overall
BP size and the number of possible implementation choices).

A representative sample of the experimental results is presented below. Figure 6.3(a) examines the execution times (in seconds) of \textsc{Top-K} and \textsc{WC} for increasing number (in thousands) of equivalence classes. (The scale for the time axis in all graphs is logarithmic). Since our experiments showed that the shape of the BP graphs and the history bound do not affect the performance (given a fixed number of equivalence classes), we show here one representative sample where the history bound is 5. The number \( k \) of requested results here is 100. (We will consider varying \( k \) values below). The Figure shows the performance of \textsc{Top-K} for \( c\text{Weight} \) values in the range \([0,1]\) with different distributions. This includes uniform and normal distributions with average value of 0.5 and varying standard deviation of 0.2, 0.1, and 0 (the latter corresponding to all-equal \( c\text{Weight} \) values). \textsc{WC} always fills in all entries of the FTable, thus is not sensitive to the \( c\text{Weight} \) distribution, and we show only one curve for it. We can observe that \textsc{Top-K} generally shows 90-99\% improvement over \textsc{WC}, and that the greater the variance in \( c\text{Weight} \) values, the better the performance of \textsc{Top-K} is. This is because variance in \( c\text{Weights} \) implies variance in the \( \text{EX-flows} \) \( f\text{Weight} \), exploited by the greedy nature of \textsc{Top-K} which quickly separates the top-\( k \) results from the rest. As we shall see below, such variance of \( c\text{Weights} \) is indeed common in real-life BPs. In the extreme (unrealistic) case where all \( c\text{Weight} \) values are identical, i.e. standard deviation 0, the performance of \textsc{WC} and \textsc{Top-K} became the same (as the early stop condition does not hold, and the flows table must be fully filled), thus we show only the \textsc{WC} curve. Figure 6.3(b) examines the execution times of \textsc{WC} and \textsc{Top-K} for a growing number \( k \) of requested results (for the same distributions of \( c\text{Weights} \) as above). The number of equivalence classes here is 200K and the history bound is 5. We can see that the running time increases only moderately as \( k \) grows, with \textsc{Top-K} steadily showing significantly better performance than \textsc{WC} and
exhibiting similar behavior to what have been described above. (The increase for WC is less visible in the graph due to the logarithmic time scale). Figure 6.3(c) examines the effect of the monotonicity strength of the weight function, on the execution time of TOP-K. We fix $k$, the number of equivalence classes, and the history bound (to 100, 40K, and 5, resp.), and vary the percentage of neutral weights, with the non-neutral weights uniformly distributed. At the left-most end, there are no neutral weights and TOP-K performs significantly better than WC; at the right-most (very unlikely) case all weights are neutral, and TOP-K and WC exhibit the same execution times (as the flows table must be fully filled). We see that the performance of TOP-K is significantly superior in most cases. In particular, in all realistic scenarios where less than 90% of the weights are neutral, TOP-K improves over WC by more than 75%.

Experiments with real-life data Our second set of experiments considered the real-life BP used in the context of ShopIT, namely the BP modeling part of the Yahoo! Shopping Computer Store. Its specification consists of 5976 activities with an average of 2.6 implementation choices per compound activity and a history bound of 4, yielding approx. 840K equivalence classes. The variance in cWeight values (costs) for the choices in each compound activity (product type) is very high here. For example, the average RAM price is 192$, with a standard deviation of 510$. We considered increasingly large parts of the BP specification (corresponding to the outcome of evaluating decreasingly selective queries). Figure 6.3(d) depicts results for 15 representative such parts of the specification, involving increasing counts of equivalence classes the leading factor in the performance of the TOP-K algorithm. At the extreme right, all equivalence classes participate in the computation. Observe that in the context of this real-life data, TOP-K outperforms WC by a factor of over 98%, demonstrating scalability and
good performance.

### 6.4 Related Work

We conclude this chapter with a short review of related work. We have already mentioned popular shopping web-sites such as [99, 39]. Unlike ShopIT, their ranking mechanism ranks, separately, items in each distinct category, based on a built-in, specific, ranking metric. The global effect of a full navigation flow that may include, e.g., registration to customers clubs, collection of coupon discounts, specific user choices, is not accounted for.

A variety of Recommender Systems (e.g. [88, 95]) appear in the literature. However, as mentioned in [58], they provide rather low flexibility, with a recommendation method that is hard-wired and not configurable to fit user needs\(^2\). These works also typically do not support recommendations on multiple items. Successful commercial tools such as [77, 69] share similar characteristics and often specialize in specific domains, e.g. movies, music, restaurants. In contrast, we propose here a flexible generic approach that addresses the common, multi-item, shopping scenario and identifies navigation flows that best match the users criteria and preferences. The importance of customizable recommendation systems was recently recognized in [68], where such a flexible system was introduced in the context of relational data. As\(^2\) an exception are OLAP-based approaches that are still considered an open research problem.

---

\(^2\)An exception are OLAP-based approaches that are still considered an open research problem.
shown above, the (possibly recursive) semi-structured shape of Web Applications introduces unique challenges for top-k computation, that are not found in a relational environment.
Chapter 7

Conclusion and Future Work

We have developed in this thesis the foundations for an automated analysis of Web Applications and their underlying Business Processes. We first introduced models that allow to capture such processes and their executions, accounting for the uncertainty that rise in practical cases. We have then studied analysis of executions that occurred in the past as well as of future executions, under terms of uncertainty. We provided efficient analysis algorithms where applicable, and we have further exemplified the benefit gained by the results of such analysis to both the application owners as well as to its users; finally we presented a practical application of our results, realized via the ShopIT system.

There are still many challenges in the context of Web Applications analysis. We next list some of these challenges that are planned to be studied in future research.

- **Data Values.** Our research so far has mainly focused on the flow of the applications and their executions. The data manipulated by the application was only modeled at a high-level, via the notion of guarding formulas. We note in this context that the interplay of manipulated data and application execution flow has been studied in various contexts. Notably, [57] studied Business Process artifacts, relating to the data manipulated by such processes; [38]
studied a query language called “LTL-FO” for querying the application flow via a temporal logic and the manipulated data via First Order Logic, and have demonstrated its efficiency in real-life cases. It is an interesting challenge to incorporate such data analysis over our nested DAGs model of Business Processes, while finding a correct balance in terms of computational complexity and expressive power.

• **Richer Query Language.** Our query language supports the equivalent of several common features of query languages for Databases, such as selection and projection. It is interesting to study ways in which additional features such as value-based joins, aggregates, negation etc. may be incorporated within our query language while maintaining tractability of query evaluation. Initial attempts to introduce joins in a straightforward manner, for instance, led to undecidability of query evaluation.

• **Further Optimizations.** As performance is a key issue in a practical deployment of our algorithms, we are constantly studying optimizations that will speed-up our algorithms, as well as the identification of restricted tractable cases where our results are negative (e.g. for Type Checking).

• **Inference of BP specifications out of EX-traces.** There are cases where the structure of the BP specification and / or the weight function of interest are unknown to us, due to confidentiality, legacy applications etc. One possible solution is to *mine* the specification structure and the weight function out of a set of observed execution flows, and there indeed exist many such works in this and similar contexts (e.g. [89, 55, 96]). Incorporating such algorithms within our framework that further assumes only partial information regarding the execution traces themselves (partial tracing), is a future challenge.
Further Application Domains. Our analysis approach may be applied in various additional contexts, rather than the context of Web Applications studied here. For instance, the emerging Mashup platform (see e.g. [100]) allows users to combine various applications in a complex, portal-like manner, with the participating applications interacting in various ways. While this enables users to perform complicated tasks, it also renders the navigation within the Mashup intricate, and its analysis is required. We have studied ([29]) the analysis of Mashups, and evidently these intricate connections not only cause a challenge to users, but also render the application analysis much more complicated (NP-hard even for very simple cases). Nevertheless we have employed strong optimization techniques and developed a practical system called COMPASS, assisting the users of such Mashups. Interestingly, analysis of recursive processes is valuable in additional contexts. For instance, Data Cleaning may be represented as a gradual, probabilistic process, in which the database is cleaned tuple by tuple: at each step one possible correction to the Database is probabilistically chosen out of a given set of options. We have developed ([30]) a declarative language for the formulation and evaluation of such probabilistic and recursive rules, and we intend to apply this framework for Data Cleaning tasks. Another possible application domain is Active XML [5] mentioned above. As Active XML may be considered to be a special case of our model, with all implementation graphs being trees, we may adapt our (positive) results to perform analysis of Active XML documents, complementary of the thorough work (e.g. [2, 1, 3]) already done in this context.
Bibliography


174


