Labeling Recursive Workflow Executions On-the-Fly

Zhuowei Bao
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104, USA
zhuowei@cis.upenn.edu

Susan B. Davidson
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104, USA
susan@cis.upenn.edu

Tova Milo
School of Computer Science
Tel Aviv University
Tel Aviv, Israel
milo@post.tau.ac.il

ABSTRACT

This paper presents a compact labeling scheme for answering reachability queries over workflow executions. In contrast to previous work, our scheme allows nodes (processes and data) in the execution graph to be labeled *on-the-fly*, i.e., in a dynamic fashion. In this way, reachability queries can be answered as soon as the relevant data is produced. We first show that, in general, for workflows that contain recursion, dynamic labeling of executions requires long (linear-size) labels. Fortunately, most real-life scientific workflows are *linear recursive*, and for this natural class we show that dynamic, yet compact (logarithmic-size) labeling is possible. Moreover, our scheme labels the executions in linear time, and answers any reachability query in constant time. We also show that linear recursive workflows are, in some sense, the largest class of workflows that allow compact, dynamic labeling schemes. Interestingly, the empirical evaluation, performed over both real and synthetic workflows, shows that our proposed dynamic scheme *outperforms the state-of-the-art static scheme* for large executions, and creates labels that are shorter by a factor of almost 3.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*scientific databases*

General Terms

Algorithms, Performance, Theory

1. INTRODUCTION

Scientific workflow systems are now becoming "provenance aware" by automatically recording data and module dependency during execution (*e.g.*, Taverna [14], VisTrails [7] and Kepler [5]). By using such information, provenance queries such as "Was data item A (or Module M) used to produce data item B, either directly or indirectly?" are enabled. Answering such queries entails evaluating reachability queries over large, graph-structured data, which can be expensive [8].

Reachability labels are an important tool for efficiently processing reachability queries on large graphs. The main idea is to assign each vertex a label such that, using only the label of any two vertices, we can quickly decide if one can reach the other. However, the effectiveness of this approach crucially depends on the ability to develop *compact* and *efficient* labeling schemes that take small storage space and allow fast query processing. More precisely, we say that a labeling scheme is *compact* if it uses logarithmic-size labels ($O(\log n)$ bits for any graph with n vertices), and *efficient* if it answers any reachability query in constant time [1]. This is indeed the best one can hope for, as even assigning unique ids for n vertices requires labels of $\log n$ bits.

An important observation in the context of workflow systems is that the execution graph (or *run*) from which provenance information is obtained is not arbitrary, but is derived from a workflow *specification.* Workflow specifications are commonly modeled as directed graphs whose vertices denote modules and whose edges denote data flow; furthermore, they are typically fairly small graphs (10s of vertices). A specification can be executed many times, using different data inputs or parameter settings, and generating multiple runs. A run is modeled as a directed, acyclic graph (DAG) in which vertices represent module executions and whose edges carry the data output by the source and input by the sink. Workflow runs can be much larger (1000s of vertices) and structurally more complex than the specification due to repeated execution of sub-workflows, *e.g.*, sequentially (*loops*), in parallel (*forked* executions) or through *recursion.*

Much research has been devoted recently to develop compact and efficient labeling schemes for workflow runs [6, 13] and graphs in general [24, 16, 17, 15, 2, 9]. A significant shortcoming, however, of the existing schemes is that they all need to examine the entire graph before labeling is performed. This may not be realistic in our setting since scientific workflows can take a long time to execute and users may want to ask provenance queries over partial executions. Labeling must therefore be done on-the-fly. That is, we must label modules as soon as they are executed and data as soon as it is produced, and cannot modify the labels subsequently.

Our goal is thus to develop a *dynamic* labeling scheme for workflow runs. Dynamic labeling has been previously considered in the context of XML *trees* [10, 20, 23], but workflow

[1] We follow the standard assumption that any operation on two words ($\log n$ bits) can be done in constant time [6].

runs can have an arbitrarily more complex *DAG structure* [2]. Although there have been efficient dynamic algorithms [19, 11] for maintaining the transitive closure of DAGs, they all produce a linear-size index per vertex, which is unacceptable for large graphs. Nevertheless, we will show in this paper that the knowledge of the specification can be exploited to obtain a compact (logarithmic size) and efficient (constant query time) dynamic labeling scheme for runs.

We next give a brief summary of prior work on reachability labeling and highlight the contributions of this paper.

Prior Work. Reachability labeling has been studied for different classes of graphs in both static and dynamic settings. The main goal is to bound the maximum length of labels. Clearly, the more general the class of graphs is, the more difficult it is to obtain compact labeling schemes; dynamic labeling is also harder than static labeling. The maximum label lengths for different classes of graphs are summarized in Figure 1, and the main results of this paper are shaded.

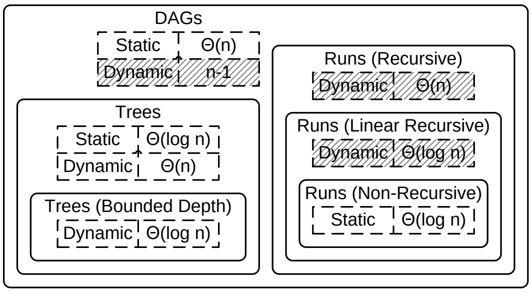

Figure 1: A Comparison of Maximum Label Length

Static. **(Trees)** The earliest work for labeling static trees [22] proposed an *interval-based* scheme that uses labels of $2\log n$ bits, where n is the number of nodes in the tree. Considerable effort [1, 4, 18, 12] was devoted to reduce the constant factor (2). The best known scheme [4] uses labels of $\log n + O(\sqrt{\log n})$ bits, which is still separated from the known lower bound of $\log n + \Omega(\log\log n)$ bits [3]. Motivated by the fact that XML trees are not deep, recent work [12] developed a scheme that uses labels of $\log n + 2\log d + O(1)$ bits, where d is the depth of the tree.

(Workflow Runs) Workflow runs are modeled as DAGs derived from a given specification. [6] proposed a compact static scheme for labeling runs that uses labels of $3\log n + O(1)$ bits. However, it can only be applied to non-recursive workflows (with only loops and forks). [13] also proposed a static scheme for labeling runs by transforming the graph into a tree and then applying the interval-based scheme. Since the size of the new tree can be exponential in the size of the original graph, it results in linear-size labels.

(General DAGs) In contrast to the above results, compact labeling is impossible for general directed acyclic graphs (DAGs), since a known lower bound on the maximum label length is $\Omega(n)$ bits. This triggered several alternative approaches for efficiently answering reachability queries over large DAGs: Chain Decomposition [15], Tree Cover [2] and 2-Hop [9]. Other recent work includes Path-Tree [17] and 3-Hop [16] that combine the previous three approaches, and GRAIL [24] that is based on randomized interval labeling.

[2] In particular, they are more general than series-parallel graphs.

Dynamic. **(Trees)** The dynamic problem is harder than the static case; it was shown in [10] that labeling dynamic trees requires labels of $\Omega(n)$ bits. [10] also proposed a *prefix-based* scheme, which provides a matching upper bound of $O(n)$ bits, and if the depth of the dynamic tree is bounded by a constant, it produces labels of $O(\log n)$ bits. Other variant prefix-based schemes with similar bounds were also studied. *e.g.*, ORDPATH [20], implemented in Microsoft SQL Server, supports frequent inserts in XML documents, and DDE [23] is tailored for both static and dynamic XML documents.

(Workflow Runs and General DAGs) To our knowledge, the present work is the first to study dynamic labeling of workflow runs (and more generally of DAGs). The main contributions of this paper are summarized as follows.

- We propose a formal model, based on graph grammars, that captures a rich class of workflows with recursion, loops and forks. Based on this model we define *execution-based* and *derivation-based* dynamic labeling problems for workflow runs (Section 2).
- To get a handle on the difficulty of the two problems, we first provide tight lower and upper bounds of $\Theta(n)$ bits on the maximum label length. As a side effect, we also give tight bounds of $n-1$ bits for the general problem of labeling dynamic DAGs (Section 3).
- Nevertheless, we identify a common class of workflows with linear recursion, and show that dynamic, yet compact ($O(\log n)$ bits) labeling is possible for linear recursive workflows. Moreover, our scheme labels a dynamic run in linear time, and answers any reachability query in constant time (Sections 4 and 5).
- We also show that linear recursive workflows are, in some sense, the largest class of workflows that allow compact dynamic labeling schemes (Section 6).
- Finally, we empirically evaluate the proposed dynamic labeling scheme over both real and synthetic workflows. Interestingly, our dynamic scheme creates even shorter labels for large runs than the state-of-the-art static scheme [6] by a factor of almost 3 (Section 7).

2. MODEL AND PROBLEM STATEMENT

We start with notations and basic definitions over graphs in Section 2.1. An informal description of our workflow model is given in Section 2.2, followed by a formalization based on graph grammars in Section 2.3. Finally, Section 2.4 formulates the dynamic workflow labeling problems.

2.1 Preliminaries

Throughout the paper, the term *graphs* refers to directed acyclic graphs with no self-loops or multi-edges. Every vertex of a graph can be associated with two kinds of labels. The one, given in the graph, denotes a module name, and the other, created by our algorithm, is used for answering reachability queries. To distinguish the two labels, we will refer to the former as vertex *name*, and the latter as *reachability label* or simply *label*. We denote by $\texttt{Name}(v)$ the name of a vertex v. Given two vertices v and v' of a graph g, let (v, v') denote an edge from v to v', and $v \leadsto_g v'$ denote that there is a path from v to v' in g. A graph g is said to be a *two-terminal graph* if it has a single source, denoted by $s(g)$,

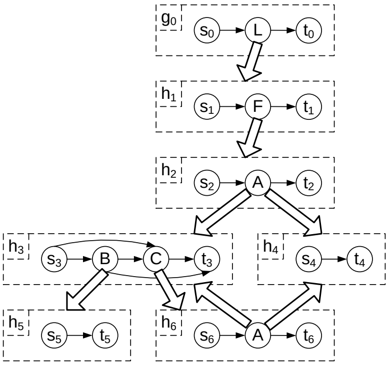

Figure 2: Workflow specification (running example)

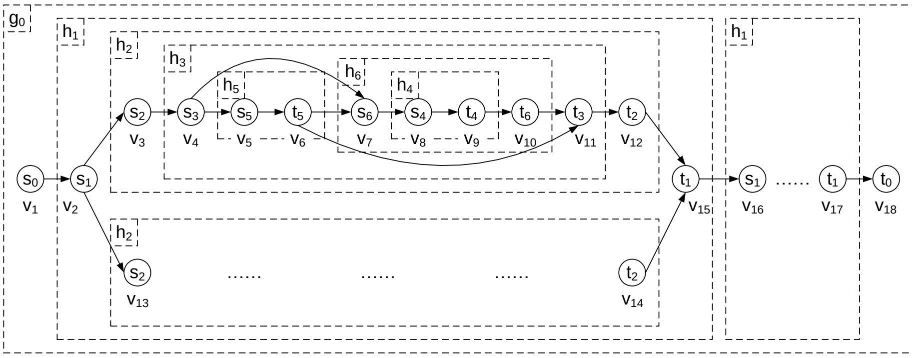

Figure 3: Workflow run (running example)

and a single target sink, denoted by $t(g)$. Given a finite set Σ of names, the set of all two-terminal graphs whose vertices are labeled by names chosen from Σ is denoted by $\mathcal{G}_\Sigma$.

Next, we introduce four graph operations, namely *series composition*, *parallel composition*, *vertex insertion* and *vertex replacement*. The first two operations are used to formalize loop and fork executions in Section 2.3. The last two operations are used to formalize execution-based and derivation-based dynamic workflow runs in Section 2.4.

Definition 1. A *series composition* of two-terminal graphs $g_1, g_2, \ldots, g_n$ forms a new two-terminal graph, denoted as $S(g_1, g_2, \ldots, g_n)$, by taking the union of their vertex sets and edges sets, and adding $(t(g_i), s(g_{i+1}))$ for all $1 \leq i \leq n-1$.

Definition 2. A *parallel composition* of two-terminal graphs $g_1, g_2, \ldots, g_n$ forms a new graph, denoted as $P(g_1, g_2, \ldots, g_n)$, by simply taking the union of their vertex sets and edge sets.

Definition 3. An *insertion* of a vertex v to a graph g, with respect to a subset C of vertices of g, forms a new graph, denoted as $g + (v, C)$, by adding v and (v', v) for all $v' \in C$.

Definition 4. A *replacement* of a vertex u of a graph g with another graph h forms a new graph, denoted as $g[u/h]$, by deleting u and all edges incident to u, and adding h and (v, s) for all predecessors v of u and all sources s of h and (t, v) for all successors v of u and all sinks t of h.

2.2 Workflow Model

Our workflow model has two components: *workflow specification* and *workflow run*. A workflow specification describes the design of a workflow, and a workflow run describes a particular execution of the given specification.

Workflow Specification. A workflow specification defines the control and data flow between a set of modules by means of a DAG. In this graph, each vertex represents a module, which takes a set of data items as input and produces a set of data items as output, and is labeled with a module name. Each directed edge represents the data flow between two modules (*i.e.*, data items that are produced by one module and consumed by the other). We also assume that each workflow has a single source (*i.e.*, with no incoming edges), which sends out all initial data and starts the execution, and a single sink (*i.e.*, with no outgoing edges), which collects all final results and stops the execution.

The modules are either *atomic* or *composite*. Atomic modules are treated as "black boxes", since their internal structure is hidden. In contrast, composite modules, treated as "white boxes", are known to be implemented by other sub-workflows. Intuitively, we can open a white box by replacing the composite module with the corresponding sub-workflow. Some composite modules are allowed to be repeatedly executed *in series* or *in parallel*. We call them *loop* and *fork* modules respectively. Note that a composite module can be implemented by a sub-workflow that contains other composite modules (including itself). This may lead to *recursion*.

Example 1. Our running example of a workflow specification is shown in Figure 2, where the uppercase letters (*i.e.*, L, F, A, B, C) are the names of composite modules, and the lowercase letters (*i.e.*, $s_0, \ldots, s_6, t_0, \ldots, t_6$) are the names of atomic modules. In particular, L and F are the names of loop and fork modules respectively; g_0 is a start graph. The thick arrows describe the possible implementations of each composite module (*e.g.*, A has two possible implementations h_3 and h_4). Also observe that A and C form a recursion.

Workflow Run. A workflow specification is repeatedly executed using different data input and parameter settings. A valid workflow run begins with the start graph, and selects one possible implementation to execute for each composite module (*"or" semantics*). For a loop or fork module, the selected implementation is repeatedly executed one or more times in series or in parallel, respectively. Moreover, it must execute all the modules in the start graph and the selected implementation graphs (*"and" semantics*). Since all composite modules are expanded during the execution, the resulting workflow run consists only of atomic modules.

Example 2. One possible run derived from the specification in Figure 2 is shown in Figure 3, where $v_1, \ldots, v_{18}$ are unique identifiers for atomic modules. In this run, h_1 (the implementation of a loop module) is replicated twice in series. In the first copy of h_1, h_2 (the implementation of a fork module) is replicated twice in parallel. For purposes of illustration, we show only the detailed execution for one copy of h_2. Observe that, due to the recursion over A and C, h_3 and h_6 may be repeatedly executed until h_4 is selected.

2.3 Workflow Grammar

We next present a formalization of our workflow model based on graph grammars. A graph grammar is similar in spirit to the well-known string grammars, such as context-free grammars. It defines a set of graph-based productions (*i.e.*, rules), and uses them to generate a set of graphs as its language. More precisely, we consider graph grammars based on vertex replacement, that is, every production defined by the grammar replaces a single vertex (*i.e.*, the head of the rule) with a graph (*i.e.*, the body of the rule). Our idea is to map every specification to a graph grammar in such a way that the set of possible runs, derived from this specification, corresponds to exactly its graph language. To capture loop and fork executions, our grammar may have an infinite (but controlled) number of productions.

Definition 5. A *workflow specification* is defined as a system $S = (\Sigma, \Delta, \Delta_{\mathcal{L}}, \Delta_{\mathcal{F}}, \mathcal{I}, g_0)$, where

- Σ is a finite nonempty set of names;
- Δ is a nonempty subset of Σ, called the set of *atomic names*, and $\Sigma \setminus \Delta$ is called the set of *composite names*;
- $\Delta_{\mathcal{L}}$ and $\Delta_{\mathcal{F}}$ are two disjoint subsets of $\Sigma \setminus \Delta$, called the sets of *loop names* and *fork names* respectively;
- $\mathcal{I}$ is a finite set of pairs (A, h), where $h \in \mathcal{G}_\Sigma$ is called an *implementation graph* of $A \in \Sigma \setminus \Delta$; and
- $g_0 \in \mathcal{G}_\Sigma$ is called a *start graph.*

A vertex labeled with an atomic name is said to be an *atomic vertex*. Similarly, we can define *composite vertex*, *loop vertex* and *fork vertex*. In the rest of this paper, we loosely follow the convention that v, v' and v_i are used for atomic vertices; and u, u' and u_i for composite vertices.

Example 3. The specification in Figure 2 is written as $(\Sigma, \Delta, \Delta_{\mathcal{L}}, \Delta_{\mathcal{F}}, \mathcal{I}, g_0)$, where $\Sigma = \{s_0, \ldots, s_6, t_0, \ldots, t_6, L, F, A, B, C\}$, $\Delta = \{s_0, \ldots, s_6, t_0, \ldots, t_6\}$, $\Delta_{\mathcal{L}} = \{L\}$, $\Delta_{\mathcal{F}} = \{F\}$ and $\mathcal{I} = \{(L, h_1), (F, h_2), (A, h_3), (A, h_4), (B, h_5), (C, h_6)\}$.

Definition 6. Given a workflow specification $S = (\Sigma, \Delta, \Delta_{\mathcal{L}}, \Delta_{\mathcal{F}}, \mathcal{I}, g_0)$, the *workflow grammar* of S is defined as a system $G = (\Sigma, \Delta, g_0, \mathcal{P})$, where Σ, Δ and g_0 are as in S, and $\mathcal{P}$ is the possibly infinite set of productions below.

$$\begin{aligned}\mathcal{P} = \{&A := h \mid (A, h) \in \mathcal{I}\} \\ \cup\, \{&A := S(\underbrace{h, h, \ldots, h}_{i\text{'s } h}) \mid (A, h) \in \mathcal{I}, A \in \Delta_{\mathcal{L}}, i > 1\} \\ \cup\, \{&A := P(\underbrace{h, h, \ldots, h}_{i\text{'s } h}) \mid (A, h) \in \mathcal{I}, A \in \Delta_{\mathcal{F}}, i > 1\}\end{aligned}$$

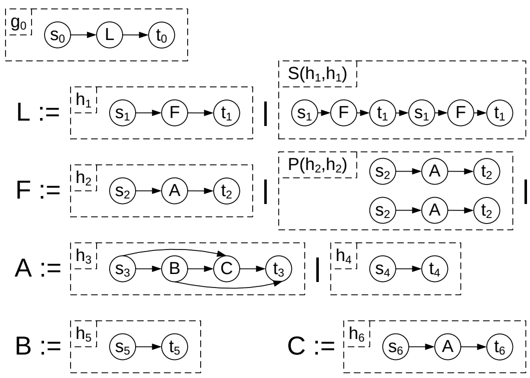

Figure 4: Workflow grammar (running example)

Example 4. The specification in Figure 2 is captured by a workflow grammar $(\Sigma, \Delta, g_0, \mathcal{P})$, where Σ and Δ are as in Example 3, and g_0 and $\mathcal{P}$ are shown in Figure 4.

Let $G = (\Sigma, \Delta, g_0, \mathcal{P})$ be a workflow grammar. We say that a graph g_2 *is directly derived from* a graph g_1 (with respect to G), denoted by $g_1 \Rightarrow_G g_2$, if there is a production $A := h \in \mathcal{P}$ such that $g_2 = g_1[u/h]$, where u is a composite vertex of g_1 with $\mathtt{Name}(u) = A$. Let $\Rightarrow_G^*$ be the reflexive and transitive closure of $\Rightarrow_G$, then g_2 *is derived from* g_1 (with respect to G), if $g_1 \Rightarrow_G^* g_2$. The *graph language* of G, denoted by $L(G)$, is defined as the set of all two-terminal graphs which can be derived from the start graph and consist only of atomic vertices. Formally,

$$L(G) = \{\ g \in \mathcal{G}_\Delta \mid g_0 \Rightarrow_G^* g\ \}$$

Definition 7. Given a workflow specification S, the set of *workflow runs*, with respect to S, is defined as $L(G)$, where G is the workflow grammar of S.

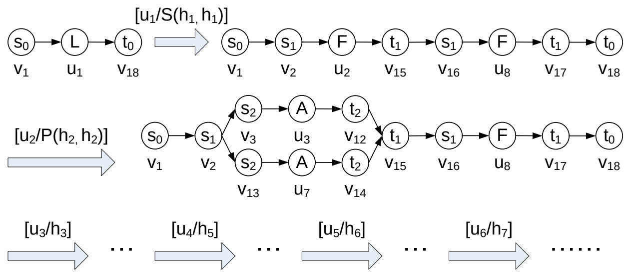

Figure 5: Graph derivation (running example)

Example 5. The workflow run in Figure 3 can be derived from the start graph using the workflow grammar in Figure 4. A graph derivation is sketched in Figure 5, where $u_1, \ldots, u_8$ are unique identifiers for composite vertices.

2.4 Dynamic Workflow Labeling Problems

The classical (static) graph reachability labeling problem is defined as follows. Given a graph g, assign each vertex of g a *reachability label* such that, using only the labels of any two vertices of g, we can decide if one can reach the other.

This paper studies the problem of labeling *dynamic* workflow runs. It differs from the above problem in two aspects. Firstly, the input graph is a workflow run derived from a given specification. Formally, $g \in L(G)$, where G is a given workflow grammar. Secondly, rather than taking the entire graph as input, we get a sequence of "updates" that leads to a graph $g \in L(G)$. We do not know the update sequence in advance, but receive them online. We must label all new vertices introduced by one update before the next update is applied, and cannot modify their reachability labels subsequently. Moreover, these labels can be used to determine reachability in any intermediate graph.

Based on different models of updates, we introduce two related dynamic workflow labeling problems. The first one describes the real life setting where run steps are reported and logged one by one, and the second one is used as an auxiliary tool for exploring the structure of workflow runs.

Execution-Based Problem. The first problem defines the update as a *vertex insertion*. Recall from Definition 3 that every insertion creates a new vertex along with a set of directed edges from existing vertices to this vertex. We begin with an empty graph $g_\emptyset$, and get a sequence of insertions that leads to a graph $g \in L(G)$, called a *graph execution.*

Definition 8. An *execution-based* dynamic reachability labeling scheme for a workflow grammar G is a pair (ϕ, π), where ϕ is a labeling function and π is a binary predicate. The input is an execution of a graph $g \in L(G)$, denoted by

$$g_\emptyset \overset{+(v_1,C_1)}{\Longrightarrow} g_1 \overset{+(v_2,C_2)}{\Longrightarrow} g_2 \overset{(v_3,C_3)}{\Longrightarrow} \dots \overset{+(v_n,C_n)}{\Longrightarrow} g_n = g$$

where $g_\emptyset$ is an empty graph and $g_i = g_{i-1} + (v_i, C_i)$, for all $1 \leq i \leq n$. In the ith step of the graph execution, ϕ assigns a reachability label $\phi(v_i)$ for the new vertex v_i. Note that by that time we can see only the first i insertions. ϕ and π are such that for any execution of a graph $g \in L(G)$, any intermediate graph g_i $(1 \leq i \leq n)$ and any two vertices v and v' of g_i, $\pi(\phi(v), \phi(v')) = \texttt{true}$ if and only if $v \rightsquigarrow_{g_i} v'$.

Derivation-Based Problem. The other problem defines the update as a *vertex replacement*. Recall from Definition 4 that every replacement substitutes an existing vertex for a new subgraph. We begin with the start graph g_0 (defined by the given workflow), and get a sequence of replacements that leads to a graph $g \in L(G)$, called a *graph derivation*.

Definition 9. A *derivation-based* dynamic reachability labeling scheme for a workflow grammar G is a pair (ϕ, π), where ϕ is a labeling function and π is a binary predicate. The input is a derivation of a graph $g \in L(G)$, denoted by

$$g_0 \overset{[u_1/h_1]}{\Longrightarrow} g_1 \overset{[u_2/h_2]}{\Longrightarrow} g_2 \overset{[u_3/h_3]}{\Longrightarrow} \dots \overset{[u_k/h_k]}{\Longrightarrow} g_k = g$$

where g_0 is the start graph and $g_i = g_{i-1}[u_i/h_i]$, for all $1 \leq i \leq k$. Initially, ϕ assigns a reachability label $\phi(v)$ for each vertex v of g_0. In the ith step of the graph derivation, ϕ assigns a reachability label $\phi(v)$ for each vertex v of h_i. Again, by that time we can see only the first i replacements. ϕ and π are such that for any derivation of a graph $g \in L(G)$, any intermediate graph g_i $(0 \leq i \leq k)$ and any two vertices v and v' of g_i, $\pi(\phi(v), \phi(v')) = \texttt{true}$ if and only if $v \rightsquigarrow_{g_i} v'$.

REMARK 1. *The derivation-based scheme labels not only atomic vertices but also composite vertices that appear during the graph derivation. However, to simplify the presentation, we will focus on the labels assigned to atomic vertices that remain in the final graph. Moreover, both insertion and replacement preserve the reachability between any pair of existing vertices. In fact, this is a necessary condition to allow persistent reachability labels. So, to prove correctness for both the execution-based and derivation-based schemes, we only need to ensure that for any two vertices v and v' of the final graph g, $\pi(\phi(v), \phi(v')) = \texttt{true}$ if and only if $v \rightsquigarrow_g v'$.*

At a first glance, the above two problems differ significantly from each other. The former receives and labels vertices one by one, while the latter by group. On the one hand, the execution-based model is more realistic, since it captures how runs advance; atomic modules of a workflow are executed in some topological ordering, due to data dependencies. On the other hand, the derivation-based model is more informative, since each step of a graph derivation describes exactly how a composite module is executed (*e.g.*, which sub-workflow is used or how many times a loop is repeated). However, our study in this paper reveals a tight relation between the two problems. We will show in Section 5.3 that a derivation-based scheme can be converted to an execution-based scheme, which creates the same reachability labels. A further study in Section 6 shows that in general, the execution-based problem allows shorter labels.

3. COMPACTNESS RESULTS

The effectiveness of reachability labeling crucially depends on the ability to design a compact labeling scheme that allows fast query processing. As mentioned before, a labeling scheme is said to be *compact* if it creates labels of $O(\log n)$ bits for any input graph with n vertices. Clearly, a compact labeling scheme creates the shortest possible labels up to a constant factor. In Section 5, we present a compact dynamic labeling scheme for a restricted class of workflows. Unfortunately, it is impossible to design a compact one for arbitrary workflows. In this section, we provide matching lower and upper bounds of $\Theta(n)$ bits on the maximum label length for both execution-based and derivation-based problems.

3.1 Lower Bounds

To establish the lower bounds, we first show in Theorem 1 that for some fixed workflow grammar, any possible dynamic labeling scheme requires linear-size reachability labels.

THEOREM 1. *There is a workflow grammar G such that for any execution-based (resp. derivation-based) dynamic labeling scheme (ϕ, π) for G, there is an execution (resp. derivation) of a graph $g \in L(G)$ with n vertices such that ϕ assigns a reachability label of $\Omega(n)$ bits for some vertex of g.*

PROOF. We first consider the execution-based problem. Let G be the workflow grammar shown in Figure 6, where A is a composite name and all the others are atomic names. Given an execution-based dynamic labeling scheme $D = (\phi, \pi)$ for G, for all $k \geq 0$, we define $L_k(G)$ to be the set of all graphs $g \in L(G)$ that are derived from g_0 by applying the production $A := h_1$ k times; and $S(D, k)$ to be the set of all reachability labels $\phi(v)$ that are assigned to a vertex v with $\texttt{Name}(v) = a$ of a graph $g \in L_k(G)$. Finally, $N(k)$ is the minimum of $|S(D, k)|$ over all possible schemes D.

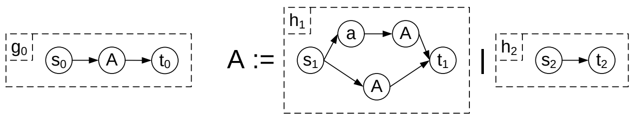

Figure 6: A workflow grammar that requires linear-size reachability labels (proof of Theorem 1).

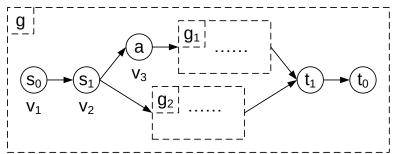

Figure 7: A graph $g \in L_{k+1}(G)$ that is derived from g_0 by applying the production $A := h_1$ $k+1$ times.

We first prove that $N(k+1) \geq 2N(k) + 1$, for all $k \geq 0$. Given a dynamic labeling scheme $D = (\phi, \pi)$ for G, the input is an execution of a graph $g \in L_{k+1}(G)$. Suppose that the first three vertices v_1, v_2, v_3 are already inserted to g, as shown in Figure 7. Consider the label domains that are reserved for two upcoming subgraphs g_1 and g_2 independently. We define S_1 and S_2 to be the sets of labels $\phi(v)$ that are reserved for all upcoming vertices v with $\texttt{Name}(v) = a$ of g_1 and g_2 respectively. Let $\phi(v_3) = l$, then $\forall l' \in S_1, \pi(l, l') = \texttt{true}$, but $\forall l' \in S_2, \pi(l, l') = \texttt{false}$. Thus, $S_1 \cap S_2 = \emptyset$ and $l \notin S_1 \cup S_2$. Since both g_1 and g_2 can be an arbitrary graph that is derived from A by applying the production $A := h_1$ k times, $|S(D, k+1)| \geq |S_1| + |S_2| + 1 \geq 2N(k)+1$. This holds for all possible schemes D, hence $N(k+1) \geq 2N(k) + 1$.

Since $N(0) = 0$ and $N(1) = 1$, we can prove by induction that $N(k) > 2^{k/2}$ for all $k \geq 2$. Therefore, we must assign a reachability label of at least $k/2$ bits for some vertex of a graph $g \in L_k(G)$. Finally, observe that g is derived from g_0 by applying the production $A := h_1$ k times and the other production $A := h_2$ $k+1$ times. Let n be the number of vertices of g, then $n = 3 + 4k + (k + 1) = 5k + 4$. So $k/2 = (n - 4)/10 = \Omega(n)$. The theorem follows.

We can use a similar proof for the derivation-based problem. The only modification is that, rather than inserting three vertices, we apply only one step of a graph derivation to obtain the intermediate graph shown in Figure 7. $\square$

3.2 Matching Upper Bounds

To match the above lower bounds, we first present a simple execution-based dynamic labeling scheme (ϕ, π), which uses linear-size reachability labels. Given an execution of a graph g with n vertices, let v_i be the ith vertex to be inserted, then $\phi(v_i)$ is a binary string of $i - 1$ bits. It simply encodes the reachability with respect to the previous $i - 1$ vertices already inserted to the graph. Formally, for all $1 \leq i \leq n$ and $1 \leq j \leq i - 1$, let $\phi(v_i)[j]$ be the jth bit of $\phi(v_i)$, then

$$\phi(v_i)[j] = \begin{cases} 1 & \text{if } v_j \rightsquigarrow_g v_i \\ 0 & \text{otherwise} \end{cases}$$

To decide if $v \rightsquigarrow_g v'$, we first compute the index of v and v' by the length of $\phi(v)$ and $\phi(v')$. Let $i = |\phi(v)| + 1$ and $i' = |\phi(v')| + 1$. Then $v \rightsquigarrow_g v'$ can be decided by

$$\pi(\phi(v), \phi(v')) = \begin{cases} \texttt{true} & \text{if } i < i' \text{ and } \phi(v')[i] = 1 \\ \texttt{false} & \text{otherwise} \end{cases}$$

The maximum length of labels used by this scheme is $n-1$ bits, which matches the lower bound of $\Omega(n)$ bits in Theorem 1. In fact, this scheme can be used to label executions of arbitrary DAGs. It was shown in [10] that even labeling dynamic trees with n nodes requires labels of $n-1$ bits. Hence, we provide as a side benefit tight lower and upper bounds of $n - 1$ bits on the maximum label length for the general problem of labeling (execution-based) dynamic DAGs.

However, the above execution-based scheme does not work for the derivation-based problem, because a graph derivation may not introduce new vertices in a topological ordering. In Section 5.2, we will present a compact derivation-based dynamic labeling scheme, which creates logarithmic-size reachability labels for a restricted class of workflows. If we use that scheme to label arbitrary workflows, it guarantees linear-size labels. Details are deferred to Section 6.

4. QUERYING DYNAMIC WORKFLOWS WITH LINEAR RECURSION

Linear-size reachability labels do not scale to large graphs. As demonstrated in the proof of Theorem 1, such large labels are required when workflows have unrestricted recursion. Luckily, workflows that one encounters in practice typically have a more restricted, linear form of recursion (to be formally defined below), which does allow for compact dynamic labeling. Indeed, we will see in Section 6 that the class of linear recursive workflows is the largest for which compact derivation-based dynamic labeling is possible.

The rest of this section is organized as follows. Section 4.1 defines the class of linear recursive workflows. To develop the labeling schemes, we first introduce a tree representation for linear recursive workflows, called the *explicit parse tree*, in Section 4.2, and then describe how to efficiently answer reachability queries using explicit parse trees in Section 4.3.

4.1 Linear Recursive Workflows

Let $G = (\Sigma, \Delta, g_0, \mathcal{P})$ be a workflow grammar. We say that a name A *directly induces* a name B (in G), denoted by $A \mapsto_G B$, if there is a production $A := h \in \mathcal{P}$ such that h has a vertex v with $\texttt{Name}(v) = B$. Let $\mapsto_G^*$ be the reflexive and transitive closure of $\mapsto_G$. We say that A *induces* B (in G), if $A \mapsto_G^* B$. Given a production $A := h \in \mathcal{P}$, a vertex u of h is said to be *recursive*, if $\texttt{Name}(u)$ induces A.

Example 6. Consider the workflow grammar in Figure 4. A directly induces B and C, due to the presence of $A := h_3$. Moreover, in this production, the vertex labeled with C is recursive, since C directly induces A by $C := h_6$.

Definition 10. A workflow grammar is said to be *linear recursive*, if any production has at most one recursive vertex.

Example 7. It can be verified that the workflow grammar in Figure 4 is linear recursive. Observe that $A := h_3$ has only one recursive vertex (labeled with C). In contrast, the workflow grammar in Figure 6 is not linear recursive, since $A := h_1$ has two recursive vertices (both labeled with A).

4.2 Explicit Parse Tree

We start by considering an arbitrary workflow grammar G. The derivation of a graph $g \in L(G)$ can be naturally captured by a *canonical parse tree* t, whose nodes represent nested subgraphs and edges represent composite vertices created during the graph derivation. The root of t corresponds to the start graph g_0. A subgraph h_1 is the parent of a subgraph h_2 if the graph derivation replaces a composite vertex v of h_1 with h_2, and the edge from h_1 to h_2 represents v.

Example 8. The canonical parse tree for the graph in Figure 3 is shown in Figure 8, whose nodes (denoted by dashed boxes) and edges (denoted by bold lines) are annotated with nested subgraphs and composite vertices that they represent. Note that $x_0', \ldots, x_8'$ are unique identifiers for nodes of the tree; $u_1, \ldots, u_8$ and $v_1, \ldots, v_{18}$ are unique identifiers for composite and atomic vertices of the graph respectively. Consider the edge (x_0', x_1') annotated with u_1. It implies that the production $L := S(h_1, h_1)$ is applied to replace the loop vertex u_1 of g_0 with a series composition of two h_1's.

For linear recursive workflow grammars, we can convert a canonical parse tree to an *explicit parse tree* by inserting three kinds of special nodes: $\mathcal{L}$ (loop) nodes, $\mathcal{F}$ (fork) nodes and $\mathcal{R}$ (recursive) nodes. The children of an $\mathcal{L}$ or an $\mathcal{F}$ node represent one or more copies of the same loop or fork subgraph, which are combined in series or in parallel respectively; and the children of a $\mathcal{R}$ node represent a sequence of nested subgraphs, which form a linear recursion.

Example 9. The explicit parse tree for the graph in Figure 3 is shown in Figure 9, where $x_0, \ldots, x_{13}$ are unique

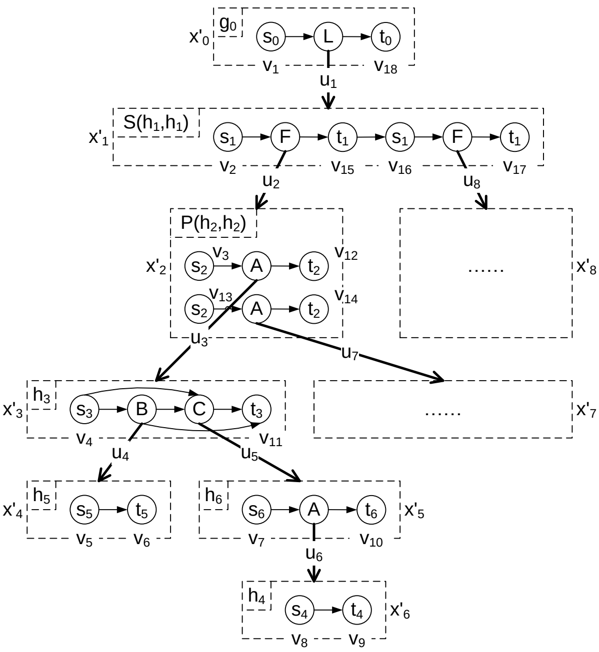

Figure 8: Canonical parse tree (running example)

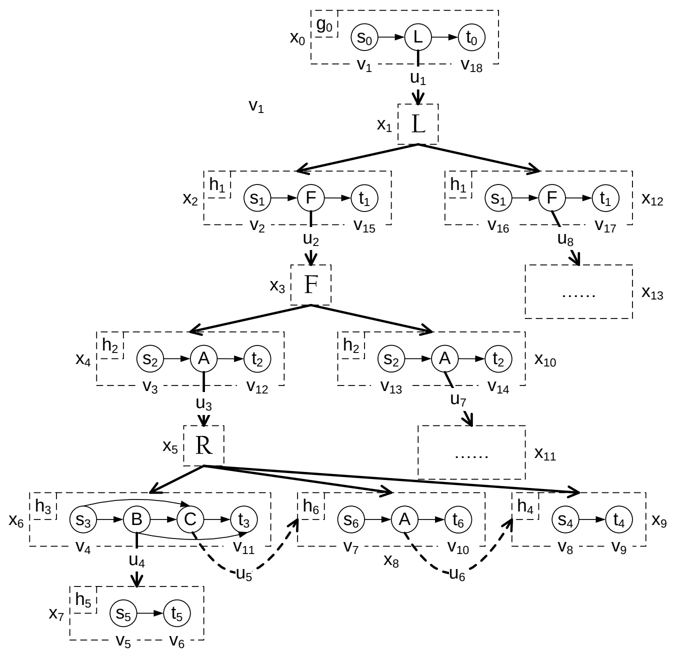

Figure 9: Explicit parse tree (running example)

identifiers for nodes of the new tree. Comparing with the canonical parse tree in Figure 8, we can see that x'_1 is split into two nodes x_2 and x_{12} rooted at a special $\mathcal{L}$ node x_1. Moreover, x'_3, x'_5, x'_6 are moved to be the children x_6, x_8, x_9 of a special $\mathcal{R}$ node x_5. Note that x_6, x_8, x_9 are linked by dashed edges annotated with recursive vertices u_5 and u_6.

It is important to note that the canonical parse tree for graphs generated by a fixed grammar may have unbounded depth due to recursion. However, in the explicit parse tree, the sequence of nested subgraphs in a linear recursion are flattened to be the children of a $\mathcal{R}$ node. Hence, for linear recursive grammars, the depth of the explicit parse tree is bounded by a constant that depends only on the grammar.

LEMMA 4.1. *Given a linear recursive workflow grammar $G = (\Sigma, \Delta, g_0, \mathcal{P})$, let t be an explicit parse tree for a graph $g \in L(G)$ and d be the depth of t, then $d \leq 2|\Sigma \setminus \Delta|$.*

PROOF. Let r be the root of t. Consider a leaf node x at the deepest level of t. Let k_1 and k_2 be the number of special and non-special nodes on the path from r to x respectively. Since the parent of a special node must be a non-special node, and both r and x are non-special nodes, we have $k_1 \leq k_2 - 1$. Hence, $d = k_1 + k_2 - 1 \leq 2k_2 - 2$. On the other hand, since each outgoing edge of a non-special node is annotated with a composite vertex, there is totally $k_2 - 1$ vertices annotated on the path from r to x. Moreover, these vertices must have distinct composite names, since all recursive vertices are annotated on dashed edges. Hence, $k_2 - 1 \leq |\Sigma \setminus \Delta|$. It follows that $d \leq 2k_2 - 2 \leq 2|\Sigma \setminus \Delta|$. $\square$

4.3 Answering Reachability Queries

Let t be an explicit parse tree for a graph g. To avoid confusion, we use x and y to refer to nodes of t and u and v to vertices of g. Note that in this section, we abuse the notation slightly by using u and v to refer to both composite and atomic vertices of g. $\texttt{Annt}(x)$ denotes the subgraph annotated on a non-special node x, and $\texttt{Annt}(x, y)$ denotes the composite vertex annotated on an edge (x, y). Recall that a graph g_2 is said to be derived from a graph g_1 if g_2 can be obtained from g_1 by applying a sequence of productions. We extend this notion to vertices as follows: A vertex v *is directly derived from* a vertex u, denoted by $u \Rightarrow v$, if a production $\texttt{Name}(u) := h$ is applied to replace u with a graph h, and v is a vertex of h. Let $\Rightarrow^*$ be the reflexive and transitive closure of $\Rightarrow$, then v *is derived from* u, if $u \Rightarrow^* v$.

To efficiently answer reachability queries using explicit parse trees, we introduce the notions of *context* and *origin*.

Definition 11. A non-special node x of t is said to be the *context* of a vertex v of g, if v is a vertex of $\texttt{Annt}(x)$.

Definition 12. A vertex u of g is said to be the *origin* of a vertex v of g *with respect to* a non-special node y of t, if v is derived from u, and y is the context of u.

Note that the context and origin are always unique and can be defined for both atomic and composite vertices.

Example 10. Consider the explicit parse tree in Figure 9. The context of v_5 (bottom left) is x_7, since v_5 is an atomic vertex of $\texttt{Annt}(x_7)$. The origin of v_5 with respect to x_2 (top left) is u_2, since u_2 is a composite vertex of $\texttt{Annt}(x_2)$ from which v_5 is derived. Similarly, the origin of v_8 (bottom right) with respect to x_6 (bottom left) is u_5 (on the dashed edge).

The main idea is as follows. To decide if v can reach v' in g, we find the *least common ancestor* of their context x and x' in t, denoted by $\texttt{LCA}(x, x')$. If $\texttt{LCA}(x, x')$ is a special $\mathcal{L}$ or $\mathcal{F}$ node, we can immediately answer "yes" or "no" by showing that v and v' belong to two distinct copies of the same loop (reachable) or fork (unreachable); otherwise (if $\texttt{LCA}(x, x')$ is a special $\mathcal{R}$ node or a non-special node), we show that the original query for v and v' over g can be reduced to a simple query for their origins u and u' with respect to a small subgraph h. The details are given in Lemma 4.2.

Lemma 4.2. *Let t be an explicit parse tree for a graph g. Given any two vertices v and v' of g, let x (resp. x') be the context of v (resp. v') in t, then*

- *if $\texttt{LCA}(x, x')$ is a special node, let y (resp. y') be a child of $\texttt{LCA}(x, x')$ who is an ancestor of x (resp. x'). Assume w.l.o.g. that y is on the left of y'.*
  - *if $\texttt{LCA}(x, x')$ is an $\mathcal{L}$ node, then $v \rightsquigarrow_g v'$;*
  - *if $\texttt{LCA}(x, x')$ is an $\mathcal{F}$ node, then $v \not\rightsquigarrow_g v'$, $v' \not\rightsquigarrow_g v$;*
  - *if $\texttt{LCA}(x, x')$ is a $\mathcal{R}$ node, let u (resp. u') be the origin of v (resp. v') with respect to y (note that $u' = \texttt{Annt}(y, z)$, where z is the right sibling of y), and $h = \texttt{Annt}(y)$, then $v \rightsquigarrow_g v'$ iff $u \rightsquigarrow_h u'$.*
- *if $\texttt{LCA}(x, x')$ is a non-special node, let u (resp. u') be the origin of v (resp. v') with respect to $\texttt{LCA}(x, x')$, and $h = \texttt{Annt}(\texttt{LCA}(x, x'))$, then $v \rightsquigarrow_g v'$ iff $u \rightsquigarrow_h u'$.*

Proof. First of all, we claim the following lemma, which easily follows from Definition 4. The proof is omitted.

Lemma 4.3. *Suppose that a graph g_2 is derived from a graph g_1. Let v and v' be two vertices of g_2, and u and u' be two vertices of g_1, such that v (resp. v') is derived from u (resp. u'). Then $v \rightsquigarrow_{g_2} v'$ if and only if $u \rightsquigarrow_{g_1} u'$.*

We prove Lemma 4.2 by four cases. (1) If $\texttt{LCA}(x, x')$ is an $\mathcal{L}$ node, let $y_1, y_2, \ldots, y_k$ be all children of $\texttt{LCA}(x, x')$ (including y and y'), and $h = S(\texttt{Annt}(y_1), \texttt{Annt}(y_2), \ldots, \texttt{Annt}(y_k))$. Let u (resp. u') be the origin of v (resp. v') with respect to y (resp. y'). Since y is on the left of y', by Definition 1, $u \rightsquigarrow_h u'$. By Lemma 4.3, $v \rightsquigarrow_g v'$; (2) If $\texttt{LCA}(x, x')$ is an $\mathcal{F}$ node, the lemma can be proved similarly by Definition 2; (3) If $\texttt{LCA}(x, x')$ is a $\mathcal{R}$ node, let $u' = \texttt{Annt}(y, z)$, where z is the right sibling of y, and let w be the origin of v' with respect to y'. Since y is on the left of y', w is derived from u'. Since v' is derived from w, v' is also derived from u'. Hence, u' is the origin of v' with respect to y. By Lemma 4.3, $v \rightsquigarrow_g v'$ if and only if $u \rightsquigarrow_h u'$; and (4) If $\texttt{LCA}(x, x')$ is a non-special node, by Lemma 4.3, $v \rightsquigarrow_g v'$ if and only if $u \rightsquigarrow_h u'$. □

Example 11. We demonstrate the above four rules using the running example. First, consider v_5 and v_{16} (top right) in Figure 9. The least common ancestor of their context x_7 and x_{12} is an $\mathcal{L}$ node x_1. By Lemma 4.2, $v_5 \rightsquigarrow_g v_{16}$, which is confirmed by Figure 3. Similarly, consider v_5 and v_{13} (middle right). The least common ancestor of their context x_7 and x_{10} is an $\mathcal{F}$ node x_3. Hence, $v_5 \not\rightsquigarrow_g v_{13}$ and $v_{13} \not\rightsquigarrow_g v_5$. Next, consider v_5 and v_8. The least common ancestor of their context x_7 and x_9 is a $\mathcal{R}$ node x_5 (note that the dashed edges are ignored). Moreover, u_4 and u_5 are the origin of v_5 and v_8 with respect to x_6 (note that u_5 is annotated on the dashed edge (x_6, x_8)). Since $u_4 \rightsquigarrow_{h_3} u_5$, by Lemma 4.2, $v_5 \rightsquigarrow_g v_8$. Finally, consider v_5 and v_{11} (bottom left). The least common ancestor of their context x_7 and x_6 is a non-special node x_6. u_4 and v_{11} are the origin of v_5 and v_{11} with respect to x_6. Hence, $u_4 \rightsquigarrow_{h_3} v_{11}$ implies that $v_5 \rightsquigarrow_g v_{11}$.

5. LABELING DYNAMIC WORKFLOWS WITH LINEAR RECURSION

Our dynamic schemes are built upon a skeleton-based labeling framework [6]. As a preprocessing step, we label the workflow specification using any static reachability scheme, and then extend the reachability labels on the specification, called the *skeleton labels*, to label workflow runs on-the-fly.

We start by discussing how to label the specification in Section 5.1. Section 5.2 presents a compact derivation-based dynamic labeling scheme for linear recursive workflows; we sketch how to adapt it to an execution-based scheme in Section 5.3. Finally, Section 5.4 proves the correctness and analyze the quality of our proposed dynamic schemes.

5.1 Labeling Workflow Specifications

Given a workflow specification $S = (\Sigma, \Delta, \Delta_{\mathcal{L}}, \Delta_{\mathcal{F}}, \mathcal{I}, g_0)$, we want to label the start graph and all implementation graphs in S. Formally, the set of graphs to be labeled is

$$\mathcal{G}(S) = \{g_0\} \cup \{h \mid (A, h) \in \mathcal{I}\}$$

It is important to note that all graphs in $\mathcal{G}(S)$ are small compared with runs derived from the specification. In practice, the largest real-life workflow that we have collected has fewer than 30 vertices, while a realistic run may repeatedly execute a loop or fork module (sub-workflow) hundreds of times. Therefore, we claim that any static scheme is scalable to label the specification. Our experiments in Section 7 show that even linear-size skeleton labels, created by the scheme described in Section 3.2, take negligible storage overhead.

5.2 Derivation-Based Dynamic Scheme

Given a labeled specification, we next explain the design of reachability labels for its runs. Let G be a linear recursive workflow grammar, and t be an explicit parse tree for a graph $g \in L(G)$. Recall from Lemma 4.2 that to decide if v can reach v' in g we only need to (1) find the least common ancestor $\texttt{LCA}(x, x')$ of their context x and x' in t; and (2) (if $\texttt{LCA}(x, x')$ is a special $\mathcal{R}$ node or a non-special node) answer an equivalent query for their origin u and u' with respect to a small subgraph h. To encode Step (1), we use a prefix-based scheme [18] [3] to label t. To encode Step (2), we enrich a prefix-based label with skeleton labels as well as other necessary information (*e.g.*, node types).

The formal description of a reachability label is given below. We use (ϕ_G, π_G) to denote the static labeling scheme for the specification, and (ϕ_g, π_g) to denote our proposed dynamic labeling scheme for runs. Recall that ϕ_G and ϕ_g are labeling functions, and π_G and π_g are reachability predicates. To label a vertex v of g, we consider a path in t

$$x_0 \xrightarrow{u_0} x_1 \xrightarrow{u_1} x_2 \xrightarrow{u_2} \ldots \xrightarrow{u_{k-1}} x_k$$

where x_0 is the root of t, x_k is the context of v, and for all $0 \le i \le k-1$, x_i is the parent of x_{i+1} and $u_i = \texttt{Annt}(x_i, x_{i+1})$ is the composite vertex annotated on the edge (x_i, x_{i+1}). Note that $u_i = \texttt{null}$ if x_i is a special node, otherwise, u_i is the origin of v with respect to x_i. To unify the notation, let $u_k = v$. Then $\phi_g(v)$ consists of a list of entries

$$\phi_g(v) = \{\texttt{Entry}(x_0, u_0), \texttt{Entry}(x_1, u_1), \ldots, \texttt{Entry}(x_k, u_k)\}$$

where $\texttt{Entry}(x_i, u_i) = (index, type, skl, rec_1, rec_2)$ is a tuple obtained from a pair (x_i, u_i) by Algorithm 1.

The details of Algorithm 1 are explained as follows. The *index* of x is a positive integer i if x is the ith child of its parent (Line 1). Note that the index of the root of t is set

[3]In a prefixed-based scheme, every node of a tree is assigned an index i, if it is the ith child of its parent. The prefix-based label for a node consists of the indexes of all its ancestors.

Algorithm 1 *Entry Construction*

Input: (x, u) is a pair of a node of t and a vertex of g
(ϕ_G, π_G) is a static scheme for labeling specification
Output: $\texttt{Entry}(x, u)$ is an entry for (x, u)

1: $index \leftarrow$ the index of x
2: $type \leftarrow$ the type of x
3: $skl, rec_1, rec_2 \leftarrow \texttt{null}$
4: **if** x is a non-special node **then**
5: $\quad skl \leftarrow \phi_G(u)$
6: $\quad$ **if** $\texttt{Annt}(x)$ has one recursive vertex **then**
7: $\quad\quad$ /* the parent of x must be a special $\mathcal{R}$ node */
8: $\quad\quad w \leftarrow$ the recursive vertex of $\texttt{Annt}(x)$
9: $\quad\quad rec_1 \leftarrow \pi_G(\phi_G(u), \phi_G(w))$
10: $\quad\quad rec_2 \leftarrow \pi_G(\phi_G(w), \phi_G(u))$
11: $\quad$ **end if**
12: **end if**
13: **return** $(index, type, skl, rec_1, rec_2)$

to zero. The *type* of x is either $\mathcal{L}$ (loop) or $\mathcal{F}$ (fork) or $\mathcal{R}$ (recursive) or $\mathcal{N}$ (non-special) (Line 2). If x is a non-special node, then $\texttt{Annt}(x)$ is already labeled by (ϕ_G, π_G). So the *skeleton label* assigned to u is given by $\phi_G(u)$ (Line 5) [4]. Finally, if $\texttt{Annt}(x)$ has one recursive vertex (note that the parent of x must be a special $\mathcal{R}$ node), then the *recursion flags* for u are two booleans, indicating if u can reach the recursive vertex w in $\texttt{Annt}(x)$ or vice versa. Note that they are computed by comparing the skeleton labels (Line 9 and 10). For other cases, skl, rec_1 and rec_2 are set to $\texttt{null}$.

Example 12. We label the running example using the explicit parse tree in Figure 9. For example,

$$\begin{aligned}\phi_g(v_5) = \{&\texttt{Entry}(x_0, u_1), \texttt{Entry}(x_1, \texttt{null}), \texttt{Entry}(x_2, u_2),\\ &\texttt{Entry}(x_3, \texttt{null}), \texttt{Entry}(x_4, u_3), \texttt{Entry}(x_5, \texttt{null}),\\ &\texttt{Entry}(x_6, u_4), \texttt{Entry}(x_7, v_5)\}\end{aligned}$$

where

$$\begin{aligned}&\texttt{Entry}(x_0, u_1) = (0, \mathcal{N}, \phi_G(u_1), \texttt{null}, \texttt{null})\\ &\texttt{Entry}(x_1, \texttt{null}) = (1, \mathcal{L}, \texttt{null}, \texttt{null}, \texttt{null})\\ &\ldots\ldots\\ &\texttt{Entry}(x_6, u_4) = (1, \mathcal{N}, \phi_G(u_4), \texttt{true}, \texttt{false})\\ &\texttt{Entry}(x_7, v_5) = (1, \mathcal{N}, \phi_G(v_5), \texttt{null}, \texttt{null})\end{aligned}$$

Since u_5 is the recursive vertex of h_3, by Algorithm 1,

$$\begin{aligned}\texttt{Entry}(x_6, u_4).rec_1 = \pi_G(\phi_G(u_4), \phi_G(u_5)) = \texttt{true}\\ \texttt{Entry}(x_6, u_4).rec_2 = \pi_G(\phi_G(u_5), \phi_G(u_4)) = \texttt{false}\end{aligned}$$

Similarly,

$$\phi_g(v_{16}) = \{\texttt{Entry}(x_0, u_1), \texttt{Entry}(x_1, \texttt{null}), \texttt{Entry}(x_{12}, v_{16})\}$$

where the first two entries are defined above, and

$$\texttt{Entry}(x_{12}, v_{16}) = (2, \mathcal{N}, \phi_G(v_{16}), \texttt{null}, \texttt{null})$$

The dynamic labeling algorithm ϕ_g can be divided into two interleaved steps. First, we generate the explicit parse tree in a top-down fashion by Algorithm 2. During this process, we also label all new vertices introduced in each step by Algorithm 3. Details are explained as follows.

[4]Since the skeleton labels are shared by multiple runs, *skl* is implemented as a pointer to the label, rather than the label itself.

Algorithm 2 *Dynamic Generation of Explicit Parse Tree*

Input: $g_0 \overset{[u_1/h_1]}{\Longrightarrow} g_1 \overset{[u_2/h_2]}{\Longrightarrow} g_2 \overset{[u_3/h_3]}{\Longrightarrow} \ldots \overset{[u_k/h_k]}{\Longrightarrow} g_k = g$
is a derivation of a graph $g \in L(G)$
Output: t is an explicit parse tree for g

1: Create a node r annotated with g_0
2: Insert r as the root of t
3: **for all** $i := 1$ to k **do**
4: $\quad y \leftarrow$ the context of u_i in t
5: $\quad$ **if** u_i is not recursive **then**
6: $\quad\quad$ **if** $\texttt{Name}(u_i)$ is a loop or a fork name **then**
7: $\quad\quad\quad$ /* h_i has no recursive vertices */
8: $\quad\quad\quad$ Let $h_i = S(\underbrace{h, h, \ldots, h}_{l\text{'s } h})$ or $P(\underbrace{h, h, \ldots, h}_{l\text{'s } h})$
9: $\quad\quad\quad$ Create a special $\mathcal{L}$ or $\mathcal{F}$ node x
10: $\quad\quad\quad$ **for all** $j := 1$ to l **do**
11: $\quad\quad\quad\quad$ Create a node x_j annotated with h
12: $\quad\quad\quad\quad$ Insert x_j as the jth child of x
13: $\quad\quad\quad$ **end for**
14: $\quad\quad$ **else**
15: $\quad\quad\quad$ **if** h_i has one recursive vertex **then**
16: $\quad\quad\quad\quad$ Create a special $\mathcal{R}$ node x
17: $\quad\quad\quad\quad$ Create a node x' annotated with h_i
18: $\quad\quad\quad\quad$ Insert x' as the single child of x
19: $\quad\quad\quad$ **else**
20: $\quad\quad\quad\quad$ Create a node x annotated with h_i
21: $\quad\quad\quad$ **end if**
22: $\quad\quad$ **end if**
23: $\quad\quad$ Insert x as the next child of y
24: $\quad\quad$ Annotate the edge (y, x) with u_i
25: $\quad$ **else**
26: $\quad\quad$ Create a node x annotated with h_i
27: $\quad\quad$ /* the parent of y must be a special $\mathcal{R}$ node */
28: $\quad\quad$ Insert x as the right sibling of y
29: $\quad\quad$ Create a dashed edge (y, x) annotated with u_i
30: $\quad$ **end if**
31: **end for**
32: **return** t

Algorithm 2: We begin with the start graph g_0, and get as input a derivation of a graph $g \in L(G)$. Initially, we create a node r annotated with g_0 as the root of t (Line 1 to 2). Let $g_i = g_{i-1}[u_i/h_i]$ be the ith step of the graph derivation. We update t in two steps. In Step (1), we create a subtree rooted at a new node x that corresponds to h_i. Consider three disjoint cases. (1a) If $\texttt{Name}(u_i)$ is a loop or a fork name (note that u_i is not recursive and h_i has no recursive vertices), let h_i be the series or parallel composition of l copies of a loop or fork subgraph h, we create a special $\mathcal{L}$ or $\mathcal{F}$ node x with l children annotated with h (Line 9 to 13); (1b) If u_i is not recursive but h_i has one recursive vertex, we create a special $\mathcal{R}$ node x with a single child annotated with h_i (Line 16 to 18); and (1c) otherwise, we simply create a new node x annotated with h_i (Line 20 and 26). In Step (2), we insert x to t. Let y be an existing node of t whose annotated graph contains u_i (Line 4; y is defined to be the *context* of u_i in Section 4.3). Again, consider two cases. (2a) If u_i is not recursive, we insert x as the next child of y, and annotate the edge (y, x) with u_i (Line 23 to 24); and (2b) otherwise (note that the parent of y must be a special $\mathcal{R}$ node), we insert x as the right sibling of y, and create a dashed edge (y, x) annotated with u_i (Line 28 to 29).

The following lemma ensures that the three cases (1a), (1b) and (1c) in Algorithm 2 are indeed disjoint.

LEMMA 5.1. *Let $G = (\Sigma, \Delta, g_0, \mathcal{P})$ be a linear recursive workflow grammar. If a production $A := h \in \mathcal{P}$ has a recursive vertex u in h, then (1) A is not a loop or a fork name; and (2) $\texttt{Name}(u)$ is not a loop or a fork name.*

PROOF. We first prove the part (1). Suppose A is a loop or a fork name, by Definition 6, either $A := S(h, h) \in \mathcal{P}$ or $A := P(h, h) \in \mathcal{P}$. But both productions have at least two recursive vertices, which contradicts Definition 10.

We next prove the part (2). Suppose $\texttt{Name}(u)$ is a loop or a fork name. Consider two cases. (a) If $\texttt{Name}(u) = A$, then A is a loop or a fork name, which contradicts (1). (b) If $\texttt{Name}(u) \neq A$, since u is recursive, $\texttt{Name}(u)$ induces A. So there is a production $\texttt{Name}(u) := h' \in \mathcal{P}$ such that for some vertex u' of h', $\texttt{Name}(u')$ induces A. Since A directly induces $\texttt{Name}(u)$, $\texttt{Name}(u')$ induces $\texttt{Name}(u)$. Hence, $\texttt{Name}(u) := h'$ has a recursive vertex u', which again contradicts (1). $\square$

Algorithm 3: During the dynamic generation of the explicit parse tree (Algorithm 2), we also perform the following labeling. For a non-special node x, we create a label $\phi_g(v)$ for each vertex v of $\texttt{Annt}(x)$. For a special node x, we also create a temporary label. By abusing the notation, we denote this label by $\phi_g(x)$. Note that to obtain a new label for a node x, we take an existing label from its parent y, and append only one new entry built by Algorithm 1 (Line 13, 16, 21).

Algorithm 3 *Labeling Function ϕ_g*

Input: A derivation of a graph $g \in L(G)$
(ϕ_G, π_G) is a static scheme for G
Output: t is an explicit parse tree for g
ϕ_g is a labeling function for g

1: Initially, create a root r of t (by Algorithm 2)
2: **for** each vertex v of $\texttt{Annt}(r)$ **do**
3: $\quad\phi_g(v) \leftarrow \{\texttt{Entry}(r, v)\}$
4: **end for**
5: **for** each derivation step **do**
6: $\quad$Update t top-down (by Algorithm 2)
7: $\quad$**for** each newly inserted node x **do**
8: $\quad\quad y \leftarrow$ the parent of x
9: $\quad\quad$**if** y is a non-special node **then**
10: $\quad\quad\quad u \leftarrow \texttt{Annt}(y, x)$
11: $\quad\quad\quad$**if** x is a non-special node **then**
12: $\quad\quad\quad\quad$**for** each vertex v of $\texttt{Annt}(x)$ **do**
13: $\quad\quad\quad\quad\quad\phi_g(v) \leftarrow$ append $\texttt{Entry}(x, v)$ to $\phi_g(u)$
14: $\quad\quad\quad\quad$**end for**
15: $\quad\quad\quad$**else**
16: $\quad\quad\quad\quad\phi_g(x) \leftarrow$ append $\texttt{Entry}(x, \texttt{null})$ to $\phi_g(u)$
17: $\quad\quad\quad$**end if**
18: $\quad\quad$**else**
19: $\quad\quad\quad$/* x must be a non-special node */
20: $\quad\quad\quad$**for** each vertex v of $\texttt{Annt}(x)$ **do**
21: $\quad\quad\quad\quad\phi_g(v) \leftarrow$ append $\texttt{Entry}(x, v)$ to $\phi_g(y)$
22: $\quad\quad\quad$**end for**
23: $\quad\quad$**end if**
24: $\quad$**end for**
25: **end for**
26: **return** t, ϕ_g

Algorithm 4 *Binary Predicate π_g*

Input: $\phi_g(v)$ is a reachability label for v
$\phi_g(v')$ is a reachability label for v'
π_G is a binary predicate for skeleton labels
Output: $\pi_g(\phi_g(v), \phi_g(v')) = \texttt{true}$ iff $v \rightsquigarrow_g v'$

$i \leftarrow \min\{j \mid \phi_g(v)[j].index = \phi_g(v')[j].index$ and $\phi_g(v)[j+1].index \neq \phi_g(v')[j+1].index\}$
if $\phi_g(v)[i].type = \mathcal{L}$ **then**
$\quad$**return** $\phi_g(v)[i+1].index < \phi_g(v')[i+1].index$
else if $\phi_g(v)[i].type = \mathcal{F}$ **then**
$\quad$**return** $\texttt{false}$
else if $\phi_g(v)[i].type = \mathcal{R}$ **then**
$\quad$**if** $\phi_g(v)[i+1].index < \phi_g(v')[i+1].index$ **then**
$\quad\quad$**return** $\phi_g(v)[i+1].rec_1$
$\quad$**else**
$\quad\quad$**return** $\phi_g(v')[i+1].rec_2$
$\quad$**end if**
else $\{\phi_g(v)[i].type = \mathcal{N}\}$
$\quad$**return** $\pi_G(\phi_g(v)[i].skl, \phi_g(v')[i].skl)$
end if

To decide if $v \rightsquigarrow_g v'$, we compare $\phi_g(v)$ and $\phi_g(v')$ using the binary predicate π_g described in Algorithm 4, where $\phi_g(v)[i]$ denotes the ith entry of $\phi_g(v)$.

Example 13. Returning to the running example, consider v_5 and v_{16} in Figure 9. Since $\phi_g(v_5)$ and $\phi_g(v_{16})$, shown in Example 12, share the first two common entries, moreover,

$$\phi_g(v_5)[2].type = \mathcal{L} \tag{1}$$

$$\phi_g(v_5)[3].index = 1 < \phi_g(v_{16})[3].index = 2 \tag{2}$$

by Algorithm 4, $\pi_g(\phi_g(v_5), \phi_g(v_{16})) = \texttt{true}$. Note that (1) implies that the least common ancestor of their context x_7 and x_{12} is a special $\mathcal{L}$ node, and (2) implies that x_2 is on the left of x_{12}. By Lemma 4.2, $v_5 \rightsquigarrow_g v_{16}$. The other cases ($(v_5, v_{13})$, (v_5, v_8) and (v_5, v_{11})) can be verified similarly.

5.3 Execution-Based Dynamic Scheme

The above derivation-based scheme can be converted to an execution-based scheme, which creates exactly the same reachability labels. The main challenge is how to dynamically build the explicit parse tree and figure out the context and origin of a newly inserted vertex, given only an execution of a graph. We first give a solution based on a natural restriction on the workflow specification, and then discuss how to remove the restriction by using execution logs.

Let $\mathcal{G}(S)$ be the set of the start graph and all implementation graphs of a workflow specification S, as defined in Section 5.1. We assume that for all graphs $h \in \mathcal{G}(S)$,

1. All vertices of h have *distinct names*; and
2. $s(h)$ and $t(h)$ have *unique atomic names*, *i.e.*, their names do not occur in any other graph in $\mathcal{G}(S)$.

Recall that $s(h)$ and $t(h)$ denote the source and the sink of a sub-workflow h that only distribute and collect the data. We call them the *dummy modules*. In fact, any specification can be modified to satisfy the above two conditions by renaming module names and introducing new dummy modules.

The execution-based labeling algorithm is sketched as follows. For a newly inserted vertex, we can decide if it is a terminal or a non-terminal, by checking its module name (Condition 2). If it is a source, then we can infer one new step of the graph derivation, and update the explicit parse tree as before; otherwise, the context of this new vertex can be determined by any of its immediate predecessors, and the origin can be decided by again checking its module name (Condition 1). If it is a sink, then we know that the current step of the graph derivation is completed.

Example 14. Consider an execution of the graph in Figure 3, which inserts an atomic vertex v_i in the ith step. We start with an empty graph, and get the first vertex v_1 inserted. Since $\texttt{Name}(v_1) = s_0$, by checking the specification in Figure 2, we know that the start graph g_0 is being executed. So by Algorithm 2, we create the root of the explicit parse tree in Figure 9 that corresponds to g_0, and assign the reachability label $\phi_g(v_1)$ according to Algorithm 3. Next, suppose v_2 is inserted. Again, by checking $\texttt{Name}(v_2) = s_1$, we know that the first copy of the loop subgraph h_1 is being executed. So we update the explicit parse tree in Figure 9 by inserting a special $\mathcal{L}$ node x_1 along with its first child x_2. Note that although the original derivation-based scheme creates all the children of a special $\mathcal{L}$ or $\mathcal{F}$ node in a single step, the labeling function ϕ_g given by Algorithm 3 can be done on a node-by-node basis. Hence, we can assign the reachability label $\phi_g(v_2)$ without seeing other copies of h_1. The remaining vertices can be labeled in a similar manner.

During the above labeling process, when the first vertex of a new subgraph is inserted, we can predict future insertions for other atomic vertices of this subgraph. Moreover, their reachability labels can be created at this point, though we do not give out these labels until they are actually inserted. In principle, we are allowed to modify these unassigned labels based on the upcoming insertions. We will see in Section 6 that this relaxation provides execution-based schemes with the potential to create shorter reachability labels.

To remove the restrictions, the only extra information we need is a mapping from vertices of the run to vertices of the specification. Note that in the above algorithm, this is done by comparing module names. In reality, most scientific workflow systems record this mapping in execution logs, by assigning a unique id for each module in the specification.

5.4 Correctness and Quality Analysis

THEOREM 2. **(Correctness)** *Let (ϕ_g, π_g) be our dynamic labeling scheme for a linear recursive workflow grammar G. For any graph $g \in L(G)$ and any two vertices v and v' of g, $\pi_g(\phi_g(v), \phi_g(v')) = \texttt{true}$ if and only if $v \rightsquigarrow_g v'$.*

PROOF. Let t be an explicit parse tree for g. Let x (resp. x') be the context of v (resp. v'). Let $\texttt{LCA}(x, x')$ the least common ancestor of x and x'. Let $\phi_g(v)[i]$ be the ith entry of $\phi_g(v)$. Recall that $\phi_g(v)[i] = \texttt{Entry}(x_i, u_i)$ is obtained from a pair (x_i, u_i) by Algorithm 1, where x_i is the ancestor of x at the ith level, and $u_i = \texttt{null}$ if x_i is a special node, otherwise, u_i is the origin of v with respect to x_i.

We prove the correctness of Algorithm 4. Line 1 computes the maximum common prefix of entries with the same index, say the first i entries. Then $\phi_g(v)[i] = \texttt{Entry}(\texttt{LCA}(x, x'), -)$ and $\phi_g(v')[i] = \texttt{Entry}(\texttt{LCA}(x, x'), -)$, where $-$ is $\texttt{null}$ if $\texttt{LCA}(x, x')$ is a special node, otherwise, $-$ is the origin of v or v' with respect to $\texttt{LCA}(x, x')$. Consider four cases. (1) If $\texttt{LCA}(x, x')$ is a special $\mathcal{L}$ node (Line 2), then let $\phi_g(v)[i+1] = \texttt{Entry}(y, -)$ and $\phi_g(v')[i+1] = \texttt{Entry}(y', -)$, where y and y' are two distinct children of $\texttt{LCA}(x, x')$. Hence, by Lemma 4.2, $v \rightsquigarrow_g v'$ if and only if y is on the left of y'. Note that the ordering of y and y' can be decided by their indexes (Line 3); (2) If $\texttt{LCA}(x, x')$ is a special $\mathcal{F}$ node (Line 4), then by Lemma 4.2, $v \not\rightsquigarrow_g v'$; (3) If $\texttt{LCA}(x, x')$ is a special $\mathcal{R}$ node (Line 6), then again let $\phi_g(v)[i+1] = \texttt{Entry}(y, -)$ and $\phi_g(v')[i+1] = \texttt{Entry}(y', -)$, where y and y' are two distinct children of $\texttt{LCA}(x, x')$. We assume without loss of generality that y is on the left of y'. The other case can be handled in the same way. Let u (resp. u') be the origin of v (resp. v') with respect to y. Note that $\phi_g(v)[i+1] = \texttt{Entry}(y, u)$. Moreover, $u' = \texttt{Annt}(y, z)$ is a recursive vertex of $h = \texttt{Annt}(y)$ annotated on the dashed edge (y, z), where z is the right sibling of y. By Lemma 4.2, $v \rightsquigarrow_g v'$ if and only if $u \rightsquigarrow_h u'$. Hence, by Algorithm 1, $v \rightsquigarrow_g v'$ if and only if $\phi_g(v)[i+1].rec_1 = \texttt{true}$; and (4) If $\texttt{LCA}(x, x')$ is a non-special node (Line 12), then $\phi_g(v)[i] = \texttt{Entry}(\texttt{LCA}(x, x'), u)$ and $\phi_g(v')[i] = \texttt{Entry}(\texttt{LCA}(x, x'), u')$, where u (resp. u') is the origin of v (reps. v') with respect to $\texttt{LCA}(x, x')$. Let $h = \texttt{Annt}(\texttt{LCA}(x, x'))$. By Lemma 4.2, $v \rightsquigarrow_g v'$ if and only if $u \rightsquigarrow_h u'$. Hence, by Algorithm 1, $v \rightsquigarrow_g v'$ if and only if $\phi_G(\phi_g(v)[i].skl, \phi_g(v')[i].skl) = \texttt{true}$. □

The quality of a labeling scheme (ϕ, π) is measured by *label length*, *construction time* (*i.e.*, the time to compute ϕ) and *query time* (*i.e.*, the time to evaluate π). Among them, label length is the main factor. Since the derivation-based and execution-based schemes create same labels, they differ only in the construction time. All parameters for quality analysis are listed in Table 1, where G is a linear recursive grammar, t is an explicit parse tree for a graph $g \in L(G)$ and h is a subgraph of g. The size of a graph refers to the number of vertices. Note that for a fixed G, n_G and t_G are constants. By Lemma 4.1, d_t is also bounded by a constant.

Table 1: Parameters for Quality Analysis

n_g	the size of g	n_h	the size of h
n_t	the size of t	d_t	the depth of t
θ_t	the max outdegree of a node of t		
n_G	the max size of a specification graph		
t_G	the time to compare skeleton labels		

THEOREM 3. **(Quality Analysis)** *Let G be a linear recursive workflow grammar. For any graph $g \in L(G)$, our dynamic labeling scheme (ϕ_g, π_g) guarantees*

1. *logarithmic label length: for any vertex v of g,*

$$|\phi_g(v)| = O(\log n_g) \text{ bits}$$

2. *linear total construction time: computing $\phi_g(v)$ for each vertex v of g takes a total of $O(n_g)$ time.*

   2a *(execution-based) for any vertex insertion, $g_i = g_{i-1} + (v, C)$, computing $\phi_g(v)$ takes $O(1)$ time.*

   2b *(derivation-based) for any vertex replacement, $g_i = g_{i-1}[u/h]$, computing $\phi_g(v)$ for each vertex v of h takes a total of $O(n_h)$ time.*

3. *constant query time: for any two vertices v and v' of g, computing $\pi_g(\phi_g(v), \phi_g(v'))$ takes $O(1)$ time.*

PROOF. First, we prove the logarithmic label length. Let $\phi_g(v)[i]$ be the ith entry of $\phi_g(v)$. By Algorithm 1,

$$|\phi_g(v)[i]| \le \log \theta_t + 2 + \log n_G + 1 + 1 \text{ bits}$$

Recall that we use only a pointer to each skeleton label, rather than the label itself. So it takes only $\log n_G$ bits. By Algorithm 3, $\phi_g(v)$ has at most d_t entries, and $\theta_t \le n_t \le n_g$.

$$|\phi_g(v)| \le d_t * (\log \theta_t + \log n_G + 4) = O(\log n_g) \text{ bits}$$

Next, we prove the linear total construction time. For the derivation-based scheme, Algorithm 3 has two steps: (1) update t by inserting a new subtree t' that corresponds to h using Algorithm 2. This step can be done in $O(n'_t) = O(n_h)$ time, where n'_t is the number of nodes of t' and $n'_t \le n_h$; and (2) create $\phi_g(v)$ for each vertex v of h. By Algorithm 1, it may involve comparing two skeleton labels, and thus takes t_G time. So the total construction time is $O(n_h * t_G) = O(n_h)$. For the execution-based scheme, the only extra computation is to decide if the newly inserted vertex is a terminal by comparing its module name, which can be done in $O(1)$ time. So the total construction time remains linear.

Finally, we prove the constant query time. Since $\phi_g(v)$ and $\phi_g(v')$ have at most d_t entries, by Algorithm 4, finding the maximum common prefix of entries with same index (Line 1) takes $O(d_t)$ time. The rest of computation may involve comparing two skeleton labels (Line 13), which takes t_G time. So the query time is $O(d_t) + t_G = O(1)$. □

6. LABELING DYNAMIC WORKFLOWS WITH NONLINEAR RECURSION

Although our dynamic labeling scheme is for linear recursive workflows, it can be adapted to label nonlinear recursive workflows. The only modification is to create a simplified explicit parse tree without special $\mathcal{R}$ nodes by treating all vertices in a non-recursive way. A further optimization can be achieved by compressing at most one recursive vertex using a special $\mathcal{R}$ node, while treating other recursive vertices (if they exist) in a non-recursive way. However, the depth of the modified explicit parse tree is no longer bounded by a constant, but is proportional to the depth of recursion. This dynamic scheme may therefore create linear-size reachability labels, matching the lower bound in Theorem 1.

The remaining question is whether any nonlinear recursive workflow allows compact dynamic labeling. Theorem 4 shows that the answer is "no" for the derivation-based problem. It gives a stronger result than Theorem 1, showing that there is no compact derivation-based dynamic labeling scheme for *any given* nonlinear recursive workflow. Combining Theorems 3 and 4, we conclude that linear recursive workflows are the largest class of workflows that allow compact derivation-based dynamic labeling schemes.

THEOREM 4. *For any nonlinear recursive workflow grammar G and any derivation-based dynamic labeling scheme (ϕ, π) for G, there is a derivation of a graph $g \in L(G)$ with n vertices such that ϕ assigns a reachability label of $\Omega(n)$ bits for some vertex of g.*

PROOF. Since G is nonlinear recursive, by Definition 10, there is a production $A := h$ with at least two recursive vertices. By applying a sequence of productions, we can obtain from $A := h$ a new production $A := h'$ with two recursive vertices u_1 and u_2 both named A.

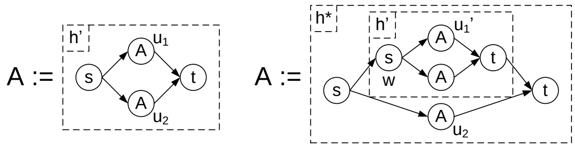

Figure 10: A new production $A := h^*$ constructed from $A := h'$ with two parallel recursive vertices u_1 and u_2.

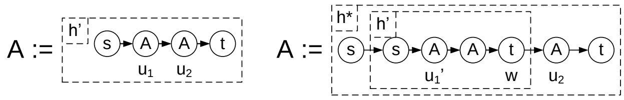

Figure 11: A new production $A := h^*$ constructed from $A := h'$ with two series recursive vertices u_1 and u_2.

Next, we want to find a *differential vertex* w that reaches exactly one of u_1 and u_2. Consider two cases. (1) If u_1 and u_2 are not reachable from each other in h' (see Figure 10), then we replace u_1 with a new copy of h', and obtain a new production $A := h^*$. Let u'_1 be the new copy of u_1, and w be the source of the new copy of h', then w reaches exactly one of u'_1 and u_2; and (2) If one of u_1 and u_2 can reach the other in h' (let $u_1 \rightsquigarrow_{h'} u_2$, see Figure 11), then again we replace u_1 with a new copy of h', and obtain a new production $A := h^*$. Let u'_1 be the new copy of u_1, and w be the sink of the new copy of h', then w reaches exactly one of u'_1 and u_2.

Recall the production $A := h_1$ in Figure 6, where the vertex named a reaches exactly one of the two vertices named A. Using the new production $A := h^*$, we can prove the theorem using the technique of Theorem 1. □

We next turn to the execution-based problem. To apply the same proof, we need to ensure that the differential vertex w precedes both recursive vertices u_1 and u_2 in the given insertion sequence, so that $\phi(w)$ can divide all reachability labels that will be later assigned to the subgraphs derived from u_1 and u_2 into two disjoint sets. *E.g.*, the proof of Theorem 1 relies on the fact that the differential vertex (named a) precedes two recursive vertices (named A) in the given insertion sequence (see Figure 6). Unfortunately, Case (2) in the proof of Theorem 4 (see Figure 11) violates the above condition. The following example inspired by Case (2) shows that some nonlinear recursive workflow indeed allows compact execution-based dynamic labeling schemes.

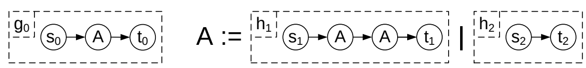

Figure 12: A nonlinear recursive workflow grammar that allows a compact execution-based dynamic scheme.

Example 15. Consider the workflow grammar G shown in Figure 12. Since any graph $g \in L(G)$ is a simple path, we simply label the ith vertex by an index of i. This naive dynamic scheme creates logarithmic-size reachability labels.

However, Case (1) (see Figure 10) respects the condition. We thus get a similar result to Theorem 4 for a subclass of nonlinear workflows, called the *parallel recursive workflows*.

Definition 13. A workflow grammar is said to be *parallel recursive*, if there is a production with two recursive vertices that are not reachable from each other (in parallel).

THEOREM 5. *For any parallel recursive workflow grammar G and any execution-based dynamic labeling scheme (ϕ, π) for G, there is an execution of a graph $g \in L(G)$ with n vertices such that ϕ assigns a reachability label of $\Omega(n)$ bits for some vertex of g.*

PROOF. (Sketch) We follow the same proof as Theorem 4. The correctness follows from Definition 13, which ensures that the new production $A := h^*$ must fall into Case (1). $\square$

This paper leaves open the problem of whether non-parallel recursive workflows (with only series recursive vertices) allow compact execution-based dynamic labeling schemes.

7. EXPERIMENTAL EVALUATION

We empirically evaluated the proposed dynamic labeling scheme in terms of label length, construction time, query time and preprocessing overhead. We performed three sets of experiments: The first uses a collected, real-life scientific workflow (Section 7.2). The second measures a variety of synthetic workflows with different characteristics (Section 7.3). The last compares our dynamic scheme against the state-of-the-art static scheme [6] (Section 7.4).

7.1 Experimental Setup

All labeling schemes are implemented in Java 6. Our experiments were performed on a local PC with Intel Pentium 2.80GHz CPU and 2GB memory running Windows XP.

Real-Life and Synthetic Datasets. The real-life workflow, called *BioAID*, was taken from the myExperiment workflow repository [21]. To focus on the specific factors that affect the labeling performance, we also created a family of synthetic workflows using the simple topology shown in Figure 13. Due to the lack of realistic workflow runs, we simulate the execution by repeating loops, forks and recursion a random number of times. For each specification, we vary the size of runs from $1K$ to $32K$ by a factor of 2, and randomly select one derivation and one execution for each run as dynamic inputs. All data are stored in XML files.

Labeling Methodology. We compare two schemes for labeling workflow runs: (1) the one presented in this paper, which is denoted by `DRL`, for (D)ynamic scheme for (R)ecursive workflows; and (2) the state-of-the-art *static* scheme [6], which is also skeleton-based, and is denoted by `SKL`, for (SK)eleton-based scheme. To obtain skeleton labels (for both schemes), we apply two simple schemes for labeling the specification: (1) `TCL` denotes the one given in Section 3.2. It precomputes the (T)ransitive (C)losure for all vertices, and can be used to label either a static graph or an execution-based dynamic graph; and (2) `BFS` does not perform any labeling, but answers a reachability query by a (B)readth (F)irst (S)earch over the graph. Since a skeleton-based scheme for labeling runs is parameterized by the scheme for labeling the specification, we denote by `DRL(TCL)` and `DRL(BFS)` (resp. `SKL(TCL)` and `SKL(BFS)`) the corresponding combinations of the two.

Evaluation Methodology. The result for label length and construction time is an average over 10^3 sample runs, and the one for query time is an average over 10^5 sample queries.

7.2 Labeling Real-Life Workflows

In the first set of experiments, we evaluate `DRL` using *BioAID*. It consists of 11 sub-workflows with an average size of 10.5 and a nesting depth [5] of 2. There are 2 loop modules, 4 fork modules and one linear recursion of length 2. Note that the derivation-based and execution-based dynamic schemes differ only in the construction time, and the scheme used to label the specification affects only the query time and the preprocessing overhead.

Figure 14 reports the maximum and average label length. As expected, both increase logarithmically with the size of the run (note that the x-axis is log scale). The average length is always shorter than the maximum length by a small constant (about 6 bits). More interestingly, both lines are almost parallel to the asymptotic line $f(n) = \log(n) + 13$. Hence, they are bounded by $c \log(n) + O(1)$, where c is a small constant factor close to 1.

Figure 15 reports the total construction time for both derivation-based and execution-based schemes. Observe that they increase linearly with the size of the run. On average, we label a new vertex on the fly in less than 5 μs, which is comparable to the time of updating the graph itself (about 6 μs). Moreover, the derivation-based scheme is faster than the execution-based scheme. This is because the latter needs to find the context and origin of the newly inserted vertex.

Figure 16 reports the query time for `DRL(TCL)` and `DRL(BFS)`. Recall that `TCL` allows constant query time, but uses linear-size labels; in contrast, `BFS` does not use any labels, but has linear query time. However, since the specification graphs are small and fixed, `DRL` answers reachability queries in almost constant time, when combined with either (Figure 16). But `DRL(TCL)` is slightly faster than `DRL(BFS)` by about 2 μs. We also measured the preprocessing overhead for `DRL(TCL)`, and found that the overhead is negligible: the skeleton labels take totally 650 bits and are built in less than 0.05 ms.

Conclusions: The experimental results confirm the theoretical quality analysis of `DRL` in Theorem 3. Moreover, `DRL` is scalable for large dynamic runs, even when combined with simple skeleton schemes like `TCL` and `BFS`. Due to the small size of the specification graphs, the benefit of using more sophisticated schemes to label the specification is limited.

7.3 Labeling Synthetic Workflows

In the second set of experiments, we evaluate `DRL` for a variety of synthetic workflows created from the specification in Figure 13. It consists of a chain of nested sub-workflows with one loop module L, one fork module F and one recursive module R. Note that the recursive sub-workflow h'_d may in general contain several R modules. All sub-workflows are random two-terminal graphs of some fixed size. The parameters are: (a) the size of sub-workflows; (b) the nesting depth of sub-workflows; and (c) if the workflow is linear recursive (*i.e.*, if h'_d has more than one R modules). Due to space constraints, we report only the main factor, label length.

First, we generate a set of linear recursive workflows by varying the size of sub-workflows from 10 to 160 by a factor of 2, and fixing the nesting depth of sub-workflows to be 5. Figure 17 reports the maximum label length for dynamic runs with $5K$ vertices. As we can see, the maximum label length increases almost logarithmically with the size of sub-workflows. To explain the result, recall that a tighter upper

[5] In a recursive workflow, the *nesting depth* of sub-workflows refers to the length of the longest path of sub-workflows, starting from the start graph, that implement distinct composite modules. *E.g.*, the nesting depth of sub-workflows in Figure 13 is d.

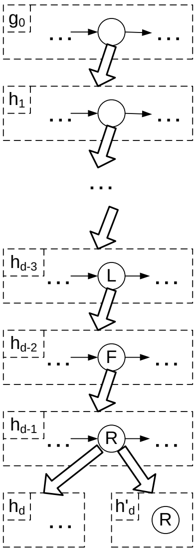

Figure 13: Synthetic Workflow

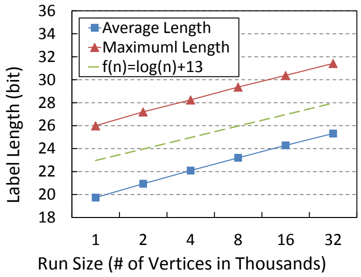

Figure 14: BioAID (Label Length)

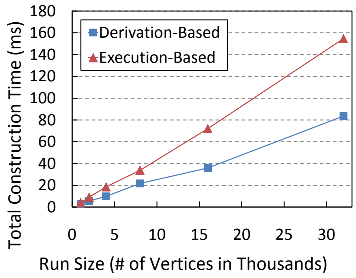

Figure 15: BioAID (Constr. Time)

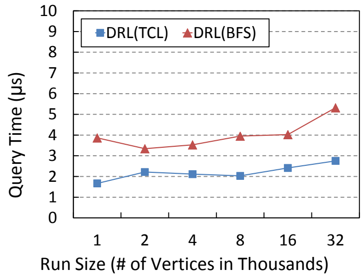

Figure 16: BioAID (Query Time)

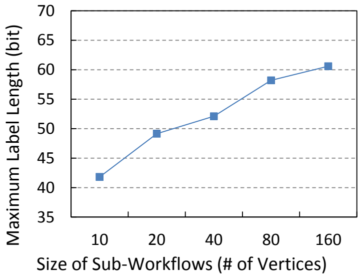

Figure 17: Varying Size

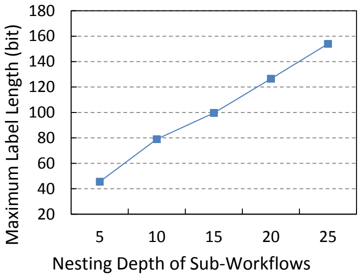

Figure 18: Varying Depth

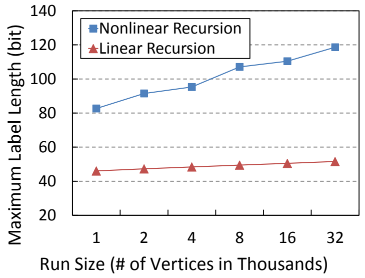

Figure 19: Nonlinear Workflows

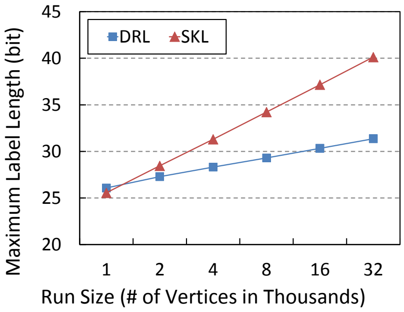

Figure 20: DRL vs SKL (Label Length)

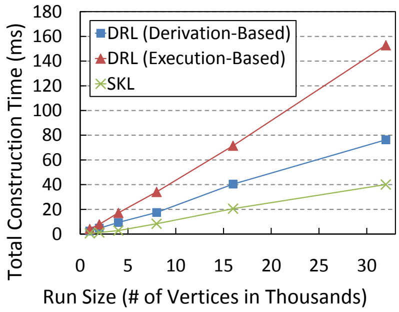

Figure 21: DRL vs SKL (Constr. Time)

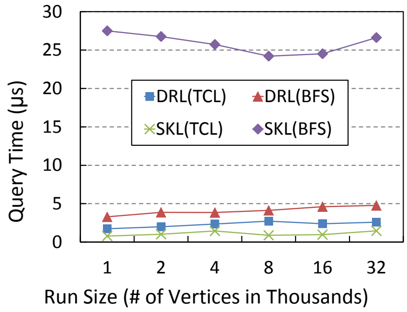

Figure 22: DRL vs SKL (Query Time)

bound of label length, given in the proof of Theorem 3, is

$$|\phi_g(v)| \leq d_t * (\log \theta_t + \log n_G + 4) \qquad (3)$$

where all parameters for quality analysis are defined in Table 1. In this experiment, d_t is fixed, and n_G increases by a factor of 2. We now estimate θ_t. Since $n_G * n_t$ is roughly the size of the run (a fixed constant of $5K$), n_t decreases by a factor of 2. Note that t is a balanced tree with fixed depth. In general, θ_t decreases much more slowly than n_t. It follows that the increase of $\log n_G$ dominates the decrease of $\log \theta_t$ in (3). This confirms the result in Figure 17.

Next, we generate a set of linear recursive workflows by varying the nesting depth of sub-workflows from 5 to 25 by a constant of 5, and fixing the size of sub-workflows to be 20. Figure 18 reports the maximum label length for dynamic runs with $5K$ vertices. Observe that the maximum label length increases linearly with the nesting depth of sub-workflows. This is again confirmed by (3), where n_G and θ_t are fixed, and d_t is proportional to the nesting depth.

Finally, we generate a nonlinear recursive workflow with two R modules in h'_d (see Figure 13) and a linear recursive one with only one R module in h'_d. For both workflows, the size of sub-workflows is 20, and the nesting depth is 5. Figure 19 reports the maximum label length. Not surprisingly, the nonlinear recursive workflow produces longer labels than the linear recursive one. Although `DRL` creates linear-size labels for nonlinear recursive workflows in the worst case, Figure 19 shows that it performs reasonably well in practice: the maximum label length for a run with $32K$ vertices is less than 120 bits. Note that if we use `TCL` to label the run dynamically, it gives a label of exactly $32K - 1$ bits.

Conclusions: The main factor that affects the performance of `DRL` is the nesting depth of sub-workflows. However, we observe from our experience that most real-life workflows are linear recursive, and have a nesting depth of less than 5. `DRL` is also effective to label nonlinear recursive workflows.

7.4 DRL (Dynamic) vs SKL (Static)

In the last set of experiments, we compare `DRL` and `SKL`. The limitations of `SKL` are: (1) `SKL` is a static scheme, which takes the entire run graph as input; (2) `SKL` supports only non-recursive workflows (with loops and forks); and (3) `SKL` entails skeleton labels over a global specification graph, in which all composite modules are replaced with corresponding sub-workflows. We show only results for the real-life workflow. To achieve a fair comparison, we remove the recursion [6]. The results for synthetic workflows are similar.

Figure 20 reports the maximum label length. Observe that `DRL` creates shorter labels than `SKL` when the run size is larger than $1.5K$. This is because `DRL` uses a prefix-based scheme [18] to label the explicit parse tree, while `SKL` uses an interval-based scheme [22]. The former performs better on

[6]It turns out that the linear recursion in this workflow can be converted to a loop which performs similar computations.

balanced trees with relatively high degrees and low depth. This is exactly the case when the run becomes large. More precisely, the upper bound of the label length for SKL is

$$|\phi_g(v)| \leq 3 * \log n_t + O(\log n_G) \tag{4}$$

where $n_t = O(n_g)$ and $n_G = O(1)$. So the logarithmic label length for SKL has a factor of 3. Recall from Figure 14 that the factor for DRL is close to 1. Hence, for large runs, DRL creates shorter labels than SKL by a factor of almost 3. This is confirmed by the slopes of the two lines in Figure 20.

Figure 21 reports the total construction time. Since SKL builds simpler (but larger) labels than DRL consisting only of three indexes and one skeleton label, SKL is faster than derivation-based and execution-based DRL by a factor of 2 and 4 respectively. However, unlike DRL, SKL cannot start labeling until the entire run is completed.

Figure 22 reports the query time for all four combinations. BFS performs a linear-time graph search over the specification. Consequently, when combined with BFS, the cost of comparing skeleton labels is the dominant factor. Note that SKL searches over a global specification graph with 106 vertices, while DRL searches over an individual sub-workflow with only 10.5 vertices on average. Hence, SKL(BFS) is slower than DRL(BFS) by one order of magnitude. In contrast, when combined with TCL, the cost of comparing skeleton labels is negligible. Given that SKL enables simple decoding which compares only three indexes and one skeleton label, SKL(TCL) is slightly faster than DRL(TCL). However, such efficiency is traded by high preprocessing overhead reported in Table 2.

Table 2: Overhead of Labeling Specification

	Total Space (bit)	Construction Time (ms)
DRL(TCL)	650	0.04375
SKL(TCL)	5565	0.16328

Conclusions: DRL creates shorter labels than SKL, and is more robust to the scheme for labeling the specification.

8. CONCLUSIONS

This paper studies derivation-based and execution-based dynamic reachability labeling problems for recursive workflows with loops and forks. We provide tight lower and upper bounds of $\Theta(n)$ bits on the maximum label length, and present a compact dynamic labeling scheme for linear recursive workflows which uses labels of $\log(n)$ bits. The evaluation, performed over both real and synthetic workflows, shows that our dynamic scheme creates shorter labels than the start-of-the-art static scheme [6] by a factor of almost 3.

This paper also shows an interesting characterization: A workflow allows a compact derivation-based dynamic scheme if and only if it is linear recursive. However, finding an execution-based characterization is still an open problem.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. This work was supported in part by the US National Science Foundation grants IIS-0803524 and IIS-0629846, by the Israel Science Foundation, by the US-Israel Binational Science Foundation, and by the EU grant MANCOOSI.

10. REFERENCES

[1] S. Abiteboul, H. Kaplan, and T. Milo. Compact labeling schemes for ancestor queries. In *SODA*, pages 547–556, 2001.

[2] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD Conference*, pages 253–262, 1989.

[3] S. Alstrup, P. Bille, and T. Rauhe. Labeling schemes for small distances in trees. In *SODA*, pages 689–698, 2003.

[4] S. Alstrup and T. Rauhe. Improved labeling scheme for ancestor queries. In *SODA*, pages 947–953, 2002.

[5] I. Altintas, C. Berkley, E. Jaeger, M. B. Jones, B. Ludäscher, and S. Mock. Kepler: An extensible system for design and execution of scientific workflows. In *SSDBM*, pages 423–424, 2004.

[6] Z. Bao, S. B. Davidson, S. Khanna, and S. Roy. An optimal labeling scheme for workflow provenance using skeleton labels. In *SIGMOD Conference*, pages 711–722, 2010.

[7] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. Vistrails: visualization meets data management. In *SIGMOD Conference*, pages 745–747, 2006.

[8] A. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *SIGMOD Conference*, pages 993–1006, 2008.

[9] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *SODA*, pages 937–946, 2002.

[10] E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic xml trees. In *PODS*, pages 271–281, 2002.

[11] C. Demetrescu and G. F. Italiano. Fully dynamic transitive closure: Breaking through the $o(n^2)$ barrier. In *FOCS*, pages 381–389, 2000.

[12] P. Fraigniaud and A. Korman. Compact ancestry labeling schemes for xml trees. In *SODA*, pages 458–466, 2010.

[13] T. Heinis and G. Alonso. Efficient lineage tracking for scientific workflows. In *SIGMOD Conference*, pages 1007–1018, 2008.

[14] D. Hull, K. Wolstencroft, R. Stevens, C. A. Goble, M. R. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34(Web-Server-Issue):729–732, 2006.

[15] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, 1990.

[16] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD Conference*, pages 813–826, 2009.

[17] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD Conference*, pages 595–608, 2008.

[18] H. Kaplan, T. Milo, and R. Shabo. A comparison of labeling schemes for ancestor queries. In *SODA*, pages 954–963, 2002.

[19] V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. In *STOC*, pages 492–498, 1999.

[20] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. Ordpaths: Insert-friendly xml node labels. In *SIGMOD Conference*, pages 903–908, 2004.

[21] D. D. Roure, C. A. Goble, and R. Stevens. The design and realisation of the my$_{\text{experiment}}$ virtual research environment for social sharing of workflows. *Future Generation Comp. Syst.*, 25(5):561–567, 2009.

[22] N. Santoro and R. Khatib. Labelling and implicit routing in networks. *Comput. J.*, 28(1):5–8, 1985.

[23] L. Xu, T. W. Ling, H. Wu, and Z. Bao. Dde: from dewey to a fully dynamic xml labeling scheme. In *SIGMOD Conference*, pages 719–730, 2009.

[24] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *PVLDB*, 3(1):276–284, 2010.