1-1-2012

Labeling Workflow Views With Fine-Grained Dependencies

Zhuowei Bao
University of Pennsylvania, zhuowei@cis.upenn.edu

Susan B. Davidson
University of Pennsylvania, susan@cis.upenn.edu

Tova Milo
Tel Aviv University, milo@cs.tau.ac.il

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-12-11.

Labeling Workflow Views With Fine-Grained Dependencies

Abstract

This paper considers the problem of efficiently answering reachability queries over views of provenance graphs, derived from executions of workflows that may include recursion. Such views include composite modules and model fine-grained dependencies between module inputs and outputs. A novel view-adaptive dynamic labeling scheme is developed for efficient query evaluation, in which view specifications are labeled statically (i.e. as they are created) and data items are labeled dynamically as they are produced during a workflow execution. Although the combination of fine-grained dependencies and recursive workflows entail, in general, long (linear-size) data labels, we show that for a large natural class of workflows and views, labels are compact (logarithmic-size) and reachability queries can be evaluated in constant time. Experimental results demonstrate the benefit of this approach over the state-of-the-art technique when applied for labeling multiple views.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-12-11.

Labeling Workflow Views with Fine-Grained Dependencies

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@cs.tau.ac.il

ABSTRACT

This paper considers the problem of efficiently answering reachability queries over *views* of provenance graphs, derived from executions of workflows that may include recursion. Such views include composite modules and model fine-grained dependencies between module inputs and outputs. A novel *view-adaptive* dynamic labeling scheme is developed for efficient query evaluation, in which view specifications are labeled statically (i.e. as they are created) and data items are labeled dynamically as they are produced during a workflow execution. Although the combination of fine-grained dependencies and recursive workflows entail, in general, long (linear-size) data labels, we show that for a large natural class of workflows and views, labels are *compact* (logarithmic-size) and reachability queries can be evaluated in constant time. Experimental results demonstrate the benefit of this approach over the state-of-the-art technique when applied for labeling multiple views.

1. INTRODUCTION

The ability to capture, manage and query workflow provenance is increasingly important for scientific as well as business applications. By maintaining information about the sequence of module executions (processing steps) used to produce a data item, as well as the parameter settings and intermediate data passed between module executions, the validity and reproducibility of data results can be enhanced. For example, if an input to a workflow execution is discovered to be incorrect, we may wish to determine whether a particular workflow output depends on it and is thus also potentially incorrect. Finding efficient techniques to answer such *reachability* queries is thus of particular interest.

However, provenance information can be extremely large, so we may wish to provide different *views* of this information. For example, users may wish to specify *abstraction views* which focus user attention on relevant provenance information and abstract away irrelevant details, an idea proposed in [7]. Workflow owners may also wish to specify *security views* which can be used to hide private information from certain user groups (e.g., sensitive intermediate data and module functionality [9]). [1] Provenance views consist of a set of *composite modules* which encapsulate subworkflows; provenance information within these subworkflows are then hidden in an associated execution.

[1]A similar notion was used for securely querying XML [11].

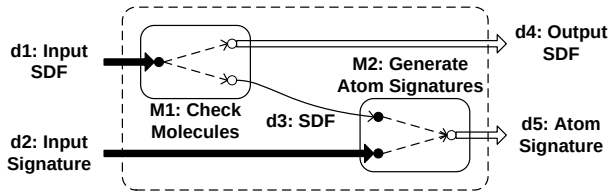

Figure 1: Views with Fine-Grained Dependencies

EXAMPLE 1. *Figure 1 shows an abstraction of a real-life scientific workflow collected from the myExperiment repository [18]. It generates atom signatures for individual compounds given a Structural Data File (SDF) as input (ignore for now the dashed edges inside modules M_1 and M_2). In a high-level view of this workflow, users see only one composite module, indicated as the big dashed box, with two inputs (d_1 and d_2) and two outputs (d_4 and d_5), while modules M_1 and M_2 and intermediate data d_3 are hidden.*

An important thing to keep in mind is that Workflow provenance not only records the order of module executions but also the dependencies between inputs and outputs of modules. Therefore, workflow views should explicitly specify the input-output dependencies for modules that are exposed to users. Previous research [12, 20, 4, 5] has adopted a simplified provenance model which assumes that every output of a module depends on every input, termed *black-box dependencies*. However, a more fine-grained provenance model captures the fact that the output of a module may depend on only a subset of its inputs.

To understand why fine-grained dependencies are useful, consider the two types of views mentioned earlier. In abstraction views, although irrelevant workflow details are hidden inside composite modules, users should still be able to see the true dependencies between inputs and outputs of composite modules (*white-box dependences*). In security views, however, one may want to hide the true dependencies between inputs and outputs of certain composite modules in order to preserve structural or module privacy [9]. To this end, one may move to somewhere on the spectrum between white-box and black-box dependencies (*grey-box dependencies*). With grey-box dependencies, additional (false) dependencies between inputs and outputs may be added. [2]

[2]Technical results in this paper hold even when true dependencies between inputs and outputs are removed.

EXAMPLE 2. *Returning to Figure 1, fine-grained dependencies between the inputs and outputs of modules M_1 and M_2 are indicated as dashed edges inside the modules. In an abstraction view, the composite module would be associated with white-box dependencies, in which d_4 depends on d_1 but not on d_2. However, in a security view, the composite module could be associated with a grey-box dependency matrix in which every output depends on every input. Hence, the answer to the reachability query "Does d_4 depend on d_2?" is different in the two views.*

This paper considers the problem of efficiently answering reachability queries over views of provenance graphs, of the types illustrated above. A common approach for processing reachability queries is to label data items so that the reachability between any two items can be answered efficiently by comparing their labels. Moreover, data items must be labeled dynamically as soon as they are produced during the execution, since scientific workflows can take a long time to execute and users may wish to query partial executions.

In contrast to previous work, we study effective dynamic labeling in the context of (1) *fine-grained dependencies* between inputs and outputs of modules; and (2) *views with grey-box dependencies*. This context introduces several new challenges. First, none of the existing dynamic labeling schemes applies to fine-grained dependencies, since they all rely on a simplified provenance model with black-box dependencies. Second, due to grey-box dependencies, the answer to a reachability query may alter in different views. A brute-force approach to handling multiple views is to label data items for each view repeatedly and separately. This has two drawbacks: (i) large index: for each data item, we must maintain one label for each view; and (ii) expensive index maintenance: when a new view is added, all existing data items must be re-labeled. To address the challenges, more effective labeling techniques must be developed. The main contributions of this paper are summarized as follows.

- We propose a formal model based on graph grammars which capture a rich class of (possibly recursive) workflows with fine-grained dependencies between the inputs and outputs of modules. We then use the model to formalize the notion of views. They are defined over the workflow specification and then naturally projected onto its runs (Section 2).

- To get a handle on the difficulty introduced by fine-grained dependencies to the dynamic labeling problem, we prove that in general, long (linear-size) labels are required. We further show that common restrictions on the workflow specification, that sufficed to reduce the label length for black-box dependencies [5], are no longer helpful. Nevertheless, we identify a large natural class of *safe views* over *strictly linear-recursive* workflows for which dynamic, yet compact (logarithmic-size) labeling is possible (Section 3).

- Based on this foundation we propose a novel labeling approach whereby view specifications are labeled *statically* (i.e. as they are created), whereas data items are labeled *dynamically* as they are produced during a workflow execution. At query time, the labeling of the view over which the reachability query is asked is used to augment the data labels to provide the correct answer in constant time. We call this a *view-adaptive* dynamic labeling scheme. It has the great advantage that, since data labels are unrelated to any view, views can be added/deleted/modified without having to touch the data. It is both space-efficient and time-efficient relative to the alternative approach where data items are labeled repeatedly and separately for each view (Sections 4 and 5).

- Finally, we evaluate the proposed view-adaptive labeling scheme over both real-life and synthetic workflows. The experimental study demonstrates the superiority of our view-adaptive labeling approach over the state-of-the-art technique [5] when applied to label multiple views (Section 6).

Related Work. Before presenting our results, we briefly review related work. The problem of reachability labeling has been studied for different classes of graphs in both static and dynamic settings. Ideally, one would like to build *compact* (logarithmic-size) labels which enable *efficient* (constant) query processing. While compact and efficient labeling is shown to be feasible for static trees [19], when labeling general directed acyclc graphs (DAGs), any possible scheme requires linear-size labels even if arbitrary query time is allowed [4]. On the other hand, dynamic labeling is also much harder than static labeling. [8] shows that even labeling dynamic trees requires linear-size labels. Fortunately, although workflow runs can have arbitrarily more complex DAG structures than trees, [4, 5] show that knowledge of the specification can be exploited to obtain compact and efficient labeling schemes for both static and dynamic runs derived from a given specification. A more detailed comparison between existing static and dynamic labeling schemes for XML trees [19, 1, 8, 17, 22], for DAGs [14, 23, 21, 15, 10] and for workflow runs [12, 4, 5] is summarized in [5]. However, as mentioned above, none of the existing dynamic labeling schemes is applicable to our problem as they neither support fine-grained dependencies nor handle views.

2. MODEL AND PROBLEM STATEMENT

We present a fine-grained workflow model with white-box dependencies in Section 2.1. Based on this model, we define views with grey-box dependencies in Section 2.2. Section 2.3 formulates the view-adaptive dynamic labeling problem.

2.1 Fine-Grained Workflow Model

Our workflow model is built upon two concepts: *workflow specification*, which describes the design of a workflow, and *workflow run*, which describes a particular workflow execution. We model the structure of a specification as a *context-free workflow grammar* whose *language* corresponds to exactly the set of all possible runs of this specification. The grammar that we use is similar to [5, 6]. However, previous work [16, 12, 20, 5, 6] adopted a simplified provenance model which implicitly assumes *black-box dependencies* – every output of a module depends on every input. In contrast, this paper proposes a more *fine-grained* provenance model which captures the fact that an output of a module may depend on only a subset of inputs. We call this *white-box dependencies*. In particular, our model associates the grammar with a *dependency assignment* that explicitly specifies the dependencies between inputs and outputs of atomic modules.

The basic building blocks of our model are *modules* and *simple workflows*. A module has a set of input ports and a set of output ports; and a simple workflow is built up from a set of modules by connecting their input and output ports.

Definition 1. **(Module)** A *module* is $M = (I, O)$, where I is a set of *input ports* and O is a set of *output ports*.

Definition 2. **(Simple Workflow)** A *simple workflow* is $W = (V, E)$, where V is a multiset of *modules* and E is a set of *data edges* from an output port of one module to an input port of another module. Each data edge carries a unique *data item* that is produced by the former and then consumed by the latter. Input ports with no incoming data edges are called *initial input ports*; and output ports with no outgoing data edges are called *final output ports*.

To simplify the presentation, we assume that (1) *pairwise non-adjacent data edges*: any pair of data edges are not incident to the same port; and (2) *acyclic simple workflow*: data edges do not form cycles among the modules. Note that the above two restrictions do not limit the expressive power of our model. For (1), adjacent data edges can be resolved by introducing dummy modules that distribute or aggregate multiple data items. For (2), we will see that loops can be implicitly captured by recursive productions.

Example 3. *The top left corner of Figure 2 shows a module S with two input ports and three output ports, which are denoted by solid and empty cycles, respectively. The top right corner of Figure 2 shows a simple workflow W_1 with six modules and ten data edges (solid edges, ignore the dashed edges inside modules for now). W_1 has two initial input ports and three final output ports, which are highlighted by solid and empty thick arrows, respectively.*

To build a new workflow, an existing (simple) workflow may be reused as a *composite module*. This is modeled by a *workflow production*.

Definition 3. **(Workflow Production)** A *workflow production* is of form $M \rightarrow_f W$, where M is a composite module, W is a simple workflow and f is a bijection that maps input ports and output ports of M to initial input ports and final output ports of W, respectively. When f is clear from the context, we simply denote a production by $M \rightarrow W$.

Example 4. *In Figure 2, each row defines one or two productions. For example, the first row defines $S \rightarrow W_1$, and the second row defines $A \rightarrow W_2$ and $A \rightarrow W_3$. Note that A also appears as a composite module in both W_1 and W_4. For simplicity, we assume that for each production $M \rightarrow W$, the (initial) input ports and (final) output ports of M and W are mapped by f from top to bottom as shown in the figure.*

The *context-free workflow grammar* is a natural extension of the well-known context-free string grammar, where modules correspond to characters, and simple workflows that are built up from modules correspond to strings that are sequences of characters. In particular, atomic and composite modules correspond to terminals and variables, respectively. We also define a start module and a finite set of workflow productions. By Definition 3, each production $M \rightarrow_f W$ replaces a composite module M with a simple workflow W. The data edges adjacent to M are connected to W based on the bijection f. The *language* of a context-free workflow grammar consists of all simple workflows that can be derived from the start module and contain only atomic modules.

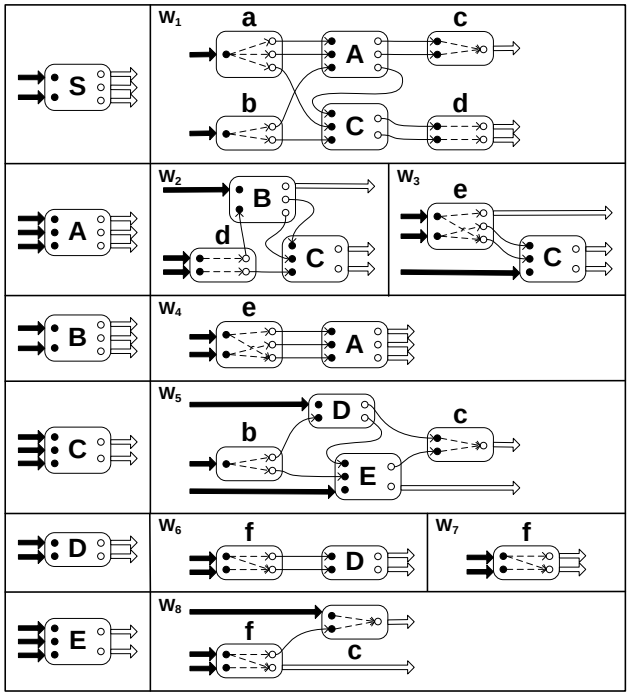

Figure 2: Workflow Specification

Following the standard notations for string grammars, given a finite set Σ of modules, let Σ^* denote the set of all simple workflows that are built up from a multiset of modules in Σ. Given two simple workflows W_1 and W_2, let $W_1 \Rightarrow_f^* W_2$ denote that W_2 can be derived from W_1 by applying a sequence of zero or more productions, and f is a bijection that maps initial input ports and final output ports from W_1 to W_2. Again, f may be omitted for simplicity.

Definition 4. **(Context-Free Workflow Grammar)** A *context-free workflow grammar* (abbr. *workflow grammar*) is $G = (\Sigma, \Delta, S, P)$, where Σ is a finite set of modules, $\Delta \subseteq \Sigma$ is a set of *composite modules* (then $\Sigma \setminus \Delta$ is the set of *atomic modules*), $S \in \Sigma$ is a *start module*, and $P = \{M \rightarrow W \mid M \in \Delta, W \in \Sigma^*\}$ is a finite set of *workflow productions*. The *language* of G is $L(G) = \{R \in (\Sigma \setminus \Delta)^* \mid S \Rightarrow^* R\}$.

Example 5. *Our running example of a workflow grammar G is shown in Figure 2. Composite modules are indicated by uppercase letters and atomic modules by lowercase letters. Formally, $G = (\Sigma, \Delta, S, P)$, where $\Sigma = \{S, A, B, \ldots, E, a, b, \ldots, f\}$, $\Delta = \{S, A, B, \ldots, E\}$, and $P = \{p_1 = S \rightarrow W_1, p_2 = A \rightarrow W_2, p_3 = A \rightarrow W_3, p_4 = B \rightarrow W_4, p_5 = C \rightarrow W_5, p_6 = D \rightarrow W_6, p_7 = D \rightarrow W_7, p_8 = E \rightarrow W_8\}$. Note that p_2 and p_4 form a recursion between A and B. p_6 forms a self-recursion over D, and along with p_7, indicates a loop (sequential execution) over f.*

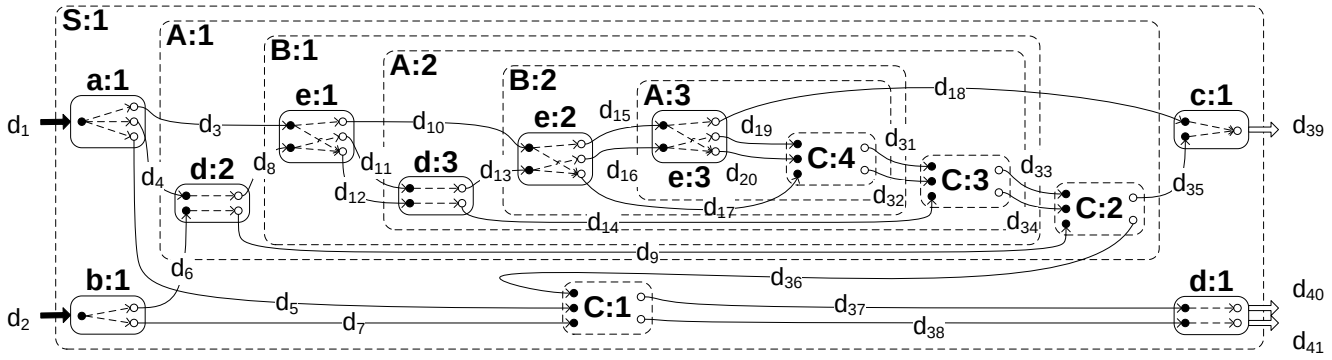

Figure 3: Workflow Run

One possible simple workflow run $R \in L(G)$ is shown in Figure 3, where the atomic modules in R are denoted by solid boxes, and the composite modules that are created during the derivation of R are denoted by dashed boxes. We create a unique id for each atomic and composite module in R by appending a distinct number to the module name. d_1, d_2, ..., d_{41} are unique ids for data items (data edges) in R. For sake of illustration, we omit details of $C:1$, $C:2$ and $C:3$, and show details of $C:4$ in Figure 4. Observe that R can be derived from S by applying a sequence of productions p_1, p_2, p_4, p_2, p_4, p_3, p_5, p_6, p_6, p_7, p_8, ...

The classical notion of *proper* context-free string grammars can also be introduced in our workflow setting.

Definition 5. **(Proper Workflow Grammar)** A workflow grammar $G = (\Sigma, \Delta, S, P)$ is said to be *proper* if it has (1) no *underivable* composite modules: $\forall M \in \Delta$, $\exists W \in \Sigma^*$, $S \Rightarrow^* W$ and W contains M; (2) no *unproductive* composite modules: $\forall M \in \Delta$, $\exists W \in (\Sigma \setminus \Delta)^*$, $M \Rightarrow^* W$; and (3) no *cycles*: $\nexists M \in \Delta$, $M \Rightarrow^* M$ (by at least one step).

One can easily show that any workflow grammar can be transformed into a proper one with exactly the same language. Without loss of expressive power, we assume that workflow grammars (and their views) are always proper.

So far we consider only workflow structure – the way in which modules are connected to construct workflows. Next, we enrich the model by defining fine-grained dependencies between inputs and outputs of atomic modules. Naturally, we assume that every input contributes to at least one output; and every output depends on at least one input.

Definition 6. **(Dependency Assignment)** Given a finite set Σ of modules, a *dependency assignment* to Σ is a function λ that, for each module $M = (I, O) \in \Sigma$, defines a set $\lambda(M)$ of *dependency edges* from I to O, such that $\forall i \in I$, $\exists o \in O$, $(i, o) \in \lambda(M)$; and $\forall o \in O$, $\exists i \in I$, $(i, o) \in \lambda(M)$.

Finally, combining all the above components, our fine-grained workflow model is formalized as follows.

Definition 7. **(Fine-Grained Workflow Model)** A *workflow specification* is G^λ, where $G = (\Sigma, \Delta, S, P)$ is a workflow grammar and λ is a proper dependency assignment to $\Sigma \setminus \Delta$. The set of all *workflow runs* w.r.t. G^λ is $L(G^\lambda) = \{R^\lambda \mid R \in L(G)\}$, where R^λ is obtained from R by adding to each module M in R a set $\lambda(M)$ of dependency edges.

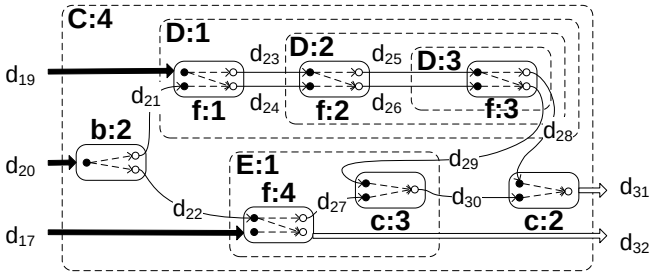

Figure 4: Details of Composite Module C:4

EXAMPLE 6. *For the grammar G in Figure 2, we define a dependency assignment λ to all atomic modules (i.e., a, b, ..., f). The dependency edges introduced by λ are shown in Figure 2 as dashed edges from input ports to output ports of atomic modules. With both data (solid) and dependency (dashed) edges, Figures 3 and 4 represent a run $R^\lambda \in L(G^\lambda)$.*

In Section 3, we will compare our fine-grained model (i.e., with white-box dependencies) to the existing coarse-grained model (i.e., with black-box dependencies) [5, 6]. Both are grammar-based, but the coarse-grained model is less expressive, and captures only a subclass of fine-grained workflows.

Definition 8. **(Coarse-Grained Workflows)** A workflow specification G^λ is said to be *coarse-grained* if (1) λ is defined such that for any atomic module, every output depends on every input; and (2) every simple workflow used by G has a single source module and a single sink module [3].

2.2 Views with Grey-Box Dependencies

Rather than showing all users full details of a workflow, one can authorize different groups of users to access different *workflow views*. We start with a simple form of views, and will extend to more general types of views in Section 5. Our view is constructed over a specification and then projected onto its runs. Such approach is common in workflows [7, 20, 9] (unlike typical database views that are defined via queries), but our work is the first to be based on a fine-grained model. Formally, a view is defined by two components. One describes the structure of a view by restricting

[3]The second restriction is used to ensure that even for composite modules, every output depends on every input.

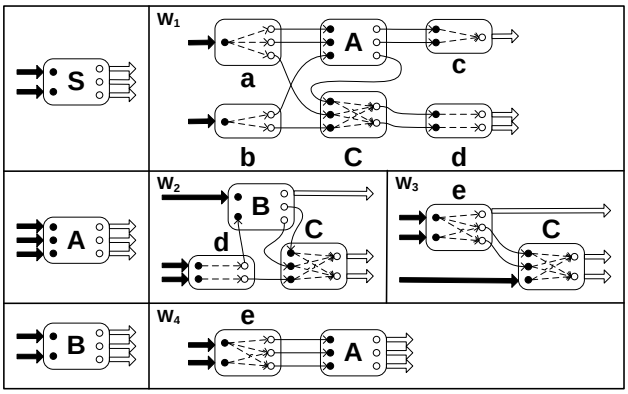

Figure 5: View of Workflow Specification

the possible expansions of workflow hierarchy to a subset of composite modules. The other specifies the "perceived" fine-grained dependencies between inputs and outputs of all unexpandable modules in this view. As mentioned in Section 1, for *abstraction views*, the perceived dependencies always reflect the true dependencies, which we call *white-box dependencies*. In contrast, for *security views*, false dependencies may be introduced in order to hide private provenance information, which we call *grey-box dependencies*.

Definition 9. **(Workflow View)** Give a workflow specification $G^\lambda = (\Sigma, \Delta, S, P)^\lambda$, a *view* over G^λ is defined by a pair (Δ', λ') , where $\Delta' \subseteq \Delta$ is a subset of composite modules and λ' is a new dependency assignment for $\Sigma \setminus \Delta'$. In particular, (Δ, λ) is said to be the *default view* over G^λ.

REMARK 1. *As will be seen in Section 3.1, from the input-output dependencies of atomic modules, we can compute those of composite modules. We thus say that a view* (Δ', λ') *has* white-box dependencies*, if* λ' *defines the same dependencies as* λ *does, otherwise, it has* grey-box dependencies*.*

A view $U = (\Delta', \lambda')$ defined over a specification G^λ produces a new grammar, denoted $G_{\Delta'}$, by restricting G to the subset of productions for composite modules in Δ'. Together with λ', it defines a new specification, denoted $G_U = (G_{\Delta'})^{\lambda'}$, which we call a *view of this specification*. Similarly, given a run $R^\lambda \in L(G^\lambda)$, by restricting the derivation of R to only productions for composite modules in Δ' and using λ', we obtain a *view of this run*, denoted $R_U = (R_{\Delta'})^{\lambda'}$.

Note that Δ' may contain underivable modules in $G_{\Delta'}$. We thus say that Δ' or (Δ', λ') is *proper* if $G_{\Delta'}$ is proper. Without loss of generality, we consider only proper views.

EXAMPLE 7. *Using the specification* G^λ *in Figure 2, we define a view* $U = (\Delta', \lambda')$*, where* $\Delta' = \{S, A, B\}$*. The new grammar* $G_{\Delta'}$ *is shown in Figure 5, which contains only the productions for* S*,* A *and* B*. Note that* C *is treated as an atomic module in this view, which makes* D*,* E *and* f *underivable. Therefore,* λ' *needs to be defined for only atomic modules* a*,* b*,* c*,* d*,* e *and* C*. The dependency edges introduced by* λ' *are shown in Figure 5 as dashed edges. Comparing with* λ *defined in Figure 2, we observe that* $\lambda'(C)$ *is newly defined,* $\lambda'(e)$ *is changed, and others are unchanged. Hence, this view introduces grey-box dependencies.*

We project this view onto the run R^λ *in Figures 3 and 4. Since* C *is treated as atomic, details of* $C:1$*,* $C:2$*,* $C:3$ *and* $C:4$ *(Figure 4) are hidden and* $R_{\Delta'}$ *has exactly the structure in Figure 3. However, all the dependency edges for* $R_{\Delta'}$ *should be given according to* λ' *as in Figure 5.*

In the rest of this paper, we may simply denote a specification by G and a run by R, since the original dependency assignment λ is irrelevant to views (i.e., overwritten by λ').

2.3 View-Adaptive Dynamic Labeling

This paper focuses on the problem of efficiently answering graph reachability queries over views of workflow runs with grey-box dependencies. Specifically, the type of queries that we consider are those which ask if one data item depends on the other, that is, if one data edge is reachable from the other. Note that due to the flase dependencies in grey boxes, the answer to whether one data item depends on another is up to the view through which the query is being asked. Such example is given below. It captures the hiding of provenance information that one intended when defining the views.

EXAMPLE 8. *Let* $U_1 = (\Delta, \lambda)$ *be the default view over the specification in Figure 2, and* $U_2 = (\Delta', \lambda')$ *be the view given in Example 7. Consider two data items* d_{17} *(an input of* $C:4$*) and* d_{31} *(an output of* $C:4$*) in Figure 3. Note that the details of* $C:4$ *in Figure 4 are hidden in* U_2*. For the query that asks whether* d_{31} *depends on* d_{17}*, the answer is "no" for* U_1*, since* d_{31} *is not reachable from* d_{17} *in Figure 4, but the answer is "yes" for* U_2*, since* $\lambda'(C)$ *is defined in Figure 5 such that every output depends on every input.*

We start with the basic dynamic labeling problem. The goal is to assign each data item a *reachability label* as soon as it is produced (*dynamically*) such that using only the labels of any two data items, we can quickly decide if one depends on the other. Two different but related dynamic labeling problems were formulated in [5]. In the *execution-based* problem, atomic modules of a run are generated one-by-one according to some topological ordering. In the *derivation-based* problem, a run is derived from the start module by applying a sequence of productions. As observed in [5], any solution for the former also provides a solution for the latter. We thus focus only on the derivation-based problem.

Definition 10. [5] **(Dynamic Labeling)** A *dynamic labeling scheme* for a given specification G^λ is (ϕ, π), where ϕ is a labeling function and π is a binary predicate. ϕ takes as input a derivation of a run $R^\lambda \in L(G^\lambda)$, that is, a sequence of productions that transform the start module S to R. Initially, ϕ assigns a label $\phi(d)$ to each input and output d of S. In the ith step of the derivation, ϕ assigns a label $\phi(d)$ to each new data item d introduced by the ith production. Note that we do not know the production sequence in advance, but receive them online. The assigned labels cannot be modified subsequently. ϕ and π are such that for any derivation of a run $R^\lambda \in L(G^\lambda)$ and any two data items d_1 and d_2 in R^λ, $\pi(\phi(d_1), \phi(d_2)) = \texttt{true}$ iff d_2 depends on d_1.

In contrast to the previous work [5], this paper studies the dynamic labeling problem in more general and useful workflow settings. Specifically, we consider (1) *fine-grained input-output dependences* and (2) *views with grey-box dependencies*. Both ingredients entail new challenges, which will be addressed in Sections 3 and 4, respectively.

To handle views, we propose in Section 4 a novel *view-adaptive* labeling approach whereby view specifications are labeled *statically* (i.e., as they are created), whereas data items are labeled *dynamically* as they are produced during a workflow execution. At query time, the label of the view over which the query is asked is combined with the labels of relevant data items to provide the correct answer. In this framework, since data labels are unrelated to any view (*view-adaptive*), views can be added/deleted/modified without having to touch the data. It is both space-efficient and time-efficient relative to the alternative approach where data items are labeled repeatedly and separately for each view.

Definition 11. **(View-Adaptive Dynamic Labeling)** A *view-adaptive* dynamic labeling scheme for a given specification G is (ϕ_r, ϕ_v, π), where ϕ_r is a labeling function for runs, ϕ_v is a labeling function for view specifications, and π is a ternary predicate. Given a derivation of a run $R \in L(G)$, ϕ_r as before assigns a label $\phi_r(d)$ (called *data label*) to each data item d as soon as it is produced during the derivation of R. Given a view U over G, ϕ_v treats U as one object and assigns a label $\phi_v(U)$ (called *view label*). ϕ_r, ϕ_v and π are such that for any derivation of a run $R \in L(G)$, any view U over G and any two data items d_1 and d_2 in R_U, $\pi(\phi_r(d_1), \phi_r(d_2), \phi_v(U)) = \texttt{true}$ iff d_2 depends on d_1 w.r.t. U.

A (view-adaptive) dynamic labeling scheme is said to be *compact* if for any derivation of a run with n data items, it creates data labels of $O(\log n)$ bits. Since just assigning unique ids requires labels of $\log n$ bits, such a scheme creates shortest possible data labels up to a constant factor.

3. FEASIBILITY OF DYNAMIC LABELING

To address the challenges brought by fine-grained dependencies, we first consider the basic dynamic labeling problem (see Definition 10), where there is only one default view defined over the specification. Note that the labels created for the default view also work for other views with white-box dependencies, but not those with grey-box dependencies.

As a formal analysis, we present in this section a classification of fine-grained workflows based on the feasibility of developing (compact) dynamic labeling schemes. In Section 3.1, we first identify a class of *safe* workflows, and show that they are the largest set of workflows that allow dynamic labeling schemes. In Section 3.2, we further identify a class of *strictly linear-recursive* workflow structures for which dynamic, yet *compact* labeling schemes are possible. Polynomial-time algorithms are also given to decide if a workflow is safe or if its structure is strictly linear-recursive.

The previous work [5] studied coarse-grained workflows (see Definition 8) which belong to a subclass of safe workflows. However, our results show that the common restriction on the workflow structure, which sufficed to reduce the label length for black-box dependencies, are no longer helpful. This formally proves the difficulty introduced by fine-grained dependencies to the dynamic labeling problem.

3.1 Safe Workflows

Some workflows cannot be labeled on-the-fly even if arbitrary label size is allowed. We illustrate by an example.

EXAMPLE 9. *Consider a simple specification shown in Figure 6 with two productions $S \to a$ and $S \to b$, where S is the start module, and a and b are two atomic modules. The dependency assignment to a and b is shown in Figure 6 as dashed edges. d_1 and d_2 are an input and an output data item of S, respectively. Observe that if $S \to a$ is applied, then d_2 depends on d_1; otherwise (if $S \to b$ is applied), d_2 does not depend on d_1. Recall from Definition 10 that the labels for d_1 and d_2 must be assigned before we see the production, and cannot be modified subsequently. Therefore, no dynamic labeling schemes exist for this example.*

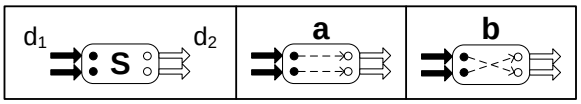

Figure 6: Unsafe Workflow

In general, if two simple workflows with only atomic modules can be derived from the same composite module, and they are *inconsistent*, in the sense that they have different dependencies between initial inputs and final outputs, then dynamic labeling is impossible for this specification. Such workflows are said to be *unsafe*, and the others are *safe*.

Definition 12. **(Consistent Simple Workflow)** Let W_1 and W_2 be two simple workflows. Let λ be a dependency assignment to all the modules in W_1 and W_2. Let f be a bijection that maps initial input ports and final output ports from W_1 to W_2. Then W_1 is said to be *consistent* with W_2 w.r.t. λ and f, if for any initial input port i of W_1 and any final output port o of W_1, o is reachable from i in W_1^λ iff $f(o)$ is reachable from $f(i)$ in W_2^λ.

Definition 13. **(Safe Workflow)** A workflow specification $G^\lambda = (\Sigma, \Delta, S, P)^\lambda$ is said to be *safe* if $\forall M \in \Delta$ and $W_1, W_2 \in (\Sigma \setminus \Delta)^*$ such that $M \Rightarrow_{f_1}^* W_1$ and $M \Rightarrow_{f_2}^* W_2$, W_1 is consistent with W_2 w.r.t. λ and $f_1^{-1} \circ f_2$. In addition, a dependency assignment λ is said to be *safe* if G^λ is safe; and a view U is said to be *safe* if G_U is safe.

REMARK 2. *Safety is a natural restriction on fine-grained workflows. It essentially says that for any module, either atomic or composite, the dependences between inputs and outputs are deterministic, in the sense that they can be predicted from the specification, and are consistent among all possible executions. In particular, by Definition 8, any coarse-grained workflow (i.e., with black-box dependencies) is always safe. Moreover, it is important to notice that from the perspective of data provenance, the output of an aggregate function depends on each of its inputs [3], even though the output may take the value from only one of its inputs (e.g., "max" or "min" functions). Therefore, a workflow that use those aggregate functions as modules is still safe.*

Our first result shows that safety characterizes the feasibility of dynamic labeling for fine-grained workflows.

THEOREM 1. *Given any workflow specification G^λ, there is a dynamic labeling scheme for G^λ iff G^λ is safe.*

PROOF. (Sketch) By Definition 13, unsafe workflows do not allow any dynamic labeling schemes. On the other hand, the view-adaptive dynamic labeling scheme, which we will

present in Section 4, can be modified to label arbitrary safe workflows, though it may create linear-size data labels.

"if" direction. We modify the adaptive dynamic labeling scheme presented in Section 4 as follows. Instead of using compressed parse trees, we simply use basic parse trees to label the runs. Note that our modified adaptive scheme can be applied to label arbitrary workflow grammar G with safe views, but may create linear-size data labels. We denote it by (ϕ_r, ϕ_v, π), and convert it to a basic dynamic labeling scheme (ϕ', π') as follows. Let $U = (\Delta, \lambda)$ be the default view over G^λ. For any data item d, $\phi'(d) = (\phi_r(d), \phi_v(U))$. For any two data items d_1 and d_2, $\pi'(\phi'(d_1), \phi'(d_2)) = \pi(\phi_r(d_1), \phi_r(d_2), \phi_v(U))$.

"only if" direction. We prove by contradiction. Suppose G^λ is unsafe. By Definition 13, there is a composite module M and two simple workflows W_1 and W_2 with only atomic modules such that $W \Rightarrow^* W_1$ and $M \Rightarrow^* W_2$. Moreover, W_1 is inconsistent with W_2 w.r.t. λ. By Definition 12, we assume without loss of generality that there is an input data item d_1 and an output data item d_2 of M such that if the sequence of productions that transform M to W_1 is applied, then d_2 depends on d_1; otherwise (if the sequence of productions that transform M to W_2 is applied), d_2 does not depend on d_1. However, by Definition 10, the data labels for d_1 and d_2 must be assigned before we see the production sequence, and cannot be modified subsequently. Therefore, no dynamic labeling schemes exist for G^λ. □

THEOREM 2. *Deciding if a given workflow specification G^λ is safe can be done in polynomial time.*

PROOF. We describe a polynomial-time algorithm to decide if a given workflow specification is safe. The idea is that if the specification is safe, then one can extend the given dependency assignment to composite modules in such a way that all the productions are consistent. More precisely,

LEMMA 1. **(Full Assignment)** *A workflow specification $G^\lambda = (\Sigma, \Delta, S, P)^\lambda$ is safe iff there is a unique dependency assignment λ^* to Σ (called the* full dependency assignment*) such that (1) $\forall M \in \Sigma \setminus \Delta$, $\lambda^*(M) = \lambda(M)$; and (2) $\forall M \rightarrow_f W \in P$, M is consistent with W w.r.t. λ^* and f.*

Given a specification G^λ, the algorithm starts by defining $\lambda^*(M) = \lambda(M)$ for each atomic module M, and proceeds to define λ^* for composite modules. We say that a production $M \rightarrow_f W$ is *verifiable* if λ^* is already defined for all the modules in W. We first find such a production, and compute $\lambda^*(M)$ as follows. For any input port i of M and any output port o of M, $(i, o) \in \lambda^*(M)$ iff $f(o)$ is reachable from $f(i)$ in W^{λ^*}. If $\lambda^*(M)$ is already defined, we also check if the new $\lambda^*(M)$ is consistent with the old one. If not, the algorithm reports that G^λ is unsafe and terminates. Otherwise, the algorithm continues to verify other productions. Since G is proper, by Definition 7, all the composite modules are productive. Therefore, there always exists a new verifiable production during this process. The algorithm finally reports that G^λ is safe, if λ^* is successfully defined for all composite modules, and outputs λ^* as a by-product.

The correctness of our algorithm follows from Lemma 1. We now analyze the time complexity. A simple way to implement this algorithm is to use a queue q to maintain all unverified productions. Initially, q contains all the productions. We always retrieve the first production p in q. If p has not been verified, we check if p is verifiable, that is, if λ^* is already defined for all the modules produced by p. Every time a new production $p = M \rightarrow W$ is verified, if $\lambda^*(M)$ is newly defined, we append to q all unverified productions that produce M. Let $|p|$ be the size of a production $p = M \rightarrow W$, that is, the total size of M and W. Then checking if p is verifiable takes $O(|p|)$ time, which needs to be done at most $|p|$ times, because every production p will be added to q at most $|p|$ times. In addition, computing $\lambda^*(M)$ for p takes $O(|p|^2)$ time, which needs to be done only once. So the total time complexity is $\sum O(|p|^2) = O(|G^\lambda|^2)$, where $|G^\lambda|$ is the size of the given specification. □

PROOF OF LEMMA 1. **"if" direction.** Let M be any composite module M, and W_1 and W_2 be any two simple workflows with only atomic modules such that $M \Rightarrow^*_{f_1} W_1$ and $M \Rightarrow^*_{f_2} W_2$. Since all the productions are consistent w.r.t. λ^*, by Definition 12, for any input port i of M and any output port o of M, $f_1(o)$ is reachable from $f_1(i)$ in W_1 iff $(i, o) \in \lambda^*(M)$ iff $f_2(o)$ is reachable from $f_2(i)$ in W_2. Thus, W_1 is consistent with W_2 w.r.t. λ^* and $f_1^{-1} \circ f_2$. Given that λ and λ^* agree on atomic modules, W_1 is also consistent with W_2 w.r.t. λ and $f_1^{-1} \circ f_2$. By Definition 13, G^λ is safe.

"only if" direction. We define $\lambda^*(M)$ for any composite module M as follows. Find an arbitrary simple workflow W with only atomic modules such that $M \Rightarrow^*_f W$. For any input port i of M and any output port o of M, $(i, o) \in \lambda^*(M)$ iff $f(o)$ is reachable from $f(i)$ in W^λ. Since G is proper, by Definition 5, M must be productive. So there is at least one such W. Moreover, since G^λ is safe, by Definition 13, all possible W's are consistent w.r.t. λ. Thus, $\lambda^*(M)$ is well-defined and unique. On the other hand, for every production $M \rightarrow_f W$, let W' be an arbitrary simple workflow with only atomic modules such that $W \Rightarrow^*_{f'} W'$, then $M \Rightarrow^*_{f \circ f'} W'$. By the definition of λ^*, for any input port i of M and any output port o of M, $(i, o) \in \lambda^*(M)$ iff $f \circ f'(o)$ is reachable from $f \circ f'(i)$ in $(W')^\lambda$ iff $f(o)$ is reachable from $f(i)$ in W^{λ^*}. Hence, M is consistent with W w.r.t. λ^* and f. □

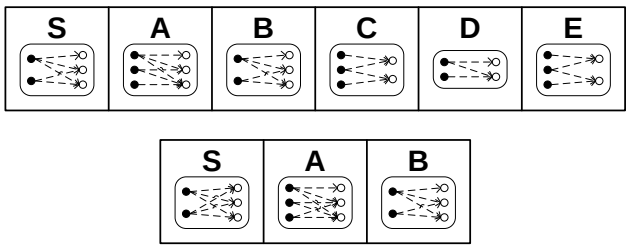

Figure 7: Full Dependency Assignment

EXAMPLE 10. *We illustrate the above algorithm using the specification G^λ in Figure 2. Initially, both $p_7 = D \rightarrow W_7$ and $p_8 = E \rightarrow W_8$ are verifiable. We compute $\lambda^*(D)$ and $\lambda^*(E)$ by p_7 and p_8. Once $\lambda^*(D)$ and $\lambda^*(E)$ are defined, $p_5 = C \rightarrow W_5$ and $p_6 = D \rightarrow W_6$ become verifiable. We compute $\lambda^*(C)$ by p_6, and verify that $\lambda^*(D)$ computed by p_6 is consistent with the one computed before by p_7. We continue this process until all the productions are verified. Hence, G^λ is safe, and λ^* is shown on the top of Figure 7.*

Similarly, one can verify that the view $U = (\Delta', \lambda')$ defined in Example 7 is safe using Figure 5. The full dependency assignment for U is shown on the bottom of Figure 7. Comparing the two full assignments in Figure 7, while B gets the same dependencies, the ones for S and A are different.

LEMMA 2. *Any coarse-grained workflow is safe.*

PROOF. It easily follows from Definition 8 that for any coarse-grained specification G^λ, every output depends on every input for both atomic and composite modules. Therefore, the full dependency assignment λ^* can be defined such that for any (either atomic or composite) module $M = (I, O)$, $\lambda^*(M) = I \times O$. By Lemma 1, G^λ is safe. □

3.2 Linear-Recursive Workflow Structures

For safe workflows, we further examine the feasibility of developing compact dynamic labeling schemes. First of all, we prove a negative result.

THEOREM 3. *There is a safe workflow specification G^λ such that any dynamic labeling scheme for G^λ requires linear-size data labels. More precisely, for any dynamic labeling scheme (ϕ, π) for G^λ, there is a derivation of a workflow run $R^\lambda \in L(G^\lambda)$ with n data items such that ϕ assigns a data label of $\Omega(n)$ bits to some data item in R^λ.*

PROOF. A negative result in [5] shows that there is a coarse-grained workflow that does not allow any compact dynamic labeling scheme. By Lemma 2, any coarse-grained workflow is safe. The theorem follows. Note that the theorem also follows from Theorems 5 and 6 given later. □

Given this, our next goal is to identify safe workflows that enable compact dynamic labeling. [5] gives an elegant characterization for coarse-grained workflows (Theorem 4).

Definition 14. [5] **(Linear-Recursive Workflow Grammar)** A workflow grammar $G = (\Sigma, \Delta, S, P)$ is said to be *linear-recursive* if $\forall M \in \Delta$ and $W \in \Sigma^*$ such that $M \Rightarrow^* W$, W has at most one instance of M.

THEOREM 4. *[5] Given any coarse-grained workflow specification G^λ, there is a compact dynamic labeling scheme for G^λ iff G is a linear-recursive workflow grammar.*

Note that this does not solve our problem, because coarse-grained workflows are a restricted class of safe workflows in the fine-grained model. In this paper, we first show that the "only if" direction can be generalized to all safe workflows.

THEOREM 5. *Given any safe workflow specification G^λ, there is a compact dynamic labeling scheme for G^λ only if G is a linear-recursive workflow grammar.*

PROOF. Note that linear recursion describes only the structure of a workflow, while safety is a property of the entire specification. Therefore, to prove the theorem, we need to show that given any nonlinear-recursive workflow grammar G, for any safe dependency assignment λ, any dynamic labeling scheme for G^λ requires linear-size data labels.

Since G is nonlinear-recursive, by Definition 14, there is a compose module M and a simple workflow W such that $M \Rightarrow^* W$ and W has at least two instances of M. Since G is proper, by Definition 5, every composite module in W is productive. So we can transform W to a simple workflow W_1 such that $M \Rightarrow^* W_1$ and W_1 has only atomic modules except for two instances of M. Similarly, we can transform W to a simple workflow W_2 such that $M \Rightarrow^* W_2$ and W_2 has only atomic modules. In addition, since G is proper, by Definition 5, M is derivable. So we can transform the start module S to a simple workflow W_0 such that $S \Rightarrow^* W_0$ and W_0 has only atomic modules except for one instance of M. We introduce three new productions: $p_0 = S \to W_0$, $p_1 = M \to W_1$ and $p_2 = M \to W_2$, as shown in Figure 8.

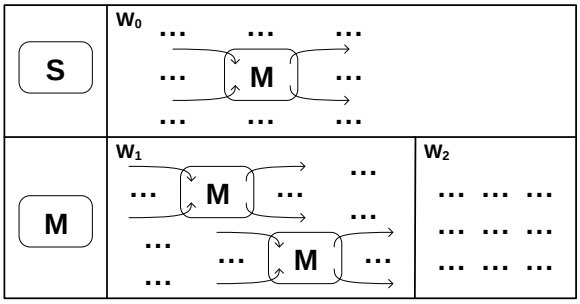

Figure 8: Workflow Grammar in Proof of Theorem 5

For any $k \geq 0$, let $L_k(G)$ denote the set of all simple workflows with only atomic modules that can be derived from S by applying p_0 only once as the first production and applying p_1 a total of k times. In addition, only p_0, p_1 and p_2 are used. Therefore, p_2 must be applied exactly $k + 1$ times, but may be interleaved with p_1.

Let λ be any given safe dependency assignment. For any dynamic labeling scheme D for G^λ, we define $F(D, k)$ to be the set of all data labels assigned by D to data items that are created after the first production p_0 of any derivation of a workflow run $R^\lambda \in L_k(G^\lambda)$. Moreover, let $N(k)$ be the minimum of $|F(D, k)|$ over all possible D's.

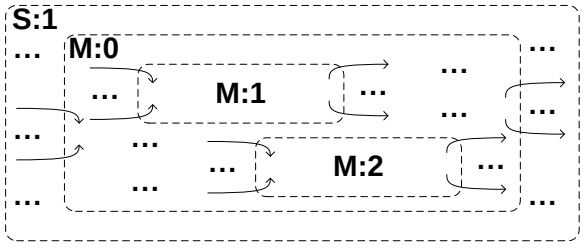

Figure 9: Workflow Run in Proof of Theorem 5

We next prove that $N(k + 1) \geq 2N(k)$, for any $k \geq 0$. Given any dynamic labeling scheme $D = (\phi, \pi)$ for G^λ, we use D to label a derivation of a workflow run $R^\lambda \in L_{k+1}(G^\lambda)$. Suppose that the first two productions p_0 and p_1 are applied, as shown in Figure 9. By Definition 10, all the data items created so far have already been labeled. We refer to the two instances of M in W_1 as $M : 1$ and $M : 2$. Let R_1^λ and R_2^λ be two upcoming simple workflows that are derived from $M : 1$ and $M : 2$, respectively. Let F_1 and F_2 be two sets of data labels that are reserved for all upcoming data items created in R_1^λ and R_2^λ, respectively.

We first show that F_1 and F_2 are disjoint. Consider any two upcoming data items d_1 and d_2 created in R_1^λ and R_2^λ,

respectively. Let $l_1 = \phi(d_1) \in F_1$ and $l_2 = \phi(d_2) \in F_2$. By Definition 2, W_1 is acyclic in the sense that there is no cycle among the modules. Without loss of generality, we assume that $M : 1$ is not reachable from $M : 2$. Therefore, for any input port i_1 of $M : 1$ and any output port o_2 of $M : 2$, i_1 is not reachable from o_2 in W_1^λ. This further implies that d_1 does not depend on any input data item of R_2^λ. On the other hand, by Definition 6, d_2 must depend on at least one input data item, say d_0, of R_2^λ. Moreover, d_0 is already labeled before d_1 and d_2 are produced. Let $l_0 = \phi(d_0)$, then $\pi(l_0, l_1) = \texttt{true}$, but $\pi(l_0, l_2) = \texttt{false}$. Hence, $l_1 \neq l_2$.

Since both R_1 and R_2 can be an arbitrary simple workflow that is derived from M by applying p_1 a total of k times, $|F_1| \geq N(k)$ and $|F_2| \geq N(k)$. So $|F(D, k+1)| \geq |F_1| + |F_2| \geq 2N(k)$. Note that this holds for any possible scheme D for G^λ. Hence, $N(k+1) \geq 2N(k)$, for any $k \geq 0$.

For any $0 \leq i \leq 2$, suppose W_i has x_i data edges, that is, an application of p_i produces x_i new data items. Note that $x_1 \geq 1$, otherwise, M and W_1 have different number of inputs and outputs. So $N(1) \geq 1$. We can prove by induction that $N(k) \geq 2^{k-1}$, for any $k \geq 1$. Therefore, any dynamic labeling scheme D for G^λ must assign a data label of at least $k-1$ bits to some data item in some $R^\lambda \in L_k(G^\lambda)$. Suppose R^λ has a total of n data items, and initially the start module S has a total of x_s inputs and outputs, then $n = x_s + x_0 + k * x_1 + (k+1) * x_2$, where x_s, x_0, x_1 and x_2 are constants. So $k - 1 = \Omega(n)$. The theorem follows. □

Unfortunately, it turns out that the "if" direction of Theorem 4 does not extend to all safe workflows.

THEOREM 6. *There is a linear-recursive workflow grammar G and a safe dependency assignment λ such that any dynamic labeling scheme for G^λ requires linear-size data labels.*

PROOF. Consider the example of G^λ shown in Figure 10. S is the start module, and a, b, c are atomic modules. G has three productions $p_a = S \to W_a$, $p_b = S \to W_b$ and $p_c = S \to W_c$, which produce a, b and c, respectively. The dependency assignment λ to a, b, c is shown as dashed edges. It is easy to check that G is linear-recursive and λ is safe.

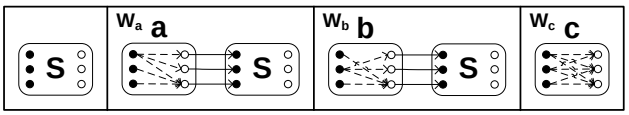

Figure 10: Workflow Grammar in Proof of Theorem 6

Both p_a and p_b produce three new data items (data edges). We focus only on the first two on the top, which are referred to as *good* data items. For any $k \geq 0$, let $L_k(G^\lambda)$ denote the set of workflow runs that are derived from S by applying p_a and p_b a total of k times. Note that p_c must be applied only once as the last production. For any dynamic labeling scheme D for G^λ, we define $F(D, k)$ to the set of all data labels assigned by D to good data items that are created by any derivation of a run $R^\lambda \in L(G_k^\lambda)$. Let $N(k)$ be the minimum of $|F(D, k)|$ over all possible D's.

We next prove that $N(k+1) \geq 2N(k)$, for any $k \geq 0$. Given any dynamic labeling scheme $D = (\phi, \pi)$ for G^λ, we use D to label a derivation of a run $R^\lambda \in L_{k+1}(G^\lambda)$. Let d be the first input data item of $S : 1$, as shown in Figure 11. By Definition 10, d is labeled initially. Let $l = \phi(d)$. If p_a is applied as the first production (see the left of Figure 11), let R_a^λ be the upcoming simple workflow that will be derived from $S : 2$, and let F_a be the set of data labels that are reserved for all upcoming good data items created in R_a^λ. Otherwise, if p_b is applied as the first production (see the right of Figure 11), we can define R_b^λ and F_b similarly.

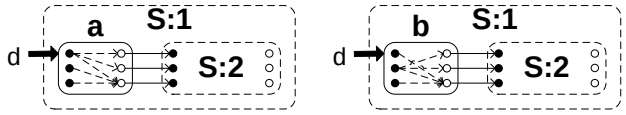

Figure 11: Workflow Run in Proof of Theorem 6

We first show that F_a and F_b are disjoint. Consider any two upcoming good data items d_a and d_b created in R_a^λ and R_b^λ, respectively. Let $l_a = \phi(d_a) \in F_a$ and $l_b = \phi(d_b) \in F_b$. Clearly, d_a must depend on d, but d_b does not depend on d. So $\pi(d, d_a) = \texttt{true}$ but $\pi(d, d_b) = \texttt{false}$. Hence, $l_a \neq l_b$.

Since both R_a^λ and R_b^λ can be an arbitrary run in $L_k(G^\lambda)$, $|F_a| \geq N(k)$ and $|F_b| \geq N(k)$. So $|F(D, k+1)| \geq |F_a| + |F_b| \geq 2N(k)$. Note that this holds for any possible scheme D for G^λ. Hence, $N(k+1) \geq 2N(k)$, for any $k \geq 0$.

Clearly, $N(1) \geq 1$. We can prove by induction that $N(k) \geq 2^{k-1}$, for any $k \geq 1$. Therefore, any possible dynamic labeling scheme D for G^λ must assign a data label of at least $k-1$ bits to some good data item in some $R^\lambda \in L_k(G^\lambda)$. Suppose R^λ has a total of n data items, then $n = 6 + 3k$. So $k - 1 = \Omega(n)$. The theorem follows. □

Theorem 6 tells us that while fine-grained dependencies increase the expressive power of the model, they limit the recursive workflow structure that allows compact dynamic labeling. We thus identify a natural class of *strictly linear-recursive* workflow grammars for which dynamic, yet compact labeling is feasible for any safe dependency assignment. To define them, we introduce a *production graph* that describes the derivation relationship between modules.

Definition 15. **(Production Graph)** Given a workflow grammar $G = (\Sigma, \Delta, S, P)$, the *production graph* of G is a directed multigraph $\mathcal{P}(G)$ in which each vertex denotes a unique module in Σ. For each production $M \to W$ in P and each module M' in W, there is an edge from M to M' in $\mathcal{P}(G)$. Note that if W has multiple instances of a module M', then $\mathcal{P}(G)$ has multiple parallel edges from M to M'.

Intuitively, every cycle in $\mathcal{P}(G)$ corresponds to a recursion in G. G is said to be *recursive* if $\mathcal{P}(G)$ is cyclic. A module in G is said to be *recursive*, if it belongs to a cycle in $\mathcal{P}(G)$.

Definition 16. **(Strictly Linear-Recursive Workflow Grammar)** A workflow grammar G is said to be *strictly linear-recursive* if all the cycles in $\mathcal{P}(G)$ are vertex-disjoint.

REMARK 3. *Strictly linear recursion is able to capture common recursive patterns that we observed from the myExperiment workflow repository [18]. In particular, consider two common forms of recursion that we encounter in real-life scientific workflows. The first is called the* loop *execution for which a sub-workflow is repeated sequentially a number of times until certain condition is met. The second*

is called the fork *execution for which multiple copies of a sub-workflow are executed in parallel. In scientific workflow systems, such as Taverna [13] and Kepler [2], fork executions are commonly used to model operations over complex data (e.g., "maps" over sets). Both loop and fork executions belong to a simple form of strictly linear recursion.*

It is easy to show that every strictly linear-recursive workflow grammar is also linear-recursive, but not vice versa.

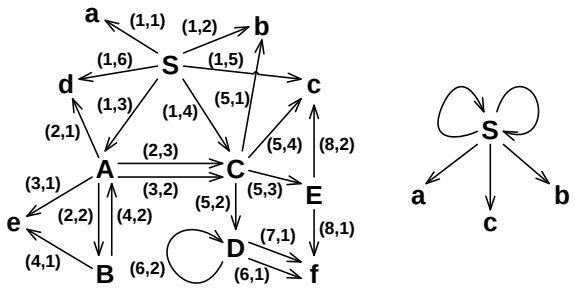

Figure 12: Production Graphs

EXAMPLE 11. *The left part of Figure 12 shows the production graph $\mathcal{P}(G)$ for the grammar G in Figure 2 (ignore number pairs on the edges). Observe that $\mathcal{P}(G)$ has two cycles: one between A and B and the other (self-loop) over D. Since they are vertex-disjoint, by Definition 16, G is strictly linear-recursive. The right part of Figure 12 shows the production graph $\mathcal{P}(G')$ for the linear-recursive grammar G' in Figure 10. Since $\mathcal{P}(G')$ has two self-loops that share S, G' is linear-recursive but not strictly linear-recursive.*

THEOREM 7. *Deciding if a given workflow grammar G is linear-recursive or strictly linear-recursive can be done in polynomial time.*

PROOF. We first describe a polynomial-time algorithm to decide if a given workflow grammar G is strictly linear-recursive. The algorithm is based on Definition 16. We start by building the production graph $\mathcal{P}(G)$ according to Definition 15, and check if $\mathcal{P}(G)$ has two vertex-disjoint cycles as follows. For each vertex v in $\mathcal{P}(G)$, we start a breadth-first search (BFS) from v. If a cycle c_1 is detected, we remove one edge of c_1 from the graph temporarily, and re-start a BFS from v. If a cycle c_2 is still detected, then G is not strictly linear-recursive. Note that searching for c_2 is done for each edge of c_1 independently. If we fail to find two cycles joint at any vertex v in $\mathcal{P}(G)$, then G is strictly linear-recursive.

Similarly, one can decide if a given workflow grammar is linear-recursive in polynomial time by using the production graph. It is based on an equivalent of Definition 14.

LEMMA 3. *A workflow grammar G is linear-recursive iff for any production $M \to W$ in G, M is reachable from at most one module of W in $\mathcal{P}(G)$.*

The algorithm to decide if a given workflow grammar G is linear-recursive straightforwardly follows Lemma 3. Again, we first build the production graph $\mathcal{P}(G)$, and compute the reachability between any pair of vertices in $\mathcal{P}(G)$. Then for each production $p = M \to W$ in G, we check if M is reachable from at most one module of W in $\mathcal{P}(G)$.

The correctness of the above two algorithms easily follows from Definition 16 and Lemma 3, respectively. We analyze their time complexity in terms of the size (i.e., the total number of vertices and edges) of the production graph $\mathcal{P}(G)$, which is linear in the size of the grammar G.

The algorithm to decide if G is strictly linear-recursive performs one BFS to find the first cycle c_1, and for each edge of c_1, performs one BFS to find the second cycle c_2. This is done for each vertex v in $\mathcal{P}(G)$. So it performs at most a total of $(1 + |\mathcal{P}(G)|) * |\mathcal{P}(G)|)$ BFS. Since each BFS takes $O(|\mathcal{P}(G)|)$ time, the total time complexity is $O(|\mathcal{P}(G)|^3)$.

The other algorithm needs to compute the reachability between every pair of vertices in $\mathcal{P}(G)$, which can be done in $O(|\mathcal{P}(G)|^2)$ time. Once this step is done, checking the reachability between any pair of vertices in $\mathcal{P}(G)$ takes only $O(1)$ time. We need to check at most one pair for each edge in $\mathcal{P}(G)$. So the total time complexity is still $O(|\mathcal{P}(G)|^2)$. □

PROOF OF LEMMA 3. **"if" direction.** By Definition 14, it suffices to show that given any composite module M and a simple workflow W such that $M \Rightarrow^* W$, W has at most one module that can reach M in $\mathcal{P}(G)$ [4]. We prove this stronger claim by an induction on the length l of the derivation from M from W. The base case, where $l = 0$ and $W = M$, clearly holds. We prove the inductive case by contradiction. Suppose W has two modules, say m_1 and m_2, that can reach M in $\mathcal{P}(G)$. Note that m_1 and m_2 may be two instances of the same module. Let $M \to W_1$ be the first production of the derivation from M to W. Since M is reachable from at most one module of W_1 in $\mathcal{P}(G)$, by Definition 15, m_1 and m_2 must be derived from the same module, say M', in W_1. Thus, there is a simple workflow W' such that $M' \Rightarrow^* W'$ and W' contains both m_1 and m_2. Moreover, the length of the derivation from M' to W' is at most $l - 1$. Since both m_1 and m_2 can reach M in $\mathcal{P}(G)$, they can also reach M'. By the inductive hypothesis, we obtain a contradiction.

"only if" direction. We prove by contradiction. Suppose G has a production $M \to W$ such that M is reachable from two modules m_1 and m_2 of W in $\mathcal{P}(G)$. Note that m_1 and m_2 may be two instances of the same module. By Definition 15, both m_1 and m_2 can be transformed to a simple workflow with at least one instance of M. Hence, there is a simple workflow W' such that $M \Rightarrow^* W'$ and W' has at least two instances of M, contradicting with Definition 14. □

The main result of this paper is to show that dynamic, yet compact labeling is feasible for strictly linear-recursive grammars with any safe dependency assignment.

THEOREM 8. *Given any strictly linear-recursive workflow grammar G, for any safe dependency assignment λ, there is a compact dynamic labeling scheme for G^λ.*

PROOF. Let (ϕ_r, ϕ_v, π) be the compact adaptive dynamic labeling scheme for G (with safe views) presented in Section 4. We convert it to a basic dynamic labeling scheme (ϕ', π') for G^λ as follows. Let $U = (\Delta, \lambda)$ be the default view over G^λ. For any data item d, $\phi'(d) = (\phi_r(d), \phi_v(U))$. For any two data items d_1 and d_2, $\pi'(\phi'(d_1), \phi'(d_2)) = \pi(\phi_r(d_1), \phi_r(d_2), \phi_v(U))$. Given any derivation of a run $R \in L(G)$ with n data items, for any data item d in R, $\phi_r(d)$ has $O(\log n)$ bits. Moreover, $\phi_v(U)$ is of constant size (for a given G). Hence, $\phi'(d)$ also has only $O(\log n)$ bits. □

The following section describes our labeling scheme.

[4] A vertex is said to be reachable from itself in any directed graph.

4. VIEW-ADAPTIVE DYNAMIC LABELING

This section presents a compact view-adaptive dynamic labeling scheme for strictly linear-recursive workflows with safe views. The rationale behind our label design is explained as follows. Both data labels and view labels encode only partial (but orthogonal) reachability information. More precisely, a data label encodes only a subsequence of the run derivation that creates this data item, while a view label encodes only the fine-grained dependencies that are defined in this view. However, a combination of two data labels and a view label provides the complete information to infer the reachability between the two data items over this view.

We start with a preprocessing step in Section 4.1. Two independent tasks for labeling dynamic runs and labeling safe views are described in Sections 4.2 and 4.3, respectively. Section 4.4 presents how to efficiently answer queries using a combination of data labels and view labels. Finally, Section 4.5 analyzes the quality of our labeling scheme.

4.1 Preprocessing

As a preprocessing step, we assign a pair of numbers to each edge in the production graph. These pairs serve as unique ids for the edges, and will be used later to label runs and views. Let $G = (\Sigma, \Delta, S, P)$ be a strictly linear-recursive grammar and $\mathcal{P}(G)$ be its production graph. First of all, we fix an arbitrary ordering among the productions in P, and for each production $M \to W$, fix an arbitrary topological ordering among the modules in W. Let $p_k = M \to W$ be the kth production in P, and M_i be the ith module in W, then we assign the edge from M to M_i in $\mathcal{P}(G)$ a pair (k, i). Hereafter, we simply refer to this edge as (k, i). In addition, we also fix an arbitrary ordering among all the (vertex-disjoint) cycles in $\mathcal{P}(G)$, and for each cycle, fix an arbitrary edge as the first edge of the cycle. We denote by $\mathcal{C}(s)$ the sth cycle in $\mathcal{P}(G)$ containing a list of number pairs.

EXAMPLE 12. *For the grammar G in Figure 2, the pairs of numbers assigned to the edges in $\mathcal{P}(G)$ are shown in Figure 12. Note that the productions p_1, p_2, ..., p_8 are simply sorted by their subscripts. In Figure 2, all the modules in W_1 are sorted topologically as $a \to b \to A \to C \to c \to d$. Therefore, the edge from S to c in Figure 12 is assigned $(1, 5)$ because $p_1 = S \to W_1$ is the first production, and c is the fifth module in W_1. Moreover, the two cycles in $\mathcal{P}(G)$ are denoted by $\mathcal{C}(1) = \{(2, 2), (4, 2)\}$ and $\mathcal{C}(2) = \{(6, 2)\}$.*

4.2 Labeling Dynamic Runs

Given a derivation of a run $R \in L(G)$, our goal is to assign a data label $\phi_r(d)$ to each data item d in R as soon as it is produced. The labeling is based on a tree representation for runs, which is introduced in Section 4.2.1. The design of data labels is described in Section 4.2.2. Finally, the dynamic labeling algorithm is given in Section 4.2.3.

4.2.1 Tree Representations

We start by describing the *basic parse tree*, and then convert it into a *compressed parse tree* with bounded depth.

Definition 17. **(Basic Parse Tree)** The *basic parse tree* for a run R is an ordered tree $\mathcal{T}'(R)$, where each leaf node of $\mathcal{T}'(R)$ denotes an atomic module of R, and each non-leaf node of $\mathcal{T}'(R)$ denotes a composite module that is generated during the derivation of R. The root node denotes the start module. If a production $M \to W$ is applied, then the children of M in $\mathcal{T}'(R)$ correspond to all the modules of W, and they are ordered by the fixed topological ordering.

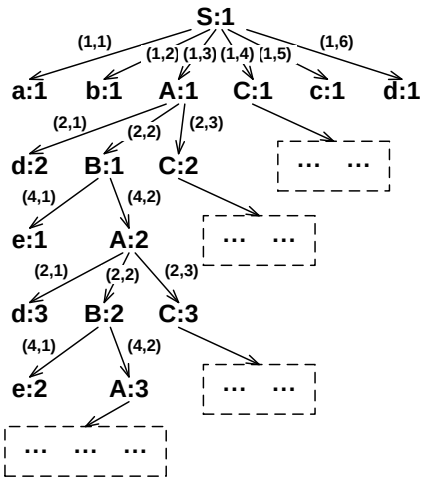

Figure 13: Basic Parse Tree

EXAMPLE 13. *For our running example R given in Figure 3, the basic parse tree $\mathcal{T}'(R)$ is depicted in Figure 13. We denote every node of $\mathcal{T}'(R)$ by a unique module id given in Figure 3, and also label every edge of $\mathcal{T}'(R)$ with an edge id assigned to the production graph $\mathcal{P}(G)$ in Figure 12. Observe that $\mathcal{T}'(R)$ can be generated from $\mathcal{P}(G)$ by unfolding the cycles (recursions) that occur in the derivation of R. So every edge of $\mathcal{T}'(R)$ can be mapped to an edge of $\mathcal{P}(G)$.*

Although the basic parse tree is well-defined for arbitrary recursive workflow grammars, the problem is that its depth can be proportional to the size of the run (we will see later this results in long (linear-size) data labels). Next, we describe an alternative tree representation for (strictly) linear-recursive grammars, called the *compressed parse tree*, whose depth is always bounded by the size of the specification.

Definition 18. **(Compressed Parse Tree)** The *compressed parse tree* for a run R is an ordered tree $\mathcal{T}(R)$, which is converted from the basic parse tree $\mathcal{T}'(R)$ by adding *recursive* nodes. Each of them denotes a linear recursion in the derivation of R. The children of a recursive node in $\mathcal{T}(R)$ correspond to a sequence of nested composite modules of R obtained by unfolding a cycle in the production graph.

Since G is strictly linear-recursive, all the cycles in $\mathcal{P}(G)$ are vertex-disjoint. Therefore, $\mathcal{T}(R)$ is well-defined.

EXAMPLE 14. *We convert the basic parse tree $\mathcal{T}'(R)$ in Figure 13 to the compressed parse tree $\mathcal{T}(R)$ in Figure 14. $R : 1$ and $R : 2$ are two recursive nodes that are newly inserted. Comparing the two trees, we can see that $A : 1$, $B : 1$, $A : 2$, $B : 2$, $A : 3$ used to be a path in Figure 13, but are now flattened to be the children of $R : 1$ in Figure 14. The edge labels in Figure 14 will be explained in Section 4.2.2.*

LEMMA 4. *Given a strictly linear-recursive workflow grammar G, for any derivation of a run $R \in L(G)$, the depth of the compressed parse tree $\mathcal{T}(R)$ is no greater than $2 * |\Delta|$, where $|\Delta|$ is the number of composite modules in G.*

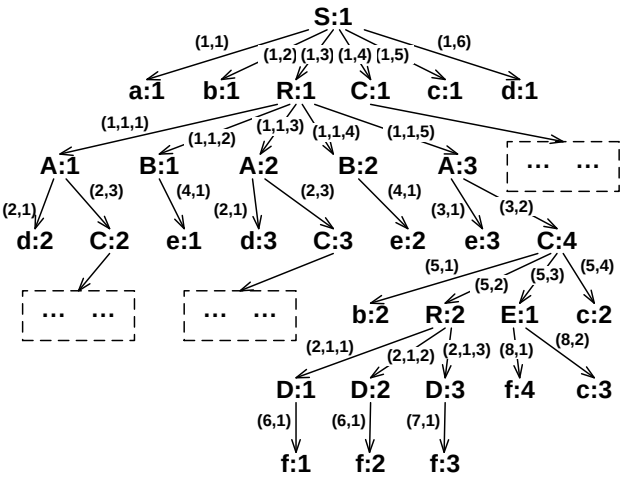

Figure 14: Compressed Parse Tree

PROOF. Consider the path from the root node to an arbitrary leaf node in $\mathcal{T}(R)$. Clearly, all the composite modules denoted by non-leaf nodes on this path are distinct, otherwise, they form a cycle in $\mathcal{P}(G)$, and therefore should be positioned as the children of a recursive node in $\mathcal{T}(R)$. On the other hand, by Definition 16, every composite module belongs to at most one cycle in $\mathcal{P}(G)$. So there is at most one recursive node for each composite module on this path. Note that the leaf node denotes an atomic module. Hence, the length of this path is no greater than $2 * |\Delta|$. □

4.2.2 Data Labels

Next, we describe the data labels that will eventually be created by our labeling scheme. Let $\mathcal{T}(R)$ be the compressed parse tree for R. We build the labels step by step.

First of all, we describe the label for an edge in $\mathcal{T}(R)$. Let e be an edge from u to v in $\mathcal{T}(R)$. We denote by $\phi_r(e)$ the label of e. (1) If u is not a recursive node, then e can be mapped to an edge e' in $\mathcal{P}(G)$. Recall from Section 4.1 that each edge in $\mathcal{P}(G)$ is uniquely identified by a pair of numbers. Let $e' = (k, i)$, then $\phi_r(e) = (k, i)$; and (2) otherwise (if u is a recursive node), let u denote the sth cycle in $\mathcal{P}(G)$ starting from the tth edge. This can be determined by the first child of u. Let v be the ith child of u, then $\phi_r(e) = (s, t, i)$.

Secondly, we use a sequence of above edge labels to construct the label for an input port i in R. We denote by $\phi_r(i)$ the label of i. Suppose i is first created as the xth input port of a module M during the derivation of R, and M is denoted by a node v in $\mathcal{T}(R)$. Let e_1, e_2, ..., e_l be the path from the root node to v in $\mathcal{T}(R)$, then $\phi_r(i) = \{\phi_r(e_1), \phi_r(e_2), \ldots, \phi_r(e_l), x\}$. For an output port o, $\phi_r(o)$ is defined similarly.

Finally, we use a pair of port labels to construct the label for a data item (data edge) $d = (o, i)$ in R. We denote by $\phi_r(d)$ the label of d, then $\phi_r(d) = (\phi_r(o), \phi_r(i))$. Note that if $d = (-, i)$ is an initial input, then $\phi_r(d) = (-, \phi_r(i))$. Similarly, if $d = (o, -)$ is a final output, then $\phi_r(d) = (\phi_r(o), -)$. Also note that o and i must be created by the same production. So $\phi_r(o)$ and $\phi_r(i)$ differ only in the last one or two edge labels. The size of $\phi_r(d)$ can be reduced almost by half by factoring out the common prefix of $\phi_r(o)$ and $\phi_r(i)$.

EXAMPLE 15. *Firstly, the edge labels for the compressed parse tree $\mathcal{T}(R)$ are shown in Figure 14. E.g., the edge from $R:1$ to $A:3$ is labeled by $(1, 1, 5)$, because $R:1$ denotes the first cycle in the production graph starting from the first edge (see Example 12), and $A:3$ is the fifth child of $R:1$. Next, we label the data items of the run R in Figures 3 and 4. E.g., consider the data item $d_{21} = (o, i)$ in Figure 4, where o is the first output port of $b:2$, and i is first created as the second input port of $D:1$ (note that i is also the second input port of $f:1$). Then, $\phi_r(d_{21}) = (\phi_r(o), \phi_r(i))$, where*

$$\phi_r(o) = \{(1, 3), (1, 1, 5), (3, 2), (5, 1), 1\}$$
$$\phi_r(i) = \{(1, 3), (1, 1, 5), (3, 2), (5, 2), (2, 1, 1), 2\}$$

Note that the first three edge labels can be factored out.

4.2.3 Dynamic Labeling Algorithm

We finally present the dynamic labeling algorithm. Given a derivation of a run R, we build the compressed parse tree $\mathcal{T}(R)$ in a top-down manner. During this process, we also label the new edges that are introduced to $\mathcal{T}(R)$.

Initially, we create a node v for the start module S. (1) If S is a recursive module, that is, it belongs to a cycle in $\mathcal{P}(G)$, we also create a recursive node r as the root node of $\mathcal{T}(R)$, and insert v as the first child of r. Let r denote the sth cycle in $\mathcal{P}(G)$ starting from the tth edge, then the edge e from r to v in $\mathcal{T}(R)$ is labeled by $\phi_r(e) = (s, t, 1)$; and (2) otherwise, we simply insert v as the root node of $\mathcal{T}(R)$.

Suppose the kth production $M \to W$ in P is applied. Note that the node u that denotes M is already created in $\mathcal{T}(R)$. We create a new node v for each module in W, and insert them to $\mathcal{T}(R)$ according to the fixed topological ordering. Let M_i be the ith module in W. (1) If M_i is not a recursive module, we insert v as the next child of u. The edge e from u to v is labeled by $\phi_r(e) = (k, i)$; and (2) otherwise, we consider two subcases. (2a) If M is also a recursive module and M is reachable from M_i in $\mathcal{P}(G)$ (i.e., M and M_i are in the same cycle of $\mathcal{P}(G)$), we insert v as the next sibling of u. Note that the parent r of u must be a recursive node. Let e_u be the edge from r to u, which is already labeled by $\phi_r(e_u) = (s, t, i)$. Then the edge e_v from r to v is labeled by $\phi_r(e_v) = (s, t, i + 1)$; and (2b) otherwise, we create a new recursive node r as the next child of u, and insert v as the first child of r. The edge e_1 from u to r is labeled by $\phi_r(e_1) = (k, i)$. Let r denote the sth cycle in $\mathcal{P}(G)$ starting from the tth edge, then the edge e_2 from r to v is labeled by $\phi_r(e_2) = (s, t, 1)$.

Given the above edge labels in $\mathcal{T}(R)$, we can easily construct the labels for input and output ports (and therefore the data labels) in R as soon as they are produced during the derivation of R, as described in Section 4.2.2.

4.3 Labeling Safe Views

Given a safe view $U = (\Delta, \lambda)$ over G, our goal is to create a view label $\phi_v(U)$ which can be combined with above data labels to infer reachability over U. Using the algorithm in Section 3.1, we first compute the full dependency assignment λ^* by extending λ to all the composite modules in Δ.

Next, we define three functions, $\mathcal{I}$, $\mathcal{O}$ and $\mathcal{Z}$. Recall from Section 2.2 that G_Δ denotes the grammar obtained by restricting G to Δ. Let $\mathcal{P}(G_\Delta)$ be the production graph of G_Δ, then $\mathcal{P}(G_\Delta)$ is a subgraph of $\mathcal{P}(G)$. Recall from Section 4.1 that each edge in $\mathcal{P}(G)$ is uniquely identified by a pair of numbers (k, i). The input of $\mathcal{I}$ and $\mathcal{O}$ is an edge in

$\mathcal{P}(G_\Delta)$, denoted by a pair (k,i). The input of $\mathcal{Z}$ is a pair of edges in $\mathcal{P}(G_\Delta)$ of form (k,i) and (k,j). For simplicity, we also denote them by a triple (k,i,j). The output of all three functions is a reachability matrix, which is defined next.

Functions $\mathcal{I}$ and $\mathcal{O}$. Given an edge (k,i) in $\mathcal{P}(G_\Delta)$, let $p_k = M \to W$ be the kth production in P, and M_i be the ith module in W, then (1) $\mathcal{I}(k,i)$ is defined as a reachability matrix from the inputs of M to the inputs of M_i (w.r.t. λ^*); and (2) $\mathcal{O}(k,i)$ is defined as a *(reversed) reachability matrix* from the outputs of M to the outputs of M_i (w.r.t. λ^*).

Function $\mathcal{Z}$. Given a pair of edges (k,i) and (k,j) in $\mathcal{P}(G_\Delta)$, let $p_k = M \to W$ be the kth production in P, and M_i and M_j be the ith and jth module in W, respectively, then $\mathcal{Z}(k,i,j)$ is defined as a *reachability matrix* from the outputs of M_i to the inputs of M_j (w.r.t. λ^*). Note that $\mathcal{Z}(k,i,j)$ is an empty matrix (with only false values) if $i \geq j$, since M_i and M_j are sorted in topological ordering.

Finally, $\phi_v(U)$ consists of all the above three functions, along with $\lambda^*(S)$ for the start module S. That is,

$$\phi_v(U) = \{\lambda^*(S), \mathcal{I}, \mathcal{O}, \mathcal{Z}\}$$

Basically, the above view label encodes all the fine-grained dependency information that is specific to this view and is necessary for our decoding algorithm given in Section 4.4.

EXAMPLE 16. *For the running example, we first label the default view $U_1 = (\Delta, \lambda)$ for which λ^* is computed in Example 10, and is shown on the top of Figure 7. Using λ^*, we can compute the functions $\mathcal{I}$, $\mathcal{O}$ and $\mathcal{Z}$. E.g., consider the edge $(1,5)$ from S to c in Figure 12. The first production $p_1 = S \to W_1$ is shown in Figure 2. $\mathcal{I}(1,5)$ denotes the reachability from the inputs of S (i.e., the initial inputs of W_1) to the inputs of c (i.e., the fifth module in W_1); similarly, $\mathcal{O}(1,2)$ denotes the (reversed) reachability from the outputs of S (i.e., the final outputs of W_1) to the outputs of b (i.e., the second module in W_1); and $\mathcal{Z}(1,2,5)$ denotes the reachability from the outputs of b to the inputs of c in W_1.*

$$\mathcal{I}(1,5) = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \mathcal{O}(1,2) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \mathcal{Z}(1,2,5) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Similarly, we can label the other view $U_2 = (\Delta', \lambda')$ defined in Example 7, whose full dependency assignment is shown on the bottom of Figure 7. Using Figure 5, we have

$$\mathcal{I}(1,5) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \mathcal{O}(1,2) = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \mathcal{Z}(1,2,5) = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

As we can see above, the functions encoded by the view labels $\phi_v(U_1)$ and $\phi_v(U_2)$ may evaluate to different values for the same input. Moreover, they are defined over different domains. E.g., $\mathcal{I}(5,1)$ is defined for U_1 but not for U_2.

Space-Efficient View Labeling. By default, we precompute all the reachability matrices for $\mathcal{I}$, $\mathcal{O}$ and $\mathcal{Z}$, and materialize them in the view label. Alternatively, one can compute them on-the-fly by performing a graph search over the view of a specification during the query time. In general, more sophisticated approaches (e.g., [14, 23, 21]) can be used to label the view, in order to find a better balance between the overhead of labeling views and query efficiency. We will further explore this tradeoff in the experiments.

4.4 Decoding Data Labels with View Labels

Using only two data labels $\phi_r(d_1)$ and $\phi_r(d_2)$ and a view label $\phi_v(U)$, one can decide if d_2 depends on d_1 w.r.t. U by a decoding predicate π. We first define in Section 4.4.1 two procedures used by π, namely, `Inputs` and `Outputs`, and then describe π in Section 4.4.2. Section 4.4.3 presents fast matrix multiplication used to achieve constant query time.

4.4.1 *Procedures* `Inputs` *and* `Outputs`

Algorithm 1 *Procedure* `Inputs`

Input: $\phi_r(e) = (k,i)$ **or** (s,t,i)
$\phi_v(U) = \{\lambda^*(S), \mathcal{I}, \mathcal{O}, \mathcal{Z}\}$
Output: `Inputs`$(\phi_r(e), \phi_v(U))$

1: **if** $\phi_r(e) = (k,i)$ **then**
2: **return** $\mathcal{I}(k,i)$
3: **else** $\{\phi_r(e) = (s,t,i)\}$
4: **let** $\mathcal{C}(s) = \{(k_1,i_1),(k_2,i_2),\ldots,(k_l,i_l)\}$
5: // $\mathcal{C}(s)$ denotes the sth cycle in $\mathcal{P}(G)$ of length l
6: **let** $\forall a \geq 1$, $k_{a+l} = k_a$ **and** $i_{a+l} = i_a$
7: **return** $\prod_{a=1}^{i-1} \mathcal{I}(k_{t+a-1}, i_{t+a-1})$
8: **end if**

Let e be an edge from u to v in the compressed parse tree $\mathcal{T}(R)$. Recall from Section 4.2.2 that $\phi_r(e) = (k,i)$ if u is not a recursive node; otherwise, $\phi_r(e) = (s,t,i)$. Given an edge label $\phi_r(e)$ and a view label $\phi_v(U)$, our procedure `Inputs` computes a reachability matrix `Inputs`$(\phi_r(e), \phi_v(U))$ by Algorithm 1.

Case 1. [Line 1 to Line 2] If $\phi_r(e) = (k,i)$, that is, if u is not a recursive node, let M_u and M_v be the module denoted by u and v, respectively, then `Inputs` computes the reachability matrix from the inputs of M_u to the inputs of M_v in R_U. Let I be the function encoded by $\phi_v(U)$ in Section 4.3, then `Inputs`$(\phi_r(e), \phi_v(U))$ is simply given by $\mathcal{I}(k,i)$.

Case 2. [Line 3 to Line 8] If $\phi_r(e) = (s,t,i)$, that is, if u is a recursive node, let M_a be the module denoted by the ath child of u, then v denotes M_i, and M_1, M_2, ..., M_i form a sequence of nested composite modules in R_U obtained by unfolding the sth cycle in $\mathcal{P}(G)$ starting from the tth edge. `Inputs` thus computes the reachability matrix from the inputs of M_1 to the inputs of M_i in R_U. Let I be the function encoded by $\phi_v(U)$. Let $\mathcal{C}(s) = \{(k_1,i_1),(k_2,i_2),\ldots,(k_l,i_l)\}$ be the sth cycle in $\mathcal{P}(G)$ (see Section 4.1). For any $a \geq 1$, we also assume $k_{a+l} = k_a$ and $i_{a+l} = i_a$. Therefore, for any $1 \leq a \leq i-1$, $I(k_{t+a-1}, i_{t+a-1})$ is the reachability matrix from M_a to M_{a+1} in R_U. Multiplying all these $i-1$ matrices produces `Inputs`$(\phi_r(e), \phi_v(U))$. Note that if $i = 1$, `Inputs`$(\phi_r(e), \phi_v(U))$ is simply an identity matrix.

The other procedure `Outputs` is defined similarly. Given $\phi_r(e)$ and $\phi_v(U)$, it computes a (reversed) reachability matrix for output ports, denoted by `Outputs`$(\phi_r(e), \phi_v(U))$.

EXAMPLE 17. *Let e be the edge from $R:1$ to $A:3$ in Figure 14 and U_1 be the default view. $\phi_r(e) = (1,1,5)$ and $\phi_v(U_1)$ are explained in Examples 15 and 16. For this pair of labels, Algorithm 1 computes the reachability matrix from the inputs of $A{:}1$ to the inputs of $A{:}3$ in R_{U_1}. By Example 12, the first cycle is $\mathcal{C}(1) = \{(2,2),(4,2)\}$. Therefore,*

`Inputs`$(\phi_r(e), \phi_v(U_1)) = \mathcal{I}(2,2) \times \mathcal{I}(4,2) \times \mathcal{I}(2,2) \times \mathcal{I}(4,2)$

Algorithm 2 *Decoding Predicate* π

Input: $\phi_r(d_1) = (\phi_r(o_1), \phi_r(i_1))$
$\phi_r(d_2) = (\phi_r(o_2), \phi_r(i_2))$
$\phi_v(U) = \{\lambda^*(S), \mathcal{I}, \mathcal{O}, \mathcal{Z}\}$
Output: $\pi(\phi_r(d_1), \phi_r(d_2), \phi_v(U))$

1: **if** $\phi_r(i_1) =$ `null` **or** $\phi_r(o_2) =$ `null` **then**
2: **return** `false`
3: **end if**
4: **if** $\phi_r(o_1) =$ `null` **and** $\phi_r(i_2) =$ `null` **then**
5: **let** $\phi_r(i_1) = \{x\}$ **or** $\{\phi_r(e), x\}$
6: **let** $\phi_r(o_2) = \{y\}$ **or** $\{\phi_r(e), y\}$
7: **return** $\lambda^*(S)[x, y]$
8: **end if**
9: **if** $\phi_r(o_1) =$ `null` **then**
10: **let** $\phi_r(i_1) = \{x\}$ **or** $\{\phi_r(e_1), x\}$
11: **let** $\phi_r(i_2) = \{\phi_r(e_1), \phi_r(e_2), \ldots, \phi_r(e_l), y\}$
12: $I_a \leftarrow$ `Inputs`$(\phi_r(e_a), \phi_v(U))$
13: **return** $(\prod_{a=1}^{l} I_a)[x, y]$
14: **end if**
15: **if** $\phi_r(i_2) =$ `null` **then**
16: Similar from Line 9 to Line 14
17: **end if**
18: **let** $\phi_r(o_1) = \{l_1, x\}$ **and** $\phi_r(i_2) = \{l_2, y\}$
19: **if** $l_1 = l_2$ or one is a prefix of the other **then**
20: **return** `false`
21: **else**
22: **let** $l_1 = \{\phi_r(e_1), \ldots, \phi_r(e_{l-1}), \phi_r(e_l), \ldots, \phi_r(e_p)\}$
23: **let** $l_2 = \{\phi_r(e_1), \ldots, \phi_r(e_{l-1}), \phi_r(e'_l), \ldots, \phi_r(e'_q)\}$
24: **if** $\phi_r(e_l) = (k, i)$ **and** $\phi_r(e'_l) = (k, j)$ **then**
25: **if** $i > j$ **then**
26: **return** `false`
27: **else**
28: $O \leftarrow \prod_{a=l+1}^{p}$ `Outputs`$(\phi_r(e_a), \phi_v(U))$
29: $Z \leftarrow \mathcal{Z}(k, i, j)$
30: $I \leftarrow \prod_{a=l+1}^{q}$ `Inputs`$(\phi_r(e'_a), \phi_v(U))$
31: **return** $(O^T \times Z \times I)[x, y]$
32: **end if**
33: **end if**
34: **if** $\phi_r(e_l) = (s, t, i)$ and $\phi_r(e'_l) = (s, t, j)$ **then**
35: **if** $i < j$ **then**
36: **if** $p = l$ **then**
37: **return** `false`
38: **else**
39: **let** $\phi_(e_{l+1}) = (k', i')$
40: **let** $\mathcal{C}(s) = \{(k_1, i_1), (k_2, i_2), \ldots, (k_l, i_l)\}$
41: **let** $\forall a \geq 1$, $k_{a+l} = k_a$ **and** $i_{a+l} = i_a$
42: **let** $(k_{t+i-1}, l_{t+i-1}) = (k', j')$
43: **if** $i' > j'$ **then**
44: **return** `false`
45: **else**
46: $O \leftarrow \prod_{a=l+2}^{p}$ `Outputs`$(\phi_r(e_a), \phi_v(U))$
47: $Z \leftarrow \mathcal{Z}(k', i', j')$
48: $I' \leftarrow$ `Inputs`$((s, t+i, j-i), \phi_v(U))$
49: $I \leftarrow \prod_{a=l+1}^{q}$ `Inputs`$(\phi_r(e'_a), \phi_v(U))$
50: **return** $(O^T \times Z \times I' \times I)[x, y]$
51: **end if**
52: **end if**
53: **end if**
54: **if** $i > j$ **then**
55: Similar from Line 35 to Line 53
56: **end if**
57: **end if**
58: **end if**

4.4.2 Decoding Predicate

Given a pair of data labels $\phi_r(d_1)$ and $\phi_r(d_2)$ and a view label $\phi_v(U)$, our decoding predicate π evaluates to `true` iff d_2 depends on d_1 w.r.t. U by Algorithm 2. We first explain the boundary cases, where at least one of d_1 and d_2 is either an initial input or a final output of R (Line 1 to Line 17).

Case I. [Line 1 to Line 3] If $\phi_r(d_1) = (\phi_r(o_1), -)$ or $\phi_r(d_2) = (-, \phi_r(i_2))$, that is, if d_1 is a final output or d_2 is an initial input, then it is clear that d_2 does not depend on d_1 in R_U. So π immediately evaluates to `false`.

Case II. [Line 4 to Line 8] If $\phi_r(d_1) = (-, \phi_r(i_1))$ and $\phi_r(d_2) = (\phi_r(o_2), -)$, that is, if d_1 is an initial input and d_2 is a final output, then d_2 depends on d_1 w.r.t. U iff o_2 is reachable from i_1 in R_U. Let $\phi_r(i_1) = \{x\}$ or $\{\phi_r(e), x\}$ and $\phi_r(o_2) = \{y\}$ or $\{\phi_r(e), y\}$. Note that $\phi_r(i_1)$ and $\phi_r(o_2)$ may contain one edge label $\phi_r(e)$ if the start module S is recursive. Therefore, i_1 is the xth input port of S and o_1 is the yth output port of S. Let $\lambda^*(S)$ be the reachability matrix encoded by $\phi_v(U)$. So π evaluates to $\lambda^*(S)[x, y]$.

Case III. [Line 9 to Line 14] If $\phi_r(d_1) = (-, \phi_r(i_1))$ and $\phi_r(d_2) = (\phi_r(o_2), \phi_r(i_2))$, that is, if d_1 is an initial input and d_2 is an intermediate data item, then d_2 depends on d_1 w.r.t. U iff i_2 is reachable from i_1 in R_U. Let $\phi_r(i_1) = \{x\}$ or $\{\phi_r(e_1), x\}$ and $\phi_r(i_2) = \{\phi_r(e_1), \phi_r(e_2), \ldots, \phi_r(e_l), y\}$. Therefore, i_1 is initially the xth input port of the start module S, and i_2 is first created as the yth input port of some module M during the derivation of R. Let $I_a =$ `Inputs`$(\phi_r(e_a), \phi_v(U))$ be the reachability matrix computed by Algorithm 1 for each edge label $\phi_r(e_a)$ in $\phi_r(i_2)$, then $\prod_{a=1}^{l} I_a$ gives the reachability matrix from the inputs of S to the inputs of M. So π evaluates to $(\prod_{a=1}^{l} I_a)[x, y]$.

Case IV. [Line 15 to Line 17] If $\phi_r(d_1) = (\phi_r(o_1), \phi_r(i_1))$ and $\phi_r(d_2) = (\phi_r(o_2), -)$, that is, if d_1 is an intermediate data item and d_2 is a final output, this case is symmetric to Case III. So π is evaluated similarly using output ports.

Next, we explain the main cases, where both d_1 and d_2 are intermediate data items of R (Line 18 to Line 58). Let $\phi_r(d_1) = (\phi_r(o_1), \phi_r(i_1))$ and $\phi_r(d_2) = (\phi_r(o_2), \phi_r(i_2))$, then d_2 depends on d_1 w.r.t. U iff i_2 is reachable from o_1 in R_U. Let $\phi_r(o_1) = \{l_1, x\}$ and $\phi_r(i_2) = \{l_2, y\}$, where l_1 and l_2 are two lists of edge labels. Therefore, during the derivation of R, o_1 is first created as the xth output port of some module M_1 and i_2 is first created as the yth input port of some module M_2. Suppose M_1 and M_2 are denoted by two nodes v_1 and v_2 in the compressed parse tree $\mathcal{T}(R)$.

Case 1. [Line 19 to Line 21] If $l_1 = l_2$ or one is a prefix of the other, that is, $v_1 = v_2$ or one is an ancestor of the other in $\mathcal{T}(R)$, then $M_1 = M_2$ or one is derived from the other during the derivation of R. As illustrated by the left column of Figure 15, for all three subcases of Case 1, i_2 is not reachable from o_1 in R_U. So π evaluates to `false`.

Case 2. Otherwise, suppose l_1 and l_2 agree on the first $l-1$ edge labels, but differ on the lth edge label. Moreover, let the length of l_1 and l_2 be p and q, respectively. That is,

$$l_1 = \{\phi_r(e_1), \ldots, \phi_r(e_{l-1}), \phi_r(e_l), \ldots, \phi_r(e_p)\}$$
$$l_2 = \{\phi_r(e_1), \ldots, \phi_r(e_{l-1}), \phi_r(e'_l), \ldots, \phi_r(e'_q)\}$$

where $\phi_r(e_l) \neq \phi_r(e'_l)$. We denote by $v = LCA(v_1, v_2)$ the *least common ancestor* of v_1 and v_2 in $\mathcal{T}(R)$. Let e_l be an edge from v to v'_1 and e'_l be an edge from v to v'_2. Let M'_1 and M'_2 be the module denoted by v'_1 and v'_2, respectively.

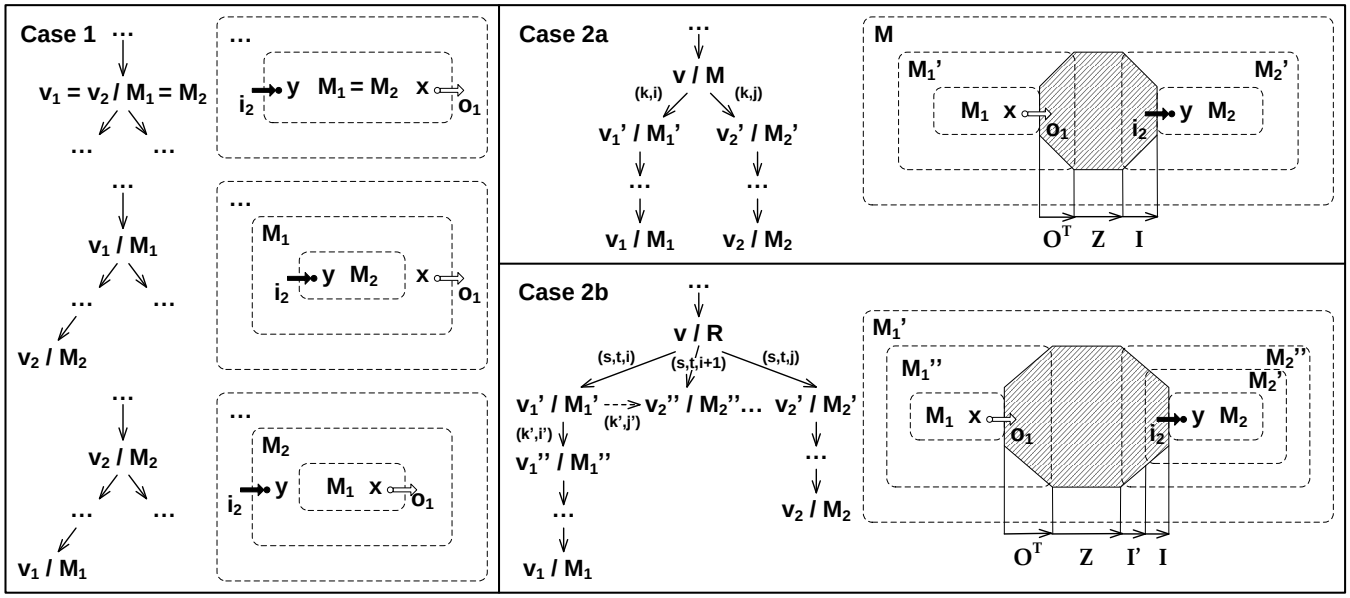

Figure 15: Compressed Parse Tree and Workflow Run for Main Cases of Decoding Predicate

Case 2a. [Line 24 to Line 33] If $\phi_r(e_l) = (k, i)$ and $\phi_r(e'_l) = (k, j)$, that is, $v = LCA(v_1, v_2)$ is not a recursive node, as illustrated by the top right corner of Figure 15, O computed by Line 28 is the (reversed) reachability matrix from the outputs of M'_1 to the outputs of M_1, Z computed by Line 29 is the reachability matrix from the outputs of M'_1 to the inputs of M'_2, and I computed by Line 30 is the reachability matrix from the inputs of M'_2 to the inputs of M_2. Thus, $O^T \times Z \times I$ gives the reachability matrix from the outputs of M_1 to the inputs of M_2, where O^T is the transpose of O. So π evaluates to $(O^T \times Z \times I)[x, y]$. Note that if $i > j$, then $Z = \mathcal{Z}(k, i, j)$ must be an empty matrix (with only false values). So we can immediately evaluate π to `false` without computing O and I by Algorithm 1 (Line 25 to Line 27).

Case 2b. [Line 34 to Line 57] If $\phi_r(e_l) = (s, t, i)$ and $\phi_r(e'_l) = (s, t, j)$, that is, $v = LCA(v_1, v_2)$ is a recursive node, we consider the case where $i < j$ (Line 35 to Line 53). The other case where $i > j$ can be handled in a similar manner (Line 54 to Line 56). First of all, if $p = l$, that is, $v_1 = v'_1$ and $M_1 = M'_1$, then M_2 is derived from M_1 during the derivation of R. By Case 1, we know that i_2 is not reachable from o_1 in R_U. So π evaluates to `false` (Line 36 to Line 38). Otherwise, as illustrated by the bottom right corner of Figure 15, let e_{l+1} be an edge from v'_1 to v''_1, which is labeled by $\phi_r(e_{l+1}) = (k', i')$. Let v''_2 be the right sibling of v'_1, then the edge from v to v''_2 is labeled by $(s, t, i + 1)$. Since the children of v are obtained by unfolding the sth cycle in $\mathcal{P}(G)$ starting from the tth edge, we can compute the label (k', j') for the dashed edge from v'_1 to v''_2 as shown in Figure 15 (Line 40 to Line 42). Let M'_1 and M''_2 be the module denoted by v''_1 and v''_2, respectively. Therefore, O computed by Line 46 is the (reversed) reachability matrix from the outputs of M''_1 to M_1, Z computed by Line 47 is the reachability matrix from the outputs of M''_1 to the inputs of M''_2, I' computed by Line 48 is the reachability matrix from the inputs of M''_2 to the inputs of M'_2, and I computed by Line 49 is the reachability matrix from the inputs of M'_2 to the inputs of M_2. In order to compute I', we create a new edge label $(s, t + i, j - i)$, since the subsequence of children of v from v''_2 to v'_2 is obtained by unfolding the sth cycle in $\mathcal{P}(G)$ starting from the $t + i$th edge, and has a length of $j - i$. Finally, $O^T \times Z \times I' \times I$ gives the reachability matrix from the outputs of M_1 to the inputs of M_2. So π evaluates to $(O^T \times Z \times I' \times I)[x, y]$. As before, if $i > j$, then $Z = \mathcal{Z}(k', i', j')$ must be an empty matrix, and therefore π immediately evaluates to `false` (Line 43 to Line 45).

Notice that we also use the function $\mathcal{C}$ which records the edge sequence for each cycle in the production graph (Line 4 in Algorithm 1 and Line 40 in Algorithm 2). Since $\mathcal{C}$ depends only on the specification (but neither views nor runs), we maintain it as a global index, which takes negligible space.

THEOREM 9. *Let (ϕ_r, ϕ_v, π) be our view-adaptive dynamic labeling scheme for a strictly linear-recursive specification G. For any derivation of a run $R \in L(G)$, any safe view U over G and any two data items d_1 and d_2 in R_U, $\pi(\phi_r(d_1), \phi_r(d_2), \phi_v(U))$ = `true` iff d_2 depends on d_1 w.r.t. U.*

PROOF. All boundary cases (**I** to **IV**) and main cases (**1**, **2a** and **2b**) of Algorithm 2 have been justified above. □

4.4.3 Fast Matrix Multiplication

To achieve constant query time, we need to show that `Inputs` and `Outputs` can be implemented in constant time.

LEMMA 5. *Given a fixed strictly linear-recursive grammar G, for any edge label $\phi_r(e)$ and any data label $\phi_v(U)$,* `Inputs` *and* `Outputs` *can be computed in constant time.*

PROOF. Let c be the maximum number of inputs or outputs of any module. That is, c is the maximum cardinality of any reachability matrix. Note that c is a constant for a fixed grammar G. Then a naive implementation of `Inputs` and `Outputs` would take $O(c^3 * i) = O(i)$ time to compute the multiplication of $i-1$ reachability matrices. In the worst case, i can be linear in the size of the run.

However, more efficient algorithms are possible by observing the repeated pattern of length l in $i-1$ matrices. Let X be the multiplication of the first l matrices. Note that l is also a constant. A better approach is thus to compute $X^{\lfloor i-1/l \rfloor}$ by divide and conquer, which runs in $O(\log i)$ time.

A further observation is that there is also a repeated pattern in the sequence X, X^2, ..., $X^{\lfloor i-1/l \rfloor}$. Given that each matrix has at most 2^{c^2} possible boolean values, we can find a and b such that $a < b <= 2^{c^2}+1$ and $X^a = X^b$. Once a and b are found, $X^{\lfloor i-1/l \rfloor}$ can be computed in constant time. Note that a, b and c are all small constants in practice. □

Query-Efficient View Labeling. To speed up the query processing, one can also pre-compute a and b for each recursion in the view, and materialize a and b (as well as $X^1, X^2, \ldots, X^b$) in the view label. In contrast to space-efficient view labeling (Section 4.3), this is the other extreme alternative that will be compared in the experiments.

4.5 Labeling Scheme Quality Analysis

We analyze the *label length* and *construction time* for both data labels and view labels, as well as the *query time* for comparing a pair of data labels and a view label. Note that we take the size of a specification as constant [4, 5], and measure the complexity in terms of the size of the run. We next show that all the above parameters, guaranteed by our labeling scheme, are *optimal* up to a constant factor.

THEOREM 10. *Let (ϕ_r, ϕ_v, π) be our view-adaptive dynamic labeling scheme for a strictly linear-recursive specification G.*

1. *logarithmic label length and linear total construction time for data labels: for any derivation of a run $R \in L(G)$ with n data items and for any data item d in R, $\phi_r(d)$ has $O(\log n)$ bits, and all data labels can be constructed dynamically in a total of $O(n)$ time.*

2. *constant label length and constant construction time for view labels: for any safe view U over G, $\phi_v(U)$ has $O(1)$ bits and can be constructed in $O(1)$ time.*

3. *constant query time: for any pair of data labels $\phi_r(d_1)$ and $\phi_r(d_2)$ and for any view label $\phi_v(U)$, $\pi(\phi_r(d_1), \phi_r(d_2), \phi_v(U))$ can be evaluated in $O(1)$ time.*

PROOF. **Part1.** Let $\mathcal{T}(R)$ be the compressed parse tree for R. We denote by n_t the size of $\mathcal{T}(R)$, by d_t the depth of $\mathcal{T}(R)$ and by θ_t the maximum outdegree of a node in $\mathcal{T}(R)$. Since R has n data items, it is easy to show that $\theta_t \leq n_t = O(n)$. In addition, by Lemma 4, $d_t = O(1)$. Recall from Section 4.2.2 that for any edge e in $\mathcal{T}(R)$, $\phi_r(e)$ is either (k, i) or (s, t, i), where k, s, t are all bounded by constants for a given G, and i is bounded by θ_t. So $\phi_r(e)$ can be encoded by $O(\log \theta_t)$ bits. Moreover, for any input (or output) port i in R, $\phi_r(i)$ consists of at most d_t edge labels, followed by a port id x that is bounded by a constant. So $\phi_r(i)$ can be encoded by $O(d_t * \log \theta_t)$ bits. Finally, for any data item d in R, $\phi_r(d)$ consists of a pair of labels for an output port and an input port, and therefore has $O(2d_t * \log \theta_t) = O(\log n)$ bits. Note that by factoring out common edge labels, the length of $\phi_r(d)$ reduces to $O((d_t+2) * \log \theta_t)$ bits. On the other hand, recall from Section 4.2.3 that $\mathcal{T}(R)$ is built dynamically in a top-down fashion, which takes a total of $O(n_t)$ time. During this process, each label for any edge in $\mathcal{T}(R)$ and any port and data item in R can be created in $O(1)$ time. So the total construction time is $O(n_t + n) = O(n)$.

Part 2. Recall from Section 4.3 that $\phi_v(U) = \{\lambda^*(S), \mathcal{I}, \mathcal{O}, \mathcal{Z}\}$. Let m be the number of edges in the production graph $\mathcal{P}(G)$ and c be the maximum number of input or output ports of a module in G. Note that both m and c are constants for a given G. Since (1) $\lambda^*(S)$ can be encoded by $O(c^2)$ bits; (2) each of $\mathcal{I}$ and $\mathcal{O}$ can be encoded by $O(mc^2)$ bits; and (3) $\mathcal{Z}$ can be encoded by $O(m^2 * c^2)$ bits, $\phi_v(U)$ has a total of $O(m^2 * c^2) = O(1)$ bits. On the other hand, let p be the number of productions in G and q be the maximum size (i.e., the total number of ports) of a simple workflow in G. Again, both p and q are constants. We first compute the reachability between every pair of ports for all simple workflows, which takes $O(p * q^2)$ time. So $\phi_v(U)$ can be constructed in $O(pq^2) + O(m^2 * c^2) = O(1)$ time.

Part 3. The dominating time complexity of Algorithm 2 falls to the main cases **2a** and **2b** in which π invokes at most $2d_t - 2$ calls to the procedures `Inputs` and `Outputs`, and performs at most $2d_t - 2$ additional matrix multiplications. By Lemma 5, every call to `Inputs` and `Outputs` can be answered in $O(1)$ time. Moreover, since the cardinality of any reachability matrix is bounded by a constant c for a given G, every matrix multiplication can be done in $O(c^3) = O(1)$ time. So the overall query time is $O(2d_t - 2) = O(1)$. □

5. HANDLING USER-DEFINED VIEWS

So far we consider only views that restrict possible expansions of the workflow hierarchy to a subset of *pre-defined* composite modules. We next extend to more general types of views, called *user-defined views*, and briefly explain how they can be handled using the view-adpative labeling approach. One of operations that can be used to construct user-defined views is to introduce new composite modules to the workflow grammar by grouping existing modules.

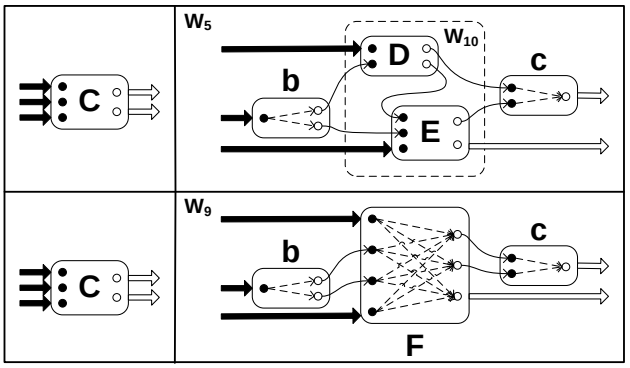

Figure 16: User-Defined View

EXAMPLE 18. *As shown in Figure 16, a user-defined view is constructed over the specification in Figure 2 by grouping the modules D and E in W_5 together as a new module F whose details are hidden from the view. Note that not only details of D and E but also the data item (data edge) flowing between D and E in W_5 are hidden. To formally define this view, we need to modify the specification by introducing F as a new composite module and replacing the production $C \to W_5$ with two new productions $C \to W_9$ and $F \to W_{10}$,*

where W_9 and W_{10} (denoted by the dashed box) are shown in Figure 16. Then, this user-defined view is given by (Δ', λ'), where $\Delta' = \{S, A, B, C\}$, and λ' is same as the default view, except that $\lambda'(F)$ is newly defined as in Figure 16.

Unfortunately, if we directly apply our scheme to label a user-defined view, the resulting view label does not work with existing data labels, because the view label is created based on the new specification containing new compoiste modules that are not captured by existing data labels.

However, reusing data labels, which is the essential goal of our view-adaptive labeling, is still possible by carefully designing the labels for user-defined views. The idea is to label user-defined views based on the old specification. To this end, we project a user-defined view onto a regular view over the old specification by expanding all newly introduced composite modules. When labeling the user-defined view, we use the structure of this regular view but with the new dependency assignment. Note that with this framework, we need to maintain only one specification, and all new specifications introduced by user-defined views are virtual. Our handling of user-defined views is illustrated below.

EXAMPLE 19. *The user-defined view in Example 18 is projected onto the default view by expanding the new module F (i.e., replacing $C \to W_9$ with $C \to W_5$). The view label is then obtained by labeling the production graph of the default view using the new dependency assignment. E.g., consider the edge $(5, 3)$ from C to E in Figure 12. The fifth production $C \to W_5$ is shown in Figure 16. Recall that $\mathcal{I}(5, 3)$ denotes the reachability matrix from the inputs of C (i.e., the initial inputs of W_5) to the inputs of E (i.e., the third module in W_5). To compute $\mathcal{I}(5, 3)$, we use the new dependency assignment given to $C \to W_9$ in Figure 16. Then,*

$$\mathcal{I}(5,3) = \begin{bmatrix} - & 0 & 0 \\ - & 1 & 0 \\ - & 0 & 1 \end{bmatrix}$$

where the first column is undefined, because the first input port of E in W_5 is invisible in W_9. Note that the new module F is expanded, so that the above view label can be combined with existing data labels to answer queries as before.

An important observation is that although user-defined views introduce new modules to the workflow hierarchy, they do not introduce any new recursions. This enables us to reuse data labels that encode the recursive structure of runs to answer queries over any user-defined views. Based on this observation, besides grouping modules, our view-adaptive labeling scheme can also support other operations to construct views, such as hiding ports or data edges.

The labels created by our view-adaptive scheme can also be used to check *data visibility*. Formally, for any data item d in a run R and any safe view U, using only a data label $\phi_r(d)$ and a view label $\phi_v(U)$, one can decide in constant time if d is visible in R_U by checking if the function $\mathcal{I}$ in $\phi_v(U)$ is defined for all the edge labels in $\phi_r(d)$. Similarly, one can also check data visibility for user-defined views.

6. EXPERIMENTAL EVALUATION

We now empirically evaluate the effectiveness of our view-adaptive labeling approach. Section 6.2 reports the main cost of labeling, which is labeling runs. Section 6.3 explores the tradeoff between the overhead of labeling views and query time by comparing three alternative implementations. Section 6.4 demonstrates the superiority of view-adaptive labeling over the state-of-the-art technique [5] when applied to label multiple views. Section 6.5 identifies important factors that influence the performance of view-adaptive labeling.

6.1 Experimental Setup

Real-Life and Synthetic Datasets. Our real-life scientific workflows were collected form the *myExperiment* workflow repository [18]. We observed that almost all of them have fairly simple recursive patterns. For simplicity, we report only the results for one representative workflow, called BioAID. It is denoted by a strictly linear-recursive grammar with 112 modules (16 are composite) and 23 productions (7 are recursive [5]). Each production produces a simple workflow with at most 19 modules, and each module has at most 4 input ports and 7 output ports. In Section 6.5, we also evaluate a family of synthetic workflows created from a linear-recursive topology shown in Figure 26. Due to the absence of real workflow executions, we simulated runs by applying a random sequence of productions, varying their sizes (i.e., the number of data items) from $1K$ to $32K$ by a factor of 2. The derivations of runs were recorded and used as dynamic inputs to labeling schemes. In addition, we obtained safe views by enumerating all possible proper subsets of composite modules and assigning random input-output dependencies to atomic modules. All the data are stored as XML files whose parsing time is omitted from the results.

Labeling Schemes. Our proposed view-adaptive dynamic labeling scheme is denoted by *FVL* for (F)ine-grained (V)iew-adaptive (L)abeling. We implemented three variants of *FVL*: (1) *Default FVL* (Section 4.3) (2) *Space-Efficient FVL* (Section 4.3) and (3) *Query-Efficient FVL* (Section 4.4.3). The three alternatives apply the same dynamic algorithm to label runs, but differ only in the way to label views, and therefore also vary in the query efficiency. A more detailed comparison will be explained (and empirically evaluated) in Section 6.3. In addition, we also compared *FVL* against the state-of-the-art scheme, called *DRL* [5], for (L)abeling (D)ynamic runs of (R)ecursive workflows. The limitations of *DRL* and the way in which we achieve a fair comparison between *FVL* and *DRL* will be further discussed in Sections 6.2 and 6.4. All the above labeling schemes were implemented in Java 6.

Evaluation Methodology. To evaluate the labeling overhead, we measure both label length (space overhead) and construction time (time overhead) for data labels and view labels, respectively. For data labels, each data point in the result is an average over 100 sample runs. We also measure the query time that is taken to compare a pair of data labels along with a view label. Each data point for query time is an average over 10^6 sample queries. All the experiments were performed on a local PC with Intel(R) Core(TM) i7-2600 3.40GHz CPU and 4GB memory running Windows 7.

6.2 Overhead of Labeling Runs

We first evaluate the overhead of labeling runs using *FVL* and *DRL*. Note that *FVL* is *view-adaptive*: the data labels created for one run can be re-used to answer queries over all safe views. In contrast, *DRL* is not view-adaptive: a run

[5]This workflow contains two loops and four forks (parallel executions). Each of them is expressed by one recursive production.

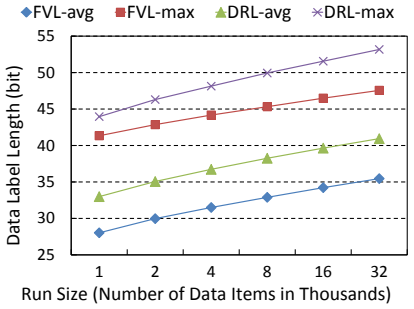

Figure 17: Space Overhead (run)

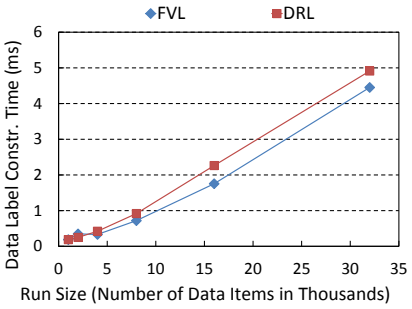

Figure 18: Time Overhead (run)

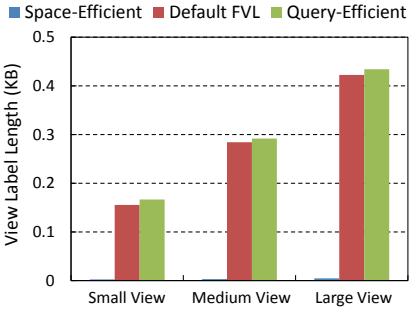

Figure 19: Space Overhead (view)

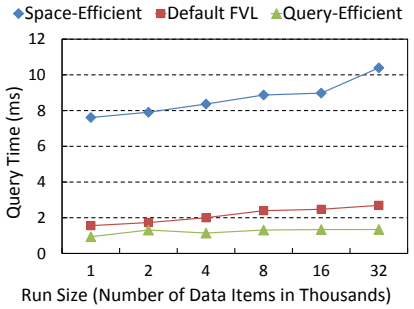

Figure 20: Query Time

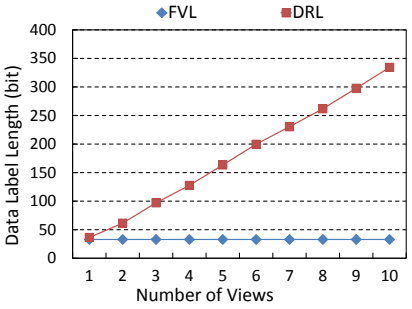

Figure 21: FVL vs DRL (space)

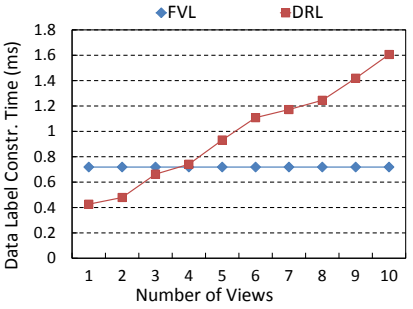

Figure 22: FVL vs DRL (time)

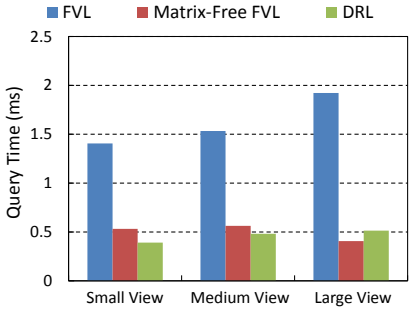

Figure 23: FVL vs DRL (query)

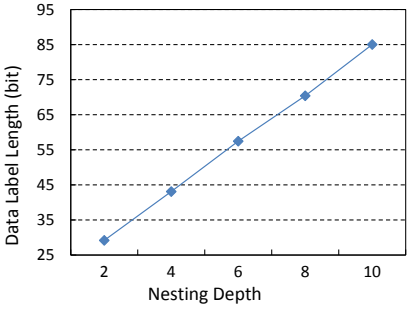

Figure 24: Nesting Depth

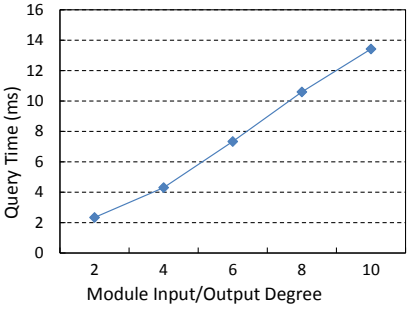

Figure 25: Module Degree

must be re-labeled for each view. Here, the comparison between them focuses on the case where only one default view is defined over the workflow. A more meaningful comparison for multiple views will be carried out in Section 6.4.

Figure 17 reports the maximum and average length of data labels created by *FVL* and *DRL*. We denote them by *FVL-max*, *FVL-avg*, *DRL-max* and *DRL-avg*, respectively. A careful analysis of Figure 17 can show that all four lines are nearly parallel to the asymptotic line $f(x) = \log x$. This implies that both *FVL* and *DRL* produce compact data labels of logarithmic length with a constant factor close to 1. Surprisingly, *FVL-avg* (*FVL-max*) is even shorter than *DRL-avg* (*DRL-max*) by about 5 bits. This small improvement is due to the compact design of data labels in *FVL* which encode only the structure of runs.

Figure 18 reports the construction time of data labels for *FVL* and *DRL*. While both build all data labels in linear time, *FVL* is faster than *DRL* by about 10% for large runs.

6.3 View Labeling Cost vs. Query Efficiency

Next, we evaluate the overhead of labeling views as well as the query time, and explore the tradeoff between them by comparing three variants of *FVL*: (1) *Default FVL* precomputes all reachability matrices for the three functions $\mathcal{I}$, $\mathcal{O}$ and $\mathcal{Z}$, and materializes them in the view label (Section 4.3); (2) *Space-Efficient FVL* pre-computes only the full dependency assignment for each view, and thus any access to $\mathcal{I}$, $\mathcal{O}$ and $\mathcal{Z}$ will be answered by performing a graph search over the view of a specification at query time (Section 4.3); and (3) *Query-Efficient FVL* materializes, in addition to $\mathcal{I}$, $\mathcal{O}$ and $\mathcal{Z}$, all intermediate states of fast matrix multiplication for each recursion in the view (Section 4.4.3).

In the experiments, we label three safe views, namely, small view, medium view and large view, with varying sizes and random dependency assignments. We estimate the size of a view by the number of composite modules that can expand. The three views contain 2, 8 and 16 composite modules, respectively. Figure 19 shows the length of view labels created by all three variants of *FVL*. As expected, *Query-Efficient FVL* creates the longest labels for all three views. However, compared with *Default FVL*, the extra space overhead is small (less than 8 bytes), since views typically have a small amount of recursions. On the other hand, *Space-Efficient FVL* creates almost no index for each view (less than 5 bytes). The results for construction time, not shown, reveal a similar trend. While *Query-Efficient FVL* labels the large view in 0.62 *ms*, *Space-Efficient FVL* needs only 0.08 *ms*. Comparing Figures 17 and 19 also shows that the main

overhead of *FVL* lies in the labeling of runs, e.g., the data labels for a small run with $1K$ data items take a total of $5KB$, while the view label created by *Query-Efficient FVL* for the large view takes only $0.4KB$. The overall difference is even bigger, since for a given workflow, the number of runs is typically much greater than the number of views.

After runs and views are both labeled (independently), we generate sample queries by randomly selecting two data items in the same run (with varying size) and randomly selecting one out of the three views. The query time for the three variants of *FVL* is reported in Figure 20. Compared to Figure 19, we can see a clear tradeoff between the overhead of labeling views and query efficiency. *Query-Efficient FVL* and *Default FVL* are faster than *Space-Efficient FVL* by almost one order of magnitude. *Query-Efficient FVL* is also significantly faster than *Default FVL* (by about 40% for large runs), while as shown in Figure 19, it takes only small extra space overhead (less than 2% for the large view).

Finally, we should notice that all three variants of *FVL* achieve constant view label length and constant query time, in terms of the size of the run. In other words, there is only a constant tradeoff between space and time for the three approaches. Therefore, *Query-Efficient FVL* is preferable to the other two variants, since it enables the fastest query processing with little extra labeling overhead. All the above results also validate our complexity analysis in Theorem 10.

6.4 Advantage of View-Adaptive Labeling

We now compare *FVL* against *DRL* when multiple views are defined over the same workflow. Since *DRL* applies only to the coarse-grained model with black-box dependencies, to make a meaningful comparison we randomly generate 10 medium-size views with black-box dependencies.

First, we compare the labeling overhead of *FVL* and *DRL*. Our focus is on the overhead of labeling runs, which is the main cost. We fix the size of runs to be $8K$ (data items), and vary the number of views from 1 to 10. Figure 21 shows the total length of data labels assigned to one data item. Since *FVL* is view-adaptive, the data label created for one data item can be re-used to query over multiple views. Therefore, in Figure 21, the total length for *FVL* remains constant. In contrast, given a data item, *DRL* has to maintain one data label for each view separately. So in Figure 21, the total length for *DRL* grows linearly with the number of views.

A similar result for the total construction time can be observed in Figure 22. Note that *DRL* is faster than *FVL* for one view, since *DRL* labels the medium-size view of a run, which is smaller than the original run. However, when there are more than 3 views, *FVL* is more time-efficient.

We next compare the query time of *FVL* and *DRL*. In order to achieve a fair comparison, we take the most *query-efficient* variant of both *FVL* and *DRL*. Since our comparison can only use coarse-grained views, many of the reachability matrices involved in the decoding of *FVL* are complete matrices (i.e., with only true values). So we also implemented a simplified version of *FVL*, called *Matrix-Free FVL*, which is optimized for coarse-grained views by avoiding redundant matrix multiplications in the decoding.

We evaluate the above three approaches over three coarse-grained views with varying sizes. As shown in Figure 23, *FVL* is about 4 times slower than *DRL*, but by removing redundent computations for coarse-grained views, *Matrix-Free FVL* achieves almost same query time as *DRL*.

6.5 Important Factors

In the last set of experiments, we examine the effectiveness of our view-adaptive labeling approach over a variety of synthetic workflows. The goal is to identify important factors that affect the performance of *FVL*. To this end, we generate synthetic workflows using four parameters: (1) *workflow size*: the number of modules in a simple workflow (default = 40); (2) *module degree*: the number of input/output ports of a module (default = 4); (3) *nesting depth*: the depth of nested composite modules (default = 4); and (4) *recursion length*: the number of composite modules in a recursion (default = 2). As an example, the production graph of a synthetic workflow is shown in Figure 26, whose nesting depth is 5 (i.e., $C_{1,1} \to C_{2,1} \to C_{3,1} \to C_{4,1} \to C_{5,1} \to C_{6,1}$) and whose recursion length is 3 (i.e., $C_{1,1} \to C_{1,2} \to C_{1,3}$). Note that all atomic modules are omitted from Figure 26.

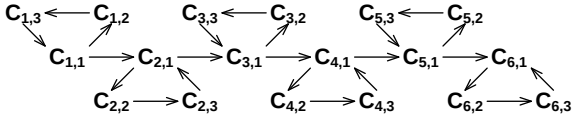

Figure 26: Production Graph of Synthetic Workflow

We created a family of synthetic workflows as Figure 26 by varying each of the four parameters and fixing the rest to be default value. For each synthetic workflow, we evaluate the performance of *FVL* by measuring (1) the overhead of labeling a run R with $8K$ data items; (2) the overhead of labeling a safe view U with all composite modules and random dependency assignment; and (3) the query time for a pair of data items in R over U. A brief summary of results is listed in Table 1. We discuss only two interesting cases.

One factor that has high impact on the data label length is nesting depth. As shown in Figure 24, the (average) data label length created by *FVL* grows linearly as the nesting depth increases from 2 to 10 by a constant of 2. This is due to the fact that the nesting depth determines the depth of the compressed parse tree which is used to build data labels.

Another factor that has high impact on the query time is module degree. As shown in Figure 25, the query time for *Query-Efficient FVL* grows almost linearly as the module degree increases from 2 to 10 by a constant of 2. This is mainly because that the module degree determines the cardinality of reachability matrices, and multiplying large matrices during the label decoding can be expensive.

7. CONCLUSIONS

This paper considers the problem of efficiently answering reachability queries over views of workflow provenance graphs. For that we design a novel *view-adaptive* labeling scheme that supports *fine-grained dependencies* between inputs and outputs of modules and combines *static* labeling of views with *dynamic* labeling of data items. In particular, we identify a natural class of *safe views* over *strictly linear-recursive* workflows for which dynamic, yet *compact* labeling is feasible. The experimental results demonstrate the advantage of our view-adaptive labeling approach over the state-of-the-art technique [5] when applied to label multiple views. Previous work [11] considers efficient evaluation of XPath queries over XML views. Extending our work to similarly rich query constructs in the context of workflow views is an interesting direction for future research.

	data label length	data label time	view label length	view label time	query time
workflow size	no impact	no impact	high impact	high impact	no impact
module degree	no impact	no impact	low impact	low impact	**high impact**
nesting depth	**high impact**	low impact	low impact	low impact	low impact
recursion length	low impact	low impact	low impact	low impact	low impact

Table 1: Important Factors and Their Impact on View-Adaptive Labeling Performance

8. 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 European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant 291071-MoDaS, by the Israel Ministry of Science, and by the Binational (US-Israel) Science Foundation.

9. REFERENCES

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

[2] 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.

[3] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *PODS*, pages 153–164, 2011.

[4] 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.

[5] Z. Bao, S. B. Davidson, and T. Milo. Labeling recursive workflow executions on-the-fly. In *SIGMOD Conference*, pages 493–504, 2011.

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

[7] O. Biton, S. C. Boulakia, S. B. Davidson, and C. S. Hara. Querying and managing provenance through user views in scientific workflows. In *ICDE*, pages 1072–1081, 2008.

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

[9] S. B. Davidson, S. Khanna, S. Roy, J. Stoyanovich, V. Tannen, and Y. Chen. On provenance and privacy. In *ICDT*, pages 3–10, 2011.

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

[11] W. Fan, C. Y. Chan, and M. N. Garofalakis. Secure XML querying with security views. In *SIGMOD Conference*, pages 587–598, 2004.

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

[13] 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.

[14] 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.

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

[16] L. Moreau, J. Freire, J. Futrelle, R. E. McGrath, J. Myers, and P. Paulson. The Open Provenance Model: An overview. In *IPAW*, pages 323–326, 2008.

[17] 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.

[18] D. D. Roure, C. A. Goble, and R. Stevens. The design and realisation of the myExperiment Virtual Research Environment for social sharing of workflows. *Future Generation Comp. Syst.*, 25(5):561–567, 2009.

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

[20] P. Sun, Z. Liu, S. B. Davidson, and Y. Chen. Detecting and resolving unsound workflow views for correct provenance analysis. In *SIGMOD Conference*, pages 549–562, 2009.

[21] S. J. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *SIGMOD Conference*, pages 913–924, 2011.

[22] 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.

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