A Navigation Model for Exploring Scientific Workflow Provenance Graphs

Manish Kumar Anand[1], Shawn Bowers[2], Bertram Ludäscher[1,3]
[1]Dept. of Computer Science, University of California, Davis
[2]Dept. of Computer Science, Gonzaga University
[3]Genome Center, University of California, Davis
{maanand, ludaesch}@ucdavis.edu, bowers@gonzaga.edu

ABSTRACT

Many scientific workflow systems record provenance information in the form of data and process dependencies as part of workflow execution. Users often wish to explore these dependencies to reproduce, validate, and explain workflow results, e.g., by examining the data and processes that were used to produce particular workflow outputs. A natural interface for determining relevant provenance information, which is adopted by many systems, is to display the complete provenance dependency graph. However, for many workflows, provenance graphs can be large, with thousands or more nodes and edges. Displaying an entire provenance graph for such workflows can result in "provenance overload," where the large amount of provenance information available makes it difficult for users to find relevant information and explore data and process dependencies. In this paper, we address the challenges of "provenance overload" through a novel navigation model that provides operations for creating different views of provenance graphs along with approaches for easily navigating between different views. Further, our proposed navigation model provides an integrated approach for exploring, summarizing, and querying portions of provenance graphs. We also discuss different architectures for efficiently navigating large provenance graphs against an underlying provenance database.

1. INTRODUCTION

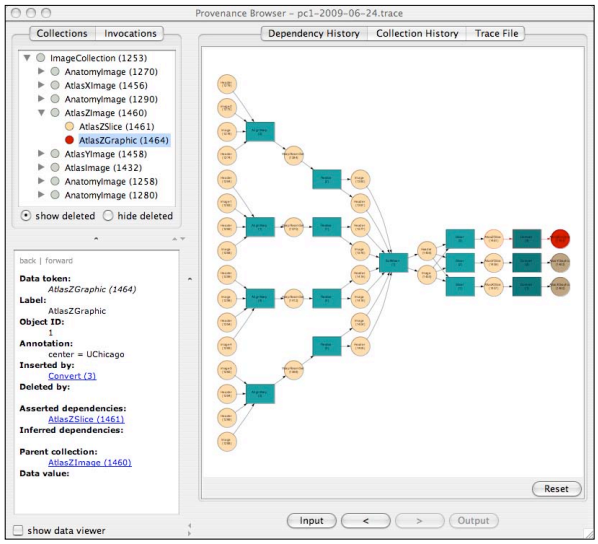
Most scientific workflow systems provide mechanisms for recording *workflow provenance*, i.e., the details of a workflow run including data and process dependencies [12, 21, 26]. This provenance information is often displayed to users visually as one or more dependency graphs [17, 6], e.g., where nodes denote data items or processes, and edges denote causal relationships between nodes. Displaying such graphs is especially useful for small workflows, involving only a few data sets and processes, since users can quickly see every data product, process, and dependency associated with a run. However, for many real-world scientific workflows provenance graphs may be large (e.g., thousands of nodes and edges) due to the complexity of the workflow, the size of input data sets, and the number of intermediate data sets produced [10, 1, 12]. For large provenance graphs, understanding and exploring provenance information becomes a significant challenge for users.

For example, Fig. 1 shows two different provenance graphs generated from real-world workflows: Fig. 1a shows a run of the fMRI analysis from the first provenance challenge [21], and Fig. 1b shows a run of a standard phylogenetic tree inference workflow [6]. While the provenance graph of Fig. 1a can be quickly understood (since it contains only a few nodes and dependencies), the graph of Fig. 1b is much larger and requires considerable effort to fully understand the associated provenance information.

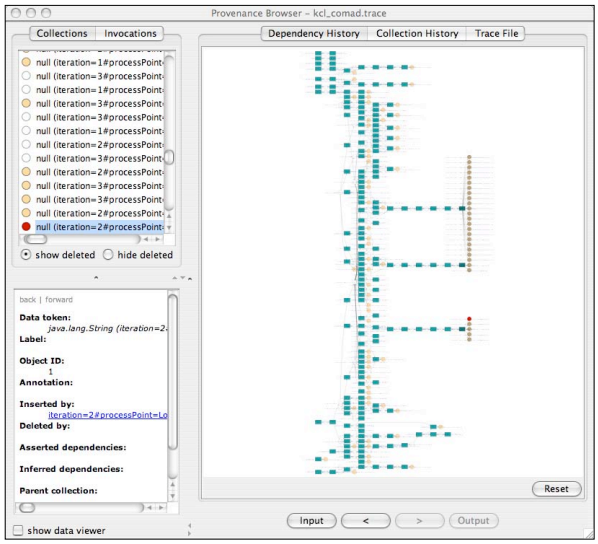
The goal of the work described here is to allow users to specify and navigate between different abstractions (or *views*) of provenance graphs such that by displaying these views, users can obtain the same benefits of quickly understanding and exploring provenance information as for complete (but small) provenance graphs. Specifically, we present a *navigation model* for scientific workflow provenance that consists of operations for creating, refining, and switching (or navigating) between different views of workflow provenance. We consider three main levels of granularity. An *actor dependency graph* consists of the types of processes (or actors) used in a workflow run and the flow of data between them. An *invocation dependency graph* consists of individual processes (or invocations) of the workflow run and the corresponding flow of data. And a *flow dependency graph* consists of the detailed data items input to and produced by the workflow run and their causal dependencies. In addition, the model supports operations that allow all or a portion of each provenance view to be expanded or collapsed, grouped into composite structures, and filtered or exposed using a high-level provenance query language. The navigation model is also based on a generic model of provenance that subsumes conventional approaches for representing workflow provenance while supporting more advanced workflow computation models that permit structured data and update semantics [1].

This paper is organized as follows. We describe the provenance representation scheme and the associated high-level provenance query language used by our navigation model in Section 2. Based on the provenance model, we describe the views and operations supported by the navigation model in Section 3. We also present in Section 3 different architectures for efficiently navigating large provenance graphs against an underlying provenance database. Finally, we describe the relationship betwen our work and existing work on representing, querying, and visualizing provenance graphs in Section 4, and summarize our contributions in Section 5.

WORKS 09, November 15, 2009, Portland Oregon, USA.



(a) Provenance graph of a small trace



(b) Provenance graph of a large trace

Figure 1: Two different provenance graphs displayed using the provenance browser of [6]: (a) shows a provenance graph for a relatively small trace, whereas (b) shows a provenance graph for a much larger trace.

2. PROVENANCE MODEL

Consider the workflow in Fig. 2a denoting a straightforward XML-based implementation of the fMRI image processing pipeline used in the first provenance challenge [17]. We refer to steps in the workflow as *actors* that are *invoked* over input data supplied by previous steps. This workflow takes a set of anatomy images representing 3D brain scans and a reference image, and applies the actors in Fig. 2a as follows.

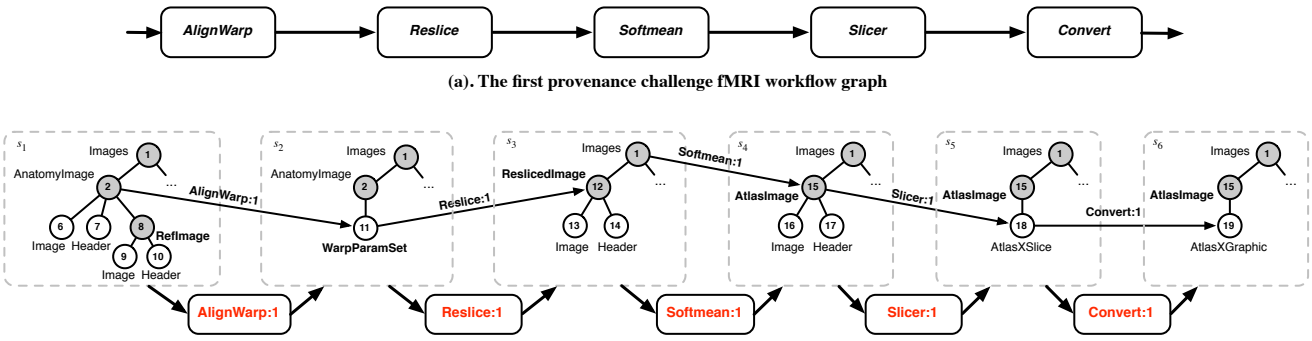
1. *AlignWarp* is invoked over each anatomy image to produce a set of "warping" parameters;
2. *Reslice* is invoked over each set of warping parameters to transform the associated anatomy image;
3. *Softmean* averages transformed images into an atlas image;
4. *Slicer* produces three different 2D slices of the atlas; and
5. *Convert* creates a graphical image for each 2D slice.

In this implementation of the workflow, each invocation of an actor receives an XML structure, performs an update on a portion of that structure, and then sends the updated version of the structure to downstream actors (see Fig. 2b). Here we assume that each XML structure denotes an unranked, labeled ordered tree representing workflow data products, each tree node has a unique identifier, and tree nodes represent either *collection tokens* or *data tokens* (which wrap complex objects or reference external data, e.g., stored within a file). A collection token may be an internal node (for non-empty collections) or a leaf node (for empty collections), whereas data tokens are leaf nodes only.

Representing Provenance with Flow Graphs. Fig. 2b shows the first invocation of each actor for a typical run of the workflow using our provenance model [1]. The invocation of the AlignWarp actor (shown as *AlignWarp:1*) modifies the first AnatomyImage collection (node 2), and replaces its contents with a WarpParamSet data token (node 11). Similarly, the invocation of the Reslice actor uses this WarpParamSet to generate a new Image and Header data token (nodes 13 and 14, respectively). Since only a portion of an input data structure D is typically modified by an invocation, we also represent explicit (i.e., "fine-grained" or node-level) data dependencies as part of the provenance of a run. For example, the arrow from node 2 to node 11 in Fig. 2b states that the WarpParamSet was created from the AnatomyImage collection by the first invocation of AlignWarp. Note that implicitly, node 11 depends on each of the descendents of node 2 (which includes nodes 6-10 in the figure). Similarly, each descendent of a collection implicitly inherits the dependencies of its ancestors. In our example, node 13 is a descendent of node 12 (a ReslicedImage collection), and thus implicitly depends on node 11. Taken together, Fig. 2b denotes a portion of the *flow graph* for a run of Fig. 2a; in particular, this flow graph shows only the information associated with the first invocation of each workflow actor.

Flow graphs can be used to derive standard process (or invocation) and data dependency graphs. For instance, Fig. 3a shows an invocation dependency graph for the flow graph shown of Fig. 2b, where nodes represent invocations of actors and edges represent dependency relationships between invocations. Fig. 3b shows a corresponding data dependency graph, where nodes denote data items and edges denote dependency relations between data items. In general, edges within data dependency graphs are not labeled. In Fig. 3b, we explicitly label dependency edges with the invocation that created the item at the end of the arrow. For example, a dependency $x \xrightarrow{i} y$ states that the data item y was produced by invocation i from the data item x. Note that while the flow graph can be used to infer data and invocation dependency graphs, the flow graph cannot be reconstructed from these two graphs alone.

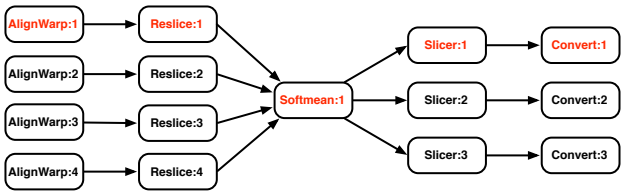
Querying Provenance with QLP. Our provenance model supports a high-level query language for provenance (QLP) [2], that allows users to easily express complex provenance queries. QLP queries can be posed against flow graphs through a number of different QLP constructs, some of which are described below. These constructs are used to query distinct *dimensions* of the flow graph representing: (i) *dependency paths* over nodes and invocations; (ii) *flow relations* among input and output structures of invocations; and (iii) *structural relations* among nodes within and across data structures.



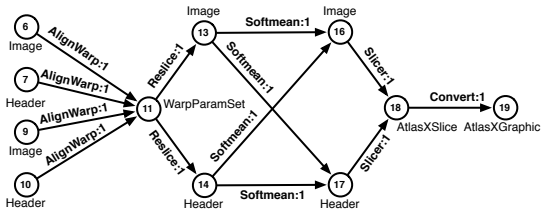
(a). The first provenance challenge fMRI workflow graph

(b). Flow graph with fine-grained node dependencies for the first invocation of each actor

Figure 2: (a) Example XML-based workflow implementing the fMRI image analysis of the first provenance challenge; (b) The flow graph showing the first invocations of each actor for a typical run.



(a). Invocation dependency graph



(b). Fine-grained data dependency view for (a)

Figure 3: (a) The implied invocation dependency graph for the run, with the first invocations of each actor shown in red; and (b) The implied fine-grain data dependency graph for the data items in Fig. 2 (b).

A QLP *path query* acts as a filter over the paths of a dependency graph to return a corresponding subgraph. For example, the following QLP dependency queries

```
* derived 19                                   (1)
6 derived *                                    (2)
#Softmean through #Convert derived *           (3)
```

return (1) dependencies denoting the set of paths starting from any node and ending at node 19, (2) dependencies denoting the set of paths starting at node 6 and ending at any node, and (3) dependencies denoting the set of paths starting at invocations of Softmean and go through invocations of Convert.

A QLP *flow-relation query* is a type of path query that additionally filters dependency graphs based on flow relations. In particular, these queries select specific versions of nodes within a flow graph based on whether the nodes were used as inputs or outputs of specific invocations. For example, the following QLP queries

```
* @in derived 19                               (4)
18 @out Slicer:1 derived *                     (5)
```

return (4) dependencies denoting paths that start at a node in the input data structure of the workflow run and end at node 19, and (5) dependencies denoting paths that start at node 18 *positioned* in the output of the first invocation of Slicer.

A QLP *structural query* is a type of path query that additionally filters dependency graphs based on data types and structural relationships. In particular, these queries select nodes within a flow graph based on whether they satisfy XPath expressions.

For example, the following query

```
* derived //AtlasXGraphic                      (6)
```

returns (6) lineage relations denoting paths that end at AtlasXGraphic nodes.

Finally, QLP queries can combine simple path, flow-relation, and structural filters to query all dimensions of flow-graph simultaneously. For example, the following query returns the set of dependencies denoting paths that end at a descendent node of an AtlasImage collection output by a Slicer invocation.

```
* derived //AtlasImage//* @out Slicer          (7)
```

Combined queries such as (7) can be (naively) evaluated by (i) obtaining the structures resulting from @in and @out version operators, (ii) applying XPath expressions to these structures, and (iii) applying lineage queries to the resulting nodes. For example, when applied to the portion of the flow graph shown in Fig. 2b, query (7) is evaluated by: (i) obtaining the output structure of the Slicer invocation; (ii) executing the XPath query '//AtlasImage//*' over the structure obtained in (i), returning nodes 16–19; and (iii) issuing a separate lineage query for each node, i.e., '* derived 16', '* derived 17', '* derived 18', and '* derived 19', where the answer contains the unique set of resulting lineage relations.

3. NAVIGATION MODEL

While provenance query languages such as QLP can help users manage the complexity of large provenance graphs, they require knowledge of the provenance graph before queries can be issued. These languages also provide limitted support for navigating provenance graphs (namely, by repeatedly issuing different queries). Thus, additional techniques are required to help users explore large provenance graphs who *a priori* do not know which portions are relevant, who want to display the graph in an aggregated or summarized form, or who wish to quickly navigate between different provenance views. This section describes a *provenance navigation model* that is designed to help address these issues.

The navigation model provides an integrated approach for exploring, summarizing, and querying all or select portions of provenance graphs through a set of *navigation operators* (see Fig. 4). These operators allow users to: (i) explore and navigate various

Operator	*Versions*	*Effect on Current Provenance View*
expand	expand : $T \times A \rightarrow \text{Set}(I)$ expand : $T \times I \rightarrow \text{Set}(D)$	Replace an actor with its invocations, and an invocation with its dependencies
collapse	collapse : $T \times \text{Set}(I) \rightarrow A$ collapse : $T \times \text{Set}(D) \rightarrow I$	Replace invocations with their actor, and dependencies with their invocation
group	group : $T \times \text{Set}(A) \rightarrow G_A$ group : $T \times \text{Set}(I) \rightarrow G_I$	Replace actors with a composite actor, and invocations with a composite invocation
ungroup	ungroup : $T \times G_A \rightarrow \text{Set}(A)$ ungroup : $T \times G_I \rightarrow \text{Set}(I)$	Replace a composite actor with its actors, and a composite invocation with its invocations
filter	filter : $T \times Q \rightarrow \text{Set}(D)$	Filter flow graph according to a given query Q
navigate	navigate : $T \times V \times \text{Set}(Op) \rightarrow V$	Apply a set of operations to the current provenance view
standard views	ADG : $T \times V \rightarrow V$ IDG : $T \times V \rightarrow V$ FDG : $T \times V \rightarrow V$	Replace current view with actor, invocation, or flow graph view, respectively
flow-graph views	EDEP : $T \times V \rightarrow V$ CDEP : $T \times V \rightarrow V$ DDEP : $T \times V \rightarrow V$	Replace current view with expanded, collapsed, or data flow dependency view, respectively

Figure 4: Navigation model operators, where T, A, I, D, G_A, G_I, Op, Q, and V are the set of traces (flow graphs), actors, invocations, dependencies, grouped actors, grouped invocations, navigation operations, QLP queries, and views, respectively.

provenance views at different levels of granularity; (ii) summarize (or abstract) portions of views through grouping; and (iii) filter (or query) provenance views using QLP. We first describe the different views (i.e., levels of granularity) of flow graphs supported by the navigation model, we then describe the navigation approach and operators supported by the model, and end this section by describing architectural issues associated with implementing the model.

3.1 Provenance Views

A workflow specification is composed of actors together with inter-actor connections. These connections specify the desired flow of data between actors. During workflow execution, actors are executed such that data flow is constrained by the given actor connections. Each actor may be invoked multiple times during workflow execution, where each invocation receives specific data items and produces new data items that are dependent on some or all of the given input data. Thus, we consider three separate levels of granularity for viewing flow graphs in the navigation model, corresponding (from highest to lowest granularity) to the actor level, the invocation level, and the data-dependency level.

Specifically, an *actor dependency graph* (ADG) is a high-level view of a flow graph that consists of actors and their dataflow connections. An ADG consists of only those actors (and corresponding connections) that were used during workflow execution. Fig. 5a shows an example ADG in which nodes *a* and *b* are actors, and edges represent the flow of data of the workflow run. An *invocation dependency graph* (IDG) is the next lower-level view of a flow graph that consists of invocations and their dataflow connections. Fig. 5b shows an example IDG in which *a*:1, *a*:2, *b*:1, and *b*:2 denote invocations of actors *a* and *b*, respectively. The lowest level of granularity is a *flow dependency graph* (FDG), which is a view of a flow graph that contains node-level data dependencies between input and output structures received and produced by invocations. Fig. 5c shows an example FDG containing different versions of the tree structure rooted at node 1 together with the corresponding dependencies (labeled with invocations). Note that an IDG is similar to the invocation-dependency graph of Fig. 3a, whereas an FDG is similar to the data-dependency graph of Fig. 3b.

In addition, we also consider three distinct representations of an FDG (see Fig. 6) that can further simplify the display of dependencies of a flow graph. An *expanded flow dependency view* (EDEP) shows only those nodes in an FDG that participate in a dependency relationship together with the descendents of these nodes (for dependency nodes that are collections). A *collapsed flow dependency view* (CDEP) is similar to an EDEP, but does not show the corre-

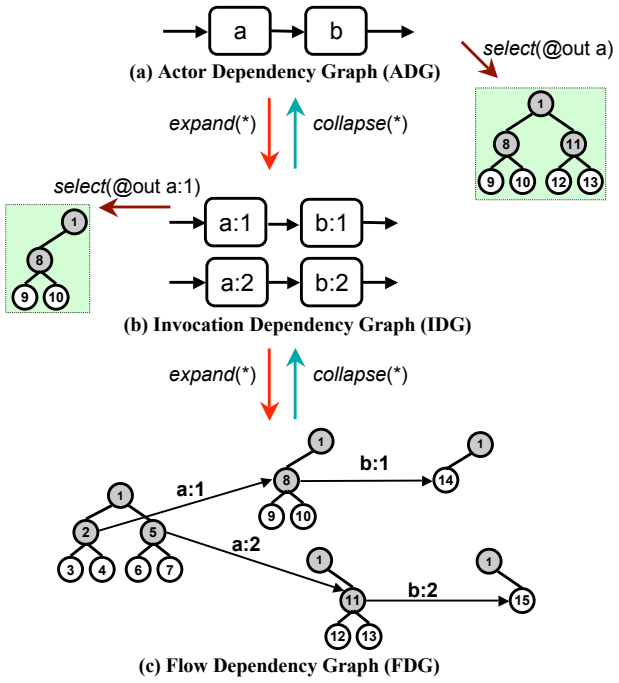


Figure 5: Three separate levels of granularity for viewing flow graphs (from highest to lowest granularity) at : (a) the actor level; (b) the invocation level; and (c) the data-dependency level.

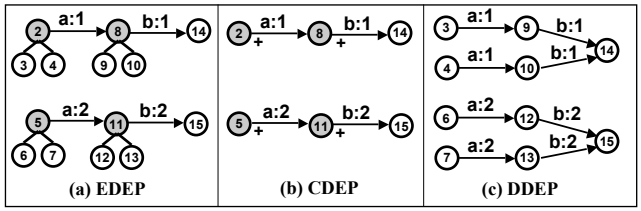


Figure 6: Three representations of a flow dependency graph: (a) Expanded Flow Dependency (EDEP) view; (b) Collapsed Flow Dependency (CDEP) view; and (c) Data Flow Dependency (DDEP) view.

sponding descendent nodes of collections. A *data flow dependency view* (DDEP) shows only dependencies of data nodes. Fig. 6 shows each of these three views for the FDG of Fig. 5c. We note that the DDEP view can be used to construct a standard data-dependency graph as in Fig. 3b. In general, the CDEP view (collapsed at the collection level) will produce the smallest flow-dependency graph and the EDEP will produce the largest flow-dependency graph of the three sub-views (CDEP, EDEP, and DDEP). Each of these, however, will be smaller than the FDG, which displays all nodes input to and output by each invocation and not just those that were used to derive new data items.

Within the navigation model, a user can switch to any of these provenance views and sub-views (ADG, IDG, FDG, EDEP, CDEP, and DDEP) from their current provenance view. That is, each view can be used as a simple form of a navigation operator (see Fig. 4) that replaces the current view with the corresponding view. This allows users to bring all elements in the provenance view to the same level of granularity.

Although not shown in Fig. 4, the navigation model also supports a *select* operator that allows users to pick items within a view to display various details of the item. Selecting an item, however, does not modify the current provenance view. For example, by selecting an invocation, a user can determine the parameter values passed to the invocation, the duration of the invocation, and so on. Similarly, by selecting a connection between two invocations, a user can see the details of the data structure passed between them. For example, in Fig. 5b, if a user selects the edge between invocation a:1 and b:1, (shown as a QLP expression), a new pop-up window would open in a browser and display the structure produced by invocation a:1. Selecting an edge in an actor dependency graph would similarly display the structures of the data passed between the invocations of each of the two actors. For example, in Fig. 5a, if a user selects the output edge of actor a (shown as a QLP expression), then all outputs of invocations of a that were provided to invocations of actor b are displayed (referred to as a "combined" view of the data structure).

3.2 Navigating Provenance Views

Besides navigating to specific, pre-defined provenance views, the navigation model also provides the collapse, expand, group, ungroup, and filter operators for constructing new views (see Fig. 4). Given a set of these navigation operators, a new view is constructed using the *navigate* operation. If v_i is the current provenance view (i.e., a provenance graph), t is the trace, and $\{op_1, op_2, \dots\}$ is a set of operators,

$$\text{navigate}(t, v_i, \{op_1, op_2, \dots\}) = v_{i+1}$$

returns the new provenance view v_{i+1} that results from applying the navigation operators to v_i over trace t.

We describe each of the navigation operators below. We assume that w is a workflow consisting of actors A, and that t is a trace (i.e., a flow graph) of a run of w that contains invocations I, dependencies D, and data structures S. Note that a dependency $\langle n_1, i, n_2 \rangle \in D$ states that node n_1 was used by invocation i to produce node n_2 where n_1 and n_2 are each part of structures s_1 and s_2 input to and output by i, respectively.

3.2.1 Expand and Collapse

Instead of displaying the entire provenance graph at the same level of granularity, the *expand* and *collapse* operators allow users to explore separate portions of the graph at different levels of detail. We consider two versions of the expand operator. Given an actor

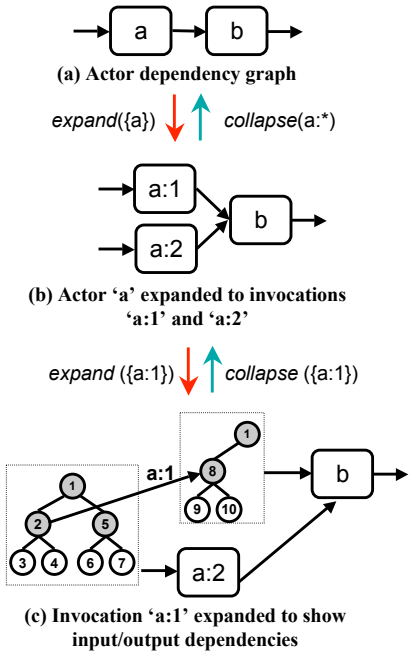


Figure 7: Applying expand and collapse operators to only a portion of provenance views.

$a \in A$ of trace t,

$$\text{expand}(t, a) = \{i_1, i_2, \dots\}$$

returns the set of invocations $i_1, i_2, \dots \in I$ of a in t. Alternatively, given an invocation $i \in I$ of trace t,

$$\text{expand}(t, i) = \{d_1, d_2, \dots\}$$

returns the set of dependencies $d_1, d_2, \dots \in D$ introduced by i in t, where $d = \langle x, i, y \rangle$ for nodes x and y.

The collapse operator acts as the inverse of expand. Namely, given a set of of dependencies $\{d_1, d_2, \dots\} \in \text{Set}(D)$ generated by an invocation i, where $\text{Set}(D)$ denotes the powerset of D,

$$\text{collapse}(t, \{d_1, d_2, \dots\}) = i$$

returns invocation i. Note that a user will typically select a single dependency to collapse, which will result in all such dependencies of the same invocation to also collapse.

Similarly, given a set of invocations $\{i_1, i_2, \dots\} \in \text{Set}(I)$ of an actor a,

$$\text{collapse}(t, \{i_1, i_2, \dots\}) = a$$

returns actor a. Note that when a user selects only a single invocation to collapse, this operation will result in all such invocations of the same actor to also collapse.

To illustrate, Fig. 5 shows the result of applying the expand operator to the actor dependency graph in Fig. 5a, resulting in the new graph view shown in Fig. 5b. In this example, all actors are expanded using the operator expression expand(*). Here we use the wildcard symbol '$*$' to denote the set of all actors in the view. When expand is applied to each invocation of Fig. 5b the view in Fig. 5c is returned. Alternatively, Fig. 7 shows the result of applying the expand operator to only a portion of the corresponding dependency graph. As shown, only actor a is expanded in the actor dependency graph of Fig. 7a, which results in the (mixed) view of Fig. 7b. Expanding invocation a:1 in Fig. 7b results in the

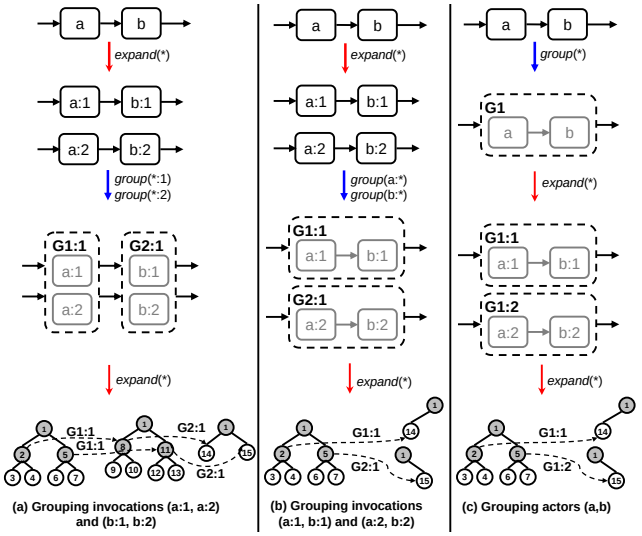


Figure 8: Actors and invocations combined into composite structures: (a) groups of invocations of the same actors $(a:1,\ a:2)$ and $(b:1,\ b:2)$; (b) groups of invocations with the same invocation numbers $(a:1,\ b:1)$ and $(a:2,\ b:2)$; and (c) grouping of actors a and b followed by the expansion of the composite actor into composite invocations.

provenance view shown in Fig. 7c, which contains each level of granularity—actors, invocations, and flow dependencies—within a single graph.

3.2.2 Group and Ungroup

The *group* and *ungroup* navigation operators allow actors and invocations to be combined into composite structures. Unlike other approaches [12] that infer groups based on a user's selection of "relevant" actors, the navigation model explicitly allows users to control which items should be grouped and supports both actor and invocation granularity.

We consider two versions of the group and ungroup operators. Given a set of actors $\{a_1, a_2, \ldots\}$,

$$\text{group}(t, \{a_1, a_2, \ldots\}) = g_{\{a_1, a_2, \ldots\}}$$

returns a composite actor $g_{\{a_1, a_2, \ldots\}}$ over the given set of actors. Similarly, given a composite actor $g_{\{a_1, a_2, \ldots\}}$,

$$\text{ungroup}(t, g_{\{a_1, a_2, \ldots\}}) = \{a_1, a_2, \ldots\}$$

returns the set of actors corresponding to the group (i.e., ungroup is the inverse of group). Similarly, for a set of invocations $\{i_1, i_2, \ldots\}$,

$$\text{group}(t, \{i_1, i_2, \ldots\}) = g_{\{i_1, i_2, \ldots\}}$$

returns a composite invocation $g_{\{i_1, i_2, \ldots\}}$; and given a composite invocation $g_{\{i_1, i_2, \ldots\}}$,

$$\text{ungroup}(t, g_{\{i_1, i_2, \ldots\}}) = \{i_1, i_2, \ldots\}$$

returns the set of invocations that comprise the group.

Fig. 8 shows three examples of using the group operator. In Fig. 8a, invocations of the same actor are grouped, i.e., a:1 and a:2 form one group and invocations b:1 and b:2 form a different group. Here we use the shorthand notation $*$:1 and $*$:2 to construct these groups. In Fig. 8b, invocations with the same invocation number are grouped, i.e., a:1 and b:1 form one group and invocations a:2 and b:2 form a different group. Similar to Fig. 8a, we use the shorthand notation a:$*$ and b:$*$ to form the groups in Fig. 8b. In Fig. 8c, actors a and b are grouped, resulting in a composite actor with two distinct invocations. Unlike in Fig. 8b, these invocations are of the same actor group and have different invocation numbers, whereas in Fig. 8b two distinct groups are created. In general, forming invocation groups explicitly, as opposed to first forming actor groups and then expanding actor groups, supports grouping at a finer-level of granularity by allowing various patterns of composite invocations that are not possible to express at the actor level.

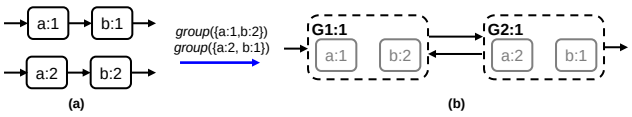


Figure 9: An example of an invalid grouping of invocations causing a cycle in the resulting invocation dependency graph.

As shown in Fig. 8, composites created by the group operator are assigned new identifiers. In addition, the inputs, outputs, and dependencies associated with grouped items are inferred from the underlying inputs, outputs, and dependencies of the invocations of the groups. For dependencies in particular, this often requires computing the transitive closure of dependencies associated with invocations of the group, e.g., as in Fig. 8b-c.

When a group is created at the actor level, expanding the group results in a correspondingly grouped set of invocations, e.g., as shown in Fig. 8c. These invocations are constructed based on the invocation dependency graph. In particular, each invocation group of the actor group contains a set of connected invocations, and no invocation within an invocation group is connected to any other invocation in a different invocation group. Thus, the portion of the invocation graph associated with the actor group is partitioned into connected subgraphs, and each such subgraph forms a distinct invocation group of the actor group. Similarly, when an invocation group is expanded, this composite invocation is used in the flow dependency graph, resulting in a provenance view where dependencies are established between output and input data, without intermediate data in between. This approach allows scientists to continue to explore dependencies for grouped invocations (since the dependencies are maintained through groups).

We limit the use of the group operator such that the resulting actor and invocation dependency graphs with composites remain cycle free. Thus, grouping a set of invocations or actors should not introduce cycles in the ADG or IDG. For example, Fig. 9 shows invocations *a:1* and *b:2* grouped as *G1:1* and *a:2* and *b:1* grouped as *G2:1*. The invocation dependency graph that results has the output of *G1:1* connected to the input of *G2:1* and vice-versa, thereby resulting in a cycle between *G1:1* and *G2:1*. The navigate operator checks to ensure that a given group operation will not result in cyclic dependency graphs.

3.2.3 Filter

Besides grouping and expanding provenance views, the navigation model also allows provenance views to be queried using QLP. Issuing a query results in the portion of the provenance view to be displayed that corresponds to query answer. A provenance graph is refined in this way using the *filter* operation of Fig. 4. Given a QLP query q,

$$\text{filter}(t, q) = \{d_1, d_2, \ldots\}$$

returns the set of dependencies that result from applying the query to the flow graph. The navigate operation uses these dependencies

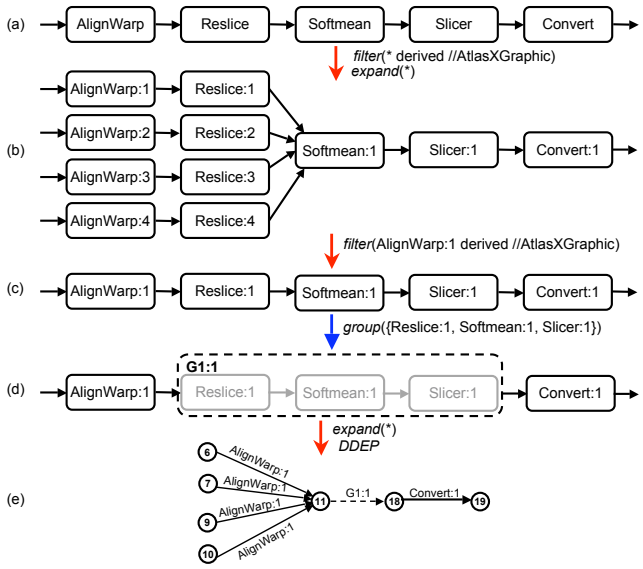


Figure 10: Exploring and managing provenance graphs: visualizing, summarizing, filtering, and navigating relevant sections of provenance views.

to either remove or add items to the current view. Items are added to a view if the current view is based on a more selective query.

3.2.4 *Combining Operators*

Here we describe a simple example that illustrates how the integrated environment provided by the navigation model enables users to explore provenance information by visualizing, summarizing, and filtering relevant sections of provenance views. Consider a user who is interested in exploring the provenance of an execution of the fMRI workflow of Section 2. Using the navigation model, the user begins by displaying the provenance of the run at the highest level of granularity, i.e., by visualizing the actor dependency graph (ADG) as shown in Fig. 10a.

Assume that the user first decides to view the outputs generated by the workflow, which is performed by issuing the *select* operation over the outgoing edge of the actor *Convert* in Fig. 10a. After examining the output structure, assume that the user notices that one of the data products—namely, the *AtlasXGraphic*—is not the expected output, i.e., the data product seems to be incorrect.

To check whether the product was generated correctly, the user decides to display the lineage of the *AtlasXGraphic* in more detail by navigating from the current actor dependency view to the invocation dependency view for this data item. The user creates this view by configuring a *filter* operator using the QLP query

* derived //AtlasXGraphic,

which selects only those items that share dependency relationships with the data item *AtlasXGraphic*. The user also applies the operator expand(*) to display the invocation dependency graph (relative to the filter). After specifying these operators, the user applies the *navigate* function to generate and display the new provenance view, which is shown in Fig. 10b.

While analyzing this new provenance view, the user suspects that the first invocation of each displayed actor (*:1) might have led to the incorrect *AtlasXGraphic* output (e.g., based on inspecting the parameters or intermediate products produced by one or more of the invocations). To explore whether the problem is due to these invocations, the user further refines the provenance view by displaying only the first invocation of actors that share a dependency relationship with the *AtlasXGraphic* data product by issuing the *filter* operator using the QLP query

AlignWarp:1 through //AtlasXGraphic,

which selects the lineage items that start from *AlignWarp:1* and end in *AtlasXGraphic*. After applying the navigate function, the resulting provenance view is shown in Fig. 10c.

Before looking at the detailed data items and dependency relationships, the user decides to further simplify the invocation dependency graph by grouping the *Reslice*, *Softmean*, and *Slicer* invocations into a single composite invocation. In particular, this group restricts the data items in the view to the output of *AlignWarp* and the input to *Convert*. The summarization operation is performed by specifying a group over the three invocations, resulting in the provenance view of Fig. 10d.

The user is now ready to analyze the fine-grained data dependencies among the relevant portions of the provenance graph. The user first expands the resulting invocations (producing a flow dependency graph), and then applies the *DDEP* view operator, as shown in Fig. 10e. The user can now analyze the dependency relationships to verify that the relevant intermediate data products are correct and that they were correctly derived based on the dependencies (e.g., to check wether the correct warping parameters were used and that the slice was correctly generated). Note that the dashed edge between node 11 and 18 denotes that the dependency is the result of a composite invocation. If the user wishes to further explore the dependencies represented by the composite invocation, the edge can be ungrouped by operator ungroup(G1:1) to expose the detailed dependency information, which in this case would result in the data dependency graph of Fig. 3b.

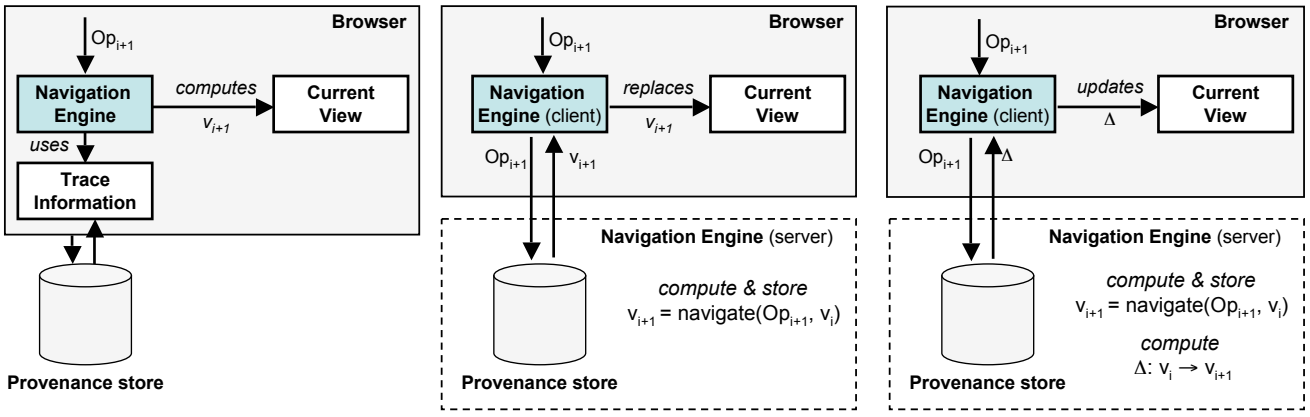
This example demonstrates how the operations of the navigation model can provide a flexibile approach for summarizing, refining, and navigating different provenance views, which is essential for users needing to explore and manage large provenance graphs.

3.3 System Architecture

The navigation model allows users to navigate from the current provenance view v_i to another provenance view v_{i+1} by applying a set of navigation operators Op_{i+1}. Once a user issues a set of navigation operators Op_{i+1}, the navigation system must compute a new provenance view v_{i+1}, which then replaces the display of the current view v_i. While exploring large provenance graphs, we expect users to repeatedly issue navigation operations, which would result in frequent changes to the current provenance view. Especially for large provenance graphs, it is important to ensure fast response time so that new views can be computed and displayed quickly. Here we compare different architectures and approaches for implementing the navigation model over large provenance graphs, as shown in Fig. 11.

Each of the alternative architectures of Fig. 11 have the following common components: a server-side *Provenance Store* that acts as a database repository for provenance traces; a current provenance view v_i that is displayed by the *Current View* component; and a *Navigation Engine* that stores the current view, receives a new set of navigation operations Op_{i+1} from the user, computes the new provenance view v_{i+1} from user operations, and updates the current view v_i with the new computed view v_{i+1} via the *Current View* component.

In the architecture of Fig. 11a, the Navigation Engine is located at the client. To compute the initial provenance view v_1 for a workflow run, all related provenance information stored in the Prove-



(a) Compute views on client (browser)

(b) Compute views on server (database)

(c) Compute Δ of views on server (database)

Figure 11: Different architectures for implementing the navigation model: (a) new views computed at the client (browser); (b) new views computed at the server (database); and (c) the difference (Δ) between current and new views computed at the server (database).

nance Store is initially loaded into a client-side Trace Information component. The Navigation Engine uses the trace stored at the Trace Information component to compute the initial view v_1 (i.e., the actor dependency graph), and this view is displayed by the Current View component. When a user issues a new set of navigation operations Op_{i+1} against the current view v_i, the Navigation Engine computes a new view v_{i+1} by applying the set of operators Op_{i+1} to the recently stored view v_i, and then updates the current view v_i with new computed view v_{i+1}. This approach is largely an in-memory-based approach, i.e., the related provenance information is first loaded from the server (via the Provenance Store) to the client (via the Trace Information component). This in-memeory approach can speed up view computation, as the required provenance information is temporarily stored at the client. However, depending on the size of the trace, this approach may require significant CPU and memory resources [1] for answering queries and for storing the initial trace, data products, query results, and views. This cost also includes the time required to transfer the trace information from the Provenance Store to the client-side application, which for larger traces can take a considerable amount of time.

Fig. 11b suggests a different architecture in which the Navigation Engine is split between the client and server. In particular, when the client-side Navigation Engine receives a new set of navigation operators Op_{i+1} from the user, these operations are sent to the server-side Navigation Engine (which includes the Provenance Store). The server-side Navigation Engine computes and stores the new view v_{i+1} on the server and then sends the new view to the client. Thus, the architecture of Fig. 11b differs from Fig. 11a by storing the trace and computing and storing new views on the server, thus avoiding the storage and computation costs needed for the client in Fig. 11a.

We can extend the approach of Fig. 11b in Fig. 11c by additionally minimizing the amount of information sent to construct a new view, thus (for many cases) reducing the communication costs between the client and server. In particular, in addition to computing the new view v_{i+1} on the server, we also compute a set of changes (or "diffs") $\Delta : V \rightarrow V$ between the current view v_i and the new view v_{i+1} such that $\Delta(v_i) = v_{i+1}$. In the normal way, each Δ consists of a set of nodes and edges that should be added to and removed from v_i to generate v_{i+1}. Instead of returning v_{i+1} (which will typically overlap with v_i), we only return the set of changes Δ, which are used by the Current View component to display the new view. Both of the approaches in Fig. 11b and Fig. 11c can help to reduce the storage and processing costs of the client application in terms of computing new views for large-sized provenance graphs, since these approaches push view processing and provenance storage to the server. Also, computing and sending only the changes between views (for the cases when changes to views are small) as in Fig. 11c can further speed up navigation response time compared to sending the entire view as in Fig. 11b. This can be seen in even the small exampe of Fig. 10, where each navigation step generally only makes small changes to the previous view. We can also further extend Fig. 11c by sending the smaller of (i) the changes Δ between views and (ii) the new view. In particular, if the new view is smaller than the corresponding Δ, then the server-side Navigation Engine would return the new view, whereas if the size of the changes Δ is smaller, then the set of changes are sent.

In our prior work [6] we describe an interactive tool for browsing provenance that can display different provenance views for scientific workflow traces. The provenance browser has been integrated with the Kepler Scientific Workflow System [18], and can also be run as a standalone application. Using the provenance browser, a user can connect to a provenance store to display various provenance views of the execution trace: the *dependency history* view (Fig. 1) combines data dependency and process invocation graphs (where data nodes are denoted as cirlces and invocations as squares); the *collection history* view shows the data structures input and output by invocations; and the invocation graph view shows process dependencies. Each of these views are synchronized, e.g., selection of a data item in the dependency history view also selects the corresponding item in the collection history view. In a view, users can also step forward and backward ("VCR-style") through the execution history to display corresponding portions of the XML structures and data dependencies. As future work, we intend to leverage the provenance browser to support the proposed architecture of Fig. 11c. Our goal is to extend the provenance browser to support the navigation model described here, thereby providing users with an integrated environment to flexibly visualize, summarize, query, and navigate large provenance graphs.

4. RELATED WORK

Automatically recording provenance information during workflow execution is one of the important added values of scientific workflow systems over more conventional script-based approaches

[19, 25]. However, providing techniques for effectively representing, managing, and accessing the large amounts of provenance information generated by workflow systems presents a number of technical challenges [12, 25]. The navigation model presented here helps to address a number of these challenges by providing users with an approach for exploring and viewing relevant portions of provenance information using intuitive and natural graph-based provenance representations. Our navigation model is based on a generic provenance representation scheme [1] that extends conventional approaches to support a wide range of workflow systems. Our model also combines approaches for querying (i.e., using QLP [2]) and summarizing workflow graphs (based on composites), while offering additional abstract views of provenance information (actor, invocation, and flow dependency graphs) and the ability to navigate between views (e.g., using expand and collapse). The remainder of this section describes related work and compares our work to existing approaches.

Scientific workflow systems are being used in many scientific domains, and many approaches have been proposed recently for representing workflow provenance (e.g., [26, 21, 12, 25, 7, 17]). Most existing approaches for representing provenance do not consider workflow computation models that work over structured data, including XML. Standard provenance representation schemes (e.g., [17, 15, 3, 5] among others) largely assume that workflow models are based on *transformation semantics* in which each workflow step consumes all input data and produces entirely new output data. Alternatively, workflow models that work over structured data (e.g., [6, 27, 20, 28]) often employ *update semantics*, where only a portion of an incoming XML stream is modified by each workflow step. Our navigation model is based on a generic model of provenance that subsumes conventional approaches for representing workflow provenance while supporting more advanced workflow computation models permitting multiple invocations of processes (e.g., for pipelining and loops), structured data, and update semantics [1].

Current approaches for exploring workflow provenance are based on visualizing entire provenance graphs or specific views of these graphs, such as data and invocation dependency graphs [6, 25, 16, 17, 21]. In these approaches, provenance graphs are typically displayed at the lowest level of granularity. Some systems further divide provenance information into distinct layers. For example, myGrid [29] divides provenance into data, process, organisational, and knowledge levels; VisTrails [8, 25, 4] divides provenance information into workflow evolution, workflow, and execution layers; Redux [3] divides provenance into runtime execution, data instantiation, abstract service, and service instantiation layers; and the Provenance Aware Storage System (PASS) [22] divides provenance into data and process layers. In all of these aproaches, however, these levels are largely either orthogonal or hierarchical, whereas the provenance views supported by our navigation model (i) combine both hierarchical abstractions (i.e., ADGs, IDGs, and FDGs) with (ii) the ability to seamlessly navigate between these different levels of granularity, while (iii) allowing users to summarize, group, and filter portions of these views to create new views for further exploration of relevant provenance information.

Unlike standard provenance approaches, the Zoom*UserViews system [11, 5] provides a mechanism for defining composite actors to abstract away non-relevant provenance information. The basic approach is to allow users to select one or more "relevant" actors from a workflow specification graph, and based on these selections, the system creates associated composite actors that contain at most one relevant actor. The composite actors are constructed in such a way as to maintain certain dataflow connections, thereby generating a workflow over the composites that is similar (in terms of dataflow) to the original. However, unlike in our approach, users of the Zoom*UserViews system cannot explicitly define their own composites, and composition is defined only at the actor level (where each actor is assumed to have at most one invocation). Our approach also maintains grouping across views (including the ability to ungroup composites within these views), maintains the original data dependencies (i.e., dependencies within composites are maintained, unlike in general within the Zoom*UserViews approach), and we support a more general provenance model that explicitly handles structured data.

Our navigation approach is inspired by and has similarities to those proposed previously for exploring object-oriented and XML databases, where graphical environments were developed that allow users to "drill-down" from schema to instances and navigate relationships among data. For example, PESTO [9] provides an integrated browsing and querying environment that allows users to employ a "query-in-place" paradigm for exploring the contents of object databases. In particular, PESTO allows users to mix navigation and query in which queries can be issued relative to a position in the database reached through navigating object relationships. Similarly, in Blended Browsing and Querying (BBQ) [23], a graphical user interface is provided that supports both browsing and querying of XML data. Like PESTO, querying in BBQ is schema driven, requiring users to know the details of the (Object-Oriented or XML) schema prior to issuing queries. Also, a standard XML-based web-based navigation and visualization approach tailored to clinical provenance information is proposed in [13]. In contrast, provenance information is largely schema-free, i.e., the information contained within an ADG, IDG, and FDG is not constrained by an explicit schema, and queries in our model are posed directly against the items contained within these views (or generally the flow graph). In addition, our provenance model is considerably more specialized than the more generic data models supported by PESTO and BBQ, resulting in navigation operators (such as expand and group) that are tailored specifically to provenance information.

Finally, our navigation model is the first approach that we are aware of that combines navigation, abstraction (through composition), and query capabilities. A number of systems allow users to query provenance information, however, these approaches largely rely on physical representations of provenance information [12] (e.g., relational, XML, or RDF schemas), where users express provenance queries against these schemas using corresponding query languages (i.e., SQL, XQuery, or SPARQL). Provenance queries, however, often require computing transitive closures over dependency relations, and expressing such queries using standard approaches is typically done using recursion or stored procedures [14, 10, 1]. Expressing such queries is both cumbersome and error-prone, and requires considerable user expertise. Instead, high-level languages such as QLP provide a separation between the logical provenance model and its underlying physical representation, which allows for the use of different representation schemes and additional optimization techniques. Our approach, in particular, automatically translates QLP queries to equivalent relational queries expressed against the provenance storage schemes described in [1]. Standard approaches for querying provenance information (e.g., [24, 15, 30, 5]) return sets of nodes (either sets of data items or process invocations) as the query result. This approach requires additional steps (queries) to reconstruct causal relations among nodes within a query answer. Instead, QLP is closed under lineage relations, where answers to lineage queries are sets of lineage dependencies (edges) forming provenance subgraphs, and thus query results are "provenance preserving". This approach has a number

of advantages, e.g., for supporting provenance views, incremental querying, and for supporting visualization [12, 6].

5. CONCLUSION

We propose an approach to address a number of open issues in effectively exploring large provenance information generated from complex scientific workflows. Specifically, we define a novel *navigation model* for scientific workflow provenance that consists of operations for creating, refining, and switching between different provenance views. The navigation model provides an integrated approach to: (i) create abstract views of provenance information (actor, invocation, and flow dependency graphs); (ii) seamlessly navigate between these views; (iii) summarize portions of views through grouping actors or invocations; and (iv) filter (or query) provenance views using QLP, a high-level provenance query language. We also present different architectures for efficiently navigating large provenance graphs against an underlying provenance database. Our navigation model is the first approach that we are aware of that provides capabilities of visualizing, navigating, summarizing, and querying provenance views in an integrated environment. The approaches described here extend our prior work on browsing and query provenance information by providing explicit navigation operations and views for abstracting and summarizing provenance graphs. Combined with the provenance browser, these approaches can provide a powerful environment for scientists to explore and validate the results of scientific workflows, especially those that involve large and complex data sets and large numbers of interconnected processes.

Acknowledgments. This work supported in part by NSF grants IIS-0630033, OCI-0722079, IIS-0612326, ATM-0619139, and DOE grant DE-FC02-07ER25811.

6. REFERENCES

[1] M. K. Anand, S. Bowers, T. M. McPhillips, and B. Ludäscher. Efficient provenance storage over nested data collections. In *EDBT*, 2009.

[2] M. K. Anand, S. Bowers, T. M. McPhillips, and B. Ludäscher. Exploring scientific workflow provenance using hybrid queries over nested data and lineage graphs. In *SSDBM*, 2009.

[3] R. S. Barga and L. A. Digiampietri. Automatic capture and efficient storage of e-science experiment provenance. *Concurr. Comput. : Pract. Exper.*, 20(5), 2008.

[4] L. Bavoil, S. P. Callahan, C. E. Scheidegger, H. T. Vo, P. Crossno, C. T. Silva, and J. Freire. Vistrails: Enabling interactive multiple-view visualizations. In *IEEE Visualization*, 2005.

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

[6] S. Bowers, T. McPhillips, S. Riddle, M. Anand, and B. Ludäscher. Kepler/pPOD: Scientific workflow and provenance support for assembling the tree of life. In *IPAW*, 2008.

[7] S. Bowers, T. M. McPhillips, B. Ludäscher, S. Cohen, and S. B. Davidson. A model for user-oriented data provenance in pipelined scientific workflows. In *IPAW*, 2006.

[8] S. Callahan, J. Freire, E. Santos, C. Scheidegger, C. Silva, and H. Vo. VisTrails: Visualization meets data management. In *SIGMOD*, 2006.

[9] M. J. Carey, L. M. Haas, V. Maganty, and J. H. Williams. Pesto: An integrated query/browser for object databases. In *VLDB*, 1996.

[10] A. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *SIGMOD*, 2008.

[11] S. Cohen, S. C. Boulakia, and S. B. Davidson. Towards a model of provenance and user views in scientific workflows. In *DILS*, 2006.

[12] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD*, 2008.

[13] V. Deora, A. Contes, O. F. Rana, S. Rajbhandari, I. Wootten, K. Tamas, and L. Z. Varga. Navigating provenance information for distributed healthcare management. In *WI*, 2006.

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

[15] D. Holland, U. Braun, D. Maclean, K.-K. Muniswamy-Reddy, and M. Seltzer. A data model and query language suitable for provenance. In *IPAW*, 2008.

[16] J. Hunter and K. Cheung. Provenance explorer-a graphical interface for constructing scientific publication packages from provenance trails. *Int. J. Digit. Libr.*, 7(1), 2007.

[17] L. Moreau, *et al*. The open provenance model. Technical Report 14979, ECS, Univ. of Southampton, 2007.

[18] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurr. Comput. : Pract. Exper.*, 18(10), 2006.

[19] T. McPhillips, S. Bowers, D. Zinn, and B. Ludäscher. Scientific workflow design for mere mortals. *Future Generation Computer Systems*, 25(5), 2009.

[20] P. Missier, K. Belhajjame, J. Zhao, and C. Goble. Data lineage model for taverna workflows with lightweight annotation requirements. In *IPAW*, 2008.

[21] L. Moreau*, et al.* The first provenance challenge. *Concurr. Comput. : Pract. Exper.*, 20(5), 2008.

[22] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor. Layering in provenance systems. In *USENIX*, 2009.

[23] K. D. Munroe and Y. Papakonstantinou. Bbq: A visual interface for integrated browsing and querying of xml. In *VDB 5*, 2000.

[24] C. Scheidegger, D. Koop, E. Santos, H. Vo, S. Callahan, J. Freire, and C. Silva. Tackling the provenance challenge one layer at a time. *Concurr. Comput. : Pract. Exper.*, 20(5), 2008.

[25] C. Silva, J. Freire, and S. P. Callahan. Provenance for visualizations: Reproducibility and beyond. *Computing in Science & Engineering*, 9(5), 2007.

[26] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD*, 34(3), 2005.

[27] A. Slominski. Adapting bpel to scientific workflows. In *Workflows for e-Science Scientific Workflows for Grids*. 2007.

[28] Tom Oinn, *et al.* Taverna: lessons in creating a workflow environment for the life sciences. *Concurr. Comput. : Pract. Exper.*, 18(10), 2006.

[29] J. Zhao, C. Goble, R. Stevens, and D. Turi. Mining taverna's semantic web of provenance. *Concurrency and Computation: Practice and Experience*, 20(5), 2008.

[30] Y. Zhao and S. Lu. Logic programming approach to scientific workflow provenance querying. In *IPAW*, 2008.