Project Histories: Managing Data Provenance Across Collection-Oriented Scientific Workflow Runs*

Shawn Bowers[1], Timothy McPhillips[1], Martin Wu[1], and Bertram Ludäscher[1,2]

[1] UC Davis Genome Center, University of California, Davis
[2] Department of Computer Science, University of California, Davis
{sbowers, tmcphillips, martinwu, ludaesch}@ucdavis.edu

Abstract. While a number of scientific workflow systems support data provenance, they primarily focus on collecting and querying provenance for single workflow runs. Scientific research projects, however, typically involve (1) many interrelated workflows (where data from one or more workflow runs are selected and used as input to subsequent runs) and (2) tasks between workflow runs that cannot be fully automated. This paper addresses the need for recording data dependencies across multiple workflow runs and accommodating data management activities performed between runs. We define a new conceptual model for representing project-level provenance based on the notion of project histories and folders, and describe mechanisms to support this model in the collection-oriented modeling and design framework of KEPLER. Our approach allows users to conveniently organize their projects and data using the familiar folder-hierarchy metaphor, while at the same time integrating this information with detailed provenance of data products generated via automated scientific workflows.

1 Introduction

Scientific workflows promise to *automate* complex and repetitive operations, *model* (*i.e.*, clarify for the scientist) the tasks being automated, and *record* how results of workflow runs were computed from input data. However, few results of great significance are likely to emerge from a single run of one scientific workflow. Novel research involves project organization, data exploration, decision making, and trial-and-error activities that cannot be automated in advance. Researchers employing scientific workflow automation generally wish to run a number of distinct workflows in the context of a single project, apply workflows multiple times on different data or with different parameter settings, modify workflows, and compose completely new workflows as needed. As a result, data provenance support in scientific workflow systems is likely to be of limited usefulness unless the flow of data can be tracked rigorously across multiple workflow runs. Because researchers require the freedom to organize their data and projects as they see fit, comprehensive workflow provenance support also necessitates the recording of data management activities performed manually by researchers between workflow runs. This paper describes how the Collection-Oriented Modeling and Design (COMAD) paradigm

* This work supported in part by NSF grants DBI-053368, EAR-0225673, IIS-0630033, IIS-0612326, and EF-0228651; and DOE grant DE-FC02-01ER25486.

S. Cohen-Boulakia and V. Tannen (Eds.): DILS 2007, LNBI 4544, pp. 122–138, 2007.

[11] may be extended to provide project-scale data provenance support while enabling users to freely organize their projects and data using the familiar nested-folder metaphor.

1.1 Nested Collections for Scientific Data Management

Nested collections are a ubiquitous organizational scheme for scientific data. The large number of standards based on XML testify to the naturalness of representing data as nested collections. Digital libraries and data archives also commonly exploit this metaphor to provide a primary organization for discovering data and exploring collections. While technological inertia may explain some use of the nested collection metaphor, there are more fundamental reasons for its predominance in the realm of scientific data management.

- **Hierarchical structure of natural systems.** Much scientific data can be represented readily as nested collections due to the hierarchical structure of the natural world. For example, the structure of a protein molecule can be described as successively nested collections of polypeptide chains, amino acid residues, and atoms each with their own identifiers and attributes.
- **Intuitive nested file folder metaphor.** Many individuals (including scientists) find the nested collection metaphor an intuitive way to organize information [9]. Folders are often nested and named in ways that reflect significant associations between files. Further, the meaning attached to a particular folder generally cascades to subfolders, where the resulting hierarchy represents a nested collection of metadata that annotates contained files.
- **Projects, tasks, and subtasks.** Projects are generally structured hierarchically. Project tasks can be broken down into (sometimes ordered) collections of subtasks. For this reason, files created during a project often are organized according to a nested folder scheme reflecting the task-hierarchy of the project [9]. The contents of the folders represent data used or created while carrying out the project and together reflect the state of the project at a particular point in time. For scientists in particular, storing information in nested collections is a natural way of persisting data between research tasks.
- **Operations that generate lists of lists.** Many experiments and calculations generate lists of results. For example, a BLAST search can take one molecular sequence and return a list of genes containing similar sequences. A search within the promoter for a gene can reveal a list of over-represented sequences or motifs. When tasks are performed in series the results often are more deeply nested hierarchies, *e.g.*, collections of genes with sub-collections of sequence motifs.

For the above reasons it is important that systems for automating scientific workflows respect, preserve, and ideally exploit the hierarchical structure of scientific data and project information. Workflow systems that lack such support require users to repeatedly map back and forth between a project data organization intuitive to them and the data models employed by the automation system. In contrast, CoMaD was envisioned precisely for the purpose of managing data organized within nested collections during workflow execution and facilitating this relatively unrecognized yet common component of doing science.

1.2 Leveraging Nested Collections in Scientific Workflow Automation

The CoMAD framework in KEPLER [10] extends the conventional actor-oriented framework provided by PTOLEMY II[1] to support collection-oriented workflows. Unlike workflows built from conventional actors, workflows employing collection-aware actors, or *co-actors*, transparently manage data organized as nested collections, maintain the associations implied by the collections, and exploit generic approaches for operating on streams of collections. Abstracting the generic data management tasks associated with collections from the workflow definition in this way often drastically reduces the complexity of the workflow graph (*i.e.*, fewer actors and actor connections are required) with the result that collection-oriented workflows generally are simpler to compose, understand, and maintain than their conventional counterparts. The approach also largely decouples the structure of the workflow from the structure of the data flowing through it, producing workflows that are more reusable, *i.e.*, workflow definitions need not change when the structure of input data changes. In KEPLER, CoMAD leverages the nested structure of scientific data to: (1) enable concurrency-safe pipelined execution across actors connected in series or parallel; (2) associate annotations with particular collections and data; (3) dynamically deliver customized parameter values to actors operating on specific collections; and (4) efficiently capture the detailed provenance of all data and collections created during the course of workflow execution. These advantages derive directly from the CoMAD approach of exploiting the hierarchical properties of scientific data and projects, much as scientists have been doing manually for many years.

The CoMAD implementation in KEPLER delivers the above capabilities by streaming nested collections of data through co-actors as "flat" token sequences where collections are delimited using paired (opening and closing) control tokens. CoMAD provides services to actors for managing collections, *e.g.*, for constructing internal representations of collections from input token sequences, inserting and deleting collection elements, and (re-)serializing collections to output token sequences. Co-actors can declare the types of collections and data they process via *read scope* expressions. The CoMAD framework iteratively invokes actors over portions of the input stream matching these expressions. Data and collections that fall outside of an actor's read scope are automatically forwarded by the framework to succeeding actors, enabling "assembly-line" style data processing. Annotations (*e.g.*, represented as name-value pairs) are modeled explicitly in CoMAD, and may be used to represent data and collection metadata that actors create and access during workflow execution. Annotations may also be used to override actor parameters, *e.g.*, allowing co-actor behavior to be changed at runtime within the context of particular collections. Like data and collections, annotations are represented as tokens and are automatically streamed through co-actors by the CoMAD framework.

1.3 Data Provenance Within Single Workflow Runs

Accurately recording the provenance of workflow products is a critical step towards enabling scientists to incorporate workflow systems into their day-to-day research processes. The CoMAD framework records the events (*i.e.*, actor invocations) involved

[1] `http://ptolemy.eecs.berkeley.edu/ptolemyII/`

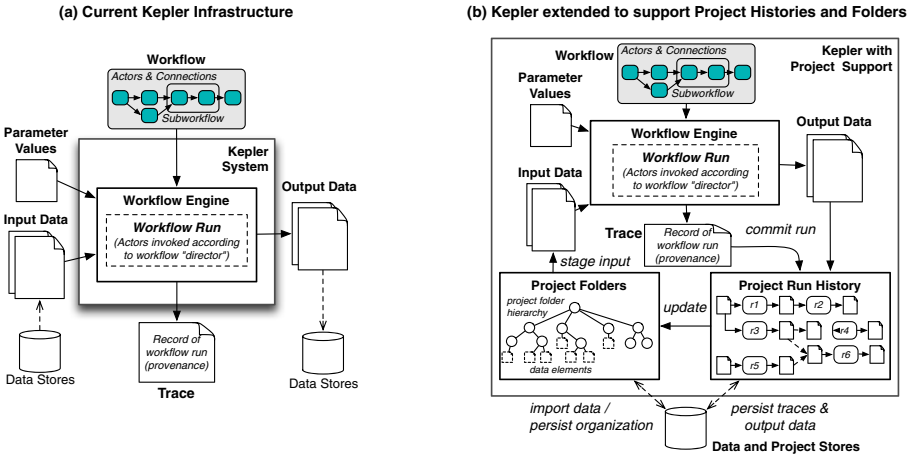

Fig. 1. Proposed extensions to Kepler for supporting project-scale provenance: (a) data management between workflow runs occurs outside of the current implementation of Kepler; and (b) envisioned support for managing data and provenance between workflow runs

in computing each workflow product along with the dependencies of these products on input and intermediate data. However, the current CoMaD implementation supports provenance for events that occur within a single run of a workflow. In general, the scope of most scientific workflow systems, *e.g.*, [5,14,15,6], is limited to single workflow runs. An exception is [4], which tracks changes to workflow definitions and caches output data to optimize subsequent workflow runs. In CoMaD, users may specify the collections, data, and metadata to be applied as input to a workflow run using an XML input file. Similarly, the results of a run, including references to any new data, collections, metadata and provenance records may be persisted as an XML trace file. However, these XML files, and any referenced external data, must be managed directly by the user. Thus, like most other scientific workflow systems, Kepler and the CoMaD implementation are largely ignorant of data management tasks carried out between workflow runs. Figure 1a depicts the current state of affairs and emphasizes that workflow definitions, input and output data, and records of workflow runs are generally maintained outside the workflow system. The remainder of this paper describes extensions to the CoMaD framework to capture and manage provenance information throughout research projects employing scientific workflows.

2 Project Histories and Folders

Figure 1b illustrates our vision for a project-aware version of Kepler. The Kepler system boundary is expanded to include workflow definitions and project data as part of the system state. Data may be imported or exported from the project folders, whereupon this data may be supplied to workflow runs. Data and traces produced by a workflow run are retained within the system and are added to the project history, together with the workflow definition and input data sets, when the run is committed. This section

introduces the primary use cases of the envisioned system through a realistic usage scenario.

A researcher $\mathcal{R}$ wishes to build an improved phylogenetic tree (*i.e.*, evolutionary history) of the bacterial kingdom using molecular sequences common to representatives of major taxa in the bacterial tree of life. Figure 2 shows the state of the project history and project folders at three points in time.

Creating the organizational structure for the project. $\mathcal{R}$ creates a new project and within it a folder named 'Genomes' for holding the annotated genomes of each of seventy bacterial species. $\mathcal{R}$ downloads files representing these genomes from NCBI[2] and The Institute for Genomic Research[3] (TIGR), imports these files into the project, and stores them in sub-folders of Genomes. $\mathcal{R}$ now creates another folder to hold information related to the thirty phylogenetic markers he intends to use to infer the evolutionary relationships between the bacterial taxa. Each marker is a protein sequence corresponding to a highly conserved gene expected to be present and identifiable in every bacterial genome. $\mathcal{R}$ creates a folder under Markers for each marker and data associated with it. Into each he imports a single reference protein sequence that will be used as a search pattern for identifying homologous sequences in each bacterial genome. The system is now aware of the raw data to be used throughout the rest of the project. Subsequent manual and automated operations will fill out the Markers sub-folders and create new top-level folders corresponding to attempts to build an updated bacterial tree of life based on available genome sequences.

Identifying markers in each genome. $\mathcal{R}$ is now ready to employ the first scientific workflow, *wf1*, which takes each reference protein sequence stored in the Marker folders, and then locates (via `BLAST` searches) and refines (using the `HMMER` [7] program suite) likely candidates for these genes in each of the bacterial genomes. The workflow accepts a stream of collections, each corresponding to a single marker and containing a reference sequence and sub-collection for each genome to search. The products of the workflow are candidate protein sequences for each marker-genome combination.

Once $\mathcal{R}$ has selected this workflow for execution, he stages the input data and workflow parameters. He drags a visual representation of the Markers collection onto a data staging widget, then requests that the contents of the Genomes collection be copied into each Marker sub-collection in the staging area. These interactive operations have no effect on the project collections themselves. After specifying values for workflow parameters, $\mathcal{R}$ starts the workflow. After the workflow run completes, $\mathcal{R}$ browses the output of the run via a workflow-product evaluation area. $\mathcal{R}$ inspects the results of the run, *e.g.*, skimming log files generated by `BLAST` and `HMMER`. Noting that the results appear reasonable, he commits the results of the workflow run to the project history and then updates the project folders with the workflow outputs. Once the run is committed, it appears in the project history panel (*wf1:r1* on left side of Figure 2a). The input and output collections of the run can also be accessed at any point in the future via the project history panel, regardless of whether the project folders are updated with the run's output data. Because $\mathcal{R}$ also updates the project folders, the protein sequences

[2] `http://www.ncbi.nlm.nih.gov/`

[3] `http://www.tigr.org/`

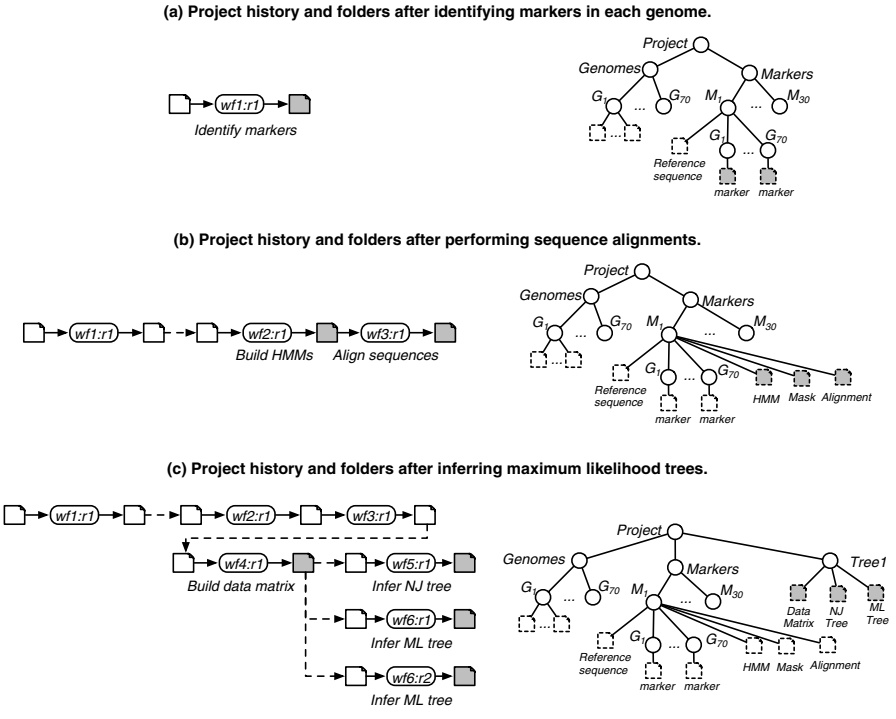

Fig. 2. Evolution of project history (left) and project folders (right) during the course of the hypothetical project described in the text

discovered during the workflow run are pasted into the corresponding project collections (right side of Figure 2a). Note that the copies of the genomes placed into the staging area are not duplicated in the project tree. Only new data is propagated to the project collections during an update.

$\mathcal{R}$ must now confirm that the markers have been unambiguously identified in each of the genomes. He browses the project folders and notes how many candidate marker sequences were discovered for each genome-marker pair. If more than one candidate was identified for a particular pair, he compares these sequences to each other and the reference sequence, and inspects statistics produced by the workflow run in an attempt to determine which sequence is orthologous to the reference sequence. $\mathcal{R}$ discards the non-orthologous sequences where this distinction can be made, and annotates any folders that still contain multiple candidate markers with metadata that will prevent these sequences from being used in downstream computations.

Creating the data matrix and maximum likelihood tree. $\mathcal{R}$ now selects a second workflow that will build a hidden Markov model (HMM) for each marker using the sequences identified in the previous run. He drags the Markers collection from the project folders panel into the staging area, enters parameter values, and starts the workflow. He evaluates the results of the run, commits the run to the project history, and updates the project collections. Each marker folder now contains a file representing the HMM.

$\mathcal{R}$ reviews the HMMs and then creates, edits, and imports a *mask* for each, indicating unreliable segments of the models. He then selects a third workflow for aligning each marker sequence to the corresponding HMM and again specifies the Markers collection as the workflow input. After committing and updating, each Markers sub-collection contains a multiple sequence alignment (see project history and folders in Figure 2b).

$\mathcal{R}$ runs a fourth workflow that concatenates the alignments into a single data matrix for further analysis. After updating the project folders, $\mathcal{R}$ visualizes the data matrix interactively to check for problems. Satisfied with its quality, $\mathcal{R}$ moves the data matrix from the Markers collection to a new, top-level collection named 'Tree1'. He then selects a fifth workflow for rapidly calculating a neighbor-joining (NJ) tree from the data matrix, updates the tree to the Tree1 collection, and visualizes the tree in an interactive application. Noting that no unusual groupings are evident in the tree, $\mathcal{R}$ stages the input data to a sixth workflow that calculates a maximum-likelihood (ML) tree from the data matrix, where $\mathcal{R}$ again visualizes the tree interactively. To check the tree topologies under different evolutionary models, $\mathcal{R}$ re-runs workflow six specifying a slightly different model of evolution via the workflow parameters. The project history and folders now appear as shown in Figure 2c.

Re-running the analysis with additional bacterial genomes. $\mathcal{R}$ periodically re-runs the above analysis to take advantage of new bacterial genome sequences that have become available. Usually he does not find it necessary to recompute the hidden Markov models or alignments from scratch. Instead, $\mathcal{R}$ simply imports the new genomes and re-runs the alignment workflow specifying (via a parameter value) that the alignments should be updated with the new sequences, and then computes the data matrix, the NJ tree, and ML tree as before, storing the results in a new top-level collection.

Querying provenance for maximum likelihood tree. One year after starting the project $\mathcal{R}$ has computed eight maximum likelihood trees, each based on more bacterial genomes than the previous analyses. During this time, $\mathcal{R}$ has updated the hidden Markov models three times. At this point a collaborator asks for the hidden Markov model for the third marker used in computing the maximum likelihood tree produced by the fourth iteration of the analysis. Because $\mathcal{R}$ has been replacing the hidden Markov models stored in the project folders each time a new HMM is computed, he cannot simply browse the folders to answer this question. Instead, he right-clicks on the maximum likelihood tree in question and selects *Show dependencies* from the context menu, whereupon a data dependency graph appears (*e.g.*, see Figure 5). $\mathcal{R}$ selects the requested HMM from the visual display and exports it for sharing with the collaborator.

3 Collection-Oriented Workflows and Provenance

Here we describe the basic CoMAD data model and an approach for recording and querying provenance in *single workflow runs*. We extend the provenance approach to support multiple workflow runs (via project histories and folders) in the next section. Figure 3 shows a collection-oriented workflow in Kepler for inferring phylogenetic trees. This workflow is similar in intent to the project described in the previous section, *i.e.*, it combines simpler approaches analogous to those used in the workflows described

in Section 2 into a single automated workflow (for the purpose of describing CoMAD for single workflow runs). The Collection Reader actor inputs a nested data collection containing DNA sequences for homologous genes from a number of taxa. The Align Sequences actor performs an initial alignment of the sequences, and the Refine Alignment actor refines this initial alignment. The Infer Trees actor infers a set of phylogenetic trees from the aligned sequences, and the Compute Consensus actor computes the consensus of these trees (see Figure 4). Finally, the Collection Writer actor stores the output of workflow runs, including the provenance information described below.

3.1 Basic CoMAD Data Model

A CoMAD nested data collection forms a node-labeled, ordered tree. Within workflow runs, these trees are flattened into sequences of data, metadata, and collection delimiter tokens (similar to SAX-based parsing[4] of XML documents). Co-actors work concurrently on nested data collections via these token sequences, ***inserting*** new data items and collections into the token stream and ***deleting*** (i.e., not forwarding) existing data items and collections from the token stream. Insertions and deletions always occur within the context of a co-actor's read scope.

The following relations can be used to represent the types of nodes that occur within nested data collections.

- Data($label, id_{node}, id_{obj}$)
- Collection($label, id_{node}, [node]$)
- Metadata($label, id_{node}, id_{obj}$)
- Parameter($label, id_{node}, id_{actor}, id_{obj}$)

Every node in a nested data collection has a label and is assumed to have a unique token identifier (denoted id_{node} above). ***Data nodes*** are used to encapsulate values. The label of a data node is used to represent the type of value encapsulated. We distinguish between primitive data values (*e.g.*, strings and integers) and complex values (*e.g.*, Java objects and external resources such as files). We generally refer to both primitive and complex values as objects. All object values are assumed to have unique identifiers (denoted id_{obj} above), *e.g.*, represented using URIs. A value can either be stored directly within a data-node token (*i.e.*, inlined), or stored externally and resolved on demand via the object identifier. Inlining complex data values can be used to cache values within a workflow run (*e.g.*, reducing the number of derefences made during workflow execution). ***Collection nodes*** contain (possibly empty) ordered sequences of "child" nodes, shown as a node list above. ***Metadata nodes*** can be used to assign annotations to data nodes and collections. Metadata nodes always precede the collections or data items being annotated. When metadata is assigned to a collection, it applies ("cascades") to all collection descendents (*i.e.*, collection data and sub-collections). ***Parameter nodes*** are used to configure actors during workflow execution. The actor identifier and label arguments of a parameter node specify a target actor and corresponding parameter name. The value of a parameter node is used to set the target actor's parameter value prior to invocation. Parameters are embedded within collections and can be overridden by parameters within sub-collections. In this way, parameter nodes provide a convenient mechanism to dynamically change the default configuration of a workflow.

[4] http://www.saxproject.org/

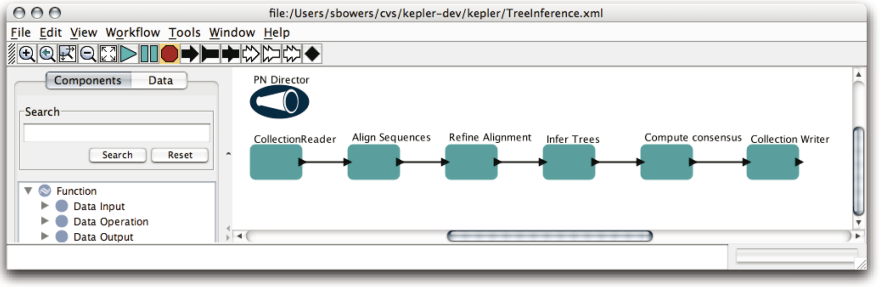

Fig. 3. A collection-oriented workflow for computing phylogenetic trees from DNA sequences

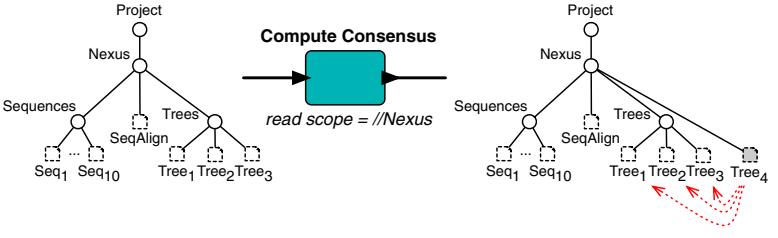

Fig. 4. An example invocation of Compute Consensus from Figure 3

3.2 Representing and Recording Provenance in Workflow Runs

The main goals of the current CoMAD provenance implementation are to (1) enable scientists to ask "scientific" questions about a workflow run by providing convenient queries against the run's execution trace, and (2) have the system track the true data and actor invocation dependencies within a run so that answers to such scientific questions may be as accurate as possible. For example, in most collection-oriented workflows only a portion of an actor's input data is used to produce output data. In addition, workflow input collections are often organized into distinct data sets that enable *independent* "sub-runs" of the workflow. Each sub-run corresponds to one workflow execution over an input data set, and the set of sub-runs are executed concurrently via pipelining. In each of these cases, assuming that all output data from either an actor invocation or a workflow depends on all input data would result in false dependencies [2,3].

The approach described here for single-workflow runs extends our previous work on recording and querying provenance in conventional KEPLER scientific workflows [3]. The approach was successfully applied to the Provenance Challenge [13], as briefly reported in [2]. In the following we describe the basic CoMAD model of provenance, the provenance-related annotations recorded during workflow execution, and examples of using this provenance information for querying CoMAD workflow traces.

Model of provenance. We adopt a simple model of provenance for capturing data dependencies and corresponding source events (such as actor invocations) related to creating and modifying nodes of nested data collections. Instances of the provenance model can be represented using the following relation

– Dependency$(node_w, \{node_r\}, \{event\})$

This relation records the set of nodes ($\{node_r\}$) and events that were used to produce a particular node ($node_w$). Note that $node_w$ is a primary key for the relation, *i.e.*, a given node has at most one associated set of dependency nodes and events. Here we only consider events that correspond to actor invocations, where each $node_r$ was read (*i.e.*, was in the actor's read scope) and each $node_w$ was written (*i.e.*, inserted) by the invocation. Actor invocations are represented via their actor identifiers and invocation number, *e.g.*, where the first invocation of actor a_1 is written a_1:1. Figure 4 shows an invocation of the Compute Consensus actor. The invocation creates a new tree, inserting it under the input Nexus collection. This tree was derived from three of the input trees. For data, metadata, and parameter nodes, a dependency always represents a *one-step derivation* (*i.e.*, via one actor invocation) with respect to a workflow run. Thus, for non-collection nodes, dependencies may be due to at most one actor invocation. For collections, multiple actor invocations may contribute to distinct portions of intermediate "versions" of the collection. Thus, dependency relations for collections record their changes within a workflow run. We can view a set of dependency relations for a workflow run as a directed acyclic graph, where vertices correspond to nested data-collection nodes and edges correspond to dependencies labeled by their invocation events. Figure 5 shows a portion of the dependency graph for a consensus tree output from a run of the workflow in Figure 3. Each vertex in the tree is labeled with its corresponding node type and identifier.

Recording provenance annotations. We infer data dependencies for a workflow run from lower-level ***provenance annotations*** that are directly embedded into the token stream by co-actors. Three different types of annotations are recorded:

- Insertion$(id_{ins}, \{id_r\}, id_{invoc})$
- Deletion(id_{del}, id_r)
- InvocationDependency$(id_{invoc_1}, id_{invoc_2})$

Each of these relations store only node *identifiers* as opposed to entire nodes, as in dependency relations. An ***insertion annotation*** records that a particular node (id_{ins}) was inserted by an actor invocation based on the presence of a given set of nodes ($\{id_r\}$). A ***deletion annotation*** records that a particular node was deleted (*i.e.*, input to, but not forwarded) by an actor invocation. Nodes can be inserted and deleted at most once within a workflow run. An ***invocation-dependency annotation*** records that an actor invocation (id_{invoc_1}) modified a collection (*i.e.*, inserted into or deleted from the collection), and the modified collection was used by another actor invocation (id_{invoc_2}). The set of invocation dependencies induces a partial ordering of actor invocations in a workflow run.

Our approach of including provenance annotations within the data stream contrasts with existing workflow systems (*e.g.*, [16,1,12]) that maintain separate provenance stores. While our approach does not prevent the use of a separate provenance database, it simplifies the overall implementation by not requiring additional communication protocols between the workflow engine and the provenance store (possibly requiring synchronization overhead), and allows provenance and run results to be easily viewed and archived outside of any given system implementation. The result of a workflow run is serialized into a single, self-contained XML ***trace*** file containing all run output and provenance annotations. Figure 6 shows a portion of a trace file generated from a run of the workflow in Figure 3.

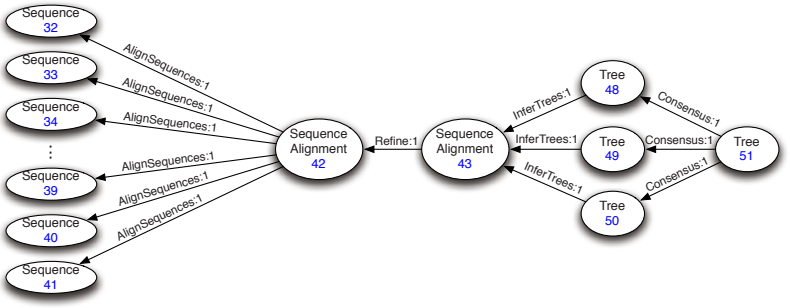

Fig. 5. A portion of a simple dependency graph starting from an output consensus tree

The current implementation of CoMAD requires actors to declare dependencies when new items are inserted into collections during workflow execution. The CoMAD framework validates declared dependencies (*e.g.*, checking that each of the items referred to are within the actor's current read scope), and inserts appropriate provenance annotations into the output token stream of the actor. CoMAD can automatically infer dependencies in certain cases. For example, *composite co-actors* [11] are composed from sub-workflows comprising conventional KEPLER actors, enabling data dependencies to be automatically inferred based on the read scope of the co-actor and the data actually accessed by the contained sub-workflow. The CoMAD framework inserts appropriate deletion annotations for input items not forwarded by an actor invocation. The framework ensures that items having deletion annotations are inaccessible to subsequent downstream actor invocations. Retaining deleted items is essential to inferring complete data dependencies when input or intermediate items are deleted. Invocation dependencies are automatically inferred by the framework based on insertion and deletion dependencies. For example, when a new item is inserted into a collection, an invocation dependency is inferred between the current invocation and each invocation used to create the item's immediate insertion dependencies. These invocation dependencies are then inserted into the token stream.

Inferring dependencies from provenance annotations. Following a workflow run, data dependencies can be *inferred* from a workflow's trace file, *i.e.*, from the provenance annotations generated by the CoMAD system together with the nested data collections output by the workflow run. In general, we aim at minimizing the number and size of provenance annotations that must be recorded to compute data dependencies. For example, when an actor inserts a new collection node, a single insertion annotation is created for the collection that cascades to all descendents of the collection.[5] Similarly, if an inserted node is derived from an entire collection (*i.e.*, the collection structure including all subnodes), an insertion annotation is created that refers just to the collection identifier (and not the various subnodes). Shorthands are also used to specify dependen-

[5] If items are inserted within this collection by subsequent actors, the insertion annotations for these items override the collection insertion annotation.

```
<Trace id="1">
  <Collection type="Project" id="4">
    <Collection type="Nexus" id="19">
      <Collection type="Sequences" id="20">
        <Data type="Sequence" id="32" objectId="1"/>
          ...
        <Data type="Sequence" id="41" objectId="10"/>
      </Collection>
      <Deletion item="42" invocation="RefineAlignment:1"/>
      <Insertion item="42" dep="32 33 34 35 36 37 38 39 40 41" actor="AlignSequence:1"/>
      <Data type="SequenceAlignemnt" id="42" objectId="22"/>
      <Insertion item="43" dep="42" actor="RefineAlignment:1"/>
      <Data type="SequenceAlignment" id="43" objectId="23"/>
      <InvocationDependency from="AlignSequence:1" to="RefineAlignment:1"/>
    </Collection>
    ...
  </Collection>
</Trace>
```

Fig. 6. An example portion of an XML trace file for Figure 3

cies on subsets of collections, *e.g.*, by referring to the collection node identifier and one or more of its subnode identifiers.

Informally, the dependencies of a data node are computed by (1) identifying the insertion annotation for the node; (2) obtaining the set of node identifiers given as dependencies by the insertion annotation; (3) for those dependency nodes that are collections, pruning away nodes inserted after the actor invocation and nodes deleted prior to the actor invocation; and (4) pruning away non-selected subnodes of collection dependencies (if any such subnodes are specified). Collection node dependencies are computed in a similar way, except that collection dependencies may span multiple actor invocations, since invocation dependencies are in general partially ordered.

Finally, the following relations are computed to store the input and output collection structure of a trace.

- input($Trace, [Node]$)
- output($Trace, [Node]$)

These relations are computed directly using insertion and deletion annotations (as opposed to first computing the dependency graph for the run). In this case, the input structure is computed by removing all nodes inserted by the run, and the output structure is computed by removing all nodes deleted by the run. We use these operations in the following section for managing multiple-run provenance via project histories.

3.3 Querying Provenance

The current CoMAD implementation includes a prototype subsystem for querying traces. The prototype is implemented as an SWI-Prolog[6] program, and operates over XML trace files output by workflow runs. The system provides basic operations (*i.e.*, Prolog rules) for accessing trace nodes, constructing dependency relations, and querying corresponding dependency graphs. The operations are defined as views over the underlying CoMAD XML schema (as shown in Figure 6). Dependency graphs are constructed by applying various inference rules. Methods also are provided to reconstruct

[6] http://www.swi-prolog.org/

parameter settings and metadata annotations attributed to data and collection nodes. The main operation in our current implementation computes dependency edges and is defined as follows.

$$\text{dependencyEdges}(Trace, [Node], [Edge])$$

This operation takes an XML trace file and a list of nodes, and returns a list of dependency edges denoting paths that start from each of the given nodes. For example, the following Prolog query selects the set of sequence alignments used to compute the output consensus tree of Figure 5.

$$q(S) :\!- \mathit{traceId}(1,T), \text{nodeForId}(T,51,N), \text{dependencyEdges}(T,[N],E), \text{edge}(E,N_1,S,I), \text{nodeType}(S,\text{`SequenceAlignment'}).$$

In this query, we (1) select the trace with identifier 1 (traceId); (2) get the Tree node having identifier 51 (nodeForId); (3) get the set of dependency edges that start from the Tree node; (4) select an edge; and (5) return the edge node S if it is of type Sequence Alignment (nodeType). As another example, the following query gives the set of actor invocations involved in creating the final consensus tree.

$$q(I) :\!- \text{traceId}(1,T), \text{nodeForId}(T,51,N), \text{dependencyEdges}(T,[N],E), \text{edge}(E,N_1,N_2,I).$$

A number of additional operations for querying traces, along with more complex query examples are given in [2]. As ongoing work, we are developing a specialized query language built upon a minimal set of low-level graph-based operations, as well as KEPLER-based tools for displaying and navigating CoMaD workflow run results.

4 Extensions for Supporting Project Histories and Folders

Here we present extensions to CoMaD for enabling provenance support across scientific workflow runs via project histories and folders. The extensions are designed to address the following challenges in supporting the project-history approach.

- ***Staging input data from project collections.*** Staging involves the selection of relevant data and sub-collections from the project-folders view, and organizing the selected items to conform to the desired collection schema of the target workflow. Selected items must be tracked and appropriately associated as input to the run, given that items may be organized in new ways, *e.g.*, data may be copied into different collections, new collections may be introduced, collection nesting may be inverted, and so on.
- ***Updating project folders from workflow run results.*** Once a user has committed a run to the project history, they have the option of updating the project-folders view using the run result. Updating requires the identification of new items generated by the run and determining where these items should be placed within the project folders (given the restructuring described above). The update process should be semi-automatic, *e.g.*, users should be asked whether to apply deletions and where to put items unrelated to the existing structure of the project folders.
- ***Tracking dependencies between workflow runs.*** In general, workflow runs (like actor invocations) are partially ordered, where all or a subset of data output by one run can be used as input to another run. The system must track these "run dependencies", *i.e.*, the order of runs and their data dependencies, for display within the project run history.

– ***Querying data dependencies spanning multiple workflow runs.*** Latent data dependencies currently exist between workflow runs, where an output of one run may have depended on input that was generated from a previous run, and so on. To expose these dependencies, provenance queries must be extended to leverage run dependencies (*e.g.*, to obtain the set of markers used to generate a phylogenetic tree in the scenario of Section 2).

These tasks are supported by the following additions to the CoMAD provenance framework.

Collection identifiers. As described in the previous section, we require every node in a nested data collection to have a unique identifier. However, as a result of staging data, nodes of project folders may be copied into multiple input sub-collections. For instance, in the marker identification example of Section 2, the contents of each genome collection are copied into each marker sub-collection. Each marker sub-collection will contain nodes that have the same identifiers as nodes in each of the other marker sub-collections. To address these problems, we create new node identifiers for all nodes in a staged nested data collection. Note that with new node identifiers, collections can no longer be tracked back to the project folder. Thus, similar to object identifiers for data values, we add collection identifiers as a mechanism to distinguish, track, and merge collection nodes. These new collection nodes can be represented using the following relation

– Collection$(label, id_{node}, id_{col}, [node])$

where the argument id_{col} is the collection identifier for the node. Unlike object identifiers, collection identifiers are used only to uniquely identify a particular occurrence of a collection, and do not prescribe a collection structure. Thus, two collection nodes that refer to the same collection identifier may have different content.

Run dependencies. To construct the project-history graph (see Figure 2), we explicitly track the order of runs for those runs with data dependencies, *i.e.*, where the output of one run is used as input to another run. The following relation is used to represent run order

– RunDependency$(id_{run_1}, id_{run_2}, type)$

Each run is assigned a unique identifier (denoted id_{run}). Run identifiers can be associated with additional metadata, *e.g.*, with the version of the workflow used in the run, the date and time of the run, and so on. As shown in Figure 2, we distinguish between partial and full run dependencies (where partial dependencies are shown using dashed arrows). A ***partial run dependency*** between a run A and B means that (1) run B occurred after run A; and (2) some of the collection and object identifiers *produced* (*i.e.*, inserted) by A were included in the input to B. In a ***full run dependency***, the input to B is identical to the output of A, *i.e.*, there is a one-to-one correspondence between node identifiers and nesting is preserved. The *type* argument above denotes whether the run dependency is full or partial, which can be easily computed directly from the associated output and input collections (via the input and output operations described in Section 3). There are a number of ways we envision users staging workflows to produce data dependencies

between runs. In particular, a user can select data from project folders, can select some or all of the output data from the project history, or can select from both.

Integrated trace storage. In the current CoMAD implementation, each trace generated from a workflow run is stored in a separate XML document. We extend this approach by allowing each trace result to be stored in a single, integrated "project store" (see Figure 1), providing central database storage and access to project traces. The project store supports both the project-history and project-folders views, as well as queries against one or multiple traces. Individual trace XML documents can be generated from the project store if needed, *e.g.*, to exchange run results between workflow systems or for archival purposes. We add the following relation to represent traces in the project store

– Trace(id_{run}, [*node*])

This relation maps run identifiers to the nodes of the trace. When a user decides to ***commit*** a run (*i.e.*, save the run in their project history), a new trace record is constructed in the project store, the run's XML trace file is used to update the project store, and associated run dependencies are created.

Although related to the problem of schema matching [8], the introduction of unique collection identifiers significantly simplifies the task of updating project folders from a committed run result. Figure 7 shows an example update operation, based on applying the following update rules for modifying project folders from run outputs.

– Only new data and collection nodes inserted by the run are added to project folders. The new items in Figure 7c include D_7, D_8, and D_9. Collections C_6 and C_7 are not considered new since they were input to the run (*i.e.*, introduced during staging).
– New data and collection nodes are added to the collection in the project-folder corresponding to their nearest ancestor collection in the run output. For example, D_7 is added to collection C_4 in Figure 7d.
– Data and collection nodes without a corresponding ancestor collection in the project-folder are added directly under the root collection of the project folders. For example, data items D_8 and D_9 are added directly under C_1 in Figure 7d.
– Data and collection nodes marked as deleted in the run output are removed from project folders. Project-folder nodes that are not marked as deleted, but with parents that are marked as deleted, are nested under their nearest non-deleted ancestor collection in the project folders. For example, D_4 is placed within C_3 in Figure 7d, since C_5 was deleted by the run.

We intend to allow users to *incrementally* accept or reject modifications to project folders resulting from an update operation. In addition, users can freely modify and rearrange project folders and their contents as needed.

Finally, the addition of run dependencies can facilitate provenance queries across multiple runs. One of our requirements is to allow users to specify the workflow runs to use to answer a particular provenance query. That is, without modifying the query expression, the user should be able to easily specify the run or set of runs to query over. We extend the rules used to infer dependency graphs (as described in Section 3.2) to include run dependency information. In this way, an input node of a workflow run may

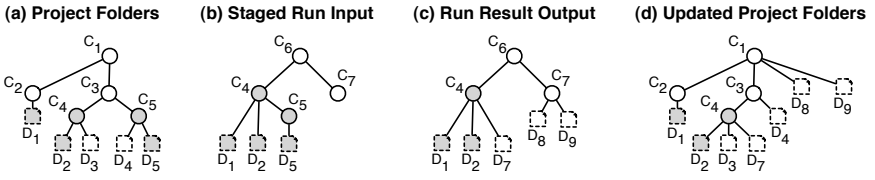

Fig. 7. An example update operation: (a) the initial project folders; (b) the staged run input, where collection and object identifiers taken from project folders are shaded; (c) the result of the run on the staged input; and (d) the resulting updated project folders

depend on an intermediate node created within a different run. For example, assume the set of data dependencies have been computed for a workflow run *B*, and a run dependency exists between *A* and *B* (*i.e.*, the output of *A* was used as input to *B*). For each input node to *B*, we additionally compute the data dependencies for corresponding nodes (*i.e.*, nodes with the same collection or data identifier) in workflow run *A*.

5 Conclusion

This paper introduces the notion of project histories and folders as a natural model for managing provenance information across scientific-workflow runs. The model leverages the file-folder metaphor for organizing project data, provides a simple and intuitive project-history view of workflow runs that emphasizes run dependencies, and leverages our previous work on collection-oriented workflows and provenance. In addition, we propose extending the single-run provenance support in CoMaD with new constructs to support project histories, *e.g.*, for tracking collections and data across runs, and updating project folders with run results. These extensions allow multiple workflow traces to be stored in a single, integrated repository, *e.g.*, to better support provenance queries across runs.

References

1. Barga, R.S., Digiampietri, L.S.: Automatic generation of workflow provenance. In: Moreau, L., Foster, I. (eds.) IPAW 2006. LNCS, vol. 4145, Springer, Heidelberg (2006)
2. Bowers, S., McPhillips, T.M., Ludäscher, B.: Provenance in collection-oriented scientific workflows. Concurrency and Computation: Practice and Experience (To appear 2007)
3. Bowers, S., McPhillips, T.M., Ludäscher, B., Cohen, S., Davidson, S.B.: A model for user-oriented data provenance in pipelined scientific workflows. In: Moreau, L., Foster, I. (eds.) IPAW 2006. LNCS, vol. 4145, Springer, Heidelberg (2006)
4. Callahan, S.P., Freire, J., Santos, E., eidegger, C.E.S., Silva, C.T., Vo, H.T.: Managing the evolution of dataflows with VisTrails. In: IEEE Workshop on Workflow and Data-Flow for Scientific Applications (SciFlow) (2006)
5. Churches, D., Gombas, G., Harrison, A., Maassen, J., Robinson, C., Shields, M., Taylor, I., Wang, I.: Programming scientific and distributed workflow with Triana services. Concurrency and Computation: Practice and Experience, Special Issue on Scientific Workflows (2005)

6. Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Patil, S., Su, M.-H., Vahi, K., Livny, M.: Pegasus: Mapping scientific workflows onto the grid. In: European Across Grids Conference (2004)
7. Eddy, S.R.: Profile hidden markov models. Bioinformatics 14(9), 755–763 (1998)
8. Fuxman, A., Hernández, M.A., Ho, C.T.H., Miller, R.J., Papotti, P., Popa, L.: Nested mappings: Schema mapping reloaded. In: VLDB, pp. 67–78 (2006)
9. Jones, W., Phuwanartnurak, A.J., Gill, R., Bruce, H.: Don't take my folders away!: Organizing personal information to get things done. In: CHI Extended Abstracts (2005)
10. Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger-Frank, E., Jones, M., Lee, E., Tao, J., Zhao, Y.: Scientific workflow management and the Kepler system. Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows (2005)
11. McPhillips, T.M., Bowers, S., Ludäscher, B.: Collection-oriented scientific workflows for integrating and analyzing biological data. In: Leser, U., Naumann, F., Eckman, B. (eds.) DILS 2006. LNCS (LNBI), vol. 4075, pp. 248–263. Springer, Heidelberg (2006)
12. Miles, S., Groth, P., Branco, M., Moreau, L.: The requirements of recording and using provenance in e-science experiments. Journal of Grid Computing (To appear 2006)
13. Moreau, L., Ludäscher, B., et al.: The first provenance challenge (editorial). Concurrency and Computation: Practice and Experience (To appear 2007)
14. Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M., Wipat, A., Li, P.: Taverna: A tool for the composition and enactment of bioinformatics workflows. Bioinformatics Journal, 20(17) (2004)
15. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the condor experience. Concurrency and Computation: Practice and Experience 17(2-4), 323–356 (2005)
16. Zhao, J., Wroe, C., Goble, C., Stevens, R., Quan, D., Greenwood, M.: Using semantic web technologies for representing e-Science provenance. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, Springer, Heidelberg (2004)