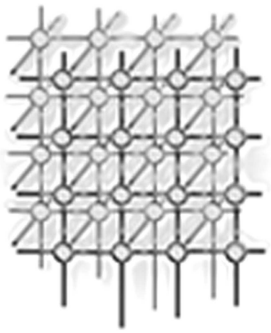
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2000; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Provenance in Collection-Oriented Scientific Workflows†



Shawn Bowers[1,*], Timothy M. McPhillips[1], Bertram Ludäscher[1,2]

[1] *Genome Center, University of California, Davis, USA*
[2] *Dept. of Computer Science, University of California, Davis, USA*

SUMMARY

We describe a provenance model tailored to scientific workflows based on the Collection-Oriented Modeling and Design paradigm. Our implementation within the Kepler scientific workflow system captures the dependencies of data and collection creation events on preexisting data and collections, and embeds these provenance records within the data stream. A provenance query engine operates on self-contained workflow traces representing serializations of the output data stream for particular workflow runs. We demonstrate this approach in our response to the First Provenance Challenge.

KEY WORDS: Provenance; Collection-Oriented Scientific Workflows; Scientific Data Management

1. INTRODUCTION

Much of the complexity of scientific workflows arises from the need to maintain associations between data. As illustrated by the Provenance Challenge, data input to a scientific workflow run generally are related, and steps (i.e., *actors*) in scientific workflows typically produce additional collections of related results. Scientific workflow systems provide little support for managing these data associations explicitly. Maintaining data associations often requires a variety of special-purpose workflow actors for record and object assembly and disassembly (often over multiple levels of nesting), explicit control parameters for data routing, and numerous actor connections (including loops) for managing data and control flow. These additional actors, parameters, and connections often make otherwise straightforward scientific workflows difficult to design, implement, maintain, and understand. The goal of *Collection-Oriented Modeling and Design* (CoMaD) is to address these problems in scientific workflows

*Correspondence to: sbowers@ucdavis.edu
†Work supported in part by NSF grants DBI-0533368, EAR-0225673, IIS 0630033, and IIS 0612326; and DOE grant DE-FC02-01ER25486.

Received 15 November 2006
Revised 08 March 2007

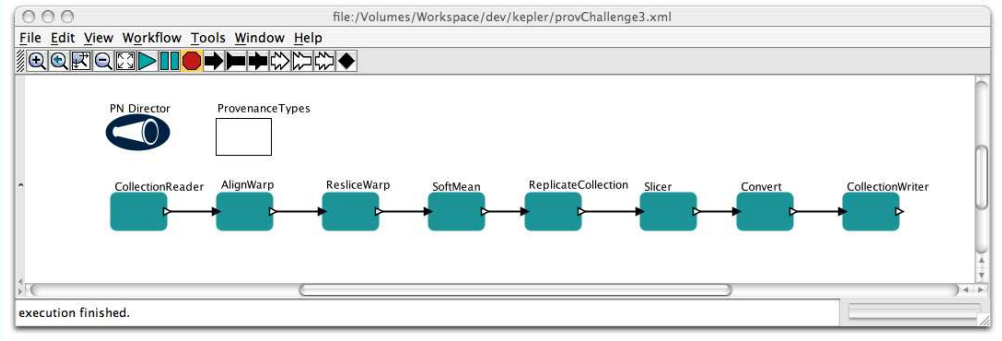


Figure 1. A CoMaD implementation of the Provenance Challenge Workflow

[5, 6]. This paper describes the provenance support available through CoMaD, and our implementation of the Provenance Challenge (e.g., see Figure 1).

CoMaD extends the conventional dataflow-based approach of scientific workflow systems such as Kepler [4] by introducing new modeling constructs and data management capabilities. The main features of CoMaD include:

• ***Nested Data Collections.*** In CoMaD, data is explicitly grouped and related using nested data collections, which are input, manipulated, and output by collection-oriented actors (*co-actors*). Data and collections are both explicitly typed in CoMaD.

• ***Collection Token Streams.*** In a manner similar to SAX-based parsing of XML documents[†], nested data collections are streamed through co-actors as "flat" token sequences in which collections are delimited using paired (opening and closing) control tokens (each token being analogous to a SAX parsing event). CoMaD provides services to co-actors for managing collections, e.g., for constructing collection structures from input token sequences, inserting and deleting collection elements, and (re-)serializing collections to output token sequences.

• ***Actor Scope Parameters.*** Co-actors can explicitly declare the types of collections and data they process via *scope expressions*. The CoMaD framework iteratively invokes co-actors over portions of the input stream matching corresponding scope expressions, thus ensuring co-actors operate only on relevant data and collections. Data and collections that fall outside of an actor's scope are automatically forwarded by the framework to succeeding actors, enabling "assembly-line" style data processing.

• ***Explicit Annotations.*** Annotations (e.g., represented as name-value pairs) are explicit data types in CoMaD. Annotations can represent data and collection metadata that actors may access and create during workflow execution. Annotations can also represent values that automatically override actor parameters, thus, e.g., allowing actor behavior to be changed at

[†]http://www.saxproject.org/

runtime. Like data and collections, annotations are automatically streamed through co-actors by the CoMaD framework.

CoMaD workflows are often simpler and more reusable than conventional workflows (see [6]). For example, collection-oriented workflow definitions are typically independent of the size of input data, i.e., changes to the number of items within a collection do not require changes to the workflow specification. CoMaD workflows are also invariant to common changes to the structure of input data, e.g., increases or decreases in the depth of collection nesting.

Figure 1 depicts a CoMaD implementation of the Challenge workflow within Kepler. The co-actors labeled *AlignWarp*, *ResliceWarp*, *SoftMean*, *Slicer*, and *Convert* correspond to the five stages of the Challenge workflow. The *ReplicateCollection* actor creates additional copies of the products of *SoftMean*—where the number of copies is specified via an actor parameter—so that downstream actors will execute the appropriate number of times, once for each desired slice of the average image. The number of copies and the desired slice computed for each copy can be specified via parameter annotations given in the workflow input. The actors labeled *CollectionReader* and *CollectionWriter* import data into the workflow and save the output of the workflow, respectively. Both input data and output data are serialized in XML using a simple CoMaD schema (e.g., see Figure 4).

Although the workflow definition is linear, it can operate on an arbitrary number of Anatomy Images in stages 1–3, and create an arbitrary number of Altlas Graphic images in stages 4–5, *without* modifying or reconfiguring the workflow definition. The workflow also can handle multiple sub-runs associated with independent sets of input Anatomy Images in a single workflow invocation. In this case, distinct sets of images are nested within separate collections (defined within the input data file that represents the desired input data stream). Each of these collections may consist of differing numbers of input images and distinct annotations and actor parameters. In processing each such sub-run, the CoMaD framework not only maintains associations between graphics images and the anatomy images from which they were derived, but also keeps distinct those results arising from different sub-runs. In contrast to the CoMaD approach, the most straightforward conventional implementation of the Provenance Challenge workflow would consist of four instances of an AlignWarp actor, one per expected input Anatomy Image, as well as three instances each of the Slicer and Convert actors. Running the workflow against an input set of five images or producing additional (or fewer) slices, in the case of a conventional workflow implementation, would require modifying the workflow graph significantly.

The rest of this paper reports our extensions to the CoMaD framework for capturing and querying comprehensive data dependency information. Section 2 describes the CoMaD provenance model and how we employ the CoMaD annotation mechanism to embed provenance information directly within the data flow. Section 3 describes a prototype for querying CoMaD provenance and demonstrates the overall system using the challenge queries.

2. COLLECTION-ORIENTED PROVENANCE

The purpose of the provenance support in CoMaD is to record sufficient information to answer scientifically relevant data-dependency questions [1], e.g., allowing scientists to investigate

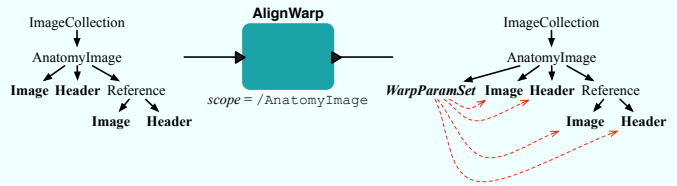


Figure 2. An invocation of the AlignWarp co-actor, with input and output streams shown as trees

and reproduce results from earlier workflow runs, explain unexpected results, and determine the input data and workflow steps that contributed to intermediate and final data products. Here we present our abstract provenance model and discuss how the model is instantiated at workflow runtime within the CoMaD framework.

Provenance in CoMaD is modeled as a set of *element nodes*, corresponding to the data, collection, and parameter items provided to or produced by a workflow run, and a set of *dependency relationships*

$$dependency : N \rightarrow \{N\} \times \{E\}$$

mapping each element node to the set of nodes and events directly involved in its creation. For example, $dependency(n_1) = (\{n_2\}, \{e\})$ asserts that node n_1 was derived from node n_2 by event e. Here, we only consider events corresponding to actor invocations. As an example, the AlignWarp invocation depicted in Figure 2 implies that the WarpParamSet depends on four data nodes (the Anatomy Image, Anatomy Header, Reference Image, and Reference Header) via a single event corresponding to the AlignWarp invocation.

For data and parameter element nodes, a dependency represents a one-step derivation (i.e., via one actor invocation) with respect to a workflow. When the dependency is on a collection, multiple, independent actor invocations may be involved because different invocations may have contributed distinct portions of the version of the collection received by the actor. These dependency relations can be used to reconstruct the "evolution" of collection versions across a workflow run. In general, we can view a set of dependency relations for a workflow run as a (possibly unconnected) directed acyclic graph. Figure 3 shows a portion of such a *dependency graph* depicting the complete derivation of the three Atlas Graphic images computed by a run of the Challenge workflow. Note that in Figure 3, each node is assigned a unique identifier by the workflow system.

Three special types of annotations are recorded during execution of a collection-oriented workflow and used to represent the provenance of results:

- $Insertion(n, N_{dep}, a)$. A node n was derived from the set of nodes N_{dep} by actor invocation a and inserted into the token stream. Note that a node can be inserted at most once.
- $Deletion(n, a)$. A node n was deleted from the token stream by actor invocation a.
- $InvocationDependency(a_1, a_2)$. Actor invocation a_1 used information modified (i.e., inserted into or deleted from the token stream) by actor invocation a_2.

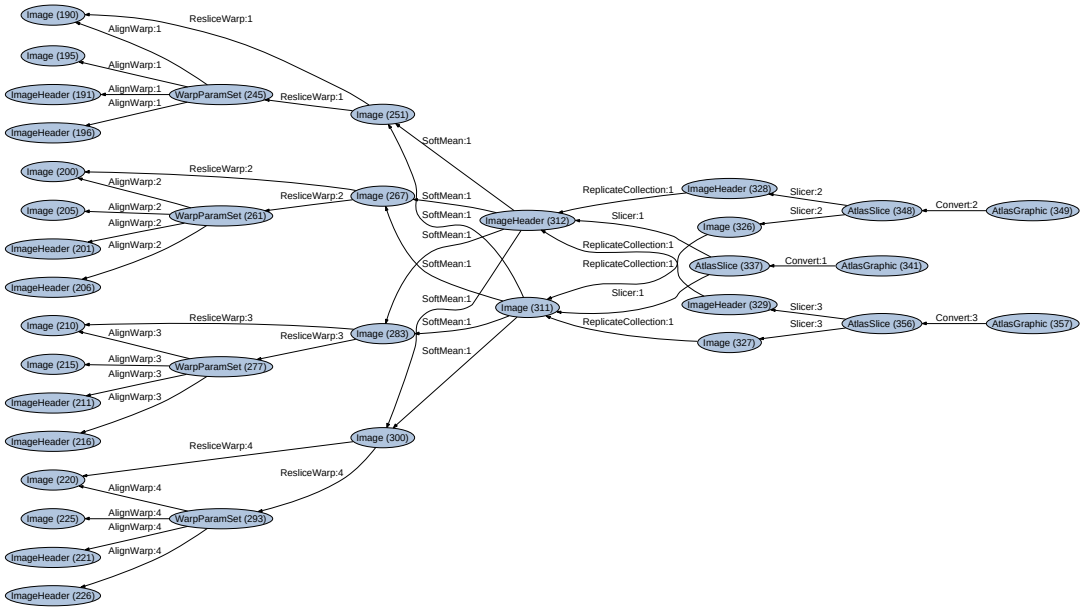


Figure 3. A portion of the dependency graph corresponding to an execution of the Challenge workflow

The goal of our provenance system is to both record these provenance annotations and to use them together with the output of a workflow run to reconstruct data dependencies (the dependency graph).

The CoMaD implementation directly embeds *Insertion*, *Deletion*, and *DependencyInvocation* annotations within the token stream. The result of a workflow run is serialized into a single, self-contained XML "trace" file containing all output data and provenance annotations. Figure 4 shows a portion of the trace file generated for a run of the Challenge workflow. Each data, collection, metadata, and parameter node is assigned an element identifier by the CoMaD framework that is unique with respect to the trace. Insertion, deletion (not shown in Figure 4), and invocation-dependency annotations also are embedded directly in the trace file, and refer to items using the element identifiers.

The provenance extensions to CoMaD require co-actors to declare dependencies when new items are inserted into collections during workflow execution. The CoMaD framework validates these declared dependencies (e.g., by checking that each of the items referred to are within the current scope of an actor), and then inserts appropriate provenance annotations into the output token stream of the actor. CoMaD can automatically infer dependencies in some cases. For example, the use of *composite co-actors* [6], in which new co-actors are composed from sub-workflows comprising conventional Kepler actors, enables data dependencies to be automatically inferred based on the scope of the co-actor and the data actually accessed by

```
...
<Collection type="AnatomyImage" id="186"/>
    <Metadata key="center" type="String" id="189">UChicago</Metatadata>
    <Data type="Image" id="190" objectId="37"/>
    <Data type="ImageHeader" id="191" objectId="38"/>
    <Collection type="ReferenceImage" id="192">
        <Data type="Image" id="195" objectId="39"/>
        <Data type="ImageHeader" id="196" objectId="40"/>
    </Collection>
    <Insertion item="245" dep="190 191 195 196" actor="AlignWarp:1"/>
    <Data type="WarpParamSet" id="245" objectId="53"/>
    <Insertion item="248" dep="190 245" actor="ResliceWarp:1"/>
    <Collection type="ResliceImage" id="248">
        <Data type="Image" id="251" objectId="54"/>
        <Data type="ImageHeader" id="252" objectId="55"/>
    </Collection>
</Collection>
<InvocationDependency from="ResliceWarp:1" to="AlignWarp:1"/>
...
```

Figure 4. An example portion of the XML trace file output by the Challenge workflow

the contained sub-workflow. The CoMaD provenance extension also validates and generates deletion annotations for "dropped" items. Dropped items are not removed from the stream. Rather, the framework ensures that items annotated by deletion records are inaccessible to subsequent downstream actor invocations. Retaining deleted items in this way is essential for inferring complete data dependencies when input or intermediate items are deleted. Similar approaches are used to efficiently manage multiple versions of structured (e.g., XML) documents [2]. Finally, invocation dependencies are automatically inferred by the framework from insertion dependencies and deletion annotations. For example, when a new item is added to a collection, an invocation dependency is generated 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.

3. COLLECTION-ORIENTED PROVENANCE QUERIES

We have developed an initial system prototype for managing and querying CoMaD provenance information. The system is implemented in SWI-Prolog[‡] and operates over the XML trace files output by the CoMaD implementation within Kepler. The system provides basic "built-in" operations (in the form of Prolog predicates) for accessing trace nodes, constructing dependency relations, and querying corresponding dependency graphs. Each operation is defined as a view over the underlying CoMaD XML schema. Dependency graphs are constructed by applying a set of inference rules defined within the system. These rules specify how to infer the dependencies of a data item or collection within a trace based on embedded provenance annotations. The rules also can be used to reconstruct parameter settings and, although not demonstrated here, reconstruct the contents of collections prior to particular

[‡]http://www.swi-prolog.org/

actor invocations via a process analogous to reverting from changes to structured documents [2]. The rest of this section describes how our approach can be used to answer the queries of the Provenance Challenge. We also include additional queries that further demonstrate the utility of the CoMaD provenance approach.

3.1. CHALLENGE QUERIES

To demonstrate the flexibility of the CoMaD implementation with respect to varying numbers of input images, and multiple sets of input images representing independent sub-runs, we supplied two different sets of inputs to the workflow shown in Figure 1. The first corresponds exactly to the Challenge workflow and contains four *AnatomyImage* collections within a single *ImageCollection* (e.g., see Figure 2). The second contains three independent image collections comprising four, three, and two *AnatomyImage* collections, respectively. In particular, the second trace was used to demonstrate that the provenance system infers correct dependencies between independent workflow sub-runs. In both cases, the intermediate and final results of running the workflow are added to the collections defined within the workflow input file, nested within each independent *ImageCollection*. Below, we refer to the result of running the Challenge workflow over these input configurations as the first and second trace.

The first three provenance challenge queries ($Q1$–$Q3$) are answered using the built-in operation

dependencyEdges(*Trace*, *Nodes*, *Edges*),

which takes a CoMaD trace (i.e., an XML tree) and a set of data or collection node identifiers, and returns the dependency-graph edges denoting paths that start from the given nodes. For example, the following (Prolog) query finds edges in the data dependency graph for the first trace that represent "everything that caused the Atlas X Graphic to be as it is" ($Q1$).

q_1(*Edges*) :– traceId('1', Trace), nodeForId(*Trace*, '341', *Node*),
dependencyEdges(*Trace*, [*Node*], *Edges*).

The built-in *traceId* operation takes a trace identifier and returns the root node of the corresponding trace. The built-in *nodeForId* operation takes a trace and node identifier, and returns the corresponding trace node. A similar query can be constructed for the second trace. Because three separate image collections are input to the workflow, the query returns a dependency graph consisting of three Atlas X Graphics instances, each having independent derivations.

Additional filtering operations can be applied to the dependency graph to answer the second and third provenance queries. The following two queries use the built-in operation *filterBeforeActor* to "exclude everything prior to the averaging of images with softmean" ($Q2$), and the built-in *selectAfterActor* operation to "include Stage 3, 4, and 5 details of the process" (Q3). Both queries in this case return the same set of edges.

q_2(*FilteredEdges*) :– traceId('1', *Trace*), nodeForId(*Trace*, '341', *Node*),
dependencyEdges(*Trace*, [*Node*], *Edges*),
filterBeforeActor(*Trace*, *Edges*, 'SoftMean', *FilteredEdges*).

q_3(*FilteredEdges*) :– traceId('1', *Trace*), nodeForId(*Trace*, '341', *Node*),
dependencyEdges(*Trace*, [*Node*], *Edges*),
selectAfterActor(*Trace*, *Edges*, 'ResliceWarp', *FilteredEdges*).

The provenance system provides a number of additional operations for accessing trace information. For example, actor invocation parameters are recovered using the following operation

traceInvocParam(*Trace*, *ParameterName*, *ParameterValue*, *Actor*, *Invoc*),

which returns actor invocations within traces having a given parameter name-value pair. The following query returns "all invocations of AlignWarp using a twelfth order nonlinear 1365 parameter model" ($Q4$).

q_4(*TraceId*, *Invoc*) :– traceId(*TraceId*, *Trace*),
traceInvocParam(*Trace*, 'warpParams', '-m 12', 'AlignWarp', *Invoc*).

The input and output nodes of a workflow run can be obtained from a trace using the built-in operations *traceInputNode* and *traceOutputNode*. For example, the following query finds "all Atlas Graphic images output from workflows where at least one of the input Anatomy Headers had an entry global maximum=4095" ($Q5$).

q_5(*TraceId*, *Graphic*) :– traceId(*TraceId*, *Trace*), traceInputNode(*Trace*, *X*),
nodeType(X, 'AnatomyHeader'), headerQuery(X), traceOutputNode(*Trace*, *Graphic*),
nodeType(*Graphic*, 'AtlasGraphic').

The built-in *nodeType* operation relates a node to its corresponding data or collection type. We assume here that *headerQuery* is a user-supplied predicate that applies the global maximum check on the header data object. The particular phrasing of $Q5$ suggests that all graphics output in a particular workflow run depend on all images and headers input to that run. As a result, if only one input image collection from the second trace were to contain an Anatomy Header with the given entry, this query would return incorrect dependencies. It is possible, however, to rewrite this query in our system to return only valid derivations.

The built-in *actorInvocation* operation computes the input and output of an actor invocation from a dependency graph. For example, the following query finds "all output averaged images of softmean procedures, where SoftMean was preceded in the workflow, directly or indirectly, by an AlignWarp procedure with argument -m 12" ($Q6$).

q_6(*TraceId*, *Image*) :– traceId(*TraceId*, *Trace*), actorInvocation(*Trace*, 'SoftMean', *X*, *Image*),
nodeType(*Image*, 'Image'), dependencyEdges(*Trace*, [*Image*], *Edges*),
member((N_1, N_2, 'AlignWarp', *I*), *Edges*),
traceInvocParam(*Trace*, 'warpParams', '-m 12', 'AlignWarp', *I*).

The built-in *nodeMetadata* operation gives the metadata key-value pairs for trace nodes. For example, the following query finds "outputs of AlignWarp where the inputs are annotation with center=UChicago" ($Q8$).

q_8(*TraceId*, *OutNode*) :– traceId(*TraceId*, *Trace*),
actorInvocation(*Trace*, 'AlignWarp', *Invoc*, *InNode*, *OutNode*), nodeType(*InNode*, 'Image'),
nodeMetadata(*Trace*, 'center', 'UChicago', *InNode*).

The following query finds "all graphical atlas *sets* that have metadata annotation studyModality with values speech, visual, or audio" ($Q9$).[§]

[§]Note that $Q9$ asks for the annotations for the returned graphics, which can be performed as an additional step using the built-in *nodeMetadata* operation.

q_9(*TraceId*, *GraphicSet*) :– traceId(*TraceId*, *Trace*), traceInvocation(*Trace*, 'SoftMean', *Invoc*), graphAtlasSet(*Trace*, *Invoc*, *GraphicSet*).

graphicAtlasSet(*Trace*, *Invoc*, *GraphicSet*) :– setOf(G, *graphicAtlas*(*Trace*, *Invoc*, *G*), *GraphicSet*), member(*Graphic*, *GraphicSet*), nodeMetadata(*Trace*, 'studyModality', *Modality*, *Graphic*), member(*Modality*, ['speech', 'visual', 'audio']).

graphicAtlas(*Trace*, *Invoc*, *AtlastGraphic*) :– traceOutputNode(*Trace*, *AtlasGraphic*), nodeType(*AtlasGraphic*, 'AtlasGraphic'), dependencyEdges(*Trace*, [*AtlasGraphic*], *Edges*), member((N_1, N_2, 'SoftMean', *Invoc*), *Edges*).

We assume that a Graphic Atlas "set" consists of all Atlas Graphics derived from an invocation of SoftMean (i.e., the Atlas X, Y, and Z Graphics generated from SoftMean correspond to a single set). The first trace results in one graphics set, while the second trace results in three graphics sets. The complexity of this query is due to the generation of these sets, which is performed using a *group-by* operation followed by filtering groups according to their metadata annotations.

3.2. ADDITIONAL QUERIES

The focus of the CoMaD approach on maintaining relationships between input, output, and intermediate data (via collections) facilitates recording the true data dependencies at each stage of workflow execution even when multiple, independent data sets are provided to a single workflow run. The queries below further demonstrate the utility of recording such explicit data dependencies.

Find all intermediate (not input or output) Images used to derive an Atlas X Graphic. The following query (1) obtains an Atlas X Graphic from the second trace, (2) obtains the dependency edges starting from the Graphic, (3) selects an image used to derive the Graphic, and (3) checks that the image was not an input to the workflow.

q_{10}(*Image*) :– traceId('2', *Trace*), nodeForId(*Trace*, '1093', *Graphic*), dependencyEdges(*Trace*, [*Graphic*], *Edges*), edgeNode(Edges, Image), nodeType(*Image*, 'Image'), ¬traceInputNode(*Trace*, *Image*), ¬traceOutputNode(*Trace*, *Image*).

Here, the built-in *edgeNode* operation gives the nodes used in the given set of edges. A variant of this query is to find the "closest" Image on the derivation path from the given output.

Find all input Images used to derive the Atlas X Graphic. This query is of particular importance for the second trace, where not all input images were used to derive each output graphic. The query (1) obtains an Atlas X Graphic from the second trace, (2) obtains the dependency edges starting from the Graphic, (3) selects an Image used to derive the Graphic, and (4) checks that the Image was an input to the workflow.

q_{11}(*Image*) :– traceId('2', *Trace*), nodeForId(*Trace*, '1093', *Graphic*), dependencyEdges(*Trace*, [*Graphic*], *Edges*), edgeNode(Edges, Image), nodeType(*Image*, 'Image'), traceInputNode(*Trace*, *Image*).

4. COMPARISON TO OTHER APPROACHES

The CoMaD framework is an extension of the Kepler Scientific Workflow System [4] (see comparison matrix in [9]), which in turn is based on Ptolemy II [10]. The RWS [3] and SDG [12] Challenge approaches also extend Kepler. Provenance support for CoMaD currently is divided between (i) the extension to Kepler which performs provenance recording and generates output XML trace files (e.g., [11, 7] also use XML for provenance serialization), and (ii) a stand-alone query and inference system implemented in Prolog (we plan to develop and integrate CoMaD provenance query tools within Kepler). CoMaD provenance queries are based on primitive operations that work over dependency graphs. Although Kepler and CoMaD allow external components to be wrapped as actors [6], the co-actors used in the Challenge Workflow (see Figure 1) do not currently invoke underlying image processing tools.

Our approach is distinguished by the benefits of employing the CoMaD paradigm: (1) The approach exploits the flexibility inherent in CoMaD to model the workflow succinctly (i.e., linearly) while capturing provenance information accurately from runs involving collections of data of varying sizes and nesting. (2) It minimizes the provenance information that must be recorded for a workflow run by allowing provenance annotations on collections to cascade to child elements. (3) It simplifies association of workflow runs with data provenance by storing workflow inputs, outputs, intermediate data products, and derivation dependencies in a single, self-contained trace file. (4) It decouples provenance representation from a particular workflow technology by representing the trace file using a system-independent XML schema. Similarly, the query system does not require information about the workflow description (similar to, e.g., [3]). (5) Finally, our provenance system can reconstruct collection "histories" (i.e., the contents of collections at specific workflow stages) from underlying dependency relations, providing additional support for managing data collections in CoMaD.

REFERENCES

1. S. Bowers, T. M. McPhillips, and B. Ludäscher. A model for user-oriented data provenance in pipelined scientific workflows. In *IPAW*, 2006.
2. S.-Y. Chien, V. J. Tsotras, and C. Zaniolo. Efficient management of multiversion documents by object referencing. In *VLDB*, 2001.
3. B. Ludäscher, N. Podhorszki, I. Altintas, S. Bowers, and T. M. McPhillips. Models of computation and provenance, and the RWS approach. In Moreau and Ludäscher [8].
4. B. Ludäscher *et al.* Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience, Special Issue: Workflow in Grid Systems*, 18(10):1039–1065, 2006.
5. T. M. McPhillips and S. Bowers. An approach for pipelining nested collections in scientific workflows. *SIGMOD Record*, 34(3):12–17, 2005.
6. T. M. McPhillips, S. Bowers, and B. Ludäscher. Collection-oriented scientific workflows for integrating and analyzing biological data. In *DILS*, pages 248–263, 2006.
7. S. Miles, P. Groth, S. Munroe, S. Jiang, T. Assandri, and L. Moreau. Extracting Causal Graphs from an Open Provenance Data Model. In Moreau and Ludäscher [8].
8. L. Moreau and B. Ludäscher, editors. *Concurrency and Computation: Practice and Experience – Special Issue on the First Provenance Challenge*. Wiley, 2007.
9. L. Moreau *et al.* The First Provenance Challenge. In Moreau and Ludäscher [8].
10. Ptolemy II project and system. Department of EECS, UC Berkeley, 2006.
11. C. Scheidegger, D. Koop, E. Santos, H. Vo, S. Callahan, J. Freire, and C. Silva. Tackling the provenance challenge one layer at a time. In Moreau and Ludäscher [8].
12. K. Schuchardt, T. Gibson, E. Stephan, and G. Chin, Jr. Applying content management to automated provenance capture. In Moreau and Ludäscher [8].