

# Data lineage model for Taverna workflows with lightweight annotation requirements

Paolo Missier<sup>1</sup>, Khalid Belhajjame<sup>1</sup>, Jun Zhao<sup>2</sup>, and Carole Goble<sup>1</sup>

<sup>1</sup> School of Computer Science, University of Manchester, UK  
{pmissier,khalidb,carole}@cs.man.ac.uk

<sup>2</sup> Department of Zoology, University of Oxford, UK  
jun.zhao@zoo.ox.ac.uk

**Abstract.** The provenance, or *lineage*, of a workflow data product can be reconstructed by keeping a complete trace of one workflow execution. This lineage information, however, is likely to be both imprecise, because of the black-box nature of the services that compose the workflow, and noisy, because of the many trivial data transformations that obscure the intended purpose of the workflow. In this paper we argue that these shortcomings can be alleviated by introducing a small set of optional lightweight annotations to the workflow, in a principled way. We begin by presenting a baseline, annotation-free lineage model for the Taverna workflow system, and then show how the proposed annotations improve the results of fundamental lineage queries.

## 1 Introduction

Workflow technology is being increasingly adopted in e-science as a way to model and automate the enactment of scientific experiments, and more generally, to specify complex sequences of distributed data manipulation operations (retrieval, transformation and analysis) in a flexible and declarative way. The workflow shown in Fig. 1, for example, is designed to look for a list of diseases in response to a single input query consisting of clinical terms, for instance “Alzheimers disease +protein”.<sup>3</sup> The output list is obtained by (i) retrieving relevant abstracts based on the query (`Lucene-query`), (ii) extracting protein names from the abstracts by means of a dedicated named entity recognizer (`NERrecognize`), and (iii) linking the proteins to disease names through the OMIM<sup>4</sup> disease database (`extract_diseases-from-OMIM`).

The final workflow output is certainly the most important product of this modelling and execution effort. If we intend to use it as a piece of scientific evidence upon which more results will be built, however, we need to provide some proof of its soundness, i.e., by showing that such a sophisticated chain of

---

<sup>3</sup> The example is due to Marco Roos at the University of Amsterdam. Full details are available through the myExperiment workflow repository and sharing facility: <http://www.myexperiment.org>.

<sup>4</sup> OMIM: <http://www.ncbi.nlm.nih.gov/sites/entrez?db=omim>

data transformations and manipulations does indeed produce the intended result. This is not just a matter of debugging, but rather, of supporting the reliability of final results. For instance, it is crucial for the experimenter to understand how proteins names were identified in the NERecognize step, if these automatically produced results are to be trusted. This suggests the need to use intermediate workflow data products as a way to explain the final result and to support any claim of reliability on it, for the benefit of both the experimenters and their community at large.

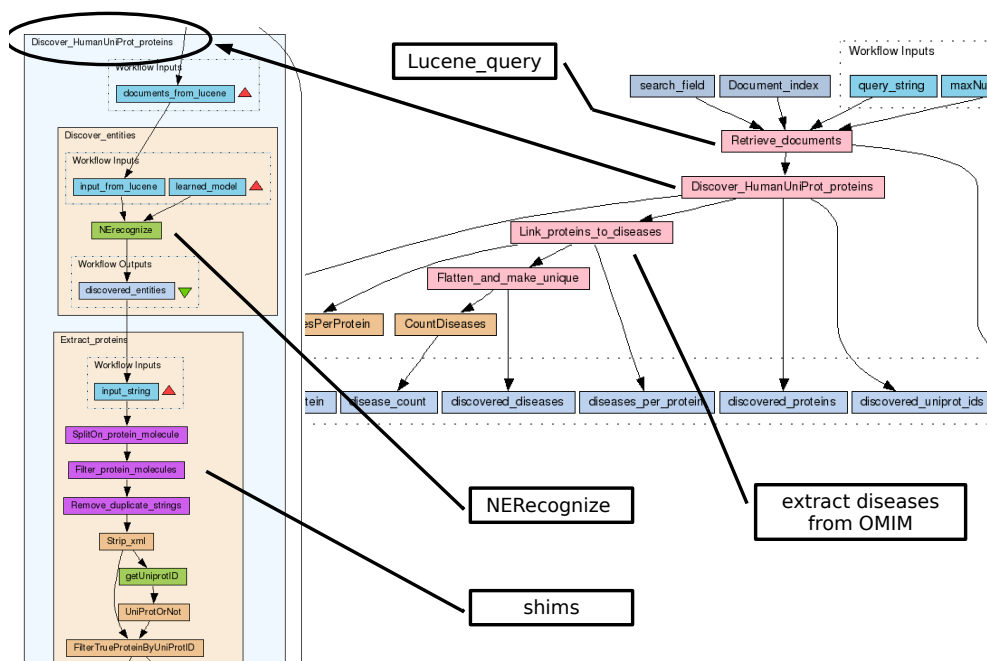


Fig. 1: Example Taverna dataflow, folded view (right) and unfolded selected sub-workflow (left)

This is only one of the reasons for collecting and analysing data provenance, broadly defined as “information that helps determine the derivation history of a data product, starting from its original sources” [16]. In this paper we focus specifically on *data lineage* obtained from one or more executions of a dataflow through multiple processors, i.e., the graph of data dependencies that account for an output value produced during the course of a dataflow execution. By *dataflow* we mean a workflow consisting only of data links, i.e., with no explicit control links between the nodes.

Issues of data lineage have been studied extensively in the context of data management in databases, originally with respect to the derivation of data elements as a result of relational operations [5], and, more recently, with the goal of helping resolve uncertainties in data, i.e., in the Trio project [3]. Despite this body of research, two main issues make the problem of capturing and presenting lineage information in the workflow context a challenging one. Firstly, a common assumption that underpins the work just mentioned is that the available data manipulation operations are limited to a well-founded set, i.e., a collection of relational algebra operators or data replication primitives. In contrast, workflows invariably include the invocation of services described only in terms of their access interfaces. It has been observed [19, 4] that the black-box nature of these services limits the specificity of the lineage information that can be captured by observing a workflow execution.

Secondly, a workflow is a detailed specification of a process that could be described, in abstract, as a set of interdependent data transformations, such as those listed at the beginning. Nevertheless, the model of the data that the process operates upon is *remains latent* and is never made explicit as part of the process specification. Thus, a conceptual model designed from the top down to represent the data managed by our example workflow, would probably include a handful of entities, such as “clinical term”, “article abstract”, “disease term”, “protein name”, along with logical associations amongst them. While one execution of the actual workflow does generate values that can potentially be used to populate the data model, doing so automatically is difficult, because the interesting values are part of a much larger collection of relatively irrelevant data products, that exist solely to enable the integration among the main data transformation steps.

While the adapters that produce these values should ideally disappear, along with their products, from a user-oriented view of the overall process, this requires an explicit abstraction mechanism. As an example, Figure 1 describes one possible mechanism for abstraction, available in Taverna [15, 11], Kepler [17], and other workflow systems, namely the nesting of workflows structures. The workflow shown in the right part of the figure actually consists of a number of sub-workflows, each rendered here as an atomic processor, while the left part shows the unfolding of one of those sub-workflows. At this finer level of detail we can see that only few of the processors, for instance `NERrecognize`, actually perform interesting data transformations, while the remaining processors, known as *shims* [10], are adapters that are required to perform mundane tasks. Note however that nesting is entirely optional and is perceived by many users more as a mechanism for reuse, rather than for abstraction. The lineage model described in this paper does not rely, and indeed does not benefit from, structural nesting, although this type of abstraction, central for example to the *Zoom* approach [4] mentioned below, is being considered as part of ongoing work.

Based on these observations, we argue that a desirable goal for a data lineage management system is to provide a variable level of specificity and abstraction, based on the specification of both workflow designers and of consumers of the lineage information. The question is then, what is a reasonable trade-off between

the effort required of users to create a complete specification, and the benefit in terms of precision of lineage information. We could, in an extreme case, transform all the black-boxes into “white-box” services by adding extensive annotations to describe their semantics. This would probably impose an unacceptable additional burden to the workflow designers, however. On the other hand, data lineage that is based exclusively on the workflow structure, i.e., its graph topology and the interface-level information about the services, is complete but it may contain too much irrelevant information that obscures the intended purpose of the workflow.

This paper explores the middle ground between these two extremes, focusing on a small set of lightweight annotations that add value to basic lineage information while requiring with little additional human effort. The analysis presented here underpins the current data lineage model for the Taverna workflow system, which will be used as a reference model throughout the paper. Specifically, our goal is twofold. Firstly, we define a simple, baseline model for annotation-free data lineage, and show that it is sufficient to answer lineage queries. Secondly, we introduce a small set of annotation types, in addition to user-defined constraints on the queries, and show their added value in terms of increased specificity and focus of the resulting lineage query results. The specific goals of the annotations are as follows:

- increase specificity, by explicitly declaring dependencies of output variables from input variables for each processor, including the fine-grained transformation of list-valued variables;
- increase focus, by letting users specify data lineage queries that select only relevant aspects of the workflow, for instance the few important processors alluded to earlier;
- enable space/time trade-offs when storing and querying lineage data. As pointed out recently [8], the size of provenance may easily outgrow the size of the data being computed by a workflow. We note that, if we instead knew that some of the workflow processors are stateless, then we would have the option to compute their transformation at lineage query time, as needed, rather than recording it explicitly. This is beneficial when the workflow includes many simple shims that add little computational cost to the query.

The lineage model described in this paper, including support for the proposed lightweight annotations, is currently being implemented as part of the Taverna provenance architecture.

## 2 Baseline model for capturing and querying data lineage

In this section we lay the foundation for the lineage model, assuming that no information besides the workflow structure is available to collect and present data lineage. For this purpose we characterise a Taverna dataflow as a DAG where the nodes denote processors<sup>5</sup>. Throughout our discussion we are going to

---

<sup>5</sup> A full account of the formal syntax and structural semantics for the Taverna language can be found in [20].

use the generic workflow pattern of Figure 2, which, in particular, captures the topology of the real-life workflow presented earlier.

Each processor may have multiple inputs and outputs, each denoted by a distinct variable name. We write

$$\langle P, [X_1 : \tau_1 \dots X_n : \tau_n], [Y_1 : \sigma_1 \dots Y_m : \sigma_m] \rangle \quad (1)$$

to denote a node in the graph, representing a processor  $P$  with  $n$  input variables  $X_1 \dots X_n$  and  $m$  output variables  $Y_1 \dots Y_m$ . Variables have a type, denoted here by  $\tau_i$  and  $\sigma_j$ , which is either a simple type (**string**, **boolean**, etc.), or is a list of values, denoted  $l(\tau)$ . Lists can be nested, i.e.,  $\tau$  is either a simple type or itself a list. Nodes in the dataflow graph are connected through directed data links  $\langle P_1, X, P_2, Y \rangle$  that transfer a value bound to output  $X$  from an upstream processor  $P_1$  to the value bound to input  $Y$  of a downstream processor  $P_2$ . Note that this simple type system does not prevent the use of bulk or multimedia types, as strings can be used to hold references, typically URIs, to external objects.

The data lineage information captured during dataflow execution reflects the available knowledge regarding the dependencies of output variables from input variables, for each node of the form (1). Unless processors are annotated with specific dependency information, as discussed later in Section 3, we must assume that every output depends on every input. We write this as a set of functional dependencies, as follows:

$$X_1 \dots X_n \rightarrow Y_1, \quad X_1 \dots X_n \rightarrow Y_2, \dots$$

When recording lineage information, we consider an instantiation of the dataflow graph consisting of:

- a binding of each variable  $X$  to a value  $x$ , denoted  $X : \tau/x$ , and
- a binding of each processor  $P$  to a process instance  $p$ , denoted  $P/p$ .

The *data lineage graph* captured during dataflow execution consists of three relations. The first, *xform*, describes data transformations through a processor:

$$xform([X_1 : \tau_1/x_1 \dots X_n : \tau_n/x_n], Y_j : \sigma_j/y_j, P/p) \quad (2)$$

A second relation *xfer* captures the transfer of value  $x$  of output  $X$  to input  $Y$  through a data link:

$$xfer(X : \tau/x, Y : \tau/x) \quad (3)$$

Note that  $X$  can be a list-typed variable; in this case,  $X/x$  denotes the binding of the *entire list*  $x$  to  $X$ . Since we have no specific information that links individual elements of a list to one another, those elements are indistinguishable. We do, however, provision for the explicit reference to list elements as part of our model, using a third relation:

$$member(x_i, x, i) \quad (4)$$

to indicate that value  $x_i$  appears at position  $i$  within list  $x$ . This can be used whenever there are good reasons to refer to individual members of a list, as

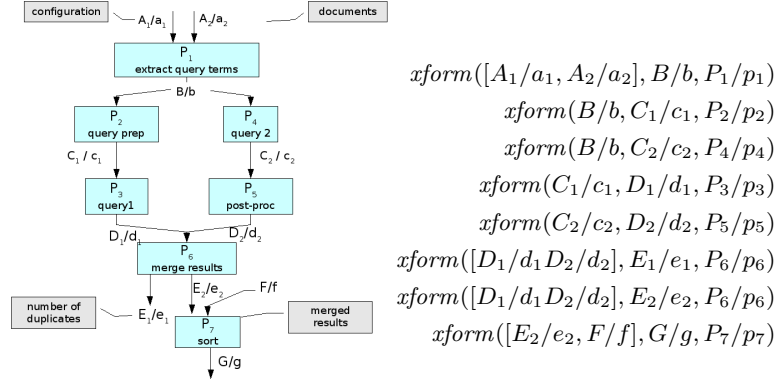


Fig. 2: Example dataflow with execution annotations, and corresponding lineage graph specification

described later (Section 2.1). In addition, we will use relation  $isInput(X/x)$  and  $isOutput(X/x)$  to denote the fact that  $X/x$  is an input (resp., output) to the entire workflow.

With this notation, the right side of Figure 2 shows the lineage graph for one sample execution of the dataflow on the left. Note that, without loss of generality, we have left the  $xfer$  relation implicit, assuming for simplicity that the variable names on corresponding outputs and inputs on a data link are the same (e.g. the output  $B$  of  $P_1$  and the corresponding input into  $P_2$ ). With this assumption, all  $xfer$  tuples are of the trivial form  $xfer(X/x, X/x)$ .

## 2.1 Explicit and implicit collections

As mentioned, each of the values in the example above may be a list. In fact, Taverna processors that manage lists can be described by the following patterns, where some of the variables have an explicit list type  $l(\tau)$ :

$$\langle P, [X : \tau], [Y : l(\sigma)] \rangle \quad (5)$$

$$\langle P, [X : l(\tau)], [Y : \sigma] \rangle \quad (6)$$

$$\langle P, [X : l(\tau)], [Y : l(\sigma)] \rangle \quad (7)$$

These patterns reflect paradigmatic transformations: (5) is representative of a search service, where  $X$  is a search string and  $Y$  the result collection; (6) captures, among other things, aggregation functions, while (7) is appropriate for a filter (i.e., a selection of elements) or a sort operation on a list.

Characteristically, however, Taverna also allows for variables with simple type to be bound to a list. For instance,  $X : \text{string}$  can be assigned a list of strings,  $x = [x_1 \dots x_k]$ . Taverna manages this type cardinality mismatch by adding an

implicit iterator on  $x$ , so that  $P$  is executed separately on each value  $x_i$ . Correspondingly, the output values  $y_i$  are collected into a list, which is then assigned to an output  $Y$  (also originally of type `string`). This case, where each element  $y_i$  depends only on the corresponding input  $x_i$ , is captured by the following tuples:

$$xform(X/x_i, Y/y_i, P/p_i), \text{ member}(x_i, x, i), \text{ member}(y_i, y, i), \quad i : 1 \dots k$$

Thus, in Taverna this is equivalent to having  $n$  instances  $p_1 \dots p_n$  of a processor  $P$ , each responsible for one element  $x_i$  of  $x$ . This is a case where we can provide a more granular lineage data than would otherwise be possible in general.

## 2.2 Data lineage queries

The lineage graph collected during one execution supports a variety of queries, including some of those proposed as part of the First Provenance Challenge<sup>6</sup>. While a complete account of the query formulation is beyond the scope of this paper, it should be clear that useful queries involve traversing the lineage graph, a task that can be accomplished in a variety of ways. Consider for example the basic lineage query: “find all derivation paths for an output value (or any intermediate value), back to the input values that contribute to it during a specific execution”. Its answer consists of the tree of paths, rooted at  $G/g$ , shown in Figure 2.2 (right). The graph-traversal algorithm that computes the tree is presented as a Prolog program on the left in the same figure<sup>7</sup>. Informally, the program computes a *derivation tree* for an input bound variable, say  $Y/y$ . The root of the tree is labelled  $Y/y$ . If  $Y/y$  is derived through a transformation of the form (2), i.e.:

$$xform([X_1/x_1 \dots X_n/x_n], Y/y, P)$$

then node  $Y/y$  has  $n$  sub-trees, each rooted at  $X_i/x_i$ , expressing the fact that  $Y/y$  is derived from *all* of the  $X_i/x_i$ . Each such sub-tree is computed recursively using other *xform* tuples in the lineage graph, until we reach the input variables (i.e., the  $X/x$  such that tuple *isInput*( $X/x$ ) exists). In practice, a derivation tree is an unfolding of a particular traversal strategy on a lineage graph, in this case a bottom-up visit (remember that the lineage graph is a DAG, just as the original workflow graph). The derivation tree  $DT$  for our example workflow corresponds to the Prolog goal: `dt(G/g,DT)`.

In a similar fashion we can support a number of additional queries; for instance, by traversing the graph in a forward fashion we can compute the set of all values that depend on a given set of inputs. Perhaps more interestingly, in the next section we consider adding constraints to these basic queries, namely to (i) focus on selected paths in the graph, and (ii) focus on selected transformations within a path.

<sup>6</sup> <http://twiki.gridprovenance.org/bin/view/Challenge/FirstProvenanceChallenge>.

<sup>7</sup> Here we use Prolog for conciseness; however, this does not reflect the actual implementation for this and additional lineage queries supported by the lineage graph.

```

// compute a derivation tree
dt(V, DT) :- isInput(V), !,
             DT = derive(V, [], in).
dt(V, DT) :- xform(Vset, V, P),
             dt1(Vset, DTlist),
             DT = derive(V, DTlist, P).

dt1([], []).
dt1([V1 | Vrest], [DT | DTrest]) :-
    dt(V1, DT1),
    dt1(Vrest, DTrest).

```

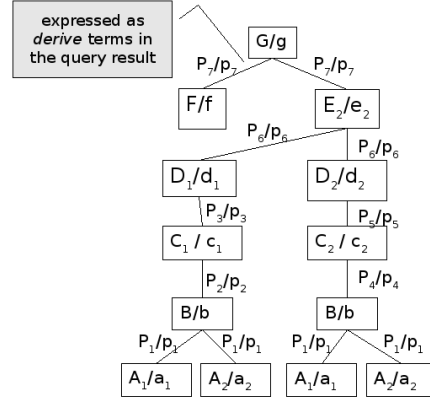


Fig. 3: A basic derivation tree computation in Prolog, and its output on a specific goal

### 3 Lightweight annotations for improving lineage data

The derivation graph described at the end of the last section exhibits some of the problems that we had stated informally at the beginning, namely:

- when services are black boxes, then we have to assume that all outputs depend on all inputs, for instance  $B/b$  depends on both  $A_1/a_1$  and  $A_2/a_2$ ; furthermore, each element in each output data collection depends on each element in all of the input collections;
- lineage derivation trees include shim services, i.e.,  $P_2$  and  $P_5$ , that add little to the understanding of the actual, latent data model that is implicit in the dataflow. In addition, the lineage data for all the shim transformations must be stored explicitly and dealt with in the same way as more critical workflow steps, although these processors usually perform mundane tasks. This additional space consumption does not translate into useful information to users.

To address these problems in a principled way, we propose a simple classification of annotation types that serve different purposes, namely *precision*, *focus*, and *optimisation* and are provided at different stages during experimentation, i.e., *workflow design*, *workflow execution*, and *lineage query*.

**Precision:** these annotations aim at improving the granularity and understandability of lineage derivation trees. We consider workflow design time annotations that:

1. Make a distinction among input variables according to their role during processing, i.e., between data that is used as part of the processor’s computation, for instance a search string, and configuration parameters, e.g. the number of results returned by the search.

2. Refine the functional dependencies between inputs and outputs for individual processors. With reference to (2) on page 5, if the designer knew that, say,  $Y_1$  only depended on  $X_1 \dots X_k$ , with  $k < n$ , then the first dependency would become  $X_1 \dots X_k \rightarrow Y_1$ , resulting in more specific *xform* tuples, i.e.,  $xform([X_1/x_1 \dots X_k/x_k], Y_1/y_1, P/p)$ .
3. Assert a 1-1 mapping between elements of an input list and corresponding elements of an output list. When this additional information is available, as in the case of cardinality mismatch described earlier, lineage can be tracked at the level of individuals within a collection.
4. Explain the nature of aggregation functions. This amounts to stating, for example, that  $E_1/e_1$  is the result of applying a function `dupCount` to the input lists  $D_1/d_1$  and  $D_2/d_2$ . Note that this is a special case of a more general semantic annotation for processors, an interesting topic of current research.

We also consider additional information that may become available during workflow execution, and that is contributed either by the workflow enactor, or by the services themselves. This includes, for example:

- the information that implicit iterators have been applied to input collections to resolve some cardinality mismatch, and
- an explicit *permutation map* provided by a processor that performs a sort operation. Such a map allows the lineage service to refine the derivation graph by applying the inverse mapping to individual elements in the input/output lists.

**Focus:** these annotations provide users with a means to select relevant lineage information at lineage query time, namely by (i) suppressing some of the paths in the graph, for example those involving  $D_1$  but not  $D_2$ , and (ii) specifying a subset of the processors of interest. In the example, it would be natural to focus on the query processors  $P_3$  and  $P_4$ , while ignoring  $P_2$  and  $P_5$ , for instance<sup>8</sup>.

**Optimisation:** these annotations, specified at workflow design time, indicate that some of the processors are *stateless*, i.e., they are guaranteed to produce the same result when executed multiple times on the same input (unlike, for example, a query to a database that may change in time). When this is the case, the lineage service has a choice between materialising the lineage tuples corresponding to those processors’ transformations, or re-executing the services themselves when the lineage tree is computed (under the realistic assumption that its implementation is available to the lineage service at query time). This is potentially beneficial for a number of small shim services that are computationally inexpensive.

Table 1 presents a summary of these annotation types (the specific annotation syntax is not relevant for the purposes of this paper). Although the framework in the table is fairly general and applicable to a variety of annotation options

---

<sup>8</sup> Taverna does include a basic feature that can be used as starting point, namely for tagging processors as “boring” so that they are excluded from the visual rendering.

Annotation type	Phase	Effect
<b>Precision:</b>		
refinement of functional dependencies between inputs and outputs	design	$xform([X_1/x_1 \dots X_k/x_k], Y/y, P)$ replaced by: $xform([X_1/x_1 \dots X_k/x_k], Y/y, P)$ , $k < n$
parameter vs. data input distinction	design	$xform([A_1/a_1 A_2/a_2], B/b, P_1)$ replaced by: $xform(A_2/a_2, B/b, P_1)$ $A_1/a_1$ reported as separate context information instead
1-1 mapping on lists	design	$xform(B/b, C_2/c_2, P_2)$ replaced by: $xform(B/b_i, C_2/c_{2i}, P_2)$ for each $i$
type of aggregation functions	design	$xform([D_1/d_1 D_2/d_2], E_1/e_1, P_6)$ reported along with <code>dupCount</code> during query answering
implicit iteration over non-collection variables	execution	equivalent to 1-1 mapping, only implicit
explicit permutation maps for list sorting processors	execution	a permutation map containing: $\Pi(E/e_{2i}) = G/g_j$ justifies the derivation: <code>derive(G/g_j, [E/e_{2i}], -)</code>
<b>Focusing:</b>		
path suppression	lineage query	disregard some of the lineage paths, e.g. $P_6 \rightarrow P_5 \rightarrow P_4 \rightarrow P_1$ not considered
processor selection	lineage query	only report on derivation through, say, $P_3$ and $P_4$ .
<b>Optimisation:</b>		
stateful vs. stateless processors	design	If $P_1, P_2, P_5, P_6, P_7$ are stateless, then the only required materialisation of lineage is now: $xform(B/b, C_2/c_2, P_4)$ $xform(C_1/c_1, D_1/d_1, P_3)$

Table 1: Summary of dataflow annotations types and their effect on lineage

and workflow systems, we focus here on a few cases that are of direct interest to Taverna workflows. The last column of the table provides examples of the effect of each of these annotations.

Let us now consider their effect on our example workflow graph, specifically:

- (1)  $A_1, F$  are configuration parameters
- (2)  $P_5$  provides a 1-1 mapping between input and output lists
- (3) Any path containing  $P_3$  should be excluded from the derivation tree
- (4) Processor  $P_6$  should be excluded from the derivation tree

Figure 4 (left) shows a new version of the workflow graph, where the nodes that will be ignored according to (3) and (4) are shown in dotted lines. The derivation tree from  $G/g$  back to the input  $A_2/a_2$  is shown on the right. Note that  $F/f$  and  $A_1/a_1$  are now mentioned only as part of the processor configuration, and that  $G/g$  now appears to be derived from  $D_2$  through a two-nodes path involving  $P_7$  as well as  $P_6$ . Also, since we know that  $P_5$  maps each element  $c_{2i}$  of its input list  $C_2$  to the corresponding element  $d_{2i}$  in  $D_2$ , we can make the derivation from  $c_{2i}$  to  $d_{2i}$  explicit in the tree, resulting in the three branches shown in the figure (this fine granularity does not extend to  $B/b$  nor to  $G/g$ ).

## 4 Discussion and conclusions

The work presented in this paper stems from the hypothesis that a model for describing the lineage of workflow data products can improve in precision by adding a few, selected annotations to the workflow, both at design time and

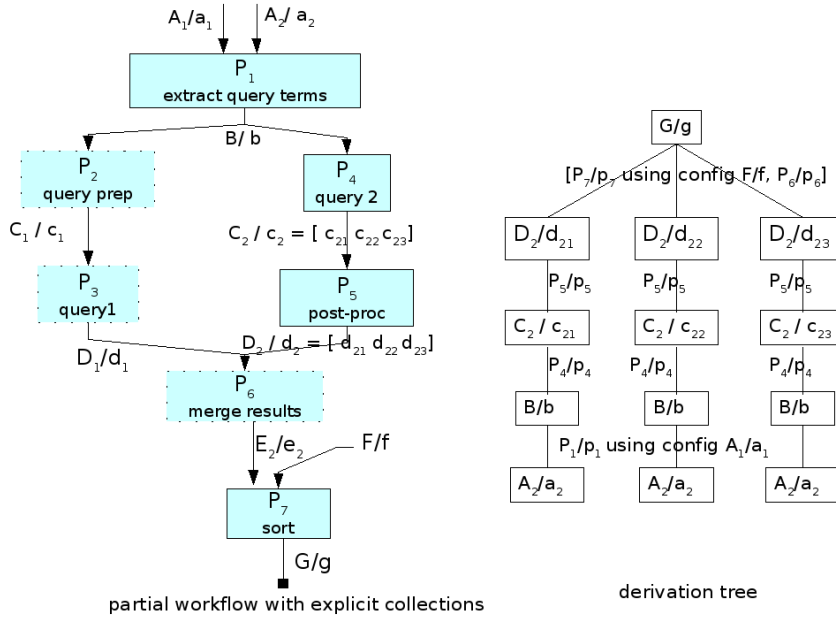


Fig. 4: Derivation tree obtained using additional annotations and user selection

at execution time. Furthermore, a simple selection of relevant processors by the users when formulating lineage queries can be effective in presenting lineage data at a suitable level of abstraction. We have proposed a simple classification of lightweight annotation types and have demonstrated their impact by comparing a derivation tree obtained as the results of a typical lineage query, with the equivalent derivation tree obtained using a baseline, annotation-free data model for lineage in Taverna.

A number of well-known workflow management systems for scientific applications have been proposed which collect and exploit provenance information for different purposes. These include enabling partial, “smart” re-runs of previously executed workflows (Vistrails [7] and Kepler [1]), debugging workflows (Kepler [1]), and comparing experiment results (Karma [18])). Hidders *et al.* [9] describe a formal functional model of *dataflow repositories* using the Nested Relational Calculus [6] (NRC). The authors show how, for dataflows described using NRC, the lineage of any occurrence of a value that appears during the course of a workflow execution can be specified using the same formalism, in terms of a path across the dataflow model. This interesting reference model could, potentially, be adopted as a starting point for our own model of data lineage. This would entail showing that Taverna workflows can be expressed using the NRC-based dataflow model, so that the provenance inference rules defined therein can be applied. Indeed it would be interesting, but beyond the scope of our current work, to investigate how the model can be used to describe the types of annotations

that we propose in this paper, and their effect on the computation of lineage paths.

It is important to emphasize that, in this paper, we are not claiming any specific element of novelty with respect to the provenance models and management systems just mentioned: tracking “raw” data lineage on a dataflow graph is, after all, a well-defined problem with known solutions, as the cited research shows. Here we focus instead on the problem of specifying and exploiting additional properties that may be known about the graph components, to bring added value to provenance users. In this respect, the *Zoom* system [4] is perhaps the closest in spirit to our efforts. *Zoom* lets users define personalised “composite modules” that are abstractions of the concrete workflow, by way of grouping some of its components and then selecting the relevant groups. The system then provides answers to provenance queries that are consistent with the abstraction level chosen by the user. The type of abstraction envisioned by the authors is similar to that described in Figure 1, i.e., by modular composition.

In a similar vein, Simon miles *et al.* [12] propose a mechanism for narrowing the scope of provenance queries. In this proposal, *p-assertions* are used when provenance is collected as a way to document the relationships among items of provenance metadata. In particular, one can use p-assertions to specify causal, functional, or other kinds of relationships. Provenance queries are then scoped based on these p-assertions types.

In contrast to both these approaches, we envision a distinction among processors that is independent of any grouping/nesting feature, and is instead based on the contribution of individual processors to the “latent data model”, as we have described it earlier.

The only work to our knowledge that considers the use of semantic annotations for analysing workflow provenance is by Miles *et al.* [13], where a method for validating scientific experiments is proposed. The validation entails reasoning over collected data provenance and the semantic descriptions of the services that compose the workflows. Its goal is to ensure that the experiments are enacted correctly, and that the results they deliver are of value. Using this method, for example, a user is able to check that the intermediate data delivered by a given service operation belongs to the appropriate domain, e.g. “protein”. It is worthwhile noting that this proposal assumes that semantic annotations of web services are always available. However, practice shows that semantics annotations are a scarce commodity in general [2]. With this in mind, the solution we propose is incremental in that semantics annotations are not mandatory inputs for analysing the lineage of workflow results. Rather, they are used for improving the analysis and facilitating the interpretation of the lineage results.

Finally, although the systems mentioned above define a variety of different data lineage models, a consensus has recently begun to emerge among different groups for a common model for the broad notion of workflow provenance. The result is an initial version of the Open Provenance Model (OPM) [14], a conceptual model that describes provenance using a pre-defined set of entities and relationships. This is an interesting reference schema onto which we hope to map

our lineage model (a detailed comparison between the lineage model proposed in this paper and the OPM is beyond the scope of this paper).

The work presented in the paper is still in progress and forms the core of the provenance architecture for Taverna, with support for a range of queries, both on a single workflow execution and across executions. A mapping of the Taverna lineage model to the Open Provenance Model is also in the plans.

## References

1. I. Altintas, O. Barney, and E. Jaeger-Frank. Provenance collection support in the Kepler scientific workflow system. In *IPAW*, pages 118–132, 2006.
2. Khalid Belhajjame, Suzanne M. Embury, Norman W. Paton, Robert Stevens, and Carole A. Goble. Automatic annotation of web services based on workflow definitions. *ACM Transactions on the Web*, 2(2), 2008.
3. O. Benjelloun, A. Das Sarma, A. Y. Halevy, M. Theobald, and J. Widom. Databases with uncertainty and lineage. *VLDB J.*, 17(2):243–264, 2008.
4. O. Biton, S Cohen-Boulakia, S. Davidson, and C. Hara. Querying and managing provenance through user views in scientific workflows. In *Procs. International Conference on Data Engineering (ICDE)*, April 2008.
5. P. Buneman, S. Khanna, and W. Chiew Tan. Why and where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.
6. Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, 1995.
7. S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, Cláudio T. Silva, and H. T. Vo. VisTrails: visualization meets data management. In *SIGMOD Conference*, pages 745–747, 2006.
8. A. Chapman and H. V. Jagadish. Issues in building practical provenance systems. *IEEE Data Eng. Bull.*, 30(4):38–43, 2007.
9. J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, and J. Van den Bussche. A formal model of dataflow repositories. In S. Cohen Boulakia and V. Tannen, editors, *DILS*, volume 4544 of *Lecture Notes in Computer Science*, pages 105–121. Springer, 2007.
10. D. Hull. Description and classification of shims in *mygrid*. Technical report, University of Manchester, 2006.
11. D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34:W729–W732, 2006.
12. Simon Miles. Electronically querying for the provenance of entities. In *Provenance and Annotation of Data, International Provenance and Annotation Workshop, IPAW 2006, Chicago, IL, USA, May 3-5, 2006, Revised Selected Papers*, pages 184–192. Springer, 2006.
13. Simon Miles, Sylvia C. Wong, Weijian Fang, Paul T. Groth, Klaus-Peter Zauner, and Luc Moreau. Provenance-based validation of e-science experiments. *J. Web Sem.*, 5(1):28–38, 2007.
14. L. Moreau, J. Freire, J. Futrelle, R. McGrath, J. Myers, and P. Paulson. The Open Provenance Model, December 2007.

15. T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, pages 3045 – 3054, November 2004.
16. Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3):31–36, 2005.
17. Y. L. Simmhan, B. Plale, and D. Gannon. A framework for collecting provenance in data-centric scientific workflows. In *ICWS*, pages 427–436, 2006.
18. Y.L. Simmhan, B. Plale, and D. Gannon. Towards a quality model for effective data selection in laboratories. *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on*, pages 72–72, 2006.
19. W. Chiew Tan. Provenance in databases: Past, current, and future. *IEEE Data Eng. Bull.*, 30(4):3–12, 2007.
20. D. Turi, P. Missier, D. De Roure, C. Goble, and T. Oinn. Taverna Workflows: Syntax and Semantics. In *Proceedings of the 3rd e-Science conference*, Bangalore, India, December 2007.