Abstract Provenance Graphs: Anticipating and Exploiting Schema-Level Data Provenance

Daniel Zinn Bertram Ludäscher

{dzinn,ludaesch}@ucdavis.edu

Abstract. Provenance graphs capture flow and dependency information recorded during scientific workflow runs, which can be used subsequently to interpret, validate, and debug workflow results. In this paper, we propose the new concept of *Abstract Provenance Graphs* (APGs). APGs are created via static analysis of a configured workflow W and input data schema, i.e., *before* W is actually executed. They summarize *all* possible provenance graphs the workflow W can create with input data of type τ, that is, for each input $v \in \tau$ there exists a graph homomorphism $\mathcal{H}_v$ between the concrete and abstract provenance graph. APGs are helpful during workflow construction since (1) they make certain workflow design-bugs (e.g., selecting none or wrong input data for the actors) easy to spot; and (2) show the evolution of the overall data organization of a workflow. Moreover, after workflows have been run, APGs can be used to validate concrete provenance graphs. A more detailed version of this work is available as [14]. [1]

1 Introduction

The ability to record, visualize, and query provenance information (in particular data lineage) is considered a key feature of scientific workflow systems and is becoming increasingly important, e.g., to help interpret, validate or debug runs of scientific workflows. So far, provenance information is provided, almost by definition, only *after* the execution of a workflow run. We propose a novel way of specifying, deriving, and exploiting a-priori (i.e., design-time) provenance information, i.e., which anticipates and summarizes the structure of workflow provenance graphs, based on (i) the given workflow specification, (ii) a description of the workflow input structure (e.g., XML DTDs), and (iii) declarative data scope expressions (i.e., actor configurations).

We focus on dataflow-oriented workflows with structured data models. Here, data is organized in nested, labeled collections much like XML data. The scientific data (*base data*) is handled opaquely by the workflow specification and the execution engine. *Actors*, which wrap external components or tools (*base functions*) use *configurations* to describe the interaction between the base data organized in nested collections and the base functions.

Example 1: Simple phylogenetics workflow. Fig. 1 shows a simple phylogenetics workflow. The input data, a set of amino acid sequences (of base type `Seq`) is stored inside the `Project` collection that will also contain the intermediary and overall output data. The actor Clustalw is configured to use all sequence objects (labeled with their type `Seq`) as input, and create a new sub-collection `Aligned` in the `Project` collection to put all output data in. Quicktree takes all `Seq` objects in the `Aligned` collection, passes the data to the Quicktree tool, and inserts the tool's output, a phylogenetic tree, directly

[1] This work was supported in part by NSF awards IIS-0612326, OCI-0722079, DBI-0619060, DE-FC02-07ER25811, ATM-0619139, and IIS-0630033.

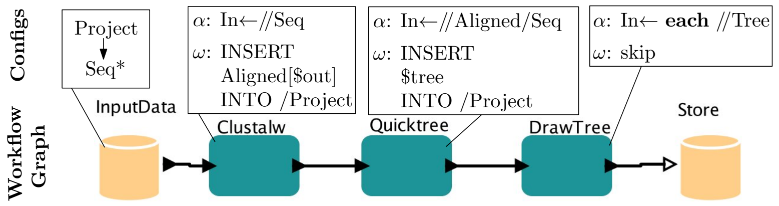

Fig. 1. A simple phylogenetics workflow consisting of three actors Clustalw, Quicktree, and DrawTree, together with a data source and sink. While the data (organized in nested, labeled collections) flows through the actors during workflow execution, each actor selects base data, calls external services, and places their results back into the stream. Actor configurations specify which base data is selected (α part of the configuration) and how results are written back into the stream (ω part of the configuration). Note that the write configuration ω of DrawTree is the no-operation `skip` since DrawTree should not have any effect on the collection data.

under `Project`. DrawTree, which is used only for display purposes, draws each tree object found in the input data. □

During workflow design, the scientist places actors on the workflow tool's canvas and subsequently provides actor configurations. The configurations play a significant role for the semantics of the workflow, and it is thus important that the designer does not introduce bugs. Our approach of providing the scientist with an abstract provenance graph during this crucial phase helps to detect errors in the configurations. Abstract provenance graphs make it obvious which base data is used and produced by which actor, and how the data organization evolves during the workflow execution.

The main ideas and steps of our approach are as follows: We compute APGs *ahead of time*, i.e., before a workflow W executes, using *static analysis* (type inference) techniques. Specifically, we infer a *schema-level summary* of the *possible* concrete provenance graphs that W can generate for the given input structures and actor configurations. Since the information is provided at the schema-level, an APG can be seen as a compile-time summary of the scientific workflow itself.

In particular, we make the following contributions: **(1)** We define abstract provenance graphs as summaries for the concrete provenance graphs a workflow can create for a given input schema. Concrete and abstract graphs are related via graph homomorphisms. **(2)** We introduce three kinds of abstract provenance graphs for workflows with a structured data model: flowgraph, time-collapsed and structure-collapsed flowgraph. **(3)** We provide examples to demonstrate the usefulness of APGs for workflow design.

2 Motivation

Recent work about scientific workflow design has demonstrated that constructing scientific workflows using an XML-like data model with XPath-like configurations leads to robust workflows with less shims and wires compared to approaches that do not deploy structured data models [10,13,12]. The key insight is that the XML data structure provides a level of indirection for actor communications and thus effectively removes the tight coupling between data flow, control flow, and the workflow graph.

Bugs introduced in the workflow configurations are hard to detect during design-time. The configurations determine which part of the input data of an actor is used as

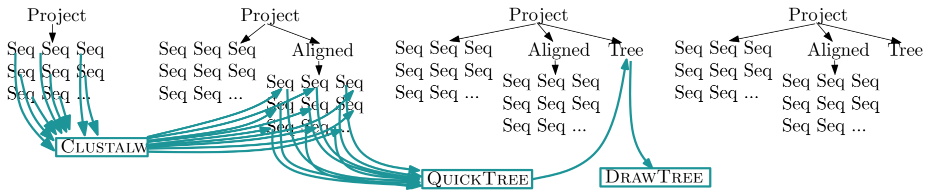

Fig. 2. Provenance flowgraph for the workflow of Fig. 1. It shows that the CLUSTALW actor reads in all `Seq` objects to create the `Seq` objects under the `Aligned` collection. These newly created aligned sequences are then used by QUICKTREE to infer a phylogenetic tree. DRAWTREE only displays the tree and not change the data stream; thus it is not connected to the last data-graph.

input to the wrapped component (base functions) and how the components output is incorporated back into the actors' XML output stream. Errors in input configurations α can cause actors to not call their base functions, simply because the XPath expressions do not match any data in the input stream. Further, even when input data is selected and base functions are called, a configuration error can cause a base function to be supplied with the wrong input data, i.e., data that the workflow designer did not intend to be input. We will now provide examples for these two kinds of errors.

Example 2: Configuration errors causing idle actors. Consider the phylogenetics workflow from Example 1 (Fig. 1). Imagine the input expression α of the QUICKTREE actor to contain a spelling error `//Alinged/Seq` instead of `//Aligned/Seq`. Then, no data would be selected from the actor's input, and consequently, its base function (here the QUICKTREE tool) would not be called; also none of the following actors would execute their base function. This bug of *idle* actors is hard to spot during design time. □

Example 3: Configuration errors causing wrong input selections. Consider again the workflow in Fig. 1 with the input expression α of QUICKTREE changed to `//Seq`. Although the actor is not idle, the data provided to the base function comprises *all* sequence data. This includes the aligned sequences as well as the unaligned ones that were part of the global workflow input. Again, this configuration error is not evident without carefully inspecting the configurations and having the overall XML structure in mind. Note that this type of bug might even be hard to notice during runtime: the base function will simply be provided with more data, potentially not creating obvious fail-stop faults, but hard-to-detect semantic errors. □

To summarize, although configurations allow us to construct flexible and adaptive workflows, they are also prone to typos and other errors that would cause the workflow to behave in ways not intended by the designer. However, once a workflow has been run, the data and its lineage (or provenance) can be visualized in several ways. A provenance *flowgraph* [3] shows how the nested collection structure and the data evolves from one workflow step to the next. The flowgraph of the workflow from Example 1 is shown in Fig. 2: the collection structure is laid out as a tree using black top-to-down edges; the green left-to-right edges show dataflow from the collection input to the actors and further to the output collection. The provenance flowgraph visualizes the detailed dataflow of a scientific workflow. It can thus be used to detect errors in the actor configurations. However, the following two reasons prevent the flowgraph being utilized during workflow design: **(i)** The provenance graph, by definition, is constructed *during* or *after* the

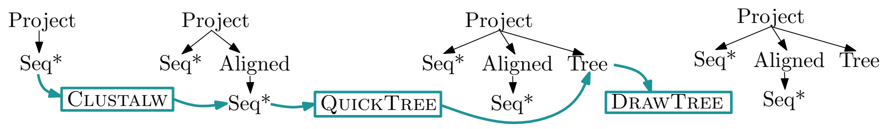

Fig. 3. Abstract provenance flowgraph for the phylogenetic workflow from Fig. 1. Similar to the concrete provenance graph (Fig. 2), a data-oriented view of the workflow is presented. However, the abstract graph uses a graphical representation at the schema-level to summarize the data involved in the computation and is thus more compact than the concrete flowgraph.

workflow execution. **(ii)** The provenance graph provides too much detail. In fact, for realistic workflows, provenance graphs can easily contain thousands of nodes [3], making them impractical to find design-errors without explicitly querying the graph structure.

3 Abstract Provenance Graphs

Similar to concrete provenance graphs, abstract provenance graphs show the collection structure and dataflow. However, (1) the graph is computed as a static analysis before the workflow is run, and (2) the data and actors are shown at a *type level* and thus in a condensed yet informative way. Fig. 3 shows the abstract flowgraph for the phylogenetics workflow from Example 1. The relationship between workflow description W, a concrete flowgraph F_W, and an abstract flowgraph A_W is shown in the following diagram.

$$\begin{tikzcd}
W \arrow[rr, "\textit{static analysis}", "\text{via } \tau"'] \arrow[dr, "\substack{\textit{workflow}\\ \textit{execution}}", "\text{via } v \in \tau"'] & & A_W(\tau) \arrow[dl, dashed, bend right=10, "\substack{\textit{summary/}\\ \textit{constraints}}"'] \\
& F_W(v) \arrow[ur, bend right=10, "\substack{\textit{embedding /}\\ \textit{graph homomorphism } \mathcal{H}_v}"'] &
\end{tikzcd}$$

During the execution of a workflow W on an input value v, provenance information can be collected to create a concrete flowgraph $F_W(v)$. However, given a workflow W together with an input type τ, we can infer an abstract flowgraph $A_W(\tau)$ via abstract interpretation, a form of static analysis.

The abstract provenance graph summarizes possible concrete provenance graphs (i.e., one for each value $v \in \tau$) via an *embedding* that gives rise to a graph homomorphism[2] on the two graphs. Thus, the APG constrains the possible provenance graphs that can be created by the specific workflow W with input schema τ. Consider the APG in Fig. 3: Since there is no edge between the left `Seq` node in the second type graph to the QuickTree actor, there is *no* input value $v \in \tau$ for which QuickTree would use any of those sequence-data as input. The abstract provenance graph can therefore be used as a data-oriented view of the workflow specification itself. Since it is created at the type-level without actually executing the workflow, it can be used during workflow design time to provide immediate feedback to the designer upon configurations changes.

We use XML to represent nested, ordered collections that can contain base data, where $\mathtt{B}^v$ and $\mathtt{C}^v$ denote the set of base data nodes and collection nodes of a value v respectively. To simplify the presentation of the rest of the paper, we consider workflow pipelines, i.e., where a workflow W is a sequence of actors: $W = A_1 \rightarrowtriangle A_2 \rightarrowtriangle \ldots \rightarrowtriangle A_n$.

[2] A graph homomorphism is a mapping between two graphs that respects their structure. More concretely, it maps adjacent vertices to adjacent vertices.

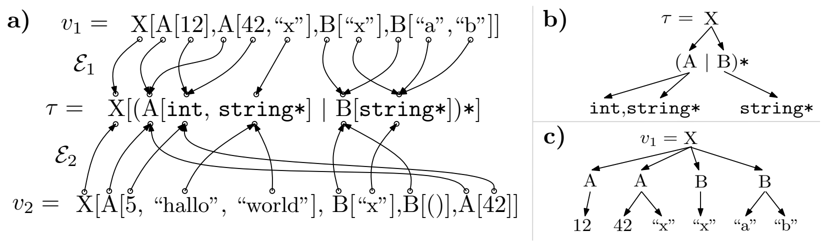
a)
v₁ = X[A[12],A[42,"x"],B["x"],B["a","b"]]
ℰ₁
τ = X[(A[int, string*] | B[string*])*]
ℰ₂
v₂ = X[A[5, "hallo", "world"], B["x"],B[()],A[42]]
b)
τ = X
(A | B)*
int,string*
string*
c)
v₁ = X
A
A
B
B
12
42
"x"
"x"
"a"
"b"

Fig. 4. **(a)** regular expression type τ and values $v_1, v_2 \in \tau$ with embeddings $\mathcal{E}_1$ and $\mathcal{E}_2$; **(b)** and **(c)** show the graphical representations of τ and v_1, respectively.

We identify each actor with a function (or update) from values to values. The execution semantics of W on input data v_0 is then simply the composition of its actors.

Provenance flowgraph. A provenance *flowgraph* $F_W(v_0)$ shows the evolution of the XML data $v_0, \ldots, v_n$ during the execution of workflow W on the XML data v_0 (Fig. 2). In particular, the provenance of base data items $d \in \mathtt{B}^v$ is illustrated. $F_W(v_0)$ is composed from (1) the individual graphs for each value, (2) nodes $i \in \mathtt{I}$ representing actor invocations, and (3) provenance edges of the kind `in`, `out`, and `copy` with $\mathtt{in} \subseteq \mathtt{B} \times \mathtt{I}$, $\mathtt{out} \subseteq \mathtt{I} \times \mathtt{B}$, and $\mathtt{copy} \subseteq \mathtt{B} \times \mathtt{B}$. Thus, our model closely ressembles the Open Provenance Model (OPM) [11]. Our `in` and `out` relations correspond to the inverses of OPM's `used` and `genBy` relations.

3.1 Abstract Provenance Flowgraphs

As an important step towards the creation of APGs, we now introduce the formalism for our types τ. We adapt regular expression types (RE types) [9] to *summarize* a set of values. Our RE types are similar to DTDs or XML-Schema, with two distinctions: (1) we disallow recursion, and (2) we restrict them to our data model, which contains no attributes. (3) As it is the case in XML Schema, we disallow ambiguous [6] RE types. Like XML Schemas, RE types can encode vertical context information (the sequence of labels from the root to the current node). Our non-recursive RE types are of the following form:

$$\tau ::= () \mid T \mid \tau, \tau' \mid \tau|\tau' \mid a[\tau] \mid \tau^* \qquad a \in \mathcal{L},\ T \in \mathcal{T} \tag{1}$$

An RE type can either be the type of the empty sequence (); a base type T (e.g., `string` or `Seq`); a sequence of two already defined types; the alternative of two types; a collection type $a[\tau]$ with a label a from the label alphabet $\mathcal{L}$; or a repetition type τ^*. The set of values of a type τ (written $[\![\, \tau \,]\!]$) is recursively defined in the usual [9] way:

$$\begin{array}{ll} (i) & [\![\, () \,]\!] = \{()\} \\ (ii) & [\![\, T \,]\!] = \{d \mid d \text{ is a base data value of type } T\} \\ (iii) & [\![\, \tau, \tau' \,]\!] = \{x, y \mid x \in [\![\, \tau \,]\!], y \in [\![\, \tau' \,]\!]\} \\ (iv) & [\![\, \tau|\tau' \,]\!] = [\![\, \tau \,]\!] \cup [\![\, \tau' \,]\!] \\ (v) & [\![\, a[\tau] \,]\!] = \{a[x] \mid x \in [\![\, \tau \,]\!]\} \\ (vi) & [\![\, \tau^* \,]\!] = \{a_0, a_1, \ldots, a_n \mid n \in \mathbb{N},\ 0 \le i \le n,\ a_i \in [\![\, \tau \,]\!]\} \end{array} \tag{2}$$

Note, how the embedding $\mathcal{E}_1$ in Fig. 4(a) is a summary for the value v_1: regardless of how many `A`-labeled subtrees there are in v_1, they are all mapped to the single `A` symbol in the type τ. In general, sequences in the value that are characterized by the repetition constructor "$*$" are collapsed in the type. Furthermore, since every $v \in \tau$ has a derivation that corresponds to an embedding, τ summarizes *all* its values. Fig. 4(a) highlights this fact by showing two different values v_1 and v_2 with their respective embeddings. We further group multiple invocations to one actor node via $\mathcal{A} : \mathtt{I} \rightarrow \mathtt{A}$. Due to space constraints, we refer to [14] for more details.

The abstract provenance flowgraph $A_W(\tau_0)$ is based on the intermediary types τ_i and the workflow output type τ_n (which are constructed via propagating τ_0 through the workflow) and provenance edges. This is similar to the concrete flowgraph, which is composed of the graphs for the individual values $v_0, \ldots, v_n$. Since there are embeddings $\mathcal{E}_i$ for each of the values into each of the types in the abstract graph, and since $\mathcal{A}$ is a mapping between the invocation nodes in $F_W(v_0)$ and the actor nodes in the abstract flowgraph $A_W(\tau_0)$, we have a complete mapping of all nodes in $F_W(v_0)$ to the nodes in $A_W(\tau_0)$. Similar mappings can be constructed for a different input value $v'_0 \in \tau$. We now require that edges in $A_W(\tau_0)$ are placed such that for all input values $v \in \tau$ the resulting mapping $\mathcal{H}_v := \mathcal{E}_v \cup \mathcal{A}$ is a "tight" graph homomorphism as described below:

Property 1. The abstract flowgraph $A_W(\tau_0)$ has a provenance edge e (e.g., `in`, `out`, or `copy` edge) between two nodes N_1, N_2 iff there is an input value $v \in \tau_0$ such that the concrete flowgraph $F_W(v)$ contains two nodes n_1, n_2 with $\mathcal{H}_v(n_1) = N_1$ and $\mathcal{H}_v(n_2) = N_2$, such that n_1 and n_2 are connected with a provenance edge e of the respective kind[3].

Corollary 1. *If there is no* `in` *edge between a base type node T and an actor node A in the abstract flowgraph $A_W(\tau)$, then in **no** execution of W on **any** value $v \in \tau$ will **any** invocation of actor A use a data item b that would be mapped to T via $\mathcal{H}_v$. In particular, if an actor node A does not have any incoming edges in the abstract flowgraph, then its base function will never be called.*

This corollary is very useful in practice, as it helps to discover errors as in Example 2. The abstract provenance graph, which indicates that none of the actors QuickTree and DrawTree will be called is shown in Fig. 5.

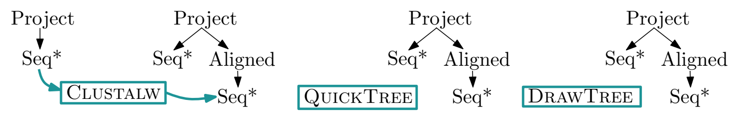

Fig. 5. Abstract flowgraph for Example 2 showing idle actors QuickTree and DrawTree

Corollary 2. *If there is an* `in` *edge between a base type node T and an actor node A in the abstract flowgraph $A_W(\tau)$, then there is at least one input value $v \in \tau$ such that executing W on v will cause an invocation of actor A that uses a data item b that corresponds to T via $\mathcal{H}_v$.*

[3] Note, that we have not drawn copy edges in our abstract provenance layouts (e.g., in Fig. 3) to avoid cluttering the graph.

This corollary helps to identify configuration errors as in Example 3, where too much data was selected as input for a particular component:

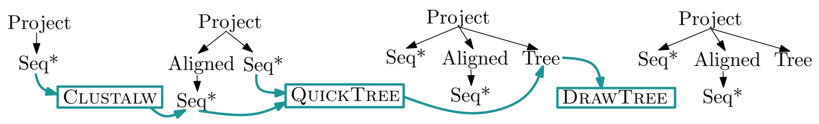

Fig. 6. Abstract flowgraph for Example 3 showing that QUICKTREE also uses the unaligned set of sequences as input and not just the aligned ones as was desired.

3.2 Variations of Abstract Provenance Graphs

Abstract provenance flowgraphs can be used as a starting point to create even more coarse-grained summaries:

Time-collapsed flowgraph. Instead of showing the evolution of intermediary data from actor to actor in the workflow, we can collapse all nodes that are connected via copy edges into one single node. This *view* is especially interesting in workflows that only add data and collections from step to step, since here each node in the collapsed graph is also a node in the output type τ_n (since no actor deletes data or collections). Thus, the time-collapsed flowgraph for add-only workflows corresponds to a summary of the output data, explaining its provenance:

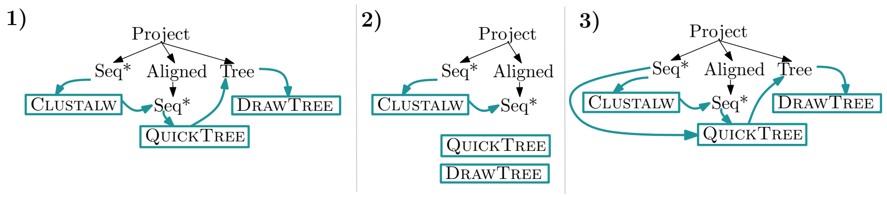

Fig. 7. Time-collapsed abstract flowgraphs for the workflows described in Examples 1-3. **1)** is the intended behavior, in **2)**, a configuration errors causes two actors to idle, and in **3)**, QUICKTREE also consumes the `Seq` data directly under the `Project` collection, which is a design error.

Structure-collapsed flowgraph. Starting from the time-collapsed flowgraph, we can additionally summarize the graph by collapsing XML nesting edges into their leaf nodes, i.e., into the data type nodes. The result (Fig. 8) shows how base data evolves.

4 Related Work & Conclusion

Our provenance model is closely related to the Open Provenance Model (OPM) [11]. OPM does not directly support nested data; although there is a proposal to handle collections in OPM [8]; we adopt the extensions of Anand et al. [3] for nested data here. Our concrete provenance flowgraph is also based on [3], which introduces a provenance model for workflows with XML-structured data models and actors with update semantics. In their work, they use a *combined structure* for efficient storage, which was the inspiration for our time-collapsed abstract graph versions. In [2], they propose summary techniques for provenance graphs along with a model to navigate between these

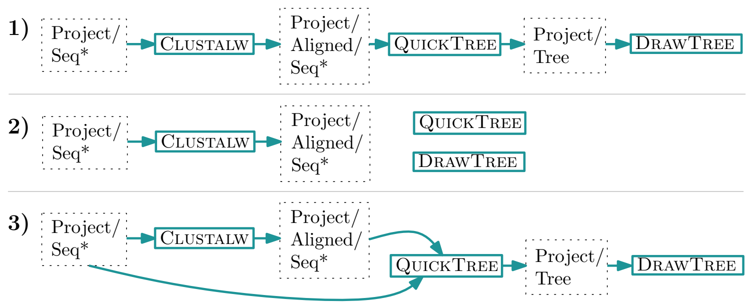

Fig. 8. Structure-collapsed flowgraphs for the workflows from Examples 1-3. The collection-structure is collapsed into the leaf nodes. This graph shows the explicit routing of data items through the set of actors. In this view, actors that work on data independently are drawn as parallel branches (not shown in these examples).

different summaries. This work is similar to ours in the sense that it also addresses the problem of summarizing provenance graphs. However, their approach is based on actual provenance information that has been gathered during a workflow run. Their created views thus summarize only one specific workflow execution—not like our approach, which summarizes all possible executions based on the workflow's input data type. Furthermore, our approach is intended to be used during workflow design-time when no actual provenance information is available yet.

In a recent paper, Acar et al. [1] investigate the relationship between provenance graphs and the computation performed by the system. They extend DFL, a dataflow-oriented extension of the nested relational calculus, to produce concrete provenance graphs. This paper is close to ours in the spirit of computing provenance graphs from the language in which the workflow is defined rather than by collecting provenance information via a rather loosely linked provenance recording mechanism. Our paper demonstrates another advantage of linking provenance closely with the model of computation by showing the usefulness of computing schema-level graphs.

Related to the summarization goal of our abstract graphs is the work from Biton et al. [5,4], where groups of actors in a workflow are replaced by a module to simplify the provenance information. Our work here is orthogonal in the sense that the ZOOM groups can be used to further collapse multiple actors in our abstract graphs. In other words, we can further summarize abstract graphs by applying the ZOOM grouping to our grouping $\mathcal{A}$ of invocations.

Our work, suggesting to use abstract provenance graphs as feedback, aims at improving the workflow design process. Viewed from this perspective, there exists related work within the scientific workflow community. In [7], Gibson et al. present a "data playground" for intuitive workflow specification, in which users can focus on their data, rather than on the processes of the workflow. It would be interesting to investigate whether our concept of abstract provenance graphs can be utilized in this system. Using abstract provenance graphs inside a GUI to create workflow configurations by having the users interactively select nodes, and possibly groupings for multiple invocations, is also an interesting avenue for future work.

Conclusion. Abstract provenance graphs make explicit use of XML typing mechanisms to summarize potential provenance graphs. We generalized embeddings that occur while validating XML documents with DTDs to graph homomorphisms between concrete and abstract provenance graphs. Similar to how an XML document is validated against a DTD, our approach allows to validate a concrete flowgraph $F_W(v)$ (recorded by a scientific workflow system) against the abstract flowgraph $A_W(\tau)$ obtained from a configured workflow and input type τ. Furthermore, based on type propagation algorithms, abstract provenance graphs can be constructed without executing the workflow. Thus, they allow the designer to anticipate the high-level (XML) structure of the workflow result, together with a summary of the result derivation in terms of the workflow's active components (actors). To the best of our knowledge, this is the first attempt to exploit provenance information during the design process of scientific workflows.

Acknowledgements. The authors thank Timothy McPhillips, Lei Dou, Sean Riddle, Sven Köhler, and Shawn Bowers for their work on collection-oriented modeling and design in Kepler, as well as for the many fruitful discussions.

References

1. U. Acar, P. Buneman, J. Cheney, J. V. den Bussche, N. Kwasnikowska, and S. Vansummeren. A graph model of data and workflow provenance. In *Proceedings of the 2nd USENIX Workshop on the Theory and Practice of Provenance (TaPP '10)*, 2010. 8
2. M. K. Anand, S. Bowers, and B. Ludäscher. A navigation model for exploring scientific workflow provenance graphs. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science (WORKS'09)*, pages 1–10, New York, NY, USA, 2009. ACM. 7
3. M. K. Anand, S. Bowers, T. M. McPhillips, and B. Ludäscher. Exploring scientific workflow provenance using hybrid queries over nested data and lineage graphs. In *SSDBM*, pages 237–254, 2009. 3, 4, 7
4. O. Biton, S. Cohen-Boulakia, and S. B. Davidson. Zoom UserViews: Querying relevant provenance in workflow systems. In *VLDB '07*, pages 1366–1369, 2007. 8
5. O. Biton, S. B. Davidson, S. Khanna, and S. Roy. Optimizing user views for workflows. In *ICDT '09: Proceedings of the 12th International Conference on Database Theory*, pages 310–323, New York, NY, USA, 2009. ACM. 8
6. A. Bruggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998. 5
7. A. Gibson, M. Gamble, K. Wolstencroft, T. Oinn, C. Goble, K. Belhajjame, and P. Missier. The data playground: An intuitive workflow specification environment. *Future Generation Computer Systems*, 25(4):453 – 459, 2009. 8
8. P. Groth, S. Miles, P. Missier, and L. Moreau. A proposal for handling collections in the Open Provenance Model, 2009. 7
9. H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(1):46–90, 2005. 5
10. T. McPhillips, S. Bowers, D. Zinn, and B. Ludäscher. Scientific workflow design for mere mortals. *Future Generation Computer Systems*, 25(5):541 – 551, 2009. 2
11. L. Moreau, B. Clifford, J. Freire, Y. Gil, P. Groth, J. Futrelle, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, Y. Simmhan, E. Stephan, and J. V. den Bussche. The Open Provenance Model - core specification (v1.1). *Future Generation Computer Systems*, 2010. 5, 7
12. D. Zinn, S. Bowers, and B. Ludäscher. XML-based computation for scientific workflows. In *Intl. Conf. on Data Engineering (ICDE)*, 2010. See also technical report. 2
13. D. Zinn, S. Bowers, T. M. McPhillips, and B. Ludäscher. Scientific workflow design with data assembly lines. In E. Deelman and I. Taylor, editors, *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science (WORKS'09)*. ACM, 2009. 2
14. D. Zinn and B. Ludäscher. Abstract provenance graphs: Anticipating and exploiting schema-level data provenance. Technical Report CSE-2010-14, UC Davis, 2010. 1, 6