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 a new concept, called *abstract provenance graph* (APG). APGs are created via static analysis of a configured workflow W and input data schema τ, i.e., *before* the workflow 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.

1 Introduction

Scientific workflows form a crucial piece of cyberinfrastructure, which allows scientists to combine existing components (e.g., for data integration, analysis, and visualization) into larger software systems to conduct "in silico" experiments [13]. Scientific workflow systems [18,15,20,19] provide a tool to create workflow specifications combined with an execution engine. While much work has focused on efficient *execution* of such scientific workflows to save "machine cycles", fewer research has been conducted geared to make their *design* as efficient as possible, and thus save valuable "human cycles". In this paper, we aim at minimizing human cycles during workflow construction time. Our contribution is to propose and develop the novel concept of an *abstract provenance graph (APG)*. APGs summarize the structure of provenance graphs a workflow can potentially create. As such, they provide a data-oriented view of the workflow under development. We show how these graphs can be used to communicate this crucial knowlegde to the workflow designer during the construction process.

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. Actor configurations describe how base data is provided as input to external components or tools (*base functions*) and how their results are incorporated back into the collection structure. *Actors*, which wrap 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.

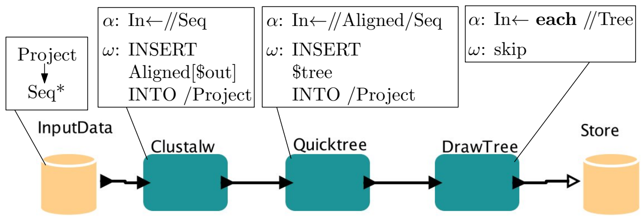

Fig. 1. A simple phylogenetics workflow consisting of three actors Clustalw, Quicktree, and DrawTree, together with a data source and sink. While the data (organized in nested, labeled collections) flows through the actors during workflow execution, each actor selects base data, calls external services, 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).

The first actor, Clustalw is configured to use all sequence objects (labeled with its type `Seq`) as input, and create a new sub-collection labeled with `Aligned` in the `Project` collection to put all output data in. The second actor, 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 under `Project`. The last actor DrawTree, which is used only for display purposes, is configured to draw each tree object found in the input data. The write scope is the no-operation `skip` since DrawTree should not have any effect on the collection 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 (i) to compute APGs *ahead of time*, i.e., before a workflow W executes, (ii) using static analysis (type inference) techniques that infer a schema-level summary of (iii) 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 doubles as a compile-time summary of the scientific workflow itself. In particular, we make the following contributions:

- 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.
- We introduce three kinds of abstract provenance graphs for workflows with a structured data model: flowgraph, time-collapsed and structure-collapsed flowgraph.
- We provide examples to demonstrate the usefulness of APGs for workflow design.

2 Motivation

Recent work about scientific workflow design has demonstrated that designing 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 [16,22,21]. The key insight here, is that the XML data structure provides a level of indirection for actor communications and thus effectively removes the tight coupling between data and control flow on the one side and the workflow graph on the other. However, there is a flip-side: to grasp the detailed the behavior of a workflow, it is necessary to understand the configurations and their interactions with each other. The workflow graph does not convey all details about the data flow anymore, instead, it merely defines the relative order of how services can be applied to the scientific data.

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 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. Imagine the input expression α of the QUICKTREE actor to contain a spelling error `//Al$\underline{\text{ing}}$ed/Seq` instead of `//Al$\overline{\text{ign}}$ed/Seq`. Then, no data would be selected from the actors' input, and consequently, its base function (here the QUICKTREE tool) would not be called. No tree would be inserted into the actor's output and consequently, also the DRAWTREE actor would not find any input tree to draw. In fact, for any input that conforms to the input type[1] `Project[Seq*]`, the actors QUICKTREE and DRAWTREE would not invoke their base function. These *idle actors* are most likely a consequence of a configuration error, i.e., they are not intended by the workflow designer, otherwise she would have built the workflow without the actors inside at the first place. Although this bug will become evident during runtime, it is hard spot it during design-time by merely inspecting the workflow graph with configurations.

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 without causing any errors at the base function. However, the base function's result data (here the phylogenetic tree) might differ from the correct data, which would have been obtained with correctly selected input. Although these errors can be identified with the help of provenance information after the workflow has run, they are hard to spot during workflow design and might even stay undiscovered.

[1] We will introduce types formally in Section 4.1.

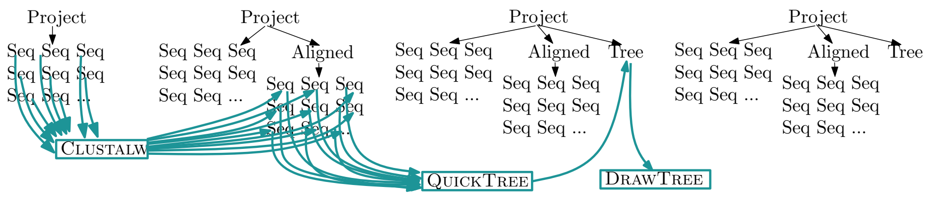

Fig. 2. Provenance flowgraph for the workflow of Fig. 1. It shows that the ClustalW actor reads in all `Seq` objects to create the `Seq` objects under the `Aligned` collection. These newly created aligned sequences are then used by QuickTree to infer a phylogenetic tree. DrawTree does only display the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

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 workflow execution. Running the overall workflow whenever the designer changes some configurations is not feasible due to two reasons: (1) since the workflow is currently under construction, running the workflow might cause un-desired side-effects. (2) Its execution can be too slow or impossible, preventing an interactive and responsive design-environment. Executing the components can be time consuming; accessing input data might be slow (or input data might not be available at all).

(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 for the provenance of specific output data items.

In the rest of the paper, we will describe how abstract provenance graphs address these shortcomings and can thus, among other things, be used to detect configuration errors as described in Example 2 and 3.

3 High-level Description of 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.

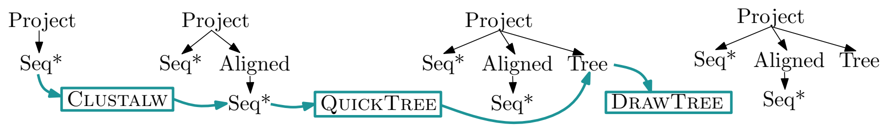

Fig. 3. Abstract provenance flowgraph for the phylogenetic workflow from Fig. 1. Similar to the concrete provenance graph (Fig. 2), a data-oriented view of the workflow is presented. However, the abstract graph uses a graphical representation at the schema-level to summarize the data involved in the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

The relationship between the workflow description W, which contains the configurations, a concrete flowgraph F_W, and an abstract flowgraph A_W is shown in the following diagram:

$$\begin{tikzcd}
W \arrow[rr, "\textit{static analysis}", "\text{via } \tau"'] \arrow[dr, "\textit{workflow execution}", "\text{via } v \in \tau"'] & & A_W(\tau) \arrow[dl, dashed, bend right, "\textit{summary/ constraints}"'] \\
& F_W(v) \arrow[ur, "\textit{embedding / graph homomorphism } \mathcal{H}_v"'] &
\end{tikzcd} \tag{1}$$

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 doubles as 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 sequences 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.

4 Details about Abstract Provenance Graphs

Abstract provenance graphs are data-oriented views of a scientific workflow. They characterize how concrete provenance graphs can look like by summarizing what operations are applied to the data during the workflow execution. This section presents APGs and their relationship to concrete provenance graphs and the workflow itself. We present our data, actor, and provenance model, describe how types summarize data and how they are visualized as graphs, and finally present APGs and variations thereof.

Data, actor, and workflow model. We use XML to represent nested, ordered collections that can contain base data. Leaf nodes are interpreted either as empty collections

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

or as base-data nodes depending on their label. We use $\mathtt{B}^v$ and $\mathtt{C}^v$ to denote the set of base data nodes and collection nodes of a value v respectively.

Each actor A comprises a configuration and a base function, which is often used to name the actor, e.g., ClustalW. During workflow execution, an actor can *invoke* its base function multiple times. XML data flows through actors that interact with it according to their configurations. In particular, actor configurations define (1) how base data is taken from the input and fed to possibly multiple invocations of the base functions, and (2) how the collection structure is updated: although arbitrary modifications are possible, actors often simply insert the output data that has been produced by the base functions. We denote the set of all actors as $\mathtt{A}$ and the set of the invocations of actor A as $\mathtt{I}_A$. The set of all invocations of all actors is denoted as $\mathtt{I}$.

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: $W = A_1 \rightarrowtriangle A_2 \rightarrowtriangle \ldots \rightarrowtriangle A_n$. We identify each actor with a function (or update) from values to values. The execution semantics of a workflow $W = A_1 \rightarrowtriangle A_2 \rightarrowtriangle \ldots \rightarrowtriangle A_n$ on input data $v_0 \in \mathcal{V}$ is then simply the composition of its individual actors (in series execution of the updates):

$$[A_1 \rightarrowtriangle A_2 \rightarrowtriangle \ldots \rightarrowtriangle A_n](v_0) := v_n \quad \text{with} \quad v_i := A_i(v_{i-1}) \;\; \text{for } i = 1 \ldots n \qquad (2)$$

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) [17]: our `in` and `out` relations correspond to the inverses of OPM's `used` and `genBy` relations. However, OPM does not model pass-through data items, i.e., data that is passed through an actor without modification. We added `copy` edges to make this notion explicit.

4.1 Type Graphs

As described in the schematic overview (1), an abstract provenance $A_W(\tau)$ graph is a summary of the concrete provenance graph $F_W(v)$ for a workflow W run on any XML value $v \in \tau$. Recall that τ is a type-formalism (e.g., a DTD) that describes a set of XML values. It is important to realize that the abstract graph A_W is not only a summary of a specific execution trace $F_W(v)$, but that it summarizes *all* execution traces that can be created by *any* input data v of type τ. As an important step towards the creation of APGs, we now introduce the formalism for our types τ and show how these types can be used as graphical summaries for values.

Type formalism. We adapt regular expression types (RE types) [14] 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. 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:

$$\tau ::= () \mid T \mid \tau, \tau' \mid \tau | \tau' \mid a[\tau] \mid \tau^* \qquad a \in \mathcal{L},\; T \in \mathcal{T} \qquad (3)$$

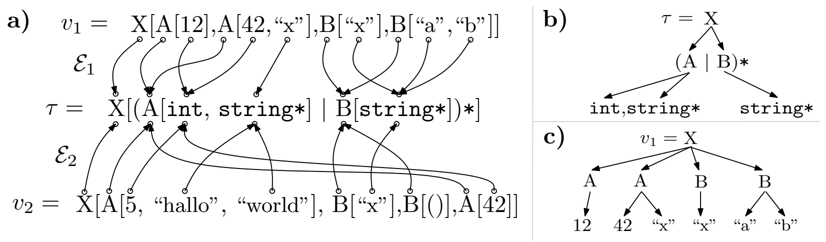
a)
$v_1 =$ X[A[12],A[42,“x”],B[“x”],B[“a”,“b”]]
$\mathcal{E}_1$
$\tau =$ X[(A[int, string*] | B[string*])*]
$\mathcal{E}_2$
$v_2 =$ X[A[5, “hallo”, “world”], B[“x”],B[()],A[42]]
b)
$\tau =$ X
(A | B)*
int,string*
string*
c)
$v_1 =$ X
A
A
B
B
12
42
“x”
“x”
“a”
“b”

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.

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 [14,4] way:

$$\begin{array}{ll} (i) & [\![\, () \,]\!] = \{()\} \\ (ii) & [\![\, T \,]\!] = \{d \mid d \text{ is a base data value of type } T\} \\ (iii) & [\![\, \tau, \tau' \,]\!] = \{x, y \mid x \in [\![\, \tau \,]\!], y \in [\![\, \tau' \,]\!]\} \\ (iv) & [\![\, \tau|\tau' \,]\!] = [\![\, \tau \,]\!] \cup [\![\, \tau' \,]\!] \\ (v) & [\![\, a[\tau] \,]\!] = \{a[x] \mid x \in [\![\, \tau \,]\!]\} \\ (vi) & [\![\, \tau^* \,]\!] = \{a_0, a_1, \ldots, a_n \mid n \in \mathbb{N},\ 0 \le i \le n,\ a_i \in [\![\, \tau \,]\!]\} \end{array} \tag{4}$$

Embedding as a summary. A derivation showing that a data value v is of a type τ naturally defines a mapping of the collection labels and base data items in v to collection labels and base type symbols in τ via the sematics rules (ii) and (v) in (4). We call this mapping the *embedding* $\mathcal{E}$ of v to τ. As common in DTDs and XML Schema, we require that RE types are *unambiguous* [7] (in W3C-lingo: have deterministic content models). A RE type τ is unambiguous if for all $v \in [\![\, \tau \,]\!]$ there is only one derivation, and consequently only one mapping between nodes in the value v to the collection labels and base type symbols in the type τ. [3]

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.

Graphical representation. Similar to XML graphs representing values, we would like to have a graphical representation of a type τ. However, since regular expression types are more complex structures than values, a graphical representation is not as obvious. While any RE type could be represented as its operator tree (i.e., with internal nodes for the operators “,”, “|”, “a[...]”, and “$*$”), in such a tree, the depth of a node from the root would not correspond with the collection-nesting depth as it is the case for values. Also, two nodes connected with a nesting edge in the data graph might not be adjacent in the type graph (Example: $v = a[d]$ and $\tau = a[D*]$). We instead propose a representation in which only the nesting structure caused by the collection constructor “a[...]” is mapped

[3] An example for a type τ that is *not* unambiguous is τ = `string*,string*` since for the value v = “x”, the single node “x” can be mapped to the first `string` or to the second `string` in τ.

to the graph, whereas the remaining operators (i.e., “$*$”,“$|$”,“,”) are not. Fig. 4(b) shows our proposed representation of the type τ. Each collection label $a[...]$ and base data symbol T in a type τ becomes a node in the type graph $G(\tau)$. Then, the type expression τ is transformed recursively (bottom-up) into a type graph by replacing each $a[\tau']$ by a with a nesting edge to each of the nodes in τ'. At the very bottom of the nesting hierarchy (recursion's base case), τ' will not have any further nesting; higher up, τ' will already be transformed into a type in which each nesting-constructor has been replaced by the label-node (with its children being attached via edges). Note, that such an expression τ' may contain multiple nodes (e.g., in Fig. 4(b), one τ' is $(A|B)*$, which contains two nodes: one labeled with A and another labeled with B); during the layout of the graph we can place these nodes next to each other and superimpose the type-expression τ' on them again (e.g., as it is done in the figure). Due to space restrictions, we omit the detailed code of the algorithm transforming a type into its type graph.

4.2 Abstract Provenance Flowgraphs

To define the structure of abstract (provenance) flowgraphs, one additional step of summarization is necessary: partitioning the set of all invocations `I` to be able to represent multiple invocations with one single node in the abstract graph. For simplicity, we just use the grouping introduced by the actors themselves, although the notion of abstract flowgraphs is not limited to this summarization. We use $\mathcal{A} : \mathtt{I} \rightarrow \mathtt{A}$ to denote the mapping of invocations to “actor-representing” nodes in the flowgraph.

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.

Note, that we have not drawn copy edges in our abstract provenance layouts (e.g., in Fig. 3) to avoid cluttering the graph. Approaches like color-codings, or dynamically displaying copy edges for base types a user clicks on, can be used to include this information without introducing too many edges.

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 below:

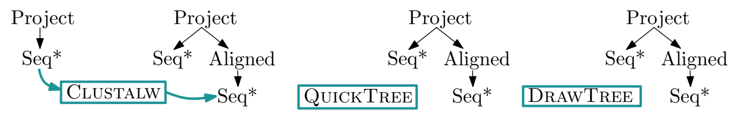

Fig. 5. Abstract flowgraph for Example 2 showing idle actors QuickTree and DrawTree

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 $\mathcal{H}_v$.*

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

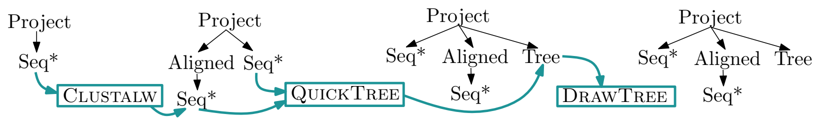

Fig. 6. Abstract flowgraph for Example 3 showing that QuickTree also uses the unaligned set of sequences as input and not just the aligned ones.

Constructing abstract provenance graphs. Abstract provenance graphs can readily be computed for XML processing languages that have a formally defined execution semantics (*semantics on values*) and a type propagation mechanism, which corresponds to a *semantics on types* or *abstract interpretation* of the XML program. The computation of the abstract flowgraph is based on existing type propagation algorithms. The key technique to exploit, is to annotate base types T in the input type τ_0 with distinct "colors" and propagate the colors from one intermediary type to the next. The *symbolic* (i.e., abstract) execution of a base function then introduces new colors for its produced type-nodes and records the provenance, i.e., the relations `in` and `out` between colors of input types-nodes and base function.

One of the languages that has a formal semantics on values and types is FLUX, a high-level XML update language introduced by Cheney in [8,9]. Here, computing abstract flowgraphs can be seen as lifting the notion of expression provenance (described in [10]) to the type level. Since "virtual data assembly line" workflows can be compiled to FLUX as demonstrated in [21], our approach is readily applicable to these workflows.

4.3 Variations of Abstract Provenance Graphs

The abstract provenance flowgraph as described in the previous section, can be used as a starting point to create even more coarse-grained summaries. We will now illustrate two of these views, namely time-collapsed and structure-collapsed flowgraphs.

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:

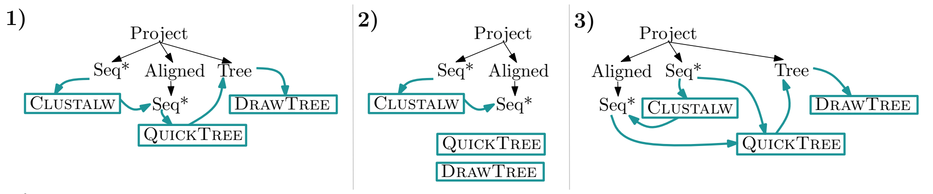

Fig. 7. Time-collapsed abstract flowgraphs for the workflows described in Examples 1-3. **1)** is the intended behavior, in **2)**, a configuration errors causes two actors to idle, and in **3)**, QUICKTREE also consumes the `Seq` data directly under the `Project` collection, which is a design error.

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

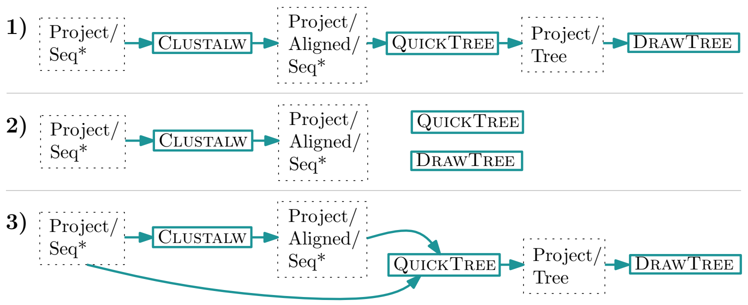

Fig. 8. Structure-collapsed flowgraphs for the workflows from Examples 1-3. The collection-structure is collapsed into the leaf nodes. This graph shows the explicit routing of data items through the set of actors. In this view, actors that work on data independently are drawn as parallel branches (not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

5 Related Work & Conclusion

Our provenance model is closely related to the Open Provenance Model (OPM) [17]. OPM does not directly support nested data; although there is a proposal to handle collections in OPM [12], 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 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. [6,5], 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 [11], 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.

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. 11
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. 11
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. 4, 10
4. V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Intl. Conf. on Functional Programming*, pages 51–63, New York, 2003. 7
5. O. Biton, S. Cohen-Boulakia, and S. B. Davidson. Zoom UserViews: Querying relevant provenance in workflow systems. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1366–1369. VLDB Endowment, 2007. 11
6. 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. 11
7. A. Bruggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998. 7
8. J. Cheney. Lux: A lightweight, statically typed XML update language. *PLAN-X*, pages 25–36, 2007. 9
9. J. Cheney. FLUX: Functional updates for XML. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 3–14, New York, NY, USA, 2008. ACM. 9
10. J. Cheney. Provenance, XML, and the scientific web. In *ACM SIGPLAN Workshop on Programming Language Technology and XML (PLAN-X 2009)*, 2009. Invited paper. 9
11. 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. 11
12. P. Groth, S. Miles, P. Missier, and L. Moreau. A proposal for handling collections in the open provenance model, 2009. 10
13. A. J. G. Hey, S. Tansley, and K. M. Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009. 1
14. 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. 6, 7
15. Kepler project. `http://http://kepler-project.org`. 1
16. T. McPhillips, S. Bowers, D. Zinn, and B. Ludäscher. Scientific workflow design for mere mortals. *Future Generation Computer Systems*, 25(5):541 – 551, 2009. 3
17. L. Moreau, J. Freire, J. Futrelle, R. E. Mcgrath, J. Myers, and P. Paulson. The Open Provenance Model: An overview. pages 323–326, 2008. 6, 10
18. Taverna project. `http://taverna.sourceforge.net`. 1
19. I. Taylor, M. Shields, I. Wang, and A. Harrison. The triana workflow environment: Architecture and applications. *Workflows for e-Science*, pages 320–339, 2007. 1
20. Viztrails project. `http://vistrails.sci.utah.edu`. 1
21. 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. 3, 9
22. 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. 3