# Declarative Rules for Inferring Fine-Grained Data Provenance from Scientific Workflow Execution Traces

Shawn Bowers[1], Timothy McPhillips[2], and Bertram Ludäscher[3]

[1] Dept. of Computer Science, Gonzaga University
[2] Stanford Synchrotron Radiation Lightsource, SLAC National Accelerator Laboratory, Stanford University
[3] Dept. of Computer Science, University of California Davis

**Abstract.** Fine-grained dependencies within scientific workflow provenance specify lineage relationships between a workflow result and the input data, intermediate data, and computation steps used in the result's derivation. This information is often needed to determine the quality and validity of scientific data, and as such, plays a key role in both provenance standardization efforts and provenance query frameworks. While most scientific workflow systems can record basic information concerning the execution of a workflow, they typically fall into one of three categories with respect to recording dependencies: (1) they rely on workflow computation steps to declare dependency relationships at runtime; (2) they impose implicit assumptions concerning dependency patterns from which dependencies are automatically inferred; or (3) they do not assert any dependency information at all. We present an alternative approach that decouples dependency inference from workflow systems and underlying execution traces. In particular, we present a high-level declarative language for expressing explicit dependency rules that can be applied (at any time) to workflow trace events to generate fine-grained dependency information. This approach not only makes provenance dependency rules explicit, but allows rules to be specified and refined by different users as needed. We present our dependency rule language and implementation that rewrites dependency rules into relational queries over underlying workflow traces. We also demonstrate the language using common types of dependency patterns found within scientific workflows.

## 1 Introduction

A key feature of scientific workflow systems is their ability to record workflow execution events at runtime, which can be used to establish various types of provenance relationships. Common events that are observed and recorded by workflow systems include the computational steps that were invoked as part of a workflow run as well as the data that were input to and output by each step. Recording these types of events in most workflow systems is straightforward, however, recording detailed provenance dependency relationships presents a number of

P. Groth and J. Frew (Eds.): IPAW 2012, LNCS 7525, pp. 82–96, 2012.

challenges. For instance, determining the sequence of computations performed to produce a data result requires understanding the fine-grained dependencies of step outputs on step inputs, which generally requires an understanding of how the underlying computation of the step is performed. While recent approaches [2] have begun to incorporate so-called "white-box" components into workflows—i.e., steps implemented in languages from which dependencies can be inferred, such as SQL and other database manipulation languages—workflow systems typically treat computation steps as "black-boxes" in which very little is known or assumed regarding the underlying implementation of steps.

Scientic workflow systems generally adopt one of three approaches for asserting provenance dependencies: (1) they rely on workflow computation steps to *declare* dependency relationships at runtime (e.g., see [4]); (2) they impose *implicit* assumptions concerning dependency patterns from which dependencies are automatically inferred (e.g., see [9,1,8,13]); or (3) they do not assert any dependency information at all. Relying on workflow steps to declare provenance relationships can be problematic, e.g., it requires a well-defined API for recording dependencies and can add considerable overhead to each step (e.g., [3]). Also, not all computational steps of interest may declare, or declare correctly, the dependencies introduced by executing the step. The use of implicit rules can be equally problematic. For example, depending on the underlying model of computation employed by a workflow system and the complexity of workflow steps, establishing implicit rules regarding dependency relationships can often lead to incomplete and incorrect dependency assertions (e.g., see [3,13]).

**Contributions.** In this paper, we describe approaches for inferring data dependencies from workflow execution traces based on *explicit* user-defined rules as opposed to implicit rules assumed by a workflow system or dependencies declared by computation steps. We propose a high-level language for expressing user-defined dependency rules that can be applied (at any time) to workow trace events to generate fine-grained dependency information. This approach takes the burden of determining provenance dependencies off of workflow systems, and allows rules to be specified and refined by different users (such as workflow developers) as needed. We present a dependency rule language and a formal implementation that converts high-level dependency rules into relational queries over an underlying workflow trace model. We also demonstrate the expressivity of the language by using the language to define common types of dependency patterns found within scientific workflows. Our approach is compatible with existing provenance standardization efforts such as OPM [15] and the W3C Prov effort [18]. In particular, both of these approaches focus on representing fine-grained data dependencies, and our approach can be used to compute these dependencies from underlying workflow execution traces.

**Organization.** This paper is organized as follows. Section 2 describes an abstract, minimal model for describing workflows, workflow traces, and data dependencies. The model is used in Section 3 as the foundation for our declarative provenance rule language. Section 3 also describes an implementation of our approach that stores execution traces within a relational schema and converts

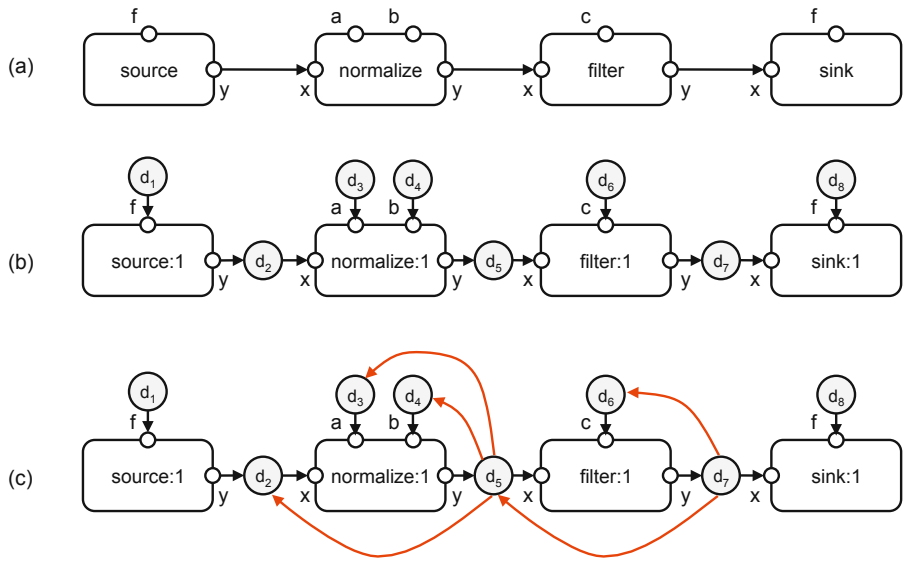


**Fig. 1.** (a) An example workflow specification, (b) an execution showing the first invocation step of each actor, and (c) the corresponding data dependencies

provenance rules into queries over schema instances. We also give examples of common dependency patterns and show how these patterns are captured using the declarative rule language of Section 3. Related work is presented in Section 4 and we summarize our contributions in Section 5.

## 2 Workflows, Traces, and Dependencies

This section describes a minimal set of observables that must be recorded by a workflow system to apply the user-defined dependency rules of Section 3. Observables are defined as part of an abstract model, and a specific (relational) implementation of the model is given in Section 3. The abstract model is divided into three distinct layers: workflow specifications, workflow traces, and fine-grained data dependencies.

Figure 1a shows a simple example workflow. In the abstract model, workflows consist of actors (which define types of steps), actor parameters, and actor channels. Actor parameters are designated as input, output, or state variables. Actor inputs receive (or read) data items, outputs produce (or write) data items, and state parameters maintain data across actor invocations (or "firings"). Actor channels define dataflow between two actors, connecting one output parameter of an actor to one input parameter of an actor. Workflow specifications in the abstract model are defined more formally as follows.

**Definition 1.** *A* workflow specification $W = (A, P, C, \sigma)$ *is a 4-tuple consisting of actor names $A$, parameter names $P$, dataflow channels $C$, and a signature*

*function* $\sigma : A \to 2^{P \times \{\texttt{in},\texttt{out},\texttt{state}\}}$ *that maps actors to their corresponding input, output, and state parameters. Each parameter in the signature of an actor must have a unique name, i.e., for each actor* $a \in A$*, if* $(p, r_1) \in \sigma(a)$ *and* $(p, r_2) \in \sigma(a)$ *then* $r_1 = r_2$*. We often write* $a.p$ *to refer to the (unique) parameter* $p$ *of actor* $a$*. Let* $A_{in}$ *and* $A_{out}$ *be the input and output parameters, respectively, of actors in* $W$*, i.e.,* $A_{in} = \{a.p \mid a \in A \wedge (p, \texttt{in}) \in \sigma(a)\}$ *and* $A_{out} = \{a.p \mid a \in A \wedge (p, \texttt{out}) \in \sigma(a)\}$*. The (directed) dataflow channels are defined as the set* $C \subseteq A_{out} \times A_{in}$ *connecting output parameters of actors in* $A$ *to input parameters of actors in* $A$*.*

For dependency inference rules, we only assume the presence of actor signatures (i.e., actor parameters and whether they represent inputs, outputs, or state). Channels are not required to apply the inference rules, and are included in the abstract model to provide a more complete view of a workflow specification. The following example describes the workflow in Figure 1a using the abstract model.

*Example 1.* Consider the workflow $W$ of Figure 1a in which the first actor reads data from a file; the second actor produces a normalized value from each value read (where `a` and `b` are the min and max values, respectively); the third actor performs a low-pass filter (i.e., outputs items received if they are less than a given cutoff value); and the last actor writes data to a file. Using Definition 1, this workflow can be represented as follows.

$$
\begin{aligned}
W &= (A, C, \sigma) \\
A &= \{\texttt{source}, \texttt{normalize}, \texttt{filter}, \texttt{write}\} \\
C &= \{(\texttt{source.y}, \texttt{normalize.x}), (\texttt{normalize.y}, \texttt{filter.x}), \\
&\qquad (\texttt{filter.y}, \texttt{sink.x})\} \\
\sigma(\texttt{source}) &= \{(\texttt{f}, \texttt{in}), (\texttt{y}, \texttt{out})\} \\
\sigma(\texttt{normalize}) &= \{(\texttt{x}, \texttt{in}), (\texttt{y}, \texttt{out}), (\texttt{a}, \texttt{in}), (\texttt{b}, \texttt{in})\} \\
\sigma(\texttt{filter}) &= \{(\texttt{x}, \texttt{in}), (\texttt{y}, \texttt{out}), (\texttt{c}, \texttt{in})\} \\
\sigma(\texttt{sink}) &= \{(\texttt{x}, \texttt{in}), (\texttt{f}, \texttt{in})\}
\end{aligned}
$$

Here, `source.f` gives the filename that data is to be read from, `filter.c` gives the cutoff value, and `sink.f` gives the filename that data is written to.

A trace of a workflow execution in the abstract model consists of a set of actor invocations, called *steps*, and their corresponding parameter *updates* (i.e., changes in parameter values). Thus workflow systems that conform to the model must be able to observe and must record both invocations and parameter updates, which are both recorded by the majority of provenance-aware workflow systems today [5,17,6]. In addition, we require only a relative order on parameter updates within actor invocations (i.e., order information across actor invocations is not required), and this information is typically provided by workflow systems using various forms of timestamps. Workflow execution traces in the abstract model are defined more formally as follows.

**Definition 2.** *A* trace $T = (S, U)$ *of a workflow* $W$ *is a pair consisting of actor steps* $S$ *and their corresponding parameter updates* $U$*. In particular,* $S \subseteq A \times \mathbb{N}$ *such that for a step* $s \in S$*,* $s = (a, i)$ *denotes an invocation step of actor* $a$*, which we typically write as* `a:i`*. No ordering constraints are placed on steps, i.e.,* `a:i` *may or may not have executed before* `a:(i+1)`*. Similarly,* $U \subseteq S \times P \times D \times \mathbb{N}$ *is the set of parameter updates, where* $D$ *is the set of data items produced by the workflow such that an update* $u = (s, p, d, j)$ *in* $U$ *denotes the* $j$*-th update of step* $s = a{:}i$ *in which the parameter* $a.p$ *was set to data item* $d$*. Updates are partially ordered for a given step. Further, each update of a specific parameter in a step must have a unique order, i.e, if* $(s, p, d_1, i) \in U$ *and* $(s, p, d_2, i) \in U$*, then* $d_1 = d_2$*.*

The only constraints placed on an execution trace with respect to a workflow specification is that all actor updates are for actors and parameters defined in the corresponding actor signatures. While it is possible to add additional constraints, these would largely be based on the computation model employed by an underlying workflow system. While most workflow systems adopt computation models based on dataflow, they often have slight differences. Examples include whether the same data item is allowed to be "written" (i.e., part of an update to an output parameter) multiple times, the number of data items that can be passed between actor invocations, and whether workflow channels define "strict" constraints on data passing (for a non-strict approach see [11]). It may also be the case that a workflow is not fully specified, or can be adapted during workflow execution. Thus, for generality, the abstract model does not presuppose any particular model of computation. The following example demonstrates how an execution (shown in Figure 1b) of the example workflow in Figure 1a can be described in the abstract model.

*Example 2.* Consider the example workflow execution shown in Figure 1b. This execution can be represented according to Definition 2 using a trace $T$ as follows.

$$
\begin{aligned}
T &= (S, U)\\
S &= \{\texttt{source:1}, \texttt{normalize:1}, \texttt{filter:1}, \texttt{sink:1}\}\\
U &= \{(\texttt{source:1}, \texttt{f}, \texttt{d}_1, 1), (\texttt{source:1}, \texttt{y}, \texttt{d}_2, 2), (\texttt{normalize:1}, \texttt{x}, \texttt{d}_2, 1)\\
&\quad\ (\texttt{normalize:1}, \texttt{a}, \texttt{d}_3, 2), (\texttt{normalize:1}, \texttt{b}, \texttt{d}_4, 3), (\texttt{normalize:1}, \texttt{y}, \texttt{d}_5, 4)\\
&\quad\ (\texttt{filter:1}, \texttt{x}, \texttt{d}_5, 1), (\texttt{filter:1}, \texttt{c}, \texttt{d}_6, 2), (\texttt{filter:1}, \texttt{y}, \texttt{d}_7, 3)\\
&\quad\ (\texttt{sink:1}, \texttt{x}, \texttt{d}_7, 1), (\texttt{sink:1}, \texttt{f}, \texttt{d}_8, 2)\}.
\end{aligned}
$$

Traces in the abstract model can be mapped to OPM provenance graphs [15]. In particular, actor invocations are similar to OPM `processes`, data items are similar to OPM `artifacts`, updates to input parameters define OPM `used` edges, and updates to output parameters define OPM `wasGeneratedBy` edges. OPM `wasTriggeredBy` edges can be obtained as follows. Assume a trace $T = (S, U)$ of a workflow $W = (A, C, \sigma)$ such that $a_1, a_2 \in A$, $(p_1, \texttt{out}) \in \sigma(a_1)$, and $(p_2, \texttt{in}) \in \sigma(a_2)$. A `wasTriggedBy` edge exists between steps $a_1{:}i_1$ and $a_2{:}i_2$ whenever there exists updates $(a_1{:}i_1, p_1, d, j_1) \in U$ and $(a_2{:}i_2, p_2, d, j_2) \in U$.

Dependency inference rules are used to infer fine-grained lineage dependencies from a given workflow trace. A lineage dependency is represented in the abstract model as a directed edge over trace updates. Thus, a dependency graph can be viewed as a separate graph of lineage edges "superimposed" over a trace (e.g., see Figure 1c). Fine-grained dependency graphs in the abstract model are defined more formally as follows.

**Definition 3.** Data (lineage) dependencies $L \subseteq U \times U$ *over a workflow trace* $T = (S, U)$ *form a directed acyclic dependency graph, where each* $(u_2, u_1) \in L$ *states that the update* $u_2$ *depended on the update* $u_1$ *(i.e.,* $u_1$ *was a dependency of* $u_2$*). We often write* $u_1 \xleftarrow{\texttt{ddep}} u_2$ *to denote that* $u_2$ *depended on* $u_1$*, i.e., that* $(u_2, u_1) \in L$*. The following additional restrictions are placed on dependency edges in* $L$*. Given updates* $u_1 = (s_1, p_1, d_1, t_1)$ *and* $u_2 = (s_2, p_2, d_2, t_2)$ *in* $T$*, if* $u_1 \xleftarrow{\texttt{ddep}} u_2$ *then*

1. $u_1$ *and* $u_2$ *must be updates of the same step, i.e.,* $s_1 = s_2$*;*
2. *update* $u_1$ *must occur before update* $u_2$*, i.e.,* $t_1 < t_2$*; and*
3. $u_2$ *must be either an output or state parameter such that if* $u_2$ *is an output,* $u_1$ *must be an input or state parameter, and if* $u_2$ *is a state parameter then* $u_1$ *can be an input, output, or state parameter, i.e., if* $W = (A, P, C, \sigma)$ *is the workflow corresponding to* $T$ *where* $s_1 = a{:}i$*,* $(p_1, r_1) \in \sigma(a)$*, and* $(p_2, r_1) \in \sigma(a)$*, one of the following must be true:* $r_2 = \texttt{out}$ *and* $r_1 \in \{\texttt{in}, \texttt{state}\}$*, or* $r_2 = \texttt{state}$ *and* $r_1 \in \{\texttt{in}, \texttt{out}, \texttt{state}\}$*.*

Dependencies in $L$ correspond to OPM's `wasDerivedFrom` edge. Specifically, a `wasDerivedFrom` edge exists from data item $d_2$ to data item $d_1$ whenever $u_1 \xleftarrow{\texttt{ddep}} u_2$ for $u_1 = (s, p_1, d_1, t_1)$ and $u_2 = (s, p_2, d_2, t_2)$. The following example demonstrates how the dependencies in Figure 1c can be represented within the abstract model.

*Example 3.* Figure 1c shows the data dependencies introduced by the workflow execution of Figure 1b. These dependencies can be represented using Definition 3 as follows:

$$\begin{aligned} L = \{&(\texttt{normalize:1}, \texttt{x}, \texttt{d}_2, 1) \xleftarrow{\texttt{ddep}} (\texttt{normalize:1}, \texttt{y}, \texttt{d}_5, 4),\\ &(\texttt{normalize:1}, \texttt{a}, \texttt{d}_3, 2) \xleftarrow{\texttt{ddep}} (\texttt{normalize:1}, \texttt{y}, \texttt{d}_5, 4),\\ &(\texttt{normalize:1}, \texttt{b}, \texttt{d}_4, 3) \xleftarrow{\texttt{ddep}} (\texttt{normalize:1}, \texttt{y}, \texttt{d}_5, 4),\\ &(\texttt{filter:1}, \texttt{x}, \texttt{d}_5, 1) \xleftarrow{\texttt{ddep}} (\texttt{filter:1}, \texttt{y}, \texttt{d}_7, 3),\\ &(\texttt{filter:1}, \texttt{c}, \texttt{d}_6, 2) \xleftarrow{\texttt{ddep}} (\texttt{filter:1}, \texttt{y}, \texttt{d}_7, 3)\}. \end{aligned}$$

Finally, we expand the notion of dependency above to consider four different kinds of possible dependency relationships among updates. First, we define the set of data items $D = D_{val} \cup D_{id}$ to be the union of the disjoint sets of data *values* $D_{val}$ and data *identifiers* $D_{id}$ (where identifiers correspond, e.g., to OPM artifacts or "tokens" in the dataflow model [7] that wrap underlying values).

For each data identifier $d_{id} \in D_{id}$ the function $v : D_{id} \rightarrow D_{val}$ gives the value $v(d_{id})$ of $d_{id}$. We do not assume any additional constraints on the interpretation of values, where values can be primitive data elements (like numbers or strings) or references to external data items.

**Definition 4.** *Given a dependency set $L$,* data derivations $L_{der}$, value-copy derivations $L_{val}$, *and* identifier-copy derivations $L_{id}$ *are subsets of $L$ such that $L_{id} \subseteq L_{val} \subseteq L_{der} \subseteq L$. If $(u_2, u_1) \in L_{der}$ (denoted $u_1 \xleftarrow{\texttt{dder}} u_2$) then the data value of update $u_1$ was involved in the derivation of the data value of update $u_2$. Derivation is a stronger assertion of lineage than dependency alone: if $(u_2, u_1) \in L$ but $(u_2, u_1) \notin L_{der}$ then the presence of $u_1$ led to the presence of $u_2$, but the value of $u_1$ was not used in this process. If $(u_2, u_1) \in L_{val}$ (denoted $u_1 \xleftarrow{\texttt{dval}} u_2$) then the value of update $u_2$ was copied from the value of update $u_1$. Similarly, if $(u_2, u_1) \in L_{id}$ (denoted $u_1 \xleftarrow{\texttt{did}} u_2$) then the identifier of update $u_2$ was copied from the identifier of update $u_1$. Thus, if $u_1 \xleftarrow{\texttt{dval}} u_2$ such that $u_1 = (s, p_1, d_1, t_1)$ and $u_2 = (s, p_2, d_2, t_2)$ then $v(d_1) = v(d_2)$. Further, if $u_1 \xleftarrow{\texttt{did}} u_2$ then $d_1 = d_2$.*

The following example further refines the fine-grained data depenencies shown in Figure 1c in terms of the types of dependencies they represent.

*Example 4.* Consider again the dependencies shown in in Figure 1c. In the case of `normalize:1`, the update of parameter `y` was derived from the `x`, `a`, and `b` values giving:

$(\texttt{normalize:1}, \texttt{x}, \texttt{d}_2, 1) \xleftarrow{\texttt{dder}} (\texttt{normalize:1}, \texttt{y}, \texttt{d}_5, 4)$,
$(\texttt{normalize:1}, \texttt{a}, \texttt{d}_3, 2) \xleftarrow{\texttt{dder}} (\texttt{normalize:1}, \texttt{y}, \texttt{d}_5, 4)$, and
$(\texttt{normalize:1}, \texttt{b}, \texttt{d}_4, 3) \xleftarrow{\texttt{dder}} (\texttt{normalize:1}, \texttt{y}, \texttt{d}_5, 4)$.

Similarly, for `filter:1` while the value of `y` was copied directly from the `x` value, it was not derived from (i.e., only depended on) the `c` value, thus:

$(\texttt{filter:1}, \texttt{x}, \texttt{d}_5, 1) \xleftarrow{\texttt{dval}} (\texttt{filter:1}, \texttt{y}, \texttt{d}_7, 3)$, and
$(\texttt{filter:1}, \texttt{c}, \texttt{d}_6, 2) \xleftarrow{\texttt{ddep}} (\texttt{filter:1}, \texttt{y}, \texttt{d}_7, 3)$.

## 3 Fine-Grained Data Dependency Rules

This section defines a set of high-level, declarative rules for specifying fine-grained data dependency patterns. Rules are expressed over actor signatures, and can be executed over traces to generate dependencies. We first describe the rule language, then describe a relational implementation of the abstract model in which Datalog queries are used to implement the patterns defined by the high-level dependency rules. We then demonstrate the dependency rules using commonly found types of dataflow actors.

**Table 1.** High-level rule language for specifying fine-grained dependencies of actors

| *Dependency Rule* | *Rule Definition* |
|---|---|
| $y$ `depends_on` $x$ `in` $a$ | If $u_1 = (a{:}i, x, d_1, t_1)$, $u_2 = (a{:}i, y, d_2, t_2)$, and $t_1 < t_2$, then assert $u_1 \xleftarrow{\texttt{ddep}} u_2$. |
| $y$ `derives_from` $x$ `in` $a$ | If $u_1 = (a{:}i, x, d_1, t_1)$, $u_2 = (a{:}i, y, d_2, t_2)$, and $t_1 < t_2$, then assert $u_1 \xleftarrow{\texttt{dder}} u_2$. |
| $y$ `derives_from_value` $x$ `in` $a$ | If $u_1 = (a{:}i, x, d_1, t_1)$, $u_2 = (a{:}i, y, d_2, t_2)$, $t_1 < t_2$, and $v(d_1) = v(d_2)$, then assert $u_2 \xleftarrow{\texttt{dval}} u_1$. |
| $y$ `derives_from_id` $x$ `in` $a$ | If $u_1 = (a{:}i, x, d_1, t_1)$, $u_2 = (a{:}i, y, d_2, t_2)$, $t_1 < t_2$, and $d_1 = d_2$, then assert $u_2 \xleftarrow{\texttt{did}} u_1$. |
| $y$ `depends_on_prev` $x$ `in` $a$ | If $u_1 = (a{:}i, x, d_1, t_1)$, $u_2 = (a{:}i, y, d_2, t_2)$, $t_1 < t_2$, and there does not exists a $u_3 = (a{:}i, x, d, t)$ such that $t_1 < t < t_2$, then assert $u_2 \xleftarrow{\texttt{did}} u_1$. |
| $y$ `derives_from_prev` $x$ `in` $a$ | If $u_1 = (a{:}i, x, d_1, t_1)$, $u_2 = (a{:}i, y, d_2, t_2)$, $t_1 < t_2$, and there does not exists a $u_3 = (a{:}i, x, d, t)$ such that $t_1 < t < t_2$, then assert $u_2 \xleftarrow{\texttt{dder}} u_1$. |
| $y$ `derives_from_value_prev` $x$ `in` $a$ | If $u_1 = (a{:}i, x, d_1, t_1)$, $u_2 = (a{:}i, y, d_2, t_2)$, $t_1 < t_2$, $v(d_1) = v(d_2)$, and there does not exists a $u_3 = (a{:}i, x, d, t)$ such that $t_1 < t < t_2$, then assert $u_2 \xleftarrow{\texttt{dval}} u_1$. |
| $y$ `derives_from_id_prev` $x$ `in` $a$ | If $u_1 = (a{:}i, x, d_1, t_1)$, $u_2 = (a{:}i, y, d_2, t_2)$, $t_1 < t_2$, $d_1 = d_2$, and there does not exists a $u_3 = (a{:}i, x, d, t)$ such that $t_1 < t < t_2$, then assert $u_2 \xleftarrow{\texttt{dval}} u_1$. |

### 3.1 Dependency Rule Language

The dependency rule language is based on the eight high-level patterns described in Table 1. Each dependency rule takes the form "$s$ $d$ $t$ `in` $a$", where $s$ is a source parameter, $d$ is a dependency type, $t$ is a target, and $a$ is an actor. Given a source $y$ and target $x$ for an actor $a$, a rule asserts dependencies from updates of parameters $a.y$ to updates of parameters $a.x$. We consider four basic types of dependencies, namely, `depends_on` which establishes a basic dependency, `derives_from` which establishes a derivation, `derives_from_value` which establishes a value-copy derivation, and `derives_from_id` which establishes an identifier-copy derivation. We also consider two dependency qualifiers. The default qualifier `all` states that each update of a parameter $a.y$ depended on every previous update of a parameter $a.x$ within an actor step. The first four dependency rules shown in Table 1 (implicitly) use the `all` qualifier. Alternatively, the `prev` qualifier states that only the most recent update of $a.x$ is a dependency of $a.y$ for a particular step. The last two rules in Table 1 use the `prev` qualifier.

### 3.2 Abstract Model and Dependency Rule Implementation

Here we briefly describe a relational implementation of the abstract model and an approach for applying dependency rules. In general, dependency rules would

be provided as part of the actor definitions of a workflow or possibly specified and refined by a workflow developer or end-user. As shown later in this section, each actor may have multiple dependency rules, in which case each rule is applied (i.e., the union of the rules is taken, instead of their intersection). Dependency rules are used to define a corresponding data-dependency *view* (or query) over a given workflow trace, and thus dependencies are decoupled from, i.e., not specified as part of, the trace itself.

Actor parameter specifications (signatures) are represented using the relation $\texttt{param}(x_a, x_p, x_t)$, which states that the parameter name $x_p$ is defined for the actor $x_a$ and has the type $x_t \in \{\texttt{in}, \texttt{out}, \texttt{state}\}$. Here we do not consider channels since only actor signatures are required for workflow specifications in the abstract model.

A workflow trace consists of parameter updates and value definitions. Parameter updates are represented using the relation $\texttt{update}(x_u, x_a, x_s, x_p, x_d, x_k, x_t)$, where $x_u$ is a unique update identifier, $x_a$ is the actor and $x_s$ is the actor invocation id (together denoting the step), $x_p$ is the parameter being updated, $x_d$ is a data item where $x_k$ is the item type such that $x_k \in \{\texttt{id}, \texttt{val}\}$, and $x_t$ is the relative update order (with respect to the step). Data values are represented using the relation $\texttt{value}(x_d, x_v)$, where $x_d$ is the data identifier and $x_v$ is the value. As an example, the updates for the `filter` actor in the trace of Figure 1b would be represented as the following facts, assuming $\texttt{d}_5$ and $\texttt{d}_7$ are both represented as the same value `v`.

$$\texttt{update}(7, \texttt{filter}, 1, \texttt{x}, \texttt{v}, \texttt{val}, 1),$$
$$\texttt{update}(8, \texttt{filter}, 1, \texttt{c}, \texttt{d}_6, \texttt{val}, 2),$$
$$\texttt{update}(9, \texttt{filter}, 1, \texttt{y}, \texttt{v}, \texttt{val}, 3).$$

We consider four separate relations for representing dependencies: $\texttt{ddep}(u_2, u_1)$, $\texttt{dder}(u_2, u_1)$, $\texttt{dval}(u_2, u_1)$ and $\texttt{did}(u_2, u_1)$, together with the following Datalog rules for capturing the subsumption hierarchy between the different dependency types.

$$\texttt{ddep}(u_2, u_1) :- \texttt{dder}(u_2, u_1).$$
$$\texttt{dder}(u_2, u_1) :- \texttt{dval}(u_2, u_1).$$
$$\texttt{dval}(u_2, u_1) :- \texttt{did}(u_2, u_1).$$

Each provenance rule is represented as a fact in the relation $\texttt{prov\_rule}(x_a, x_{p2}, x_{p1}, x_t)$, where $x_a$ denotes the actor, $x_{p2}$ specifies the target parameter, $x_{p1}$ specifies the source parameter, and $x_t$ is the type of the dependency. Given a set of `prov_rule` facts, the following Datalog rules define a program for inferring all explicitly defined dependencies of a trace. Dependencies are inferred for rules of the form "$y$ `depends_on` $x$ `in` $a$" using the Datalog query:

$$\texttt{ddep}(u_2, u_1) :- \texttt{prov\_rule}(a, y, x, \texttt{depends\_on}), \texttt{update}(u_1, a, s, x, d_1, k_1, t_1),$$
$$\texttt{update}(u_2, a, s, y, d_2, k_2, t_2), t_1 < t_2.$$

A similar query with `depends_on` replaced by `derives_from` and `ddep` replaced by `dder` is used for the case of "$y$ `derives_from` $x$ `in` $a$". For rules of the form "$y$ `derives_from_value` $x$ `in` $a$", we have three separate cases depending on the type of data item:

$$\begin{aligned}
\texttt{dval}(u_2, u_1) :- \; & \texttt{prov_rule}(a, y, x, \texttt{derives_from_value}),\\
& \texttt{update}(u_1, a, s, x, v, \texttt{val}, t_1), \texttt{update}(u_2, a, s, y, v, \texttt{val}, t_2), t_1 < t_2.\\
\texttt{dval}(u_2, u_1) :- \; & \texttt{prov_rule}(a, y, x, \texttt{derives_from_value}),\\
& \texttt{update}(u_1, a, s, x, d_1, \texttt{id}, t_1), \texttt{update}(u_2, a, s, y, d_2, \texttt{id}, t_2),\\
& \texttt{value}(d_1, v), \texttt{value}(d_2, v), t_1 < t_2.\\
\texttt{dval}(u_2, u_1) :- \; & \texttt{prov_rule}(a, y, x, \texttt{derives_from_value}),\\
& \texttt{update}(u_1, a, s, x, v, \texttt{val}, t_1), \texttt{update}(u_2, a, s, y, d, \texttt{id}, t_2),\\
& \texttt{value}(d, v), t_1 < t_2.\\
\texttt{dval}(u_2, u_1) :- \; & \texttt{prov_rule}(a, y, x, \texttt{derives_from_value}),\\
& \texttt{update}(u_1, a, s, x, d, \texttt{val}, t_1), \texttt{update}(u_2, a, s, y, v, \texttt{id}, t_2),\\
& \texttt{value}(d, v), t_1 < t_2.
\end{aligned}$$

For rules of the form "$y$ `derives_from_id` $x$ `in` $a$" we use the query:

$$\begin{aligned}
\texttt{did}(u_2, u_1) :- \; & \texttt{prov_rule}(a, y, x, \texttt{derives_from_id}),\\
& \texttt{update}(u_1, a, s, x, d, \texttt{id}, t_1), \texttt{update}(u_2, a, s, y, d, \texttt{id}, t_2), t_1 < t_2.
\end{aligned}$$

Finally, for rules of the form "$y$ `depends_on_prev` $x$ `in` $a$" we define the following two queries:

$$\begin{aligned}
\texttt{ddep}(u_2, u_1) :- \; & \texttt{prov_rule}(a, y, x, \texttt{depends_on_prev}),\\
& \texttt{update}(u_1, a, s, x, d_1, k_1, t_1), \texttt{update}(u_2, a, s, y, d_2, k_2, t_2),\\
& t_1 < t_2, \neg\texttt{after}(u_1, t_2).\\
\texttt{after}(u, t) :- \; & \texttt{update}(u, a, s, p, d, k, t_1), \texttt{update}(u_2, a, s, p, d_2, k_2, t_2),\\
& \texttt{update}(u_3, a, s, p_3, d_3, k_3, t), t_1 < t_2, t_2 < t.
\end{aligned}$$

where $\texttt{after}(u, t)$ states that an update occurred in the same step and on the same parameter after $u$ but before $t$. A similar query is used for rules of the form "$y$ `derives_from_prev` $x$ `in` $a$", again, where `depends_on` is replaced by `derives_from` and `ddep` is replaced by `dder`.

### 3.3 Dependency Rules for Common Actor Invocation Patterns

Here we provide examples of different types of actor dependency patterns found within scientific workflow systems (and in particular, those systems supporting dataflow models of computation [7] such as Kepler [10] and Taverna [14], among others). For each type of actor we give the corresponding rules for describing the data dependencies generated by each actor invocation. Our goal is to highlight

the benefits of our approach by showing that for these common types of patterns, the high-level rules both capture the dependency patterns and are easier (more concise) to specify than, e.g., the underlying queries implementing the rules.

**Transform and Filter.** The `normalize` actor in Figure 1 is an example of a basic transformer. The following provenance rules capture the dependencies for the `normalize` actor, in which each output `y` is derived from each corresponding input parameter `x`, `a`, and `b`.

```
y derives_from x in normalize,
y derives_from a in normalize,
y derives_from b in normalize.
```

Similarly the `filter` actor in Figure 1 is an example of a (low-pass) filter. The provenance rules for `filter` are

```
y derives_from_value x in filter,
y depends_on c in filter.
```

We note that the first rule could also be defined using `derives_from_id` if the actor implementation copies the input identifier to the output parameter (assuming this is also supported by the underlying workflow system).

As a simple example of the relational implementation, the above inference rules for the `filter` actor would result in the following two facts being asserted within the `prov_rule` relation.

```
prov_rule(filter, y, x, derives_from),
prov_rule(filter, y, c, depends_on).
```

Using the example updates of Figure 1b, the queries for `dder` and `ddep` given in the previous subsection together with the above rules would infer the following dependencies, where in update 7 parameter `x` was written to (as input to the invocation), in update 8 parameter `c` was written to (as input to the invocation), and in update 9 parameter `y` was written to (as output to the invocation such that `y` receives the value given to `x`, implying the value satisfied the conditional value in `c`).

```
dval(9, 7),
ddep(9, 8).
```

**Delay.** A typical `delay` actor consists of three parameters: an input `x`, a state parameter `s`, and an output `y`. The state parameter is set to a default value at the beginning of the initial invocation. At each invocation, the state value is copied to the output `y`, and the input value in `x` is then copied to the state parameter `s`. The current input is output on the *next* invocation (which, e.g., make approaches based on implicit dependency rules problematic). Delay actors are often used to initiate a loop. The dependency rules for a typical `delay` actor are as follows.

```
y derives_from_value s in delay,
s derives_from_value x in delay.
```

A similar pattern is to perform a transformation of `x` at each step. In this case, the second rule above would be changed from `derives_from_value` to `derives_from`.

**Sliding Window.** A sliding window performs an aggregate operation over an overlapping, fixed size number of input elements. For each window, an output is produced. For instance, consider the simple case of a sliding window actor `swp` that performs a product over a window size of two (see Figure 2). This actor has an input parameter `x`, a state parameter `s`, and an output parameter `y`. On each invocation, `s` contains the last element of the previous window, `x` is updated to the next value, `y` is then computed from `x` and `s`, and then `s` is updated to the new value of `x`. The dependency rules for `swp` are the following.

```
y derives_from x in swp,
y derives_from s in swp,
s derives_from x in swp.
```

Figure 2 shows an example trace and the dependencies inferred from the above rules.

**Monotonic Integer Stream Merge.** Often, two dataflow paths within a workflow must be merged, and various strategies have been developed for performing merge operations (e.g., depending on whether order must be preserved, only unique data items should be output, and so on). Here we consider a simple case of an order-preserving merge operation, which takes two input data streams represented by parameters `x` and `y`, and produces one output stream represented by parameter `z`. The data items arriving on each respective parameter `x` and `y` are assumed to be ordered. On the first invocation of the actor, data items are read into both `x` and `y`, with the smallest value being copied to output parameter `z` and the larger value copied to a state variable `s`. The actor also records the input parameter having the smallest value. On subsequent invocations, the parameter with the smallest previous value is read into, if this value is smaller than the current state parameter value, its value is copied to `z`, otherwise the state value is copied to `z` (and the next execution will read from the other input parameter). Assuming that data identifiers are copied between parameters by the merge actor, the dependency rules can be expressed as follows.

```
s derives_from_id x in merge,
s derives_from_id y in merge,
z derives_from_id x in merge,
z derives_from_id y in merge,
z derives_from_id s in merge,
z depends_on x in merge,
z depends_on y in merge,
z depends_on s in merge.
```

The `depends_on` rules state that the particular output depended on each parameter update (but was derived via a copy from only one of the parameters). If only data values are copied (as opposed to identifiers), the above rules can be modified to use `derives_from_value`, however, two parameters with the same value will result in multiple derivations (i.e., either the initial `x` and `y` or subsequent updates of `s` with `x` or `y`).

**List Transformer.** A list transformer is an instance of a standard `map` operation. In particular, given a sequence of tokens on an input parameter `x` a list transformer outputs corresponding values on an output parameter `y`. Consider the simple case of an `add1` actor, which adds one to each element of an input list, and outputs a list with the modified values. Thus, on a single invocation, `add1` reads multiple values from `x` and produces multiple values on `y`. However, each output value on `y` is dependent only on the most recently read data item on `x`. Thus, the dependency rule for `add1` is the following.

```
y derives_from_prev x in add1.
```

**List sum.** An invocation of the list sum actor computes the sum of a given list of data items. The actor can be implemented with an input parameter `x`, state parameter `s`, and output parameter `y`. At the start of an invocation, `s` is updated with the default value 0. The actor then reads a value on `x`, adds it to `s`, and stores the result back in `s`. When all values have been read, the latest value of `s` is output on `y`. Thus, each `s` value is derived from the previous `s` value, and the final output is a copy of the value on `s`. The dependency rules for list sum are the following.

```
s derives_from_prev s in sum,
s derives_from_prev x in sum,
y derives_from_value_prev s in sum.
```

List sum is an instance of a `fold` function, and the same rules can be used for list sum implemented via `scan`, i.e., with intermediate state values also output on `y`.

## 4 Related Work

While many workflow systems provide support for recording workflow trace information [5,6], systems that provide support for fine-grained data dependencies employ either implicit rules (e.g., [1,8,14,9]) or rely on actors to declare dependencies (e.g., [11]). Our approach allows for expressing explicit rules (high-level view definitions) that are independent of the underlying workflow system and layered over standard execution traces. In [12], explicit rules are also used for efficiently tracking the provenance of stream-based continuous queries. Three types of rules are defined: two for specifying sliding windows (via time intervals and window element size), and another based on data selection queries (e.g.,

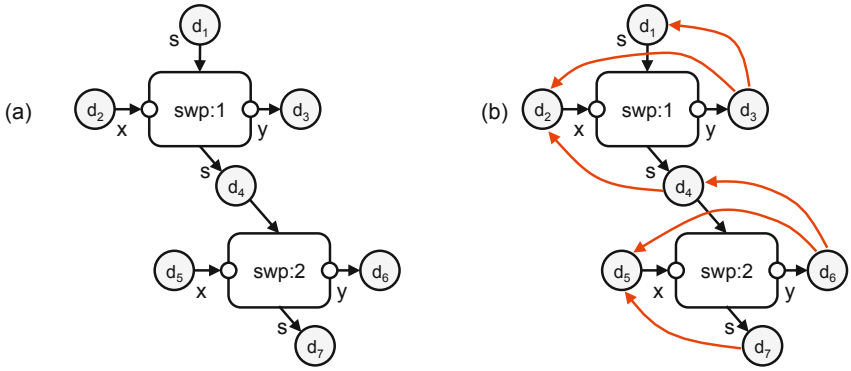


**Fig. 2.** Example sliding window product actor `swp` (with window size 2): (a) two invocations with data items $d_i$ for $i$ the update order, and (b) inferred dependencies

to assert that certain outputs depend on all inputs having specific attribute values). Our rules are more generic in that they do not rely on specific data values and cover a larger class of components than those designed for sliding window operators. Finally, fine-grained dependencies are automatically inferred from workflows composed of white-box components in [2] (based on the Pig Latin language). While we do not assume the presence of white-box actors, our rules could be used within such an approach to support cases where white-box and user-defined, black-box functions are used together.

## 5 Summary

This paper has presented an approach for addressing the problem of determining fine-grained data dependencies within scientific workflows by decoupling the specification of dependencies from the "observables" recorded within workflow execution traces. Our approach defines both an abstract model of observables and a high-level declarative rule language for specifying detailed dependency patterns. We also demonstrated how the abstract model and rule patterns can be implemented within a relational framework and provided examples of common dataflow actors expressed using our rule language. Inferring fine-grained dependencies from workflow execution traces is complementary to existing provenance standardization efforts such as OPM [15] and the W3C Prov model [18], which serve as general-purpose representing schemes for provenance information. Our basic model of provenance assumed and the dependencies generated from the inference rules presented here are compatible with both the OPM and Prov models. In this way, our framework could easily be used to produce detailed provenance information from workflow traces that conforms to the OPM and Prov representation schemes. Finally, both the model and the inference rule language described here are currently being implemented as part of the provenance framework supported by the RestFlow scientific workflow system [16].

**Acknowledgements.** This work supported in part through NSF grants IIS-1118088, DBI-0743429, DBI-0753144, DBI-0960535, and OCI-0722079.

## References

1. Altintas, I., Barney, O., Jaeger-Frank, E.: Provenance Collection Support in the Kepler Scientific Workflow System. In: Moreau, L., Foster, I. (eds.) IPAW 2006. LNCS, vol. 4145, pp. 118–132. Springer, Heidelberg (2006)
2. Amsterdamer, Y., Davidson, S.B., Deutch, D., Milo, T., Stoyanovich, J., Tannen, V.: Putting lipstick on pig: Enabling database-style workflow provenance. PVLDB 5(4) (2011)
3. Anand, M.K., Bowers, S., McPhillips, T.M., Ludäscher, B.: Efficient provenance storage over nested data collections. In: EDBT (2009)
4. Bowers, S., McPhillips, T., Riddle, S., Anand, M.K., Ludäscher, B.: Kepler/pPOD: Scientific Workflow and Provenance Support for Assembling the Tree of Life. In: Freire, J., Koop, D., Moreau, L. (eds.) IPAW 2008. LNCS, vol. 5272, pp. 70–77. Springer, Heidelberg (2008)
5. Davidson, S.B., Freire, J.: Provenance and scientific workflows: challenges and opportunities. In: SIGMOD (2008)
6. Gil, Y., et al.: Examining the challenges of scientific workflows. IEEE Computer 40(12), 24–32 (2007)
7. Lee, E., Parks, T.: Dataflow process networks. Proc. of the IEEE 83(5), 773–799 (1995)
8. Lim, C., Lu, S., Chebotko, A., Fotouhi, F.: Prospective and retrospective provenance collection in scientific workflow environments. In: IEEE SCC, pp. 449–456 (2010)
9. Ludäscher, B., Podhorszki, N., Altintas, I., Bowers, S., McPhillips, T.M.: From computation models to models of provenance: the rws approach. Concurrency and Computation: Practice and Experience 20(5), 507–518 (2008)
10. Ludäscher, B., et al.: Scientific workflow management and the Kepler system. Concurrency and Computation: Practice and Experience 18(10) (2006)
11. McPhillips, T., Bowers, S., Zinn, D., Ludäscher, B.: Scientific workflow design for mere mortals. Future Generation Computer Systems 25(5) (2009)
12. Misra, A., Blount, M., Kementsietsidis, A., Sow, D., Wang, M.: Advances and Challenges for Scalable Provenance in Stream Processing Systems. In: Freire, J., Koop, D., Moreau, L. (eds.) IPAW 2008. LNCS, vol. 5272, pp. 253–265. Springer, Heidelberg (2008)
13. Missier, P., Paton, N., Belhajjame, K.: Fine-grained and efficient lineage querying of collection-based workflow provenance. In: EDBT (2010)
14. Missier, P., Soiland-Reyes, S., Owen, S., Tan, W., Nenadic, A., Dunlop, I., Williams, A., Oinn, T., Goble, C.: Taverna, Reloaded. In: Gertz, M., Ludäscher, B. (eds.) SSDBM 2010. LNCS, vol. 6187, pp. 471–481. Springer, Heidelberg (2010)
15. Moreau, L., et al.: The open provenance model core specification (v1.1). Future Generation Computer Systems 27(6), 743–756 (2011)
16. RestFlow, `https://sites.google.com/site/restflowdocs/`
17. Simmhan, Y.L., et al.: A survey of data provenance in e-science. SIGMOD Record 34(3) (2005)
18. The W3C Provenance Working Group, `http://www.w3.org/2011/prov`