A Model for User-Oriented Data Provenance in Pipelined Scientific Workflows*

Shawn Bowers[1], Timothy McPhillips[1], Bertram Ludäscher[1,2],
Shirley Cohen[3], Susan B. Davidson[3]

[1]UC Davis Genome Center, University of California, Davis
[2]Department of Computer Science, University of California, Davis
[3]Computer and Information Science, University of Pennsylvania
{sbowers,ludaesch,tmcphillips}@ucdavis.edu
{shirleyc,susan}@cis.upenn.edu

Abstract. Integrated provenance support promises to be a chief advantage of scientific workflow systems over script-based alternatives. While it is often recognized that information gathered during scientific workflow execution can be used automatically to increase fault tolerance (via checkpointing) and to optimize performance (by reusing intermediate data products in future runs), it is perhaps more significant that provenance information may also be used by scientists to reproduce results from earlier runs, to explain unexpected results, and to prepare results for publication. Current workflow systems offer little or no direct support for these "scientist-oriented" queries of provenance information. Indeed the use of advanced execution models in scientific workflows (*e.g.*, process networks, which exhibit pipeline parallelism over streaming data) and failure to record certain fundamental events such as *state resets* of processes, can render existing provenance schemas useless for scientific applications of provenance. We develop a simple provenance model that is capable of supporting a wide range of scientific use cases even for complex models of computation such as process networks. Our approach reduces these use cases to database queries over event logs, and is capable of reconstructing complete data and invocation dependency graphs for a workflow run.

1 Introduction

The importance of provenance information in scientific data and workflow management is widely recognized, as witnessed, *e.g.*, by specialized workshops [4,1], research projects [17], and surveys [3,20] dedicated to this topic, and by investigations on foundations of data provenance for queries and transformations [5,9,23]. However, current scientific workflow systems still offer little or no support for queries of interest to the end-users of these systems, *e.g.*, researchers in

* Work supported in part by SciDAC/SDM (DE-FC02-01ER25486), NSF/SEEK (DBI-0533368), and NSF/GEON (EAR-0225673)

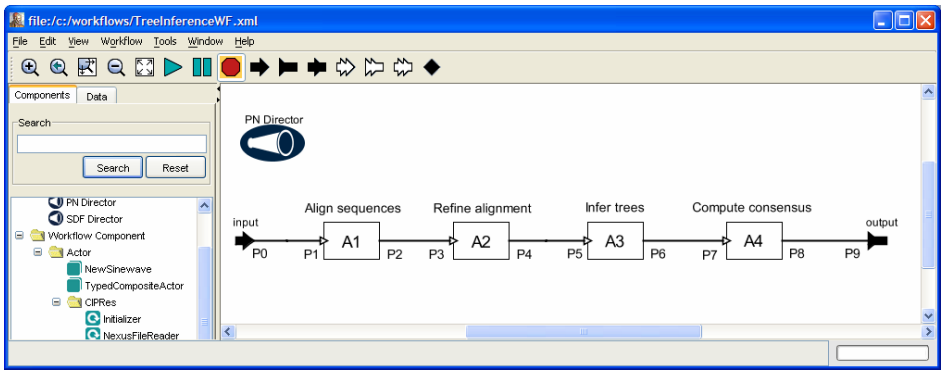

Fig. 1. A workflow for computing phylogenetic trees from input DNA sequences.

the life or physical sciences. In this paper, we argue that concrete use cases, expressed in terms that are meaningful to the scientist, should drive the design of a provenance system. Moreover, such systems should be designed in terms of the models of computation (MoC) that govern the execution of scientific workflows to ensure that all pertinent events are recorded in the execution log.

Fig. 1 shows an example workflow for inferring phylogenetic trees approximating the evolutionary relationships between organisms. DNA sequences for homologous genes from a number of taxa are provided as input to the workflow. Actor A1 performs an initial alignment of the sequences (*e.g.*, using the program `ClustalW` [22]), and actor A2 refines this initial alignment (*e.g.*, using `Gblocks` [7]).[1] Actor A3 infers a set of phylogenetic trees from the aligned sequences (*e.g.*, using `DNAPARS` [19]), and actor A4 computes the consensus of these trees (*e.g.*, using `CONSENSE` [19]).

For such scientific workflows we would like to: (a) enable scientists to ask "scientific" questions about a workflow run by providing convenient queries against the run's execution log; and (b) have the system track the true data dependencies within a run so that answers to such scientific questions may be as accurate as possible. For example, the system should recognize *independent* "sub-runs" as such: The workflow in Fig. 1 may process multiple sets $s_1, s_2, \ldots$ of independent DNA sequences (*e.g.*, corresponding to distinct genes in the taxa of interest) within a single workflow run R. In such cases, the system should *not* infer that the data products resulting from the different s_i are interdependent. Rather, the system should answer accurately questions such as:

- *Which phylogenetic trees were used to produce this consensus tree?*
- *Which DNA sequences does this consensus tree depend on?*
- *Which of the input DNA sequences were* ***not*** *used to derive any output consensus tree?*

[1] Following Kepler terminology, we call workflow components *actors*.

In this paper, we develop a provenance model designed to support such user-oriented queries for pipelined models of computation where tracking data dependencies can be complex. For another example, consider an actor A in an environmental monitoring workflow that computes a running average of temperature for each received measurement data token. Thus, upon each invocation or *firing* of A, the actor consumes a temperature token and emits a new running average token. To calculate the running average over multiple firings, A must maintain *state*. For the provenance system this means that every produced data token must be recorded as dependent not just on those input tokens received since the last time the actor fired, but on all tokens received since A was initialized. Conversely, if A is to limit the running averages to readings taken on a particular day, then A's state is *reset* once per day. There are *no* dependencies between tokens produced after a reset and tokens consumed prior to the reset. This observation naturally partitions token streams, as well as actor firings, into semantically meaningful firing *rounds*.[2] Clearly, a provenance system should be able to observe and record new rounds of firing to avoid reporting false dependencies.

The running average example described above illustrates a general property of scientific workflows implemented as process networks [12,15,13]: actors need not produce output tokens derived exclusively from tokens received since the last output token was produced. That is, actors in process networks do not generally compute functions on sets of consecutively received inputs. Rather, they may carry out arbitrarily complex transactions on streams of inputs, including running averages, filters, sliding windows, and iterative computations.

Capturing these transaction boundaries is essential for accurately recording scientific workflow provenance. In this paper, we show how this essential information can be represented in a simple tabular event log. Our approach is easy to implement, *e.g.*, in the Kepler scientific workflow system, where token-read and token-write events can be automatically captured by the workflow framework. Announcing a new round of firing (*e.g.*, by signaling a reset event), on the other hand, is performed by actors themselves, which "know" when they are beginning an independent task (such as a "sub-run" s_i above, or new daily average temperatures).

The rest of this paper is organized as follows. Section 2 briefly overviews scientific workflows within Kepler, focussing on pipelined execution models. Section 3 presents our provenance model, which consists of read, write, and state reset events. We also describe in Section 3 how to compute data and actor dependency graphs (*e.g.*, for computing data lineage) from corresponding event logs. Section 4 describes a set of operations (or views) over the provenance model for supporting "scientist-oriented" provenance queries. A number of examples are given, which define parameterized queries for the workflow of Figure 1. Finally, Section 5 summarizes our contributions and future work.

[2] A round is somewhat analogous to a database transaction, specifically in that it constitutes a logical unit of work.

2 Preliminaries

2.1 Workflow Graphs, Actors, and Tokens vs. Data Objects

We adopt notions and terminology from KEPLER, a scientific workflow system extending PTOLEMY II. Workflows are composed by placing *actors* on a design canvas, and "wiring" them together to form the desired workflow graph (Fig. 1). Actors communicate through their input and output *ports*. In a *workflow graph* W, output ports can be connected to input ports, establishing unidirectional dataflow *channels*. Actors communicate through these channels by passing *tokens*.

By default tokens are immutable and "disposable", *i.e.*, every token t is written only once [15] and thus lives only between its creation on an output port, and its consumption at subsequent input ports. Thus, even if an actor passes on a data object unchanged, a new token-id is created, facilitating tracking of token dependencies. A separate object-id is used to track object dependencies. By $\mathit{object}(t)$ we denote the data *object* represented by the token t. To support user-oriented queries, we associate with an object o one or more types $\mathit{types}(o)$.

The *ports* of an actor A are denoted $\mathit{ports}(A)$. We assume that port-ids are globally unique, *i.e.*, they include a unique actor-occurrence-id and a port-name which is unique to the actor occurrence. A port is either an *input* or *output*, so $\mathit{ports}(A) = \mathit{in}(A) \dot{\cup} \mathit{out}(A)$. Some input ports $\mathit{pars}(A) \subseteq \mathit{in}(A)$ may be distinguished as *parameters* for configuring A's behavior. The signature $\Sigma_W := \mathit{in}(W) \rightarrow \mathit{out}(W)$ of a workflow W is given by a set of distinguished inputs $\mathit{in}(W)$ and outputs $\mathit{out}(W)$. As shown in Figure 1, the distinguished workflow input and output ports are connected to a subset of the input and output ports of the workflow's actors.

2.2 Directors

The model of computation (MoC) of a workflow is *not* defined by actors, but specified by a separate component called a *director*. Thus, KEPLER allows workflow designers to choose among different MoCs by choosing appropriate directors. A director specifies and (effectively) mediates all inter-actor communication, separating workflow scheduling and runtime orchestration (a director's concern) from individual actor execution (an actor's concern). This separation achieves a form of *behavioral polymorphism* [14], resulting in more reusable actor components. KEPLER provides a variety of directors that implement process network (PN and SDF), discrete event (DE), continuous time (CT), and finite state transducer (FST) semantics.

2.3 Pipelined Execution

In the process network MoC, the PN director executes each actor as a separate process (or thread). Channels are used to send and to buffer token streams between actors. Each actor can decide independently how many tokens to consume

before writing out a number of output tokens. In this way, workflows that run using the PN director not only exhibit task parallelism, but also *pipeline parallelism*. For example, during a single workflow run, each actor in Figure 1 can execute multiple times, and different actors can execute concurrently.

A number of other MoCs can be considered as special cases of the basic process network model [12,15]. In the synchronous dataflow (SDF) model [13], actors *a priori* define fixed token consumption and production rates. This model allows the SDF director to statically schedule actors, while guaranteeing, *e.g.*, that (unlike in the general PN case) deadlocks cannot occur and that buffers have a fixed size. By DAG (directed acyclic graph) we denote a MoC that is common in job-centric grid workflows [21,10]: nodes represent jobs, and directed edges represent execution dependencies between jobs. Thus, a DAG director can simply execute the jobs in the partial order implied by the job dependency graph. This can be seen as a limited special case of SDF, with an acyclic workflow graph, actors having at most one input and one output port, consuming and producing a single token per workflow run, respectively, and in which each actor is invoked exactly once (unlike in the more general SDF or PN cases).

3 A Provenance Model for Pipelined Workflows

In this section we describe a provenance model that can handle the process network (PN) model of computation, and thus specialized versions such as SDF and DAG as well. To execute a *workflow* (graph) $\mathcal{W}$, we must "bind" (*i.e.*, select) *input data* $\mathbf{i}$ on which $\mathcal{W}$ will operate. Often $\mathcal{W}$ is also parameterized using initial parameter settings $\mathbf{p}$. It is customary to record identifiers for $\mathcal{W}$, $\mathbf{p}$, and $\mathbf{i}$ as part of the provenance information. Finally, a MoC M is needed (*e.g.*, PN, SDF, DAG) to determine how the workflow is executed.[3] Taken together, the equation

$$\mathbf{o} = M(\mathcal{W}_{\mathbf{p}}(\mathbf{i}))$$

denotes a workflow execution in which the output $\mathbf{o}$ is obtained by applying a suitable model of computation M to an appropriately instantiated workflow $\mathcal{W}$.

3.1 Runs, Traces, and Observables

Each MoC M formally defines the notion of legal *computations* or *runs*, such that one can determine whether a particular run R of a workflow $\mathcal{W}$ is a legal representation (*w.r.t.* M) of an execution $\mathbf{o} = M(\mathcal{W}_{\mathbf{p}}(\mathbf{i}))$. A workflow *trace* $\mathcal{T}$ is an approximation of a run R, according to a *model of provenance*. As recorded by a provenance model, a trace approximates a run by recording functional and non-functional *observables*. For example, an SDF director precomputes a static workflow schedule (based on actor consumption and production rates), and using this schedule signals each actor to fire in turn. Thus, actor firings are

[3] Some MoCs might also be aware of resources such as cluster (or grid) nodes and transport protocols, and schedule a distributed workflow accordingly.

directly observed in SDF. In contrast, the *size* of a token (or rather the object it represents) and the *timestamp* when the token was created are *non-functional* observables: according to the MoC, the outcome does not depend on these. Non-functional observables can be useful to record, *e.g.*, to benchmark actor execution times or data transfer times between actors, but are not essential for determining data dependencies.[4]

3.2 The Read, Write, State-Reset (RWS) Provenance Model

Here we consider a concrete model of provenance, called the RWS model, which records ***r**ead*, ***w**rite*, and ***s**tate-reset* events for each actor in a workflow run. These events are stored in a relational *event log*. This model focuses on only a minimal set of observables that allow us to answer many science-oriented user questions (see next section), while ignoring non-functional observables such as timestamps, although such information can be easily added. Figure 2 is an example of an event log for a run of the workflow given in Figure 1. During a workflow run, a read event is added to the event log each time an actor reads a token from a port. Similarly, a write event is added to the log each time an actor writes a token to a port. A series of reads followed by writes denotes an actor *firing*. Note that in a particular firing F_j, an actor may use data that it read in a previous firing F_i to generate output (*e.g.*, this is the typical behavior of a running-average actor, as described in Section 1). In this case, we say the actor maintains state across firings, and state-reset events denote when the state is "flushed" (reset). The firings between reset events constitute a firing round.[5]

As shown in Figure 2, each row in an event log contains: the *location* E_{loc} of the event, which is either a port (for read and write events) or an actor (for state-reset events); the *event type* E_{typ}, which is either 'r' for read events, 'w' for write events, or 's' for state-reset events; the *token identifier* E_{tok} that was read or written at the port (null for state-reset events); and a *firing count* E_{fire}.

Because actor port identifiers are unique across a workflow, and tokens are written once, the port and token identifiers recorded for each read and write event enable the reconstruction of the flow of data through the workflow run. However, these events alone are not sufficient to reconstruct data dependencies. We use the state-reset events (as described above) along with the firing count for this purpose. In particular, the firing count is incremented independently for each actor whenever (1) an actor switches from writing tokens to reading tokens, denoting a new firing of the actor, and (2) whenever a state-reset event occurs.

[4] In existing systems, such timestamps are often the only information available and thus are also used to second-guess other properties such as token and object dependencies.

[5] We use a single state-reset event as opposed to separate events for marking the start and end of a transaction.

E_{loc}	E_{typ}	E_{tok}	E_{fire}
p_0	w	t_1	1
...			
p_0	w	t_{18}	1
A_1	s	–	1
p_1	r	t_1	1
...			
p_1	r	t_7	1
p_2	w	t_{19}	1
A_1	s	–	2
p_1	r	t_8	2
...			
p_1	r	t_{16}	2
p_2	w	t_{20}	2

E_{loc}	E_{typ}	E_{tok}	E_{fire}
A_1	s	–	3
p_1	r	t_{17}	3
p_1	r	t_{18}	3
p_2	w	t_{21}	3
A_1	s	–	4
A_2	s	–	1
p_3	r	t_{19}	1
p_4	w	t_{22}	1
A_2	s	–	2
p_3	r	t_{20}	2
p_4	w	t_{23}	2
A_2	s	–	3
p_3	r	t_{21}	3

E_{loc}	E_{typ}	E_{tok}	E_{fire}
A_2	s	–	4
A_3	s	–	1
p_5	r	t_{22}	1
p_6	w	t_{24}	1
p_6	w	t_{25}	1
p_6	w	t_{26}	1
A_3	s	–	2
p_5	r	t_{23}	2
p_6	w	t_{27}	2
p_6	w	t_{28}	2
A_3	s	–	3
A_4	s	–	1
p_7	r	t_{24}	1

E_{loc}	E_{typ}	E_{tok}	E_{fire}
p_7	r	t_{25}	1
p_7	r	t_{26}	1
p_8	w	t_{29}	1
A_4	s	–	2
p_7	r	t_{27}	2
p_7	r	t_{28}	2
p_8	w	t_{30}	2
A_4	s	–	3
p_9	r	t_{29}	1
p_9	r	t_{30}	1

Fig. 2. The event log for a run of the example workflow in Fig. 1.

3.3 Complex Workflow Transactions

In the example event log of Figure 2, state-reset events denote "sub-runs", *i.e.*, independent actor firings operating on sets of associated data. Note that for the particular event log shown, state reset events occur exactly at read/write transitions (*i.e.*, after write events immediately followed by read events).[6] However, for more complex workflows and actors, read/write transitions alone will not determine state-reset events, and more complex event patterns will be required to accurately describe data dependencies. Figure 3 gives four cases in which read/write transitions do not imply actor transaction boundaries, thus requiring more complex uses of state-reset events.

Figure 3(a) shows an actor A_1 that computes a sequence of running temperature averages (ta_n) from a series of input temperature readings (t_m), along with a corresponding event log for an example run. Each average reading is dependent on all temperature readings received since the most recent state-reset of the actor (*e.g.*, at midnight each night). In the example event log, token ta_{24} depends on tokens t_1–t_{24}, while ta_{25} is dependent only on t_{25}. Note that assuming that an implicit state-reset follows each write event would be incorrect, because this would imply that each temperature average depended only on the latest temperature reading received, rather than all temperature readings received so far during a particular round of firings.

Figure 3(b) illustrates the necessity of recording state-reset events for a filtering actor. In this example, a series of protein structures are input to actor A_2, and only those structures meeting a minimum resolution requirement are output (though carried by new tokens). All other input protein structures are discarded by the actor. Thus, in the example event log, of the first six structures received

[6] Note that state-reset events are still necessary in this example to mark the beginning and end of the actor firing/round.

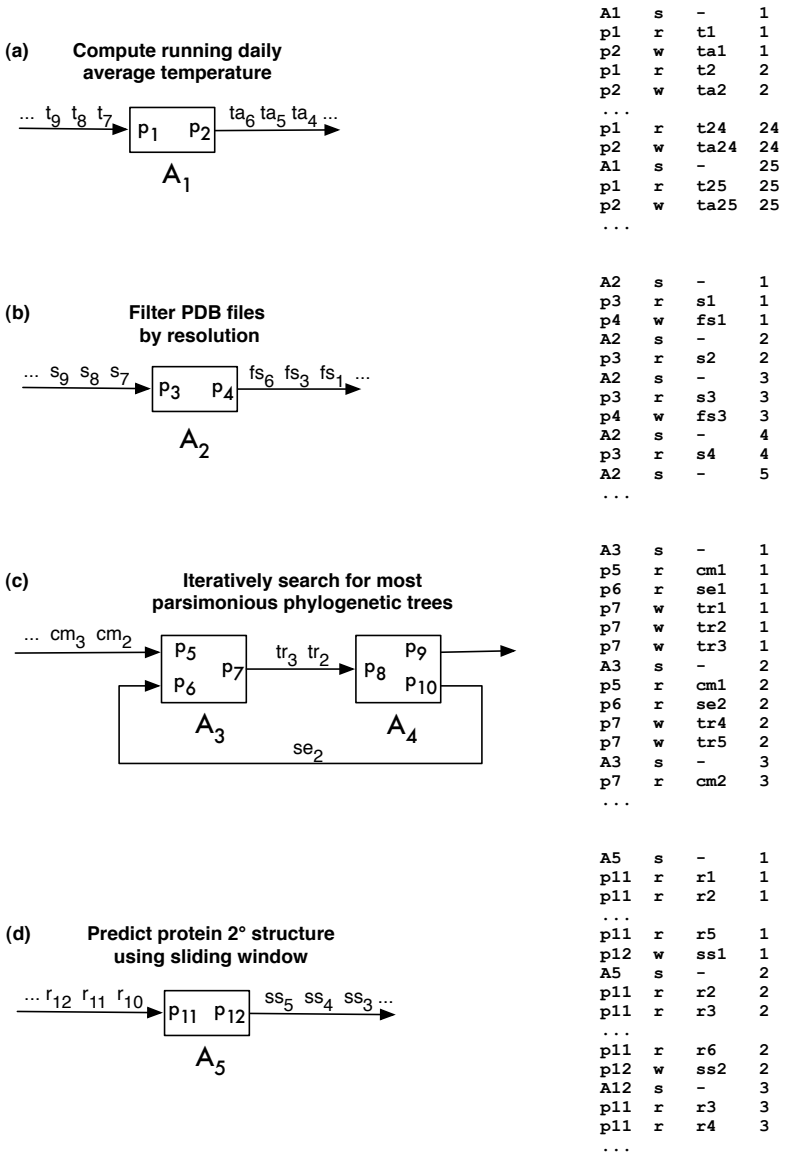

Fig. 3. Four distinct types of actors requiring complex state-reset event behavior.

by A_2, only three are output. Because the state-reset events are recorded, however, it is clear, *e.g.*, that output token fs_3 depends only on input token s_3, and not on s_2, even though no write event separates the read events for s_2 and s_3.

Figure 3(c) illustrates the more general case where an actor reuses only some of the data received during a previous firing. In this example, the tree inference

actor A_3 requires a random number seed to initiate a search for maximally parsimonious phylogenetic trees. Since any particular firing of the actor is not guaranteed to find all of the most parsimonious trees, the actor must be fired iteratively for a particular matrix of phylogenetically informative characters, using a distinct seed on each iteration. Actor A_4 collects the trees inferred by A_3 and provides the seeds needed by A_3 until a sufficient number of trees have been inferred. The RWS model allows each tree inferred in this way to be associated not only with the character matrix from which it was derived, but also with the particular random number seed used by A_3 to discover the tree. The sample event log illustrates how this works. Actor A_3 raises an 's' event prior to receiving each seed, and on receiving that seed declares that it re-reads the character matrix used previously along with the new seed. Thus, it is clear that while trees tr_1–tr_5 all depend on character matrix cm_1, only tr_4 and tr_5 were derived using seed se_2.

Finally, Figure 3(d) illustrates the requirements for recording the provenance of an actor operating on a sliding window of data. Actor A_5 predicts the secondary structure of a protein, residue by residue, based on the types of residues (*i.e.*, amino acids) within a contiguous segment of the protein chain. In this case the RWS model allows the actor to raise an 's' event after writing each output token. The actor then re-reads all tokens except the first token in the current window, along with the next token available on the input, before computing its next output.[7]

3.4 Dependency Graphs

Using the RWS model, we are able to infer from the event log the *token dependency graph*. That is, for each token t, we can know which parent tokens $\{t_1, \ldots, t_k\}$ directly contributed to the production of t (as the result of an actor firing). As an example, in the upper left of Fig. 4, $\{t_1, \ldots, t_7\}$ are parent tokens of t_{19}. Conversely, t_{22} is the parent of t_{24}, t_{25}, t_{26}. The following Datalog program illustrates how the token dependency graph can be computed from the event log. The *event* relation corresponds to the event log and the *actor* relation contains a mapping from ports to their corresponding actors.

$$\begin{aligned}
\text{depends-on}(T_1, T_2) \texttt{ :- } & \text{event}(P_1, w, T_1, C_1),\ \text{event}(P_2, r, T_2, C_2),\\
& \text{actor}(P_1, A),\ \text{actor}(P_2, A),\ \text{reset}(A, C_b, C_e),\\
& C_b \leq C_2 \leq C_1 < C_e.\\
\text{reset}(A, C_b, C_e) \texttt{ :- } & \text{event}(A, s, _, C_b),\ \text{event}(A, s, _, C_e),\ C_b < C_e,\\
& \neg\ \text{reset-between}(A, C_b, C_e).\\
\text{reset-between}(A, C_b, C_e) \texttt{ :- } & \text{event}(_, _, _, C_b),\ \text{event}(_, _, _, C_e),\\
& \text{event}(A, s, X, C),\ C_b < C < C_e.
\end{aligned}$$

We say that T_1 *depends on* T_2 whenever depends-on(T_1, T_2) is true.

In addition to the token dependency graph, we are also able to infer the *object dependency graph* using the RWS model. Object dependencies describe user data

[7] The RWS model could be optimized for cases where actors forget only a small fraction of previously read tokens during each firing by introducing an explicit 'forget' event.

lineage, and are crucial for our "user-oriented" queries. For example, the middle column of Fig. 4 shows the object dependencies for the workflow run of Fig. 1. Note that the object dependency graph differs slightly from the token dependency graph. Object dependency graphs can be computed from corresponding token dependency graphs and token-object mappings.

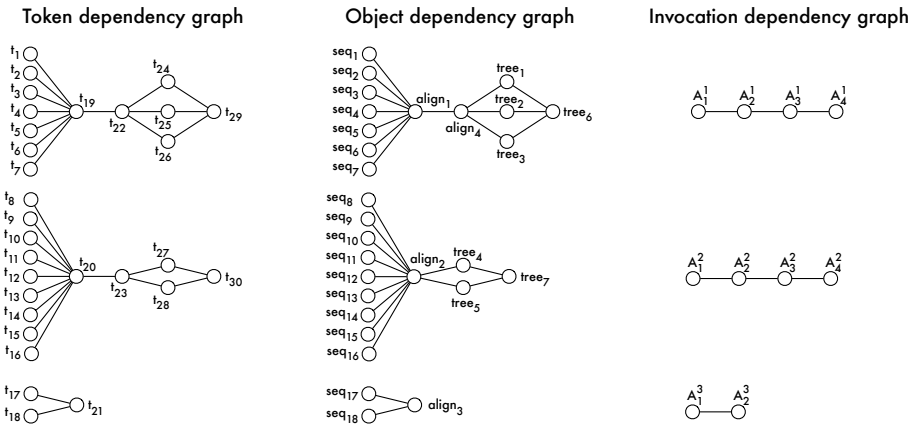

Fig. 4. Token, object, and actor invocation graphs for our example phylogenetics workflow. Dependencies are shown from left to right. Note that all but one of the token-object mappings can be inferred from the graph structures; tokens $\mathtt{t}_{20}$ and $\mathtt{t}_{23}$ both map to the object align_2.

Finally, *actor-invocation dependency graphs* can also be inferred directly from event logs in the RWS model. In particular, this graph can be built from state-reset events in the event log such that an actor invocation A_2^j depends on another actor invocation A_1^i whenever A_2^j reads a token that is written by A_1^i. Note that here, "invocation" refers to a firing round. It should be clear that all of the information stored in the event table is required to reconstruct these token, object, and invocation dependency graphs for a workflow trace. In particular, if state-reset events are not taken into account, each token written by an actor will (incorrectly) *appear* to depend on *all* previous tokens read during prior firing rounds of the actor: *e.g.*, in the absence of state-reset events, $\mathtt{t}_{21}$ would be connected to tokens $\mathtt{t}_1$ to $\mathtt{t}_{18}$ in the token dependency graph of Fig. 4.

4 Querying Workflow Traces

A wide range of scientifically relevant questions can be answered using the provenance model described above. To make access to event logs more convenient, we introduce the following primitive operations, which can be implemented, *e.g.*, as relational selections over the event log. The *writer*(t) and *reader*(t) operations

return the ports that a token t was written to and read from, respectively (a token is written to a port exactly once, but can be read multiple times). The *token-parents*(t) and *token-children*(t) operations return the set of direct token dependencies for a token t, while *token-ancestors*(t) and *token-descendents*(t) are their transitive closures. The *siblings*(t) operation returns the tokens with the same direct dependencies as t; *e.g.*, because actor A3 can infer multiple trees from an alignment, given one of these trees, *siblings* returns the other trees computed from the same alignment. The *origin*(o) and *death*(o) operations return the first and last tokens in the trace that refer to the object o; *e.g.*, the origin and death operations can be used to determine that the alignment object align_2 originated with token t_{20} (written by actor A1) and terminated with token t_{23} (written by actor A2).

The following examples illustrate how the provenance operations can be combined to answer concrete questions of interest to a scientist using the workflow in Figure 1. For each high-level question below, we define a corresponding parameterized query using set-comprehension syntax[8], along with the actual results for the event log given in Figure 2. Below, we use $\mathcal{W}$ to denote the workflow graph (in Figure 1) and $\mathcal{T}$ for the corresponding trace.

- **What DNA sequences were input to the workflow?** This is one of the first questions a scientist might ask about the workflow run. Given an object type \$c, the paramaterized query

$$q_1(\$c) := \{o \mid t \in \mathit{tokens}(\mathcal{T}) \wedge \mathit{writer}(t) \in \mathit{in}(\mathcal{W}) \wedge \mathit{object}(t) = o \wedge \$c \in \mathit{types}(o)\},$$

returns the set of objects of type \$c that were input to the workflow run. For our example trace, $q_1(\textsc{Sequence})$ returns the objects seq_1 to seq_{18}. The expression $t \in \mathit{tokens}(\mathcal{T})$ selects a token from the trace, the expression $\mathit{writer}(t) \in \mathit{in}(\mathcal{W})$ checks that the token was written by an input port of the workflow $\mathcal{W}$, the expression $\mathit{object}(t) = o$ obtains the object associated with t, and the expression $\$c \in \mathit{types}(o)$ verifies that o has \$c as a type.

- **What phylogenetic trees were output by the workflow?** This is another basic question that a scientist might initially ask after a run. Given the query

$$q_2(\$c) := \{o \mid t \in \mathit{tokens}(\mathcal{T}) \wedge \mathit{reader}(t) \in \mathit{out}(\mathcal{W}) \wedge \mathit{object}(t) = o \wedge \$c \in \mathit{types}(o)\},$$

the expression $q_2(\textsc{Tree})$ returns the objects tree_6 and tree_7.

- **What phylogenetic trees (intermediate or final) were created by the workflow?** This question requests both intermediate as well as final data products of a run. Given the query

$$q_3(\$c) := \{o \mid t \in \mathit{tokens}(\mathcal{T}) \wedge \mathit{writer}(t) \notin \mathit{in}(\mathcal{W}) \wedge \mathit{object}(t) = o \wedge \$c \in \mathit{types}(o)\},$$

[8] Queries could also be defined in Datalog or in query languages for graphs or semistructured data.

the expression $q_3(\textsc{Tree})$ returns all tree objects of Figure 4. Note that the expression $writer(t) \notin in(W)$ ensures that the returned trees were not given as input to the workflow.

- **What actor created this phylogenetic tree?** The following query returns the actors that first wrote the given object $o:

$$q_4(\$o) := \{a \mid t \in origin(\$o) \wedge actor(writer(t)) = a\}.$$

The query returns A_3 for tree_1 to tree_5, and A_4 for tree_6 and tree_7. This question is of particular interest for workflows that employ multiple approaches for inferring phylogenetic trees.

- **Which phylogenetic trees were directly used to compute this consensus tree?** This question (*i.e.*, what is this tree the "consensus" of?) asks for the intermediate data products supplied to the actor producing a particular workflow output. Given the query

$$q_5(\$c, \$o) := \{o' \mid t \in origin(\$o) \wedge t' \in token\text{-}parents(t) \wedge object(t) = o' \wedge \$c \in types(o')\},$$

the expression $q_5(\textsc{Tree}, \mathsf{tree}_6)$ returns tree_1 to tree_3; and $q_5(\textsc{Tree}, \mathsf{tree}_7)$ returns tree_4 to tree_5.

- **What sequences input to the workflow does this consensus tree depend on?** This question illustrates how a workflow output can be related to the particular workflow inputs from which it was derived. Given the query

$$q_6(\$c, \$o) := \{o' \mid t \in origin(\$o) \wedge t' \in token\text{-}ancestors(t) \wedge writer(t') \in in(\mathcal{W}) \wedge object(t') = o' \wedge \$c \in types(o')\},$$

the expression $q_6(\textsc{Sequence}, \mathsf{tree}_6)$ returns seq_1 to seq_7, and the expression $q_6(\textsc{Sequence}, \mathsf{tree}_7)$ returns seq_8 to seq_{16}.

- **Which input sequences were not used to derive any output consensus trees?** Here we are interested in whether there are any workflow inputs without corresponding workflow outputs. Such inputs may be considered the workflow equivalent of "phantom lineages" [23]. Given an input type $\$c_{in}$ and output type $\$c_{out}$, the query

$$q_7(\$c_{in}, \$c_{out}) := \{o \mid t \in tokens(\mathcal{T}) \wedge writer(t) \in in(\mathcal{W}) \wedge object(t) = o \wedge \$c_{in} \in types(o) \wedge \{t' \mid t' \in token\text{-}descendents(t) \wedge reader(t') \in out(\mathcal{W}) \wedge c_{out} \in types(object(t'))\} = \emptyset\},$$

returns the objects input to the workflow that do not produce any workflow outputs; *e.g.*, the expression $q_7(\textsc{Sequence}, \textsc{Tree})$ returns the sequences seq_{17} and seq_{18}. The query first finds workflow input tokens t that refer to objects of type $\$c_{in}$, and then checks (via a subquery) to make sure that t has no output tokens with objects of the type $\$c_{out}$.

- **What was the sequence alignment used in the process of inferring this tree?** This question requests the key intermediate data object used in

producing a workflow result. A researcher may wish to examine the alignment to assess the reliability of the results, or reuse the alignment in another workflow. Given the query

$$q_8(\$c, \$o) := \{o' \mid t \in \mathit{origin}(\$o) \wedge t' \in \mathit{token\text{-}ancestors}(t) \wedge \mathit{object}(t') = o' \wedge \\ \$c \in \mathit{types}(o') \wedge \{t'' \mid t'' \in \mathit{token\text{-}descendents}(t') \wedge \\ \$c \in \mathit{types}(\mathit{object}(t'))\} = \emptyset\},$$

the expression $q_8(\textsc{Alignment}, \mathsf{tree}_6)$ returns the sequence alignment align_4, and $q_8(\textsc{Alignment}, \mathsf{tree}_7)$ returns the sequence alignment align_2. The subquery above ensures that the object o' is the alignment directly used to infer the tree.

- **What actors were involved in creating this tree?** This question may be used, *e.g.*, when writing the methods section of a publication to cite the employed methods and implementations. Given the query

$$q_9(\$o) := \{a \mid t \in \mathit{origin}(\$o) \wedge \mathit{actor}(\mathit{writer}(t)) = a\} \cup \\ \{a \mid t \in \mathit{origin}(\$o) \wedge t' \in \mathit{token\text{-}ancestors}(t) \wedge \mathit{actor}(\mathit{writer}(t')) = a\},$$

the expression $q_9(\mathsf{tree}_6)$ returns actors A1 to A4.

- **Which actors did not produce any output for input derived from this input sequence?** This question provides an explanation for the phantom lineages revealed by q_7 above:the query

$$q_{10}(\$o) := \{a \mid t \in \mathit{origin}(o) \wedge t' \in \mathit{token\text{-}descendents}(t) \wedge \mathit{token\text{-}children}(t') = \emptyset \\ \wedge\ \mathit{actor}(\mathit{reader}(t')) = a\},$$

The expressions $q_{10}(\mathsf{seq}_{17})$ and $q_{10}(\mathsf{seq}_{17})$ both return actor A2, indicating that this actor did not forward a refined sequence alignment of these two sequences to actor A3. This result is reasonable since no informative phylogenetic trees may be inferred from only two taxa.

5 Conclusion

Tracking provenance is an important aspect of scientific workflow systems. In this paper, we have focused primarily on the problem of tracking data lineage within scientific workflow runs, for the purpose of providing an accurate provenance record for answering "scientific" (*i.e.*, user-oriented) provenance queries.

The problem of data lineage has been widely studied in the database community [5,8,2,23]. However, the primary focus has been on transformations of data items expressed as database queries. As noted in [11], current provenance approaches for workflow systems (*e.g.*, [24,25,18]) record various kinds of metadata related to provenance. Despite these developments, however, little support exists in current systems to allow end-users to query provenance information in *scientifically meaningful* ways, in particular when advanced workflow execution models go beyond simple DAGs (as in process networks).

We have shown that a simple provenance model, based on read, write, and state-reset events, is expressive enough to capture many relevant science-oriented provenance use cases. These use cases become queries against suitable views on top of the event log. Our approach also marks the beginnings of a use-case and computation-model driven approach to provenance schema design. Using our framework, it is now meaningful to ask whether a provenance schema can handle specific use cases, since the latter become queries over the former.

As future work we intend to extend our approach to support a wider array of operations, *e.g.*, so-called "smart re-runs" (a workflow system requirement in [16])[9] and crash recovery, and to extend our current Prolog-based prototype to provide direct support (including query user interfaces) for our provenance model within Kepler. We are also developing methods to optimize our approach to reduce the size of event logs for actors whose behaviors are similar to sliding window operators (*e.g.*, by introducing a "forget" event), and to support subworkflows within Kepler (*i.e.*, composite actors), *e.g.*, by inferring in a bottom-up fashion the appropriate state-reset events for the composite actor via the state-reset events of subsumed actors and the corresponding workflow graph.

References

1. D. Berry, P. Buneman, M. Wilde, and Y. Ioannidis, editors. *e-Science Workshop on Data Provenance and Annotation*, National e-Science Centre, Edinburgh, December 2003.
2. D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. In *Proc. of VLDB*, 2004.
3. R. Bose and J. Frew. Lineage retrieval for scientific data processing: A survey. *ACM Computing Surveys*, 37(1):1–28, 2005.
4. P. Buneman and I. Foster, editors. *Workshop on Data Derivation and Provenance*, Chicago, October 2002.
5. P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *Proc. of ICDT*, volume 1973 of *LNCS*, 2001.
6. S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. Managing the evolution of dataflows with vistrails. In *IEEE Workshop on Workflow and Data-Flow for Scientific Applications (SciFlow)*, 2006.
7. J. Castresana. Selection of conserved blocks from multiple alignments for their use in phylogenetic analysis. *Mol. Biol. Evol.*, 17:540–552, 2000.
8. Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *VLDB*, 2001.
9. Y. Cui, J. Widom, and J. Wiender. Tracing the lineage of view data in a warehousing environment. *ACM TODS*, 25(2), 2000.
10. E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflows onto the grid. In *Proc. of the European Across Grids Conference*, 2004.
11. C. Goble. Position statement: Musings on provenance, workflow and (semantic web) annotations for bioinformatics. In Buneman and Foster [4].

[9] Specialized provenance systems for smart re-runs exist already [6].

12. G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *Proc. of the IFIP Congress*, 1977.
13. E. A. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36, 1987.
14. E. A. Lee and S. Neuendorffer. Actor-oriented models for codesign: Balancing reuse and performance. In *Formal Methods and Models for System Design*. Kluwer, 2004.
15. E. A. Lee and T. M. Parks. Dataflow process networks. *Proc. of the IEEE*, 83(5), 1995.
16. B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice & Experience*, 2005.
17. L. Moreau, O. Rana, and D. Walker. Provenance aware service-oriented architecture (pasoa). pasoa.org, 2006.
18. T. M. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, R. M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17), 2004.
19. PHYLIP Phylogeny Inference Package. http://evolution.gs.washington.edu/phylip.html.
20. Y. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Record*, 34(3):31–36, September 2005.
21. D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The Condor experience. *Concurrency – Practice and Experience*, 17(2-4), 2005.
22. J. D. Thompson, D. G. Higgins, and T. J. Gibson. CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position specific gap penalties and weight matrix choice. *Nucleic Acids Res.*, 22:4673–80, 1994.
23. J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Conference on Innovative Data Systems Research (CIDR)*, 2005.
24. S. Wong, S. Miles, W. Fang, P. Groth, and L. Moreau. Provenance-based validation of e-science experiments. In *ISWC*, 2005.
25. J. Zhao, C. Goble, R. Stephens, and S. Bechhofer. Linking and browsing provenance logs for e-science. In *ICSNW*, 2004.