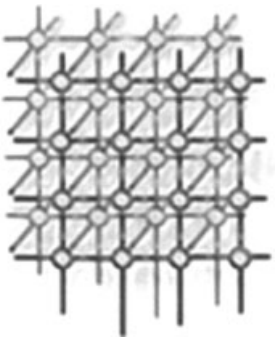
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2008; **20**:507–518
Published online 13 November 2007 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1234



From computation models to models of provenance: the RWS approach

Bertram Ludäscher[1, 2, *, †], Norbert Podhorszki[1], Ilkay Altintas[3], Shawn Bowers[2] and Timothy McPhillips[2]

[1]*Department of Computer Science, University of California, Davis, CA, U.S.A.*
[2]*Genome Center, University of California, Davis, CA, U.S.A.*
[3]*San Diego Supercomputer Center, UC, San Diego, CA, U.S.A.*

SUMMARY

Scientific workflows often benefit from or even require advanced modeling constructs, e.g. nesting of subworkflows, cycles for executing loops, data-dependent routing, and pipelined execution. In such settings, an often overlooked aspect of provenance takes center stage: a suitable *model of provenance* (MoP) for scientific workflows should be based upon the underlying *model of computation* (MoC) used for executing the workflows. We can derive an adequate MoP from a MoC (such as Kahn's process networks) by taking into account the *assumptions* that a MoC entails, and by recording the *observables* which it affords. In this way, a MoP captures or at least better approximates 'real' data dependencies for workflows with advanced modeling constructs. As a specific instance, we elaborate on the Read–Write–ReSet model, a simple and flexible MoP suitable for a number of different MoCs. Copyright © 2007 John Wiley & Sons, Ltd.

Received 8 February 2007; Revised 25 April 2007; Accepted 1 May 2007

KEY WORDS: provenance; scientific workflow; computation model

1. INTRODUCTION

Comprehensive provenance support for scientific workflow systems promises to be a key advantage of using such systems, as it gives scientists powerful means to interpret and 'debug' their analysis

*Correspondence to: Bertram Ludäscher, Department of Computer Science, University of California, 1 Shields Avenue, Davis, CA 95616, U.S.A.
†E-mail: ludaesch@ucdavis.edu

Contract/grant sponsor: NSF; contract/grant numbers: DBI-0533368, EAR-0225673, IIS 0630033, IIS 0612326
Contract/grant sponsor: DOE; contract/grant numbers: DE-FC02-01ER25486, DE-AC02-05CH11231

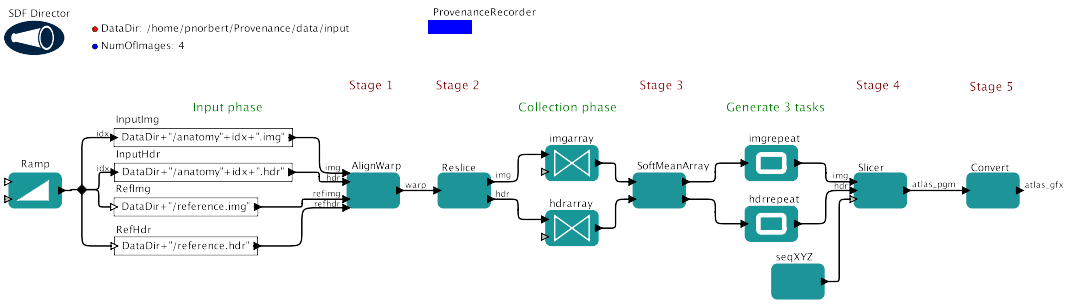


Figure 1. Main variant $W^{\mathrm{RWS}}_{\mathrm{FPC}}$ of the First Provenance Challenge workflow in KEPLER.

results. At the core of provenance support is the capability to capture the *processing history* (or *trace*) T_d that led to a workflow data product d. In its simplest form, T_d can be viewed as a tree (or a DAG in general) with root d, inner nodes representing computations, leaves representing input data sets (and parameters), and directed edges representing dataflow. For example, the First Provenance Challenge (FPC) workflow (Figure 1, [1]), produces results d_x, d_y, d_z (Atlas Graphics), the processing histories of which, $T_{d_x}, T_{d_y}, T_{d_z}$, can be seen as trees (or when combined, a single DAG) whose leaves are the input data sets (neuro-anatomy image files) and whose intermediate nodes are computations or intermediate data products, i.e. outputs of prior computation steps that constitute inputs for subsequent steps. Thus, the basic events to record for simple, DAG-style workflows like FPC state which *actor* A has executed (or *fired*) after consuming input data d, followed by writing output data d'. We denote this observation $d \overset{A}{\rightsquigarrow} d'$ and can record it as a *provenance assertion* $\mathsf{fired}(d, A, d')$. Equivalently, we sometimes say that d' *depends* on d via a firing of A, denoted $d \overset{A}{\dashleftarrow} d'$.

However, the real dependency between input and output is not always easy to determine during a workflow execution if it involves *repeated* execution of some actors and/or data-dependent routing in the workflow. In this paper we describe some *models of computation* that pose a challenge to us in creating a *model of provenance* (MoP) that can determine what depends on what. We present a specific MoP, the Read–Write–ReSet (RWS) model, which is able to capture the real dependencies under such sophisticated workflow computation models, provided actors report a certain 'reset' behavior to the recorder.

Models of computation. Consider a workflow graph W consisting of actors and *connections* (directed edges) between them[‡]. With W we can associate a set of *parameters* $\bar{p}$, input data sets $\bar{x}$, and output data sets $\bar{y}$. A *model of computation* (MoC) M prescribes how to execute the parameterized workflow $W_{\bar{p}}$ on $\bar{x}$ to obtain $\bar{y}$. Therefore, we can view a MoC as a mapping $M : \mathscr{W} \times \bar{P} \times \bar{X} \to \bar{Y}$ which for any workflow $W \in \mathscr{W}$, parameter settings $\bar{p} \in \bar{P}$, and inputs $\bar{x} \in \bar{X}$, uniquely determines the workflow outputs $\bar{y} \in \bar{Y}$ (non-deterministic MoCs can be viewed as relations). We denote this by $\bar{y} = M(W_{\bar{p}}(\bar{x}))$. Following PTOLEMY II terminology, with each MoC

[‡]Here, we ignore many details, e.g. the possibility of actor *ports*, subworkflows within W, etc.

DOI: 10.1002/cpe

M we associate a *director* of the same name, which implements M (cf. the SDF director in Figure 1, described in Section 2)[§].

For example, let $M = \text{DAG}$. Our assumptions for $W \in \mathscr{W}$ in this MoC include: (i) W is a directed, acyclic graph; (ii) each actor node in W is executed exactly once[¶]; and (iii) each actor A in W is only executed after all actors A' preceding A (denoted $A' \prec_W A$) in W have finished their execution. Note that we make no assumption whether W is executed serially or task-parallel; we only require that any DAG-compatible schedule for W must satisfy the partial order $\prec_W$ induced by W. A DAG director can obtain all legal schedules for W, i.e. the relation $\prec_W$, via a topological sort of W.

Note that a director implementing a MoC M also provides an *operational semantics* for M, which in turn can be used to define a notion of 'natural processing history' or *run* $R_{\bar{x} \rightsquigarrow \bar{y}}$ for $\bar{y} = M(W_{\bar{p}}(\bar{x}))$. In the case of $M = \text{DAG}$, we can think of $R_{\bar{x} \rightsquigarrow \bar{y}}$ as a sequence of records of the form $\mathsf{fired}(d, A, d')$ (assuming serial execution; a task-parallel run $R_{\bar{x} \rightsquigarrow \bar{y}}$ can be given as a partially ordered set of records). Given a run $R_{\bar{x} \rightsquigarrow \bar{y}}$, the operational semantics associated with M allows us to check whether $R_{\bar{x} \rightsquigarrow \bar{y}}$ is a 'faithful' (or *legal*) execution of $\bar{y} = M(W_{\bar{p}}(\bar{x}))$. For example, the order of actor firings in a legal run $R_{\bar{x} \rightsquigarrow \bar{y}}$ must conform to $\prec_W$; and $\bar{p}$, $\bar{x}$, and $\bar{y}$ must appear in $R_{\bar{x} \rightsquigarrow \bar{y}}$ as inputs and outputs, respectively.

Observables. The records of a run $R_{\bar{x} \rightsquigarrow \bar{y}}$ of a workflow execution of $\bar{y} = M(W_{\bar{p}}(\bar{x}))$ are built from basic *observables* associated with M. For $M = \text{DAG}$, the observables of a run are the (single!) *firings* of an actor A, together with the inputs d and outputs d' of the firing, recorded as $\mathsf{fired}(d, A, d')$. We may dissect firing events into smaller observables, e.g. into two records of the form $\mathsf{received}(A, m_1(d))$ and $\mathsf{sent}(A, m_2(d'))$, and a third record of the form $\mathsf{caused}(m_1, m_2)$[‖]. This can be useful for MoCs such as Communicating Sequential Processes or Message Passing Interface, where actors react to different messages, not just the implicit *read* (input consumption) and *write* (output production) that we used here for DAG. Similarly, most (if not all) job-based (or *Grid*) workflow systems are based on the DAG MoC. For these, $\mathsf{fired}(d, A, d')$ may be modeled differently, e.g. based on a job's start and finish events.

Finally, consider a 'Turing workflow' (program) W with inputs $\bar{p}$, $\bar{x}$. Transition events $q \overset{s/s'}{\rightsquigarrow} q' : h$ can be used to record the machine state q, the symbol s just read, the symbol written s', the next state q', and the head movement $h \in \{\mathsf{L}, \mathsf{R}\}$. Thus, a run $R_{\bar{x} \rightsquigarrow \bar{y}}$ of W can be captured by storing the sequence of transition events (which could be composed of smaller observables, e.g. the head movement). However, unlike Turing programs, scientific workflows typically contain *black box* components, e.g. actors for web services or external applications whose internal computations are not observable. Thus, a MoC M determines the legal runs $R_{\bar{x} \rightsquigarrow \bar{y}}$ under M only with respect to M's observables.

[§] As in PTOLEMY II [2], we separate the concern of workflow 'wiring' from that of prescribing a specific execution semantics. W defines the former, while a director is a scheduling and orchestration component, implementing the latter.

[¶] Hereby, we exclude *data-dependent routing*: for any two branches $\mathscr{B}, \mathscr{C} \in W$ that fork at an actor A (denoted $A \rightrightarrows \mathscr{B}, \mathscr{C}$), *both* branches $\mathscr{B}$ and $\mathscr{C}$ are enabled. Thus, all actors on both branches $\mathscr{B}$ and $\mathscr{C}$ must be eventually executed after the firing of A. Hence, in this DAG MoC, we cannot selectively route data or control to *either* $\mathscr{B}$ *or* $\mathscr{C}$ (other MoCs, e.g. PN allow this behavior).

[‖] The PASOA/OPA provenance model, e.g. is based on observables/assertions of the form received, sent, and caused [1,3].

DOI: 10.1002/cpe

Models of provenance. Given a MoC M and a set of observables $\mathcal{O}$ used to describe it, we can speak of the legal runs $R_{\bar{x}\rightsquigarrow\bar{y}}$ under M for the computation $\bar{x} \overset{W_{\bar{p}}}{\rightsquigarrow} \bar{y}$. With M and $\mathcal{O}$ we associate a natural (or *default*) MoP M_0. The default MoP M_0 really is just M, but instead of speaking about legal runs $R_{\bar{x}\rightsquigarrow\bar{y}}$, we speak of legal *traces* $T_{\bar{x}\rightsquigarrow\bar{y}}$ which consist of *provenance assertions* corresponding to the observables $\mathcal{O}$. The canonical use case for M_0 is *Single-Step Replay*, in which a user traces a workflow execution one observable step at a time, based on $T_{\bar{x}\rightsquigarrow\bar{y}}$.

A *general* MoP P for a MoC M is usually based on M_0 but may introduce new observables $\mathcal{O}_m$ not part of M_0. For example, $\mathcal{O}_m$ may add *timestamps* to provenance assertions, recording the local time at which an event was observed, or indicate the execution host and user id executing the observed event. Conversely, P often only *approximates* M and thus ignores or simplifies some observables $\mathcal{O}_i \subseteq \mathcal{O}$. In this sense, we can say that a trace T in P can be obtained from a run R in M by ignoring some observables $i \in \mathcal{O}_i$ and instead modeling other observables $m \in \mathcal{O}_m$. With slight abuse of notation, we may state that '$T = R - \{i\} + \{m\}$', i.e. a provenance trace T in the MoP P is a 'trimmed' workflow run R that **i**gnores some aspects i of M_0 and **m**odels some additional aspects m not in M_0.

2. CAPTURING PROVENANCE IN ADVANCED MODELS OF COMPUTATION

For DAG-style workflows like FPC, the above notions of MoC and MoP may seem complex at first. Indeed, the simple DAG MoC is common in job-based Grid workflow systems and also sufficient for the simple FPC workflow. Keeping track of data dependencies is straightforward in DAG, at least in principle, by observing actor firings and recording the corresponding data dependencies $d \overset{A}{\dashleftarrow} d'$.

However, many scientific workflows are based on MoCs that include advanced modeling constructs, e.g. nesting of subworkflows, cycles for executing loops, data-dependent routing, and pipelined execution [4–6]. In such settings, keeping track of 'real' data dependencies is much more challenging and can only be achieved by using MoPs that are 'MoC-aware', i.e. which take into account specific knowledge (assumptions and observables) about the underlying MoCs. In particular, for more sophisticated MoCs, choosing a MoP that assumes (or is derived from) the simple DAG MoC results in inadequate provenance traces that contain far too many (or possibly too few) data dependencies. To illustrate, we briefly discuss some of the implications for MoPs when advanced MoCs such as PN, SDF, and COMAD are used.

Process networks (PN). This is a MoC based on Kahn process networks [7,8]. In PN, actors execute as *concurrent* processes which communicate by sending ordered *streams* of data tokens over unidirectional first-in, first-out channels (the directed edges in the workflow graph W). PN extends DAG in several ways, e.g. the workflow graph W can have *cycles* (for executing loops), *data-dependent routing* is possible**, and in addition to task parallelism (as supported by DAG), PN exhibits *pipeline parallelism*. Not surprisingly, PN is the basic model or starting point for many dataflow-based systems with pipelined execution. Note that PN, in its most general form, does *not*

**That is for a branch $A \rightrightarrows \overset{\mathscr{B}}{\mathscr{C}}$ in W, individual tokens can *either* flow through $\mathscr{B}$ *or* through $\mathscr{C}$.

DOI: 10.1002/cpe

include an explicit observable for *firing* an actor. Instead, in the abstract PN model [7] we can only observe how the concurrently executing actors *read* (consume) and *write* (produce) tokens.

The RW$_0$ *MoP*. For each actor A of W, we observe a sequence of event occurrences $e_A^1, e_A^2, \ldots$ Each e_A^i is either $\mathsf{r}_A(x)$ or $\mathsf{w}_A(x)$, denoting a *read* or *write* of token x, respectively. Assuming that unique data objects (e.g. via unique token ids) are created with each write event, this is sufficient to chain together data dependencies in provenance traces. The resulting 'Read–Write' MoP of PN is called RW$_0$. Consider an arbitrary situation during the execution of a workflow under PN. Any actor A will have observed some sequence $e_A^1, e_A^2, \ldots, e_A^n$ of read or write events. Let the sequence of read (consumed) tokens be $x_1, \ldots, x_k$, and let $y_1, \ldots, y_\ell$ be the sequence of written (produced) tokens. Note that $n = k + \ell$, but *not* necessarily $\ell \geq k$ (in PN, actors may write any number of tokens, e.g. dependent on the value of a data token read, but independent of the number of read tokens). Assume that the next observable event at A is $\mathsf{w}_A(y_{\ell+1})$, i.e. A has just written a new token. In PN, without additional assumptions or observables, we must assume that $y_{\ell+1}$ depends on *all* tokens x_i previously read by A, hence we have to record *all* dependencies $x_i \overset{A}{\leftarrow\!\!-\!\!-} y_{\ell+1}$, for $i = 1, \ldots, k$.

This is correct and in fact desired when considering, e.g. a stateful aggregation actor A_Σ, which upon reading the kth input token x_k outputs the running sum $y_k := \sum_{i=1}^{k} x_i$ of all token values read so far. The RW$_0$ model will correctly record $x_i \overset{A}{\leftarrow\!\!-\!\!-} y_k$, for all $i = 1, \ldots, k$.

Specializing RW$_0$ *when knowing more.* While RW$_0$ is suitable for the generic PN MoC, too many dependencies may be created in situations where we have additional knowledge about a workflow or the MoC being used. For example, consider an actor $A_{F\to C}$ translating a Fahrenheit value to Celsius, or an actor A_{conv} converting files from one image-file format to another. Like A_Σ, these actors read one token, compute, then write one token (and thus allow to observe firings, while PN—without further assumptions—has no such notion). For such actors A, we would like to record $x_i \overset{A}{\leftarrow\!\!-\!\!-} y_k$ only for the single, 'real dependency' where $i = k$ (corresponding to the kth firing of A), but *not* for any $i < k$.

How can a MoP 'know' when to prune pseudo-dependencies, e.g. when considering stateless instead of stateful computations? One option, discussed elsewhere [1,6,9], is to use a MoC such as COMAD in which token streams have *structure* and thus meaning in terms of provenance. The COMAD MoC, e.g. views token streams as tagged, nested data collections (akin to the XML model). Thus, actors in a COMAD workflow perform computations not just on generic PN token streams, but on structured *collection* objects. Consequently, a COMAD MoP should record the dependencies between output and input collections appropriately [1,9], instead of 'blindly' using RW$_0$.

Synchronous dataflow (SDF). Another option, discussed below, is to include knowledge about the specific MoC or to consider additional observables. First, consider a special MoC called SDF [10]. In SDF (like in DAG, but unlike in PN) there is a notion of actor *firing*. To this end, every actor declares a priori its *token rate*, denoted $n_r : m_w$, indicating that after receiving n_r tokens it will fire (execute), and then write m_w tokens before possibly firing again. SDF is much more expressive than DAG (e.g. nested workflows, cycles, and pipelining can be handled), but not as general as PN[††].

[††]For example, data-dependent routing is not directly possible since m_w a priori fixes the number of output tokens per firing and port.

DOI: 10.1002/cpe

A practical advantage of SDF over PN is that more static analysis techniques can be applied, e.g. at compile time (serial and parallel) SDF schedules can be computed and deadlocks detected.

SDF$_1$ *MoC, and* RW$_1$ *MoP.* Let us consider an important special case, denoted **SDF$_1$**, in which all actors of a workflow have the token rate $1_r : 1_w$, i.e. all input (output) ports read (write) exactly one token per firing of A. Let us also assume that all actors of a workflow W are *stateless* (i.e. functional, like $A_{F\to C}$ and A_{conv}). Then, we can devise a suitable MoP which records $x_i \overset{A}{\leftarrow\!\!-} y_k$ (upon the kth firing of A) only for the real dependency $i=k$, avoiding all pseudo-dependencies for $i<k$ (RW$_0$ records these 'overestimates'). Let this new MoP for SDF$_1$ over stateless actors be called RW$_1$.

If we cannot assume that all actors in SDF$_1$ are stateless then we have again different options: one is to introduce a special compile-time observable which declares whether an actor is stateless or stateful. For the former, a single dependency to the input token which determines the result is recorded, while for the latter, dependencies to all previously read tokens are recorded, as in RW$_0$. Another option is to modify the workflow and model a stateful actor A using a stateless actor A' plus a *feedback* loop (e.g. A_Σ can be modeled as a stateless actor A_+ with delayed feedback loop and initial value 0). Through the feedback loop, an output token (representing A's state) is revisited by A' upon the next firing. Thus, state-dependent outputs are linked to all inputs on which they depend via revolving state tokens.

Finally, recall that DAG can be seen as a very restricted form of SDF$_1$ (and thus of PN). Therefore, although data dependencies $d \overset{A}{\leftarrow\!\!-} d'$ can be obtained for DAG simply by observing $\mathsf{fired}(d, A, d')$, alternatively we might decide to apply RW$_1$ or RW$_0$, the MoPs for SDF$_1$ and general PN, respectively. Both MoPs work as expected for DAG: the dependencies $x_i \overset{A}{\leftarrow\!\!-} y_k$ are recorded for $i=k$ (in RW$_1$) and for $i\le k$ (in RW$_0$). In DAG, because $i=k=1$, both MoPs record the same dependencies.

3. THE READ–WRITE–RESET (RWS) MODEL OF PROVENANCE

The MoPs RW$_0$ and RW$_1$ represent two extremes based on different compile-time assumptions about the underlying MoC. To handle general PN, RW$_0$ has to include dependencies to all previously read tokens, while RW$_1$ correctly handles SDF$_1$ workflows, provided they consist only of stateless actors.

The RWS model [11,12] is more flexible and subsumes both RW$_0$ and RW$_1$. It is based on an additional observable s_A for *state reset*, which indicates that an actor A has just completed a 'logical chunk' of computation, thus resetting itself to an initial state. Hence, in RWS, subsequently written tokens should *not* be considered dependent upon any token read prior to the state-reset event.

Consider, e.g. an actor A_{avg} which upon receiving a new token x_k emits the running average y_k over all $x_1, \ldots, x_k$ received so far. RW$_0$ correctly handles the dependencies of this stateful actor (RW$_1$ reports too few dependencies, unless A_{avg} is replaced by a stateless actor plus feedback loop). However, neither RW$_0$ nor RW$_1$ record the desired dependencies for actor $A_{avg_{24}}$ which outputs a daily running average, i.e. $A_{avg_{24}}$ outputs a running average similar to A_{avg}, but resets itself every 24 h, 'forgetting' all past inputs. Therefore, RW$_0$ (RW$_1$) reports too many (too few) dependencies for $A_{avg_{24}}$. For another example, consider a filter actor A_θ which outputs the tokens that satisfy a filter condition θ but which drops all other inputs. The real dependencies between A_θ's outputs and inputs cannot be determined by looking at read and write events only. In contrast, with an additional state-reset observable s_A, the RWS model can capture the actual dependencies of both

DOI: 10.1002/cpe

A_θ and $A_{avg_{24}}$. The RWS trace of A_θ corresponds to a pattern $(\mathsf{rs} \mid \mathsf{rws})^*$, where the first branch rs occurs whenever a token t does not satisfy $\theta(t)$ and thus is not written, while the second branch rws occurs for tokens that satisfy the filter condition θ. Similarly, the behavior of $A_{avg_{24}}$ can be described by the pattern $((\mathsf{rw})^{24}\mathsf{s})^*$.

RWS defines an (implicit and 'low-level') notion of firing (via r_A, w_A), plus a new notion of 'firing round' (using s_A) which captures semantically meaningful 'transactions'. More precisely, a *firing* F_A of an actor A is recognized by observing maximal series of reads followed by writes ($\mathsf{r}^+\mathsf{w}^+$), without intermediate resets s. Let F_A^i denote the ith such firing of A during a run. A *firing round* $\mathsf{R}_A^{m..n}$ is a maximal sequence of consecutive firings F_A^m, $\mathsf{F}_A^{m+1}, \ldots, \mathsf{F}_A^n$, ending in a state reset event s_A. Since s_A marks a state reset and thus delimits firing rounds, every round $\mathsf{R}_A^{m..n}$ corresponds to a pattern $((\mathsf{r}^+\mathsf{w}^+)^*\mathsf{s})$ in the provenance trace. Thus, while firings capture an actor's low-level read/write transitions (e.g. reading and writing SAX events on a XML stream), rounds capture higher-level *transactions* that define the semantically meaningful token 'chunks' on which actors operate.

In RWS a data dependency $x_i \overset{A}{\dashleftarrow} y_k$ holds if there is a round $\mathsf{R}_A^{m..n}$ with a read event $\mathsf{r}_A(x_i) \in \mathsf{F}_A^j$ and a subsequent write event $\mathsf{w}_A(y_k) \in \mathsf{F}_A^\ell$ occurring within the *same* $\mathsf{R}_A^{m..n}$, i.e. $m \le j \le \ell \le n$. In other words, no data dependency crosses an s_A event: RWS limits dependencies to within a transaction.

RW_0 is a special case of RWS in which no s events are observed. Each read/write transition in RW_0 is seen as a (low-level) RWS firing, but overall there is only one transaction, the complete workflow run. Similarly, we obtain the RW_1 model as a special case of RWS, provided every actor A signals a state reset s_A after each sequence of r_A, w_A events. Thus, every read/write transition (firing) in SDF_1/RW_1 also marks a transaction (firing round) in RWS, yielding the desired dependencies.

For the implementation of RWS, the new observable s has to be created by some entity which 'knows' when to do a state reset. Typically, the actor (or actor developer) knows when such a state reset occurs. Since many actors are stateless, an RWS implementation could by default generate an s event whenever the actor turns to reading after writing (this is equivalent to the RW_1 model). Such a default behavior is indeed implemented by the Kepler provenance recorder described in the next section. Stateful actors should, however, notify the provenance recorder about state resets themselves.

4. RWS PROVENANCE FOR THE KEPLER WORKFLOW SYSTEM

KEPLER [4] is a scientific workflow system based on PTOLEMY II [2,13] that supports different MoCs M by applying the corresponding *director* for M (usually of the same name) to a workflow graph W. The KEPLER provenance recorder [12,14] provides an extensible framework for capturing provenance information, without the need for modifying any actors within a workflow. This framework has been used to implement support for 'smart re-run', a use case where data products from previous workflow runs are reused to optimize subsequent runs with different inputs or parameter settings. For the Provenance Challenge, the provenance recorder has been extended to record RW trace information.

In KEPLER, data objects are encapsulated as *tokens* that flow between actors. The RWS recorder identifies each input token and assigns its id given at the emitter output port, that is, a recorded read event always contains the same token identifier than the write event of the sender of that

DOI: 10.1002/cpe

token, taking the burden of finding the relationship from any query engine. The unique token id is generated from the name of the emitting actor, its output port, channel, and an additional counter. The RWS recorder puts information into several relational tables: *traceTable(token, event, port)* contains the RWS events with the token references, *portTable(port, actor)* can be used to get the actor that created the token, *tokenTable(token, object)* to get the object (reference) contained by the token, and finally *objectTable(object, value, type)* to get the value of an object.

Prototype provenance query engine. The KEPLER provenance recorder deals with recording information and does not include a special querying framework for provenance questions. For example, the smart re-run application uses an *ad hoc* way for handling the recorded information. By implementing the RWS model, our goal was to extend the capabilities to record the real dependencies in all KEPLER workflows executed under different models of computation. The querying part of the provenance was not our focus, and a prototype inference engine has been implemented in Prolog to answer the first query of the challenge. The basic operation of the engine is to provide the *lineage graph* of tokens (i.e. the transitive closure of the direct dependencies), see [11,12] for more details and examples of lineage graphs. The graph is represented via its edges, which are given by pairs of node (or token) ids. From the tokens, the involved actors and generated data values are determined directly from the above-mentioned additional recorded tables.

It is important to emphasize that with RWS we have focused on single workflow runs to correctly infer the dependencies among inputs and outputs of each operation. That is, we have targeted primarily the first query in the FPC. We have not dealt with the problem of querying among different runs. The KEPLER provenance recorder stores general information about the workflow runs (date, parameters, etc.), while the implementation of the RWS model itself stores a detailed trace about the activities of a single execution. The RWS trace information is sufficient to answer the other challenge queries as well.

5. PROPERTIES OF THE RWS MODEL: EXPLAINING THE CHALLENGE MATRIX

The RWS provenance model has been implemented within the KEPLER workflow system (C1.1, C1.2). The natural execution environment for RWS is a workflow system, but RWS is obviously not restricted to KEPLER. On the other hand, the advantages of the RWS MoP appear only in workflows with MoCs that go beyond simple DAG, e.g. MoCs operating on streams of data (for pipelined execution), have control structures like conditional branches and loops, and where tasks can depend on various data for producing a particular output.

In our implementation, provenance information is recorded using *relational records* (C1.3). Specifically, since we used a Prolog inference engine for rapid prototyping, trace information is written to a file as plain text in the form of *Prolog facts*. The query language for RWS provenance data of KEPLER workflows is an *internal graph query language* (C1.4), for which the basic operation is the transitive closure over token dependencies (input–output dependencies within an actor, and output–input dependencies between actors).

Our research emphasis was on *modeling*, *execution*, and *recording* (C1.5). The recorded provenance information is sophisticated enough to provide enough information for the query engine to

DOI: 10.1002/cpe

reproduce all dependencies. Our RWS model and its KEPLER implementation enabled us to build and execute different versions W_{DAG} and W_{RWS} of the FPC workflow (see below).

We implemented the FPC workflow *partially* (C1.6), i.e. the image-processing actors are stubs that read and write the given files. Files were passed through actors via path names. Since the provenance recorder in its basic form does not record information about events outside of KEPLER, the resulting traces are exactly the same as under a 'real' execution (atomic actors are treated as black boxes).

The RWS model captures the causal graph without the need to explicitly store the workflow graph W_{FPC} (C2.1). The inference engine does not use W_{FPC} to answer the provenance queries, since the unique token ids provide all information to determine the causal flow of events. In our approach, the workflow W graph can be inferred from the RWS trace (more precisely, the executed parts of W).

Data dependencies capture the *derivation of data*. The *causal flow of events* can also be described by matching an actor's output to another actor's input. The RWS model does not need to record this, but can separately infer this information from the recorded events through token dependencies (C2.2).

The RWS model in [11,12] includes semantic information to answer user-oriented provenance questions. Semantic port annotations (recorded in read/write events) allow answering provenance queries at the user's conceptual level: e.g. the question '*What images have been created during the workflow execution?*' may provide the images but not their headers in W_{FPC}. RWS handles object annotations within the workflow (C2.3), however, it cannot handle existing external annotations of objects. The prototype implementation does not include the recording of semantic types.

The model *supports to store measured time* (C2.4)—an example of $\mathcal{O}_m$, a non-MoC observable. Date information for runs is stored so that traces can be searched by date. The execution date and duration for each actor/task is also measured during the execution (but not stored currently). However, the RWS model *does not depend* on timestamps of tasks to infer token or data dependencies. On the other hand, RWS does assume that consecutive (streaming) outputs of an actor are consumed by a subsequent actor in the same order. This is an inherent property of the underlying MoCs and enforced in KEPLER.

The recorded tokens should be named uniquely (C2.5) to be able to distinguish among them in the provenance trace. The model records provenance at the token level (C2.6). For basic data types, any changes are recorded. However, complex objects can be transferred within a token, and if an actor changes the internal state of the object without creating a new token, this 'undocumented' change will not be recognized by the provenance system (tokens are assumed to be immutable). Also, no activity of actors is recorded at the operating system level: e.g. if a file name f is given as an actor parameter, we assume that the output tokens of that actor depend on f, without investigating whether the actor actually opened the file f. The inference engine of the RWS model can be used to answer questions for nested workflows, either in full detail or in top-level tasks only (C2.7).

6. IMPLEMENTING THE CHALLENGE WORKFLOW IN KEPLER

We have implemented three different versions of the FPC workflow W_{FPC}: $W_{\mathrm{FPC}}^{\mathrm{DAG}}$ is a one-to-one implementation of W_{FPC} in KEPLER, assuming the simple DAG MoC. Thus, its provenance is correctly handled by all MoPs RW_0, RW_1, and RWS. W_{FPC} specifies a fixed set $I = \{f_1, \ldots, f_4\}$ of

DOI: 10.1002/cpe

image files. A main disadvantage of the $W_{\text{FPC}}^{\text{DAG}}$ design is that changes in I require changes to the $W_{\text{FPC}}^{\text{DAG}}$ design (not just parameter changes as one might hope).

Instead of describing $W_{\text{FPC}}^{\text{DAG}}$ here[‡‡], we consider our main variant, $W_{\text{FPC}}^{\text{RWS}}$—see Figure 1 (the third variant $W_{\text{FPC}}^{\text{COMAD}}$ is described in [1,9]). The basic idea of $W_{\text{FPC}}^{\text{RWS}}$ is to exploit the streaming capabilities of KEPLER's dataflow MoCs (here, SDF or PN) so that multiple tokens, e.g. one for each image file, can be piped through the workflow. In this way, $W_{\text{FPC}}^{\text{RWS}}$ can handle any number n of image files $\{f_i\}_{i \leq n}$, and n becomes a simple, user-definable *parameter* (`NumOfImages` in Figure 1). In $W_{\text{FPC}}^{\text{RWS}}$, all data produced is identified by file names. We use array actors to produce arrays of tokens from a token stream. There are no major differences between the $W_{\text{FPC}}^{\text{RWS}}$ traces and the $W_{\text{FPC}}^{\text{DAG}}$ traces and between the answers for the challenge queries besides the appearance of a few new actors (e.g. `imgarray`) in $W_{\text{FPC}}^{\text{RWS}}$. The $W_{\text{FPC}}^{\text{RWS}}$ workflow starts with a `Ramp` actor that generates a stream of n integer tokens to 'drive' the workflow (n is set via `NumOfImages`). The constant actors (white boxes) produce input image file names, using the integer tokens as indexes (we assume file names of the form `anatomy`k`.img`, where $k = 1, \ldots,$ `NumOfImages`). The `AlignWarp` and `Reslice` actors operate on each image file independently, so they are fed a stream of tokens, one for each file. However, we need to gather all resliced images to perform the `SoftMean` operation. Therefore, in an intermediary step (annotated *Collection Phase*), we collect all resliced images (and their header files) into an array. The `SoftMeanArray` actor is fired only once, and processes all images to produce its single output. In the next stage, we want to use the same output three times to make slices in the three axes. Therefore, in another intermediary step (*Generate 3 tasks*) we produce a stream consisting of three copies of the same token. The `Slicer` and `Convert` actors thus are fired three times, fed by the same image but with a different (dynamic) parameter. The latter is generated by the `seqXYZ` sequence producer actor. The `SDF Director` implements the SDF MoC and executes the workflow sequentially, i.e. the four input images are processed one-by-one. By replacing it with a `PN Director` we obtain pipeline-parallel execution on the token stream. Changing these directors does not change the RWS trace.

Answering the first query of the provenance challenge. The query asks to 'find the process that led to Atlas X Graphic/everything that caused Atlas X Graphic to be as it is. This should tell us the new brain images from which the averaged atlas was generated, the warping performed etc.'. By using the Prolog inference engine prototype, we can get the detailed token lineage graph of any value. Note that the engine is not part of the Kepler provenance framework. For the value in question, 'output/atlas-x.gif' we get the token that contains it using *tokenTable* and *objectTable*, then the lineage of the tokens represented as the set of edges (57 edges) can be inferred. Six edges from the whole set (the closest ones to the last Convert actor that created the data in question) are

(SoftMeanArray.atlas_hdr.0.0, hdrrepeat.output.0.0)	(hdrrepeat.output.0.0, Slicer.atlas_pgm.0.0)
(SoftMeanArray.atlas_img.0.0, imgrepeat.output.0.0)	(imgrepeat.output.0.0, Slicer.atlas_pgm.0.0)
(seqXYZ.output.0.0, Slicer.atlas_pgm.0.0)	(Slicer.atlas_pgm.0.0, Convert.atlas_gfx.0.0)

Here, we use one way to create unique token ids, i.e. from the name of the emitting actor, a port name, channel, and an additional (firing) counter. The token id can be used in lookups in relational tables but its internal structure should not be exploited. We can get the actors involved and the data

[‡‡]Instead, see `http://twiki.gridprovenance.org /bin/view/Challenge/RWS`.

DOI: 10.1002/cpe

values generated during the process leading to the particular value using the other recorded tables with the above token identifiers. From *traceTable* we get which port has emitted a given token, then *portTable* tells us, which actor is the owner of that port: e.g. token *SoftMeanArray.atlas_img.0.0* was emitted on port *SoftMeanArray.atlas_img* belonging to actor *SoftMeanArray*. Finally, from *tokenTable* and *objectTable* we can get the data value contained in the given token, e.g. for the above token, the internal object id was *o1833409252_42511136* and the value of that object was *out-stage3/atlas.img*.

7. SUMMARY

We have developed and prototypically implemented the RWS MoP in the context of the First Provenance Challenge. RWS is simple, yet allows to capture the desired data dependencies even for advanced MoCs such as PN and SDF. Such MoCs are useful not only in cases where data are inherently streaming (e.g. sensor networks), but even for simple workflows such as W_{FPC}. The simplest implementation $W_{\mathrm{FPC}}^{\mathrm{DAG}}$ can handle only a fixed number of image files (four). Instead, in $W_{\mathrm{FPC}}^{\mathrm{RWS}}$ a parameter is used to handle any number of input files. This more flexible design can be executed using different MoCs including PN (preferable when implemented on a cluster computer or Grid) or SDF (preferable for embedded or single-CPU systems). Each MoC induces a default MoP, where runs of the former correspond to traces of the latter. In this sense, RWS is a meta-MoP as it can handle many different MoCs. A major lesson learned, and special emphasis of this paper, was the importance of MoCs for workflow design and provenance management, in particular for capturing the intended data dependencies.

ACKNOWLEDGEMENTS

This work was supported by NSF grants DBI-0533368 (SEEK), EAR-0225673 (GEON), IIS 0630033 (pPOD), and IIS 0612326 (ChIP-chip); and DOE grants DE-FC02-01ER25486 (SDM) and DE-AC02-05CH11231 (CPES).

REFERENCES

1. Moreau L, Ludäscher B, Altintas I, Barga RS, Bowers S, Callahan S, Chin G Jr, Clifford B, Cohen S, Cohen-Boulakia S, Davidson S, Deelman E, Digiampietri L, Foster I, Freire J, Frew J, Futrelle J, Gibson T, Gil Y, Goble C, Golbeck J, Groth P, Holland DA, Jiang S, Kim J, Koop D, Krenek A, McPhillips T, Mehta G, Miles S, Metzger D, Munroe S, Myers J, Plale B, Podhorszki N, Ratnakar V, Santos E, Scheidegger C, Schuchardt K, Seltzer M, Simmhan YL, Silva C, Slaughter P, Stephan E, Stevens R, Turi D, Vo H, Wilde M, Zhao J, Zhao Y. The First Provenance Challenge. *Concurrency and Computation*: *Practice and Experience—Special Issue on the First Provenance Challenge* 2007; DOI: 10.1002/cpe.1233.
2. PTOLEMY II project and system. Department of EECS, UC Berkeley, 2006. http://ptolemy.eecs.berkeley.edu/ptolemyII/ [28 August 2007].
3. Miles S, Groth P, Munroe S, Jiang S, Assandri T, Moreau L. Extracting causal graphs from an open provenance data model. *Concurrency and Computation*: *Practice and Experience—Special Issue on the First Provenance Challenge* 2007; DOI: 10.1002/cpe.1236.
4. Ludäscher B, Altintas I, Berkley C, Higgins D, Jaeger E, Jones M, Lee EA, Tao J, Zhao Y. Scientific workflow management and the Kepler system. *Concurrency and Computation*: *Practice and Experience* 2006; **18**(10):1039–1065.

5. Bowers S, Ludäscher B. Actor-oriented design of scientific workflows. *24th International Conference on Conceptual Modeling* (*ER*), Klagenfurt, Austria, October 2005. Springer: Berlin, 2005.
6. McPhillips T, Bowers S, Ludäscher B. Collection-oriented scientific workflows for integrating and analyzing biological data. *Third International Workshop on Data Integration in the Life Sciences* (*DILS*), European Bioinformatics Institute, Hinxton, U.K., July 2006 (*Lecture Notes in Computer Science*). Springer: Berlin, 2006.
7. Kahn G. The semantics of a simple language for parallel programming. *Proceedings of the IFIP Congress 74*, Stockholm, Sweden, 5–10 August 1974, Rosenfeld JL (ed.). North-Holland: Amsterdam, 1974; 471–475. ISBN: 0-7204-2803-3.
8. Lee EA, Parks T. Dataflow process networks. *Proceedings of the IEEE* 1995; **83**(5):773–799.
9. Bowers S, McPhillips TM, Ludäscher B. Provenance in collection-oriented scientific workflows. *Concurrency and Computation*: *Practice and Experience—Special Issue on the First Provenance Challenge* 2007; DOI: 10.1002/cpe.1226.
10. Lee EA, Messerschmitt DG. Synchronous data flow. *Proceedings of the IEEE* 1987; **75**:1235–1245.
11. Bowers S, McPhillips T, Ludäscher B, Cohen S, Davidson SB. A model for user-oriented data provenance in pipelined scientific workflows. *International Provenance and Annotation Workshop* (*IPAW*), Chicago, IL, May 2006 (*Lecture Notes in Computer Science*, vol. 4145), Moreau L, Foster I (eds.). Springer: Berlin, 2006.
12. Moreau L, Foster I (eds.). *International Provenance and Annotation Workshop* (*IPAW*), Chicago, IL, May 2006 (*Lecture Notes in Computer Science*, vol. 4145). Springer: Berlin, 2006.
13. Brooks C, Lee EA, Liu X, Neuendorffer S, Zhao Y, Zheng H (eds.). Heterogeneous concurrent modeling and design in Java (vol. 1–3). *Technical Memorandum UCB/ERL M05/21*, *M05/22*, *M05/23*, University of California, Berkeley, CA, 2005.
14. Altintas I, Barney O, Jaeger-Frank E. Provenance collection support in the Kepler scientific workflow system. *International Provenance and Annotation Workshop* (*IPAW*), Chicago, IL, May 2006 (*Lecture Notes in Computer Science*, vol. 4145), Moreau L, Foster I (eds.). Springer: Berlin, 2006.