A Dataflow-Oriented Atomicity and Provenance System for Pipelined Scientific Workflows *

Liqiang Wang[1], Shiyong Lu[2], Xubo Fei[2], and Jeffrey Ram[3]

[1] Dept. of Computer Science, University of Wyoming, USA. wang@cs.uwyo.edu
[2] Dept. of Computer Science, Wayne State University, USA.
{shiyong, xubo}@wayne.edu
[3] Dept. of Physiology, Wayne State University, USA. jeffram@med.wayne.edu

Abstract. Scientific workflows have gained great momentum in recent years due to their critical roles in e-Science and cyberinfrastructure applications. However, some tasks of a scientific workflow might fail during execution. A domain scientist might require a region of a scientific workflow to be "atomic". Data provenance, which determines the source data that are used to produce a data item, is also essential to scientific workflows. In this paper, we propose: (*i*) an architecture for scientific workflow management systems that supports both provenance and atomicity; (*ii*) a dataflow-oriented atomicity model that supports the notions of commit and abort; and (*iii*) a dataflow-oriented provenance model that, in addition to supporting existing provenance graphs and queries, also supports queries related to atomicity and failure.

1 Introduction

Scientific workflow systems are increasingly used to execute scientific data management and analysis in many disciplines, such as biology, medicine, chemistry, physics, and astronomy. In contrast to traditional business workflows, which are task-centric and control-flow oriented, scientific workflows are data-centric and dataflow oriented. More specifically, in a business workflow model, the design of a workflow focuses on how execution control flows from one task to another (sequential, parallel, conditional, loop, or event-condition-action triggers), forming various "control-flows". In a scientific workflow model, the design of a workflow focuses on how the input data are streamlined into various data analysis steps using data channels to produce various intermediate data and final data products, forming various "dataflows".

Atomicity is an important transactional property, which requires that a transaction either runs in completion or has no partial effect (all-or-nothing). In scientific workflows, some task might fail during execution due to either the failure of the task itself or inappropriate input to a task. Despite the failure of tasks, a domain scientist might require a region of a scientific workflow to be "atomic" in the sense that either the execution of all the tasks in that region run to completion or none of them has any effect at all. However, traditional techniques for atomicity in transaction processing systems are inappropriate for complex long-running processes in distributed and heterogeneous environments. Compensation is generally considered as a proper way to handle rollback in business

* The first two authors contributed equally to this paper.

workflows [6], as it can eliminate effects of already committed transactions. The atomicity techniques based on compensation in business workflows [8, 5] are not suitable for scientific workflows. They often require the explicit definitions of transaction boundaries which are obscured in our case due to the data dependency introduced by pipelined execution of workflows. Moreover, since scientific workflows are often computation-intensive, traditional rollback techniques are inefficient because the intermediate results of aborted transactions, which might be reusable in the future, are discarded.

Data provenance is closely related to the data lineage problem [3] studied in the database community, which determines the source data that are used to produce a data item. However, in scientific workflows, datasets are not necessarily contained in a relational or XML database and data processing cannot necessarily be accomplished by a database query. Therefore, existing approaches to the data lineage problem are not sufficient for solving the data provenance problem in scientific workflows. Moreover, although several provenance models (such as [2]) have been proposed for scientific workflows, none of them supports the notion of atomicity.

This paper proposes a novel dataflow-oriented atomicity and provenance system for scientific workflows. To the best of our knowledge, our system is the first one that supports both atomicity and provenance. It captures dataflows in scientific workflows, where data communications between tasks are modeled as enqueue and dequeue operations of a recoverable queue [1]. Transaction boundaries are not necessarily defined, instead, data dependencies are tracked and logged. Our system consists of two subsystems: atomicity management subsystem, which performs commit and abort, and provenance subsystem, which infers data dependencies and processes queries. The former is contained in the workflow engine; the latter can be outside. Although our system is based on the Kepler scientific workflow management system [9], our approach is general and can be extended to other systems.

2 Background

2.1 The Kepler Scientific Workflow Management System

The Kepler system [9] is an open source application to provide generic solutions to scientific workflows. In Kepler, a workflow consists of a set of "nodes" (called *actors*), which represent components or tasks, and a set of "edges" (called *dataflow connections*), which connect actors. Actors have *input ports* and *output ports* that provide the communication interfaces to other actors. Actors communicate by passing *data tokens* (called *token* for short) between their ports. Each token is unique in the whole workflow. A unique feature of Kepler is that the overall execution and component interactions are coordinated by a separate component called *director* instead of implicitly defined in actors. Kepler provides a variety of directors that implement different computation models. In the process network model, each actor is an independent process or thread, and each dataflow connection is an asynchronous and unidirectional channel with unbounded buffers. Such scientific workflows execute in a pipelined fashion; our atomicity and provenance System is based on such a pipelined workflow model.

2.2 A Scientific Workflow in Biology

We implemented a scientific workflow in Kepler for a biological simulation project which analyzes the response of male worms to pheromone. The movement of a male worm is affected by chemical stimuli produced by female worms. Fig. 1 shows the workflow in Kepler. The actors `SampleFactory`, `EnvironmentFactory`, and `ModelFactory` provide parameters for simulations of male worms, environment, and their interactions, respectively. The actor `Simulation` repeatedly calculates the movement of worms over a time interval and the dispersion of the chemical. The actor `ImageDisplay` is used to show the result. The actor `StatisticalAnalysis` analyzes the simulations.

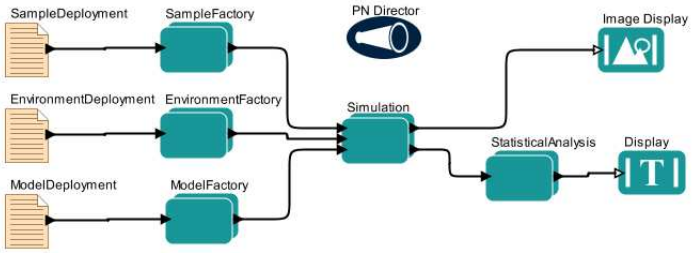

Fig. 1. A biological simulation scientific workflow in Kepler.

3 The Atomicity Management Subsystem

We first define a formal model for scientific workflows adapted from the Kepler system [9] introduced in Section 2.1.

This paper makes some assumptions about the scientific workflows that we analyze. First, each actor is "white", *i.e.*, data dependencies between input tokens and output tokens are observable. Second, message-send-response relationships between actors and services are known. Third, each retriable Web service is modeled as a local actor (this is how it is done by Kepler), which calls the remote Web service on behalf of the user. Thus, the execution of all tasks are performed in a local machine except the execution of Web or Grid Services.

A workflow $W = \langle A, E \rangle$ consists of a set A of actors, and a set E of dataflow connections. Each actor $a \in A$ has a set of associated data ports, each of which is either an input or output port. A dataflow connection bridges a set of output ports with a set of input ports.

3.1 Round and Data Dependency

In our atomicity model, a workflow execution invokes a series of actors to run. Each actor maintains a state which stores intermediate results computed from previous input tokens. A state indicates some data dependencies between the output tokens and the input tokens.

For example, Fig. 2 shows how the actors in Fig. 1 consume and produce tokens, and the data dependencies between tokens (which is shown in Fig. 2(d)). In Fig. 2(a), actor `SF` (*i.e.*, `SampleFactory`) consumes tokens `f1` (number of males) and `f2` (parameters for males), then produces token `s1` (a sample of males). When `SF` calls `reset()`, its state is flushed. Then `SF` consumes tokens

`f3` (number of females) and `f4` (parameters for females), and produces token `s2` (a sample of females). Actors `EnvironmentFactory` and `ModelFactory` work similarly, which are not shown. In Fig. 2(b), actor `S` (*i.e.*, `Simulation`) consumes `s1`, `s2`, `e1` (a set of environment parameters) and `m1` (interaction model), saves `s1` and `s2` into its state, then produces `a1` (a result). Next, `S` consumes `e2` then produces `a2`. Before the next simulation starts, `reset()` is called to flush the state. In Fig. 2(c), actor `A` (*i.e.*, `StatisticalAnalysis`) produces an analysis result for each simulation. In the meanwhile, it saves the intermediate results in its state and finally performs a full analysis based on its state. This procedure continues after `reset()` is called.

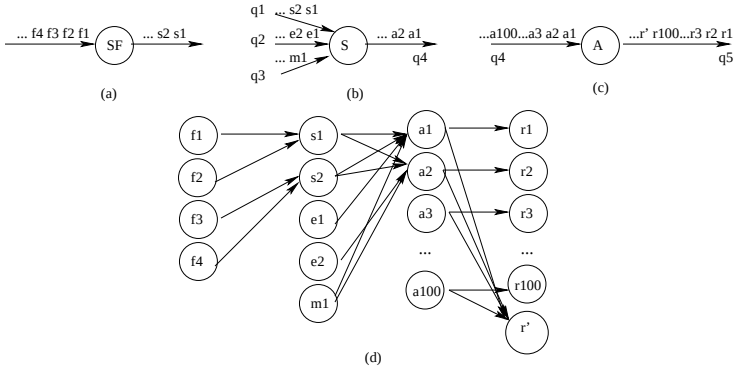

Fig. 2. Actors and data dependencies between tokens.

A *round* on an actor is the whole events that happen between two consecutive (*i.e.*, no other reset events in the middle) reset events. Each round has a unique identifier in the workflow. Thus, an invocation of a workflow contains a series of actor invocations; each invocation of an actor contains one or more rounds. Round is decided by each actor itself. When an actor calls `reset()`, it tells the workflow engine that the current round has completed. The call of `reset()` is a non-blocking operation. A reset event terminates the current round of data dependencies, and starts a new round of data dependencies. For each output token in a round, we assume that the actor can tell what input tokens that it depends on. Note that these dependent tokens might be some of the input tokens read so far (not the whole), as shown in Fig. 2(b), `a2` does not depend on `e1`. For a round $a.r$ on an actor a, let $input(a.r)$ and $output(a.r)$ denote its input and output tokens, respectively.

For two tokens t_1 and t_2, if t_2 is computed from t_1, we call t_2 *depends* on t_1, denoted $t_1 \rightarrow t_2$. For two rounds $a.r$ and $a'.r'$, if $\exists t.(t \in output(a.r) \wedge t \in input(a'.r') \wedge a \neq a')$, *i.e.*, $a'.r'$ consumes the tokens produced by $a.r$, we call $a'.r'$ *depends* on $a.r$, denoted $a.r \Rightarrow a'.r'$. Data dependencies are transitive: for token dependencies, if $t_1 \rightarrow t_2$ and $t_2 \rightarrow t_3$, then $t_1 \rightarrow t_3$; for round dependencies, if $a.r \Rightarrow a'.r'$, $a'.r' \Rightarrow a''.r''$, and $a \neq a''$, then $a.r \Rightarrow a''.r''$. Note that we do not allow cyclic transitive data dependencies on rounds. It is assumed that the workflows do not contain cyclic dataflows.

Let *depd-ancestors*$(a.r) = \{a'.r' | a'.r' \Rightarrow a.r\}$ (*i.e.*, all rounds that a round $a.r$ depends on) and *depd-descendents*$(a.r) = \{a'.r' | a.r \Rightarrow a'.r'\}$ (*i.e.*, all rounds that depend on a round $a.r$). They can be easily computed from the log introduced in Section 4.

3.2 Commit and Abort

Formally, we define the *atomicity* of a round as follows: the execution of a round $a.r$ is atomic if either it and all the rounds that depend on $a.r$ run to completion or none of them has any effect. Thus, users do not need to explicitly define transaction boundaries as in business workflows and database systems. Atomicities in the whole workflow are ensured automatically by our atomicity management subsystem. Although the atomicity granularity is based on one "round" of execution of a task in this paper, the technique can be readily extended for various granularities.

For two rounds $a.r$ and $a'.r'$, and $a.r \Rightarrow a'.r'$, if $a'.r'$ consumes only some early output tokens of $a.r$, $a'.r'$ might finish by calling `reset()` even when $a.r$ is still running. Thus, "reset" does not mean "commit" of the round, because we have to rollback both $a.r$ and $a'.r'$ if $a.r$ fails. A round $a.r$ *commits* if $a.r$ has been reset and every round in *depd-ancestors*$(a.r)$ has committed. If *depd-ancestors*$(a.r)$ is empty, $a.r$ commits once it is reset. Intuitively, a reset event indicates the ending of the current round and the starting of the next round, and a commit event makes the results of the round be observable to the users. The left column of Fig. 3 shows how the atomicity management subsystem commits a round $a.r$. When a round $a.r$ calls `reset()`, the atomicity management subsystem writes a reset event in a log, then repeatedly checks the log to see whether all rounds that $a.r$ depends on have committed. If the commit condition is satisfied, it commits $a.r$ by writing a commit event in the log.

```
Commit algorithm for a round a.r
  while (a.r has not been reset) continue;
  while (true)
    boolean toCommit = true;
    for all a'.r' ∈ depd-ancestors(a.r)
      if (a'.r' has not committed)
        toCommit = false;
    if (toCommit)
      commit(a.r);
      return;
```

```
Abort algorithm for a round a.r
  if (a.r has already committed)
    print("cannot abort.");
    return;
  Stop the execution of a.r if running;
  while (true)
    boolean toAbort = true;
    for all a'.r' ∈ depd-descendents(a.r)
      if (a'.r' has not aborted)
        toAbort = false;
    if (toAbort)
      for all t ∈ output(a.r)
        getRecoveryQueue(t).¬enq(t);
      for all t ∈ input(a.r)
        getRecoveryQueue(t).¬deq(t);
      abort(a.r);
      return;
```

Fig. 3. Commit algorithm and abort algorithm for a round $a.r$.

In our system, each dataflow connection is modeled and implemented as an extended recoverable queue adapted from [1]. An extended recoverable queue is a reliable and fault-tolerant queue which supports the following operations: *enqueue* pushes a token at the head; *dequeue* removes a token from the end and returns the token; $\neg enq$ undoes the operation of enqueue, *i.e.*, deletes an

enqueued token; $\neg deq$ undoes the operation of dequeue, *i.e.*, recovers a token that has been dequeued. After a round commits, its associated enqueue and dequeue operations cannot be undone.

When the atomicity management subsystem detects crashing of a round $a.r$, it will send *failure* messages to all actors that execute rounds in *depd-descendents*($a.r$) to abort the corresponding rounds, which are not necessarily the on-going rounds. A round $a.r$ *aborts* if all rounds in *depd-descendents*($a.r$) have aborted. The abort of a round will delete all output tokens, then recover all input tokens. Note that the "failure" event occurs only on the actor where $a.r$ runs, and "abort" events occur on each actor in *depd-descendents*($a.r$) and $a.r$ itself. The right column of Fig. 3 shows how the atomicity management subsystem aborts a round $a.r$. The atomicity management subsystem first checks whether $a.r$ has already committed, if not, tells the actor to stop the execution of $a.r$ if it is still running. Then, repeatedly check the log to see whether all rounds that depend on $a.r$ have aborted. During the abortion, the atomicity management subsystem looks up the log to find the corresponding recoverable queue for a given token t (*i.e.*, by calling `getRecoveryQueue`(t)); then it commands the recoverable queue to undo the previous operations. Finally, it writes an abort event in the log.

One optimization for the abort algorithm is: if both $a.r$ and $a'.r'$ are going to abort and $a.r \Rightarrow a'.r'$, during aborting $a'.r'$, we do not need to recover the tokens that are the output of $a.r$ and input of $a'.r'$ because they will be deleted again during aborting $a.r$.

4 The Event Log

Our atomicity & provenance system records the following events for supporting atomicity: *enqueue* (*enq*) a token; the counter-operation of *enq*, *i.e.*, $\neg enq$; *dequeue* (*deq*) a token; the counter-operation of *deq*, *i.e.*, $\neg deq$; *reset* (*rst*) a state; *failure* of an actor; *commit* (*cmt*) a round; and *abort* (*abt*) a round. These events are stored in a sequential event log. Each row in an event log contains: *event identifer*; *time stamp*; *workflow identifier*; *round identifier* (which contains actor identifier); *queue identifer*, if the event is an *enq*, *deq*, $\neg enq$, or $\neg deq$ operation; *event type*, which is one of event types listed above; *token identifier*, if the event is related with a token (such as enqueue or dequeue); and *dependent tokens*, which denote all source tokens used for producing the current token, if the event produces a token.

Fig. 4 shows a part of the log file for a run of the workflow in Fig. 2. The left column shows an aborted workflow run. Round `S`.r first dequeues s_1, s_2, e_1, and m_1 from queues q_1, q_1, q_2, and q_3, respectively. `S`.r then enqueues a_1 (which is produced by `S` based on s_1, s_2, e_1, and m_1) into q_4. After round `A`.r dequeues a_1 from q_4, `S`.r crashes. Thus, we first abort `A`.r by recovering a_1, then abort `S`.r by deleting a_1 and recovering m_1, e_1, s_2, and s_1. The right column shows a successful run, where `A`.r does not commit until `S`.r commits.

5 The Provenance Subsystem

Based on the event log, we can build *token dependency graph*, *object dependency graph*, *round dependency graph* as in [2], and *token usage graph*, which are represented as directed acyclic graphs (DAG). The event log produced by our system

evt	tm	wf	rnd	que	type	tok	depdToks
01	-	-	S.r	q_1	deq	s_1	-
02	-	-	S.r	q_1	deq	s_2	-
03	-	-	S.r	q_2	deq	e_1	-
04	-	-	S.r	q_3	deq	m_1	-
05	-	-	S.r	q_4	enq	a_1	$\{s_1, s_2, e_1, m_1\}$
06	-	-	A.r	q_4	deq	a_1	-
07	-	-	S.r	-	$fail$	-	-
08	-	-	A.r	q_4	$\neg deq$	a_1	-
09	-	-	A.r	-	abt	-	-
10	-	-	S.r	q_4	$\neg enq$	a_1	-
11	-	-	S.r	q_3	$\neg deq$	m_1	-
12	-	-	S.r	q_2	$\neg deq$	e_1	-
13	-	-	S.r	q_1	$\neg deq$	s_2	-
14	-	-	S.r	q_1	$\neg deq$	s_1	-
15	-	-	S.r	-	abt	-	-

evt	tm	wf	rnd	que	type	tok	depdToks
01	-	-	S.r	q_1	deq	s_1	-
02	-	-	S.r	q_1	deq	s_2	-
03	-	-	S.r	q_2	deq	e_1	-
04	-	-	S.r	q_3	deq	m_1	-
05	-	-	S.r	q_4	enq	a_1	$\{s_1, s_2, e_1, m_1\}$
06	-	-	A.r	q_4	deq	a_1	-
07	-	-	A.r	q_5	enq	r_1	$\{a_1\}$
08	-	-	S.r	q_2	deq	e_2	-
09	-	-	S.r	q_4	enq	a_2	$\{s_1, s_2, e_2, m_1\}$
10	-	-	S.r	-	rst	-	-
11	-	-	A.r	q_4	deq	a_2	-
12	-	-	S.r	-	cmt	-	-
13	-	-	A.r	q_5	enq	r_2	$\{a_2\}$
14	...	...	...	...	...	...	...
15	-	-	A.r	q_5	enq	r'	$\{a_1, ..., a_{100}\}$
16	-	-	A.r	-	rst	-	-
17	-	-	A.r	-	cmt	-	-

Fig. 4. A log for an execution of the workflow in Fig. 2.

contains all information of the event log in [2]. Therefore, our system can support all provenance queries listed in [2]. In addition, our system can support the atomicity and failure related queries, which are illustrated in the following examples.

- **What actors ever aborted rounds?**
  $q_1 := \{a | e \in log(\tau) \wedge type(e) = abort \wedge actor(e) = a\}$,
  where the expression $e \in log(\tau)$ selects an event from the log τ, the expression $type(e) = abort$ checks that the event is an abort, and the expression $actor(e) = a$ obtains the actor that executes the event.
- **When a round $a.r$ runs, what actors simultaneously execute the rounds that depend on $a.r$?**
  $q_2(a.r) := \{a'.r' | e \in log(\tau) \wedge a'.r' \in$ *depd-descendents*$(a.r) \wedge round(e) = a'.r' \wedge time(e) <$ *reset-time*$(a.r)\}$, where *reset-time*$(a.r)$ denotes the time when $a.r$ is reset, which is easily obtained based on the log.

6 Related Work

In recent years, scientific workflows have gained great momentum due to their roles in e-Science and cyberinfrastructure applications. There are a plethora of scientific workflows covering a wide range of scientific disciplines. A survey of various approaches for building and executing workflows on the Grid has been presented by Yu and Buyyaby [12].

Bowers et al. [2] propose the Read-Write-State-Reset (RWS) provenance model for pipelined scientific workflows within the Kepler framework [9]. The RWS model assumes that each output token depends on all tokens input so far in the current round, whereas our model refines this by assuming actors can tell what input tokens each output token depends on.

Although several provenance models [7, 11, 2, 4, 10] have been proposed for scientific workflows, there has been no work on the provenance system that supports the notion of atomicity.

Finally, although atomicity is a well studied topic in the context of databases in transaction processing and business workflows, there has been no work on atomicity in the context of "dataflows" and "pipelined execution" in scientific workflows. The read committed assumption that existing atomicity techniques are based on does not hold in pipelined scientific workflows, where both task parallelism and pipelined parallelism are present.

7 Conclusions and Future Work

We have proposed an architecture for scientific workflow management systems that supports both provenance and atomicity. We have shown that, while our atomicity system can support the notion of atomicity, currently at the round level that does not contain cyclic transitive data dependencies, our provenance system has added value to existing provenance systems as we support atomicity and failure related queries.

In the future, we will extend current atomicity and provenance models to various granularities of atomicity and for different models of computations. We will also investigate the atomicity problem for multilevel, distributed, parallel, and heterogeneous scientific workflows.

References

1. P. A. Bernstein, M. Hsu, and B. Mann. Implementing recoverable requests using queues. In *Proc. of the 1990 ACM SIGMOD international conference on Management of data*, pages 112–122. ACM Press, 1990.
2. S. Bowers, T. McPhillips, B. Ludascher, S. Cohen, and S. B. Davidson. A model for user-oriented data provenance in pipelined scientific workflows. In *Proc. of the International Provenance and Annotation Workshop (IPAW'06)*, Chicago, Illinois, USA, May 2006.
3. P. Buneman, S. Khanna, and W.-C. Tan. Why and where: A characterization of data provenance. *Proc. of the International Conference on Database Theory (ICDT)*, 1973:316–330, 2001.
4. S. Cohen, S. C. Boulakia, and S. B. Davidson. Towards a model of provenance and user views in scientific workflows. In *Data Integration in the Life Sciences*, pages 264–279, 2006.
5. W. Derks, J. Dehnert, P. Grefen, and W. Jonker. Customized atomicity specification for transactional workflows. In *Proc. of the Third International Symposium on Cooperative Database Systems for Advanced Applications(CODAS'01)*, pages 140–147. IEEE Computer Society Press, 2001.
6. H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259. ACM Press, 1987.
7. P. Groth, S. Miles, W. Fang, S. C. Wong, K.-P. Zauner, and L. Moreau. Recording and using provenance in a protein compressibility experiment. In *Proc. of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC'05)*, Research Triangle Park, North Carolina, U.S.A., July 2005.
8. F. Leymann and D. Roller. *Production workflow: concepts and techniques*. Prentice Hall, 2000.
9. B. Ludascher, 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: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1039–1065, 2006.
10. S. Miles, P. Groth, M. Branco, and L. Moreau. The requirements of recording and using provenance in e-science experiments. *Journal of Grid Computing*, 2006.
11. Y. L. Simmhan, B. Plale, and D. Gannon. A framework for collecting provenance in data-centric scientific workflows. In *Proc. of the IEEE International Conference on Web Services (ICWS'06)*, pages 427–436, Washington, DC, USA, 2006.
12. J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Record*, 34(3):44–49, Sept. 2005.