A Provenance-Based Fault Tolerance Mechanism for Scientific Workflows

Daniel Crawl and Ilkay Altintas

San Diego Supercomputer Center, UCSD, 9500 Gilman Drive,
La Jolla, CA 92093 USA
{crawl,altintas}@sdsc.edu

Abstract. Capturing provenance information in scientific workflows is not only useful for determining data-dependencies, but also for a wide range of queries including fault tolerance and usage statistics. As collaborative scientific workflow environments provide users with reusable shared workflows, collection and usage of provenance data in a generic way that could serve multiple data and computational models become vital. This paper presents a method for capturing data value- and control- dependencies for provenance information collection in the Kepler scientific workflow system. It also describes how the collected information based on these dependencies could be used for a fault tolerance framework in different models of computation.

1 Introduction and Background

Scientific workflows provide many advantages to the scientific community including provenance support. The lifecycle of scientific workflow provenance starts with workflow design and execution. The collected information can be used for evaluation of the results as well as for mining different patterns during workflow design. Different workflow users need information about different phases of the workflow. These concepts are shown in Figure 1.

Collection of Provenance Information. Provenance collection related to a scientific workflow is three-fold. Firstly, since workflow developers often change a workflow while experimenting with different computational tools and multiple scientific datasets, provenance recording can start during the design phase of the workflow. Capturing these user actions is important since it records what did and did not work, as well as how the workflow developer came up with the final workflow. The collection of provenance information continues during the experiment preparation (parameter binding) and execution of the workflow. The third aspect of collecting scientific workflow provenance that is often ignored is collection after the workflow results are published. One can verify the scientific impact of the workflow based on the citations for these results and statistics on how they are used.

Usages of Provenance Information. The collected provenance data is useful in many contexts, such as querying input/output associations, verifying results, etc. Different types of provenance information analysis based on the design and one or more runs of

J. Freire, D. Koop, and L. Moreau (Eds.): IPAW 2008, LNCS 5272, pp. 152–159, 2008.
© Springer-Verlag Berlin Heidelberg 2008

the same workflow are listed in Figure 1. The collected information on the workflow design could be used both to analyze it and to visualize the evolution of the workflow as demonstrated in the Vistrails system [1]. In addition, provenance information on data-dependencies could be used for smart reruns and fault tolerance.

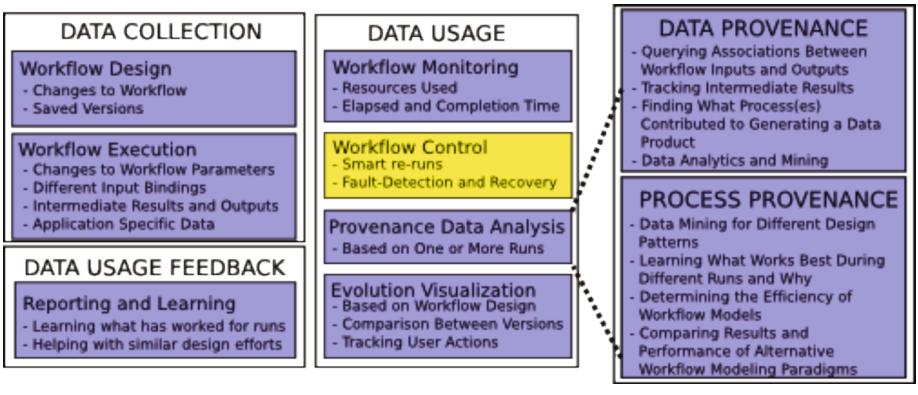

Fig. 1. Different usages of provenance data

Users of Provenance Information. Users of provenance data include the workflow developer and workflow user, scientific dashboards that help execute and monitor a workflow, and interfaces that use this data. To best serve different users, provenance recorders should provide for customized data collection through parametric interfaces.

This paper presents a method for capturing data-dependencies for provenance in the Kepler scientific workflow system, and describes how these dependencies could be used for a fault tolerance framework. Our fault-tolerance model targets different users of a workflow system. It provides mechanisms to detect errors that would be useful for workflow developers, workflow users, and programming interfaces that execute a given workflow. The level of information that needs to be collected for error-recovery is different for every workflow. Further, workflow systems could execute multiple models of computation (MoC), and the fault tolerance mechanism must work with each. To the best of our knowledge, no scientific workflow system supports these requirements for failure-detection and recovery.

In the rest of this paper, we explain a provenance collection approach in the Kepler scientific workflow system [2] and show how the collected information could be used in a fault-tolerance system that supports the requirements mentioned in the previous paragraph. We demonstrate this approach in a scientific workflow example using a part of the GEON LiDAR Workflow (GLW) [3].

2 The Kepler Provenance Framework

The Kepler scientific workflow system [2] is developed by a cross-project collaboration to serve scientists from different disciplines. Kepler provides a workflow environment in which scientists can design and execute workflows through a user interface or in batch mode from other applications. A *Provenance Recorder* that has

plug-in interfaces for new data models, metadata formats and storage destinations was designed to serve the multi-disciplinary requirements of the broad user community. The *Kepler Provenance Framework* (KPR) was presented in [4]. An extended architecture that allows for binding different data models to KPR, collection of application-specific provenance data and using results through a dashboard has been created [5]. The center of this architecture is Provenance Store: a database providing physical storage and an API to access the database. The API has three components: (1) Kepler, its actors, and external scripts use a Recording API to collect and save provenance information; (2) a Query API provides different query capabilities for dashboards, and query actors in Kepler; and (3) a Management API.

KPR uses separation of concerns principle to work with different MoCs [4, 6]. Kepler workflows are composed of a linked set of *Actor*s executing under MoC. Actors encapsulate parameterized actions and communicate between themselves by sending *Token*s, which encapsulate data or messages, to other actors through one or more output ports. Ports that receive tokens are called input ports. MoCs specify what flows as tokens between actors' input and output ports, e.g., data or messages, how the communication between the actors is achieved, when actors fire, and when the workflow can stop execution. A Kepler workflow could have different MoCs in *sub-workflows* called *Composite Actors.* KPR is a separate entity in the workflow and records provenance by communicating with the execution engine.; the information recorded depends on the MoC semantics. This can only be achieved using a data model that matches the set of observables about workflow run in a particular MoC.

2.1 Classifying Data-Dependencies

The KPR records workflow assertions and observables, including data-dependencies: data written by an actor may depend on some combination of previously read data. We categorize data-dependencies between output and input data as either *value-dependencies* or *control-dependencies*. A value-dependency occurs when an output data's *value* depends on the *value* of previously read data. For example, consider the two actors show in Figure 2. The Filter actor outputs the previously read token if the value is above a threshold. Each token written by this actor has a single value-dependency: the previously read input token. In Figure 2(a), this actor reads two input tokens T1 and T2. Only the value in T2 is above the threshold and is output in T3. The value-dependency for T3 is T2.

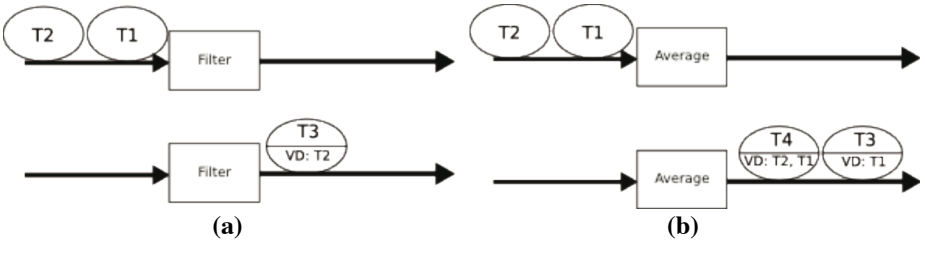

Fig. 2. Input and resulting output tokens for (a) Filter and (b) Average actors. The output tokens contain a list of value-dependencies (VD).

The Average actor outputs an average of all previously seen inputs. In this case, each output token is value-dependent on all input tokens. In Figure 2(b) the actor first reads T1, then outputs T3, which is only value-dependent on T1. Next, the actor reads T2 and outputs T4, which is value-dependent on both T2 and T1.

A control-dependency occurs when the arrival of input data causes the actor to execute and subsequently produce output data. As described above, the MoC determines when actors execute; in several the execution schedule is based on when input data are present. Many actors therefore use an input "trigger" port to determine when to execute. Since data read from this port are discarded, i.e., the *value* is not used by the actor to produce output data, the distinction between value- and control-dependencies is important for many use cases of provenance.

2.2 Recording Data-Dependencies

The KPR provides methods for recording data-dependencies, either automatically by the runtime system or manually by the actor developer.

Our method of automatically tracking dependencies is inspired by taint propagation, a technique commonly used to detect security vulnerabilities in applications [7,8]. For example, Perl's "taint mode" prohibits any data outside the application to affect something else outside the program [9]. Perl marks each variable assigned from an external input, e.g., files, web services, etc., as tainted and propagates taint to other variables when they are used in expressions with tainted variables.

To capture value-dependencies during workflow execution, the ids of tokens read by an actor are propagated "through the actor" to the produced tokens. (All tokens created during a single workflow execution are assigned a unique identifier). A token is a container for base classes such as Integer or String, and each is instrumented with a list of token ids on which its value depends. The list is created when the object is extracted from a newly read input token and updated when the object is used in an expression. For example, the assignment operator replaces the destination object's dependencies with those in the source, and the addition operator adds the source dependencies to the destination's list.

We also provide an API to capture dependencies for situations in which automatically propagating dependencies is impossible. This can occur, for example, when an actor reads from an external data source such as a web service.

In addition to value-dependencies, control-dependencies may exist between an output token and one or more previously read input tokens. KPR records a control-dependency between each output produced and each input read by an actor during the same firing cycle. A value-dependency between an output and input implies a control-dependency between the same tokens. The converse, however, is not always true. For example, consider the Ramp actor. It has two inputs: "trigger" and "step". Data must be available for both before the actor can execute. The value read in the step input increases or decreases the actor's output value. A control-dependency exists for an output token and the previously read trigger and step inputs. However, only the value in the step input is used to calculate the output; a data-dependency does not exist between the output and the trigger input.

3 Scientific Workflow Doctor: Using Provenance Data for Fault Tolerance

This section describes a fault tolerance framework using the data-dependencies recorded by the KPR. Scientific workflows commonly access a diverse set of resources such as, databases, and file systems. A scientific workflow system therefore must provide mechanisms to gracefully handle resource failures. Further, these mechanisms should be compatible with advanced modeling constructs, such as data-dependent routing, loops, and parallel processing.

Fault tolerance is provided with a composite actor called *Checkpoint*. When a sub-workflow within a *Checkpoint* produces an error event, all execution within the *Checkpoint* is stopped. *Checkpoint* handles the error itself, or passes it up the workflow hierarchy. This is similar to exception handling in text-based programming languages.

Workflow errors are detected and signaled with user-defined expressions called *port conditions*. A *port condition* can be specified for any port and is evaluated when the actor reads from the input port or writes to the output port. If the *port condition* evaluates to false, an error event is signaled. *Port conditions* are analogous to pre- and post-conditions in procedural languages and provide great flexibility for the workflow designer since they may be attached to any port in the workflow. A *port condition* uses the incoming or outgoing token's value, along with mathematical and logical operators. For example, it could check if a numerical token was above a threshold.

Actors also signal errors during workflow execution. An error API is provided for developers to generate error events based on actor-specific conditions, such as when a web service actor's request times-out.

A *Checkpoint* composite actor contains a primary sub-workflow and optionally alternate sub-workflow(s). Data read by the *Checkpoint's* input ports are first passed to the primary sub-workflow. When an error occurs in the primary sub-workflow, *Checkpoint* either re-executes the primary, or runs an alternate sub-workflow. The maximum number of times to retry the primary or an alternate sub-workflow is configurable. Once the retry limit is exceeded, the error is sent up the workflow hierarchy to the nearest enclosing *Checkpoint*.

When *Checkpoint* re-executes a sub-workflow, it resends all data read by the sub-workflow up to and including those that led to the error. The data to be resent are queried from *data-dependencies stored in the provenance database*. An actor generates an error based on the values of one or more input data. These data were written by upstream actors, which in turn created them from a set of input data. *Checkpoint* follows the data-dependencies, starting at the error, back to the data read by the *Checkpoint*'s input ports. These data tokens are sent again to the sub-workflow along with any data tokens received after it in their original order.

Figure 3 shows an example workflow containing a *Checkpoint* actor, based on the GLW [3], which allows geoscientists to analyze and interpolate Light Distance and Ranging (LiDAR) datasets. A geoscientist first selects a region to be analyzed from a web portal. The workflow retrieves the LiDAR point cloud matching the selected region from a database, which is then interpolated using GRASS [10]. GRASS outputs an ASCII grid, which must be converted to binary using Feature Manipulation

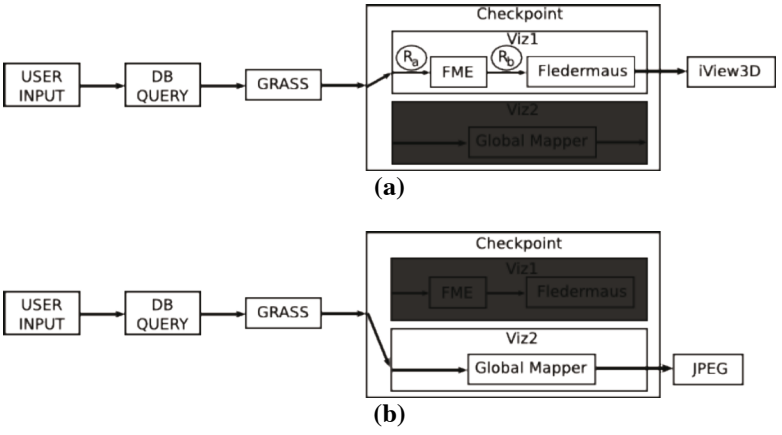

Fig. 3. GLW with fault tolerance. (a) During normal operation, the workflow executes *Viz1*. (b) If FME or Fledermaus in *Viz1* fail, Global Mapper creates a 2D image.

Engine[1] (FME) before Fledermaus[2] can read it. Fledermaus outputs the data into a format that the geoscientist can then load in an interactive 3D viewer. FME and Fledermaus are executed in the sub-workflow *Viz1*.

Since both FME and Fledermaus are web services, they may not always be available. If one fails, we would still like to provide a visualization of the selected LiDAR region. An alternative imaging tool is Global Mapper[3], shown in *Viz2*, which converts the ASCII grid data to a JPEG image.

The *Checkpoint* actor executes *Viz1*, unless FME or Fledermaus cannot be reached. When this occurs an error is signaled and *Checkpoint* runs *Viz2*. Any regions to be processed by FME or Fledermaus are instead processed by Global Mapper. R_a and R_b in Figure 3(a) represent the locations of these regions in the workflow. R_a has not been converted to binary by FME so can be sent directly to Global Mapper. However, R_b has been converted. In this case, *Checkpoint* uses the data-dependencies stored in the provenance database to find the ASCII data that led to R_b. The ASCII data can then be processed by Global Mapper. Since the datasets processed can be very large, references are passed between actors and saved in the provenance database instead of the actual data.

4 Related Work

Provenance in scientific workflow systems has become a major research track. Data-dependencies are analyzed and mapped into a data model for different systems [11,12]. Information collected on these is used to answer users' queries on different workflow aspects. Data-dependencies have also been used for a smart re-run system

[1] A GIS data conversion system (http://www.safe.com/).
[2] An interactive 3D visualization system (http://www.ivs3d.com/products/fledermaus/).
[3] A GIS visualization system (http://www.globalmapper.com/).

in Kepler [4]. In the Virtual Data System, provenance is used to reproduce data products, and can be queried with application-specific semantics [12].

Techniques have been proposed to record provenance about the computations occurring inside actors. The registry system of Wootten *et al.* records assertions of internal actor state executing a service-oriented architecture [13]. However, no method is given for automatically adding state assertions to actors. The RWS approach [14] annotates actors to signal whenever they reset to an initial state. Unlike our solution, this requires modifying each actor since the reset event is sent on actor-specific conditions. To our knowledge, no workflow provenance system distinguishes between value- and control-dependencies.

Scientific workflow systems commonly provide fault tolerance mechanisms [15], but most are not used in combination with advanced MoC. While ASKALON [16] allows constructs such as parallel loops and conditional statements, it does not support user-definable exceptions or fault-recovery for actor errors. Similar to our approach, Bowers *et al.* [17] propose embedding the primary and alternate sub-workflows in a control-flow template. However, these templates are directed by a finite state machine, which cannot be used to execute process networks [6]. The Ptolemy backtracking system [18] provides an incremental checkpoint and rollback mechanism. This is complementary to our approach, which deals primarily with stateless actors and provides user-definable fault-detection and recovery. Similar to *port conditions*, Karajan [19] performs matching on user-defined regular expressions attached to input ports. However, there is no mechanism to retry previously executed parts of the workflow.

5 Conclusions and Future Work

This paper discusses methods to identify value- and control- dependencies in the Kepler Provenance Framework. Additionally, we describe how the collected data-dependencies can be used to provide failure recovery in a scientific workflow system.

We will extend our fault tolerance system in several ways. Combining it with the Ptolemy II backtracking system will allow stateful actors to restart. Further, we are building more expressive port conditions by adding conceptual semantics. This will allow conditions such as “all Celsius temperatures (read or written by actors) must be above 5 degrees”. We are also investigating other techniques to capture dependency information.

Acknowledgements

The work in this paper is supported by DOE SciDac Award No. DE-FC02-07ER25811 for SDM Center, NSF Award No. DBI 0619060 for REAP, and NSF Award OCI-0722079 for Kepler CORE.

References

1. Freire, J., Silva, C., Callahan, S., Santos, E., Scheidegger, C., Vo, H.: Managing Rapidly-Evolving Scientific Workflows. In: Proceedings of International Provenance and Annotation Workshop, pp. 10–18 (2006)

2. Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger-Frank, E., Jones, M., Lee, E., Tao, J., Zhao, Y.: Scientific Workflow Management and the Kepler System. Special Issue: Workflow in Grid Systems. Concurrency and Computation: Practice & Experience 18(10), 1039–1065 (2006)
3. Jaeger-Frank, E., Crosby, C., Memon, A., Nandigam, V., Arrowsmith, J., Conner, J., Altintas, I., Baru, C.: A Three-Tier Architecture for LiDAR Interpolation and Analysis. In: Proceedings of International Workshop on Workflow Systems in e-Science, pp. 920–927 (2006)
4. Altintas, I., Barney, O., Jaeger-Frank, E.: Provenance Collection Support in the Kepler Scientific Workflow System. In: Proceedings of International Provenance and Annotation Workshop, pp. 118–132 (2006)
5. Altintas, I., et al.: Provenance in Kepler-based Scientific Workflow Systems. In: Microsoft e-Science Workshop, poster (2007)
6. Goderis, A., Brooks, C., Altintas, I., Lee, E.A., Goble, C.: Composing Different Models of Computation in Kepler and Ptolemy II. In: Proceedings of the International Conference on Computational Science (2007)
7. Myers, A.: JFlow: practical mostly-static information flow control. In: Proceedings Symposium on Principles of Programming Languages, pp. 228–241 (1999)
8. Haldar, V., Chandra, D., Franz, M.: Dynamic Taint Propagation for Java. In: Proceedings of Computer Security Applications Conference, pp. 303–311 (2005)
9. Wall, L., Christiansen, T., Orwant, J.: Programming Perl, 3rd edn. O'Reilly, Sebastopol
10. Mitasova, H., Mitas, L., Harmon, R.: Simultaneous spline interpolation and topographic analysis for lidar elevation data: methods for open source GIS. IEEE GRSL 2(4), 375–379 (2005)
11. Miles, S., Groth, P., Branco, M., Moreau, L.: The Requirements of Recording and Using Provenance in e-Science Experiments. Journal of Grid Computing 5(1), 1–25 (2007)
12. Zhao, Y., Wilde, M., Foster, I.: Applying the Virtual Data Provenance Model. In: Proceedings of International Provenance and Annotation Workshop, pp. 148–161 (2006)
13. Wootten, I., Rana, O., Rajbhandari, S.: Recording Actor State in Scientific Workflows. In: Proceedings of International Provenance and Annotation Workshop, pp. 109–117 (2006)
14. Ludäscher, B., Podhorszki, N., Altintas, I., Bowers, S., McPhillips, T.: From Computation Models to Models of Provenance: The RWS Approach. Concurrency and Computation: Practice & Experience 2(5), 507–518 (2007)
15. Plankensteiner, K., Prodan, R., Fahringer, T., Kertesz, A., Kacsuk, P.: Fault-tolerant behavior in state-of-the-art Grid Workflow Management Systems. TR-0091, CoreGRID (2007)
16. Fahringer, T., Prodan, R., Duan, R., Nerieri, F., Podlipnig, S., Qin, J., Siddiqui, M., Truong, H., Villazon, A., Wieczorek, M.: ASKALON: A Grid Application Development and Computing Environment. In: Proceedings of International Workshop on Grid Computing (2005)
17. Bowers, S., Ludäscher, B., Ngu, A., Critchlow, T.: Enabling Scientific Workflow Reuse through Structured Composition of Dataflow and Control-Flow. In: IEEE Workshop on Workflow and Data Flow for Scientific Applications (2006)
18. Feng, T.H., Lee, E.A.: Real-Time Distributed Discrete-Event Execution with Fault Tolerance. In: Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (2008)
19. Laszewski, G., Hategan, M.: Workflow Concepts of the Java CoG Kit. Journal of Grid Computing 3(3-4), 239–258 (2005)