Experience in Using a Process Language to Define Scientific Workflow and Generate Dataset Provenance

Leon J. Osterweil
Dept. Computer Science
Univ. of Massachusetts
Amherst, MA 01003 USA
ljo@cs.umass.edu

Lori A. Clarke
Dept. Computer Science
Univ. of Massachusetts
Amherst, MA 01003 USA
clarke@cs.umass.edu

Aaron M. Ellison
Harvard Forest
Harvard University
Petersham, MA 01366 USA
aellison@fas.harvard.edu

Rodion Podorozhny
Computer Sci. Dept.
Texas State University
San Marcos, TX 78666 USA
rp31@txstate.edu

Alexander Wise
Dept. Computer Science
Univ. of Massachusetts
Amherst, MA 01003 USA
wise@cs.umass.edu

Emery Boose
Harvard Forest
Harvard University
Petersham, MA 01366 USA
boose@fas.harvard.edu

Julian Hadley
Harvard Forest
Harvard University
Petersham, MA 01366 USA
jhadley@fas.harvard.edu

ABSTRACT
This paper describes our experiences in exploring the applicability of software engineering approaches to scientific data management problems. Specifically, this paper describes how process definition languages can be used to expedite production of scientific datasets as well as to generate documentation of their provenance. Our approach uses a process definition language that incorporates powerful semantics to encode scientific processes in the form of a Process Definition Graph (PDG). The paper describes how execution of the PDG-defined process can generate Dataset Derivation Graphs (DDGs), metadata that document how the scientific process developed each of its product datasets. The paper uses an example to show that scientific processes may be complex and to illustrate why some of the more powerful semantic features of the process definition language are useful in supporting clarity and conciseness in representing such processes. This work is similar in goals to work generally referred to as Scientific Workflow. The paper demonstrates the contribution that software engineering can make to this domain.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Foundations of Software Engineering'08, November 11–13, 2008, Atlanta, Georgia, USA.
Copyright 2008 ACM 1-58113-000-0/00/0008…$5.00.

Categories and Subject Descriptors
D.1.7 [Programming Techniques] Visual Programming, D.2.6 [Software Engineering] Programming Environments, D.3.3 [Programming Languages] Language Constructs and Features

General Terms
Documentation, Experimentation, Languages.

Keywords
Scientific Workflow, Continuous Process Improvement, Data Provenance.

1. INTRODUCTION
Scientific researchers devote considerable effort to the creation and management of data, producing collections of data, often called "datasets," that may be highly evolved. To support these efforts, scientists are increasingly turning to Scientific Workflow systems, which offer some support for invoking computational tools and managing the resulting datasets. From the software engineering perspective, it seems useful to view these scientific datasets as products of a distributed enterprise: input datasets may be stored and retrieved remotely, analytic services may be obtained from external sources, and the results of extensive computation are increasingly likely to be made directly accessible. The totality of data and capabilities produced and consumed in this way can thus be thought of as a scientific data processing enterprise. In this work we refer to it as an *analytic web* [1-3]. There is a need for strong support of this enterprise by supporting both the *production* of such datasets (a particular focus of current Scientific Workflow systems) and their *consumption* by those wanting to access them for their own subsequent scientific investigations (a need that currently seems less well served).

Support for production should consist of facilities for generating and storing new data items and datasets. Support for consumption should include annotating the data sets, and when necessary individual data items, with precise specifications of how the data item or dataset was created (*process provenance metadata*). Such metadata seems essential if other scientists are to use these results responsibly and is also essential for reproducibility, the very essence of scientific validation. An analytic web should thus support both generating and accessing such metadata, as well as the use of the process provenance metadata to support the reproduction of datasets and the generation of additional datasets.

In this paper we demonstrate that the desired annotations can be treated as attributes attached to nodes in a directed acyclic graph (DAG) created during the execution of a definition of the scientific process. We refer to this DAG as a Dataset Derivation Graph (DDG). We specify the scientific process with a graph that we refer to as a Process Derivation Graph (PDG). This paper shows that the language used to specify the PDG of a complex scientific process must incorporate a surprisingly challenging collection of semantic features in order to provide a clear and precise definition of the process. Finally, based on preliminary experiences with this approach, this paper suggests the desiderata for a system of tools to facilitate the production of the PDGs and DDGs that comprise analytic webs and to support analyses that should be used to guide their production and consumption.

Numerous Scientific Workflow projects address many of the issues raised here. A key objective of this paper, however, is to demonstrate that current Scientific Workflow technologies can be improved upon by drawing upon experiences and technologies developed to facilitate support for such key concepts as abstraction, exception management and concurrency control. In previous papers [1, 3] we presented to the ecological research community an indication of the promise of our approach. This paper discusses the challenges and research opportunities that this problem domain presents for software engineering research.

2. MOTIVATING EXAMPLE: A REALTIME HYDROLOGICAL SENSOR NETWORK

The Harvard Forest Long-Term Ecological Research (LTER) site is building a sensor network to measure the flow of water through small, forested watersheds. This real-time system will integrate ongoing meteorological, hydrological, eddy covariance, and tree physiological measurements. These measurements will enable study of variations in water flux caused by differences in topography, soils, vegetation, land use, and natural disturbance history. Frequent sampling (5-10 Hz) will enable study of water flux dynamics at a wide range of temporal scales, from minutes to days to years. Over time, the number, nature, and variety of sensors to be used in this research, and datasets resulting from it, will grow. The system is described in detail in [3].

Such systems create interesting system development and data management challenges. As the number and variety of sensor data streams grow, systems for acquiring their data must be developed. Some processing can be automated, but it must be integrated with human processing activities, that must keep pace with growth in data rates and varieties. Measurements may be delayed or corrupted for many reasons, necessitating complex contingency handling, and varied models to create substitutes for missing data.

To understand this better we describe briefly some essentials of hydrological modeling. Analysis of water flux is based on the mass balance equation:

$$dS = P - ET - Q$$

where P = precipitation, ET = evapotranspiration, Q = stream discharge, and dS = change in ecosystem water storage. The terms in this equation may represent rates that are instantaneous or integrated over fixed time intervals. Essential measurements are made at three sites, a meteorological station, an eddy flux tower, and a stream gauge. Because accurate and complete data are critically important, this system incorporates redundancy. Thus P is measured at two separate rain gauges ($P1$ and $P2$). ET is both measured directly and modeled using measured photosynthetically active radiation (*PAR*), and Q is both measured with a stream gauge and modeled using a simple linear reservoir model [4]. This redundancy requires dealing with inconsistencies in the redundant data. In addition some measurements might be missing or suspect (in which case interpolation and gap filling models create substitute values). While potentially fully automated, the system must also enable overall human oversight in real-time and must also support subsequent more leisurely human retrospective data analysis. Specifically, the system must integrate:

(1) A **real-time** sub-process to collect, analyze, and document data from the meteorological station, eddy flux tower, and stream gauge: This subprocess has to retrieve measurements every 30 minutes, do range checking, calculate best values from redundant sensors, create and apply models, choose between measured and modeled values, and calculate water storage change (dS).

(2) A **post-processing** sub-process to automatically update datasets after a fixed time period (e.g., one month): Experience has shown that models created in realtime can be improved upon by using both preceding and subsequent measurements, especially during periods of rapid ecosystem change.

(3) An **alternate measurement** sub-process to support further modifications to data items and datasets: For example, more accurate adjustments may become known too late for scheduled post-processing, original measurements may require correction for sensor drift, and missing or questionable measurements may need replacement by data from other sites. Because this activity will change some values, other (modeled) values may consequently have to be updated because of "ripple effects."

(4) A **new model** sub-process to support modifying or replacing the models used in the system. This may entail changing the model functionality, the parameters used, or the temporal range of data used to create modeled value(s). This sub-process helps scientists to construct, evaluate, and then promulgate new model(s), and the circumstances under which they should be used.

The first three of these subprocesses each produces datasets containing sequences of data items that represent either direct measurements or values that have been generated by the application of one or more models. Dataset consumers have a strong interest in being sure that datasets produced by these subprocesses will include at least the following:

• All original and alternate measurements, with appropriate metadata annotations (e.g., value missing, value out of range, or value computed by model).

• All models used to compute substituted data items, with documentation indicating which data items were processed by which models at what times.

• Estimates of P, ET, Q, and dS for each time period of interest (e.g., every 30 minutes), indicating what processing steps (e.g., model applications) were applied and in what sequence.

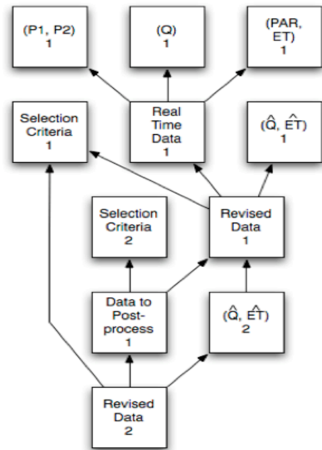

Figure 1. An excerpt of a dataset-derivation graph (DDG) for a small part of the water flux system. The boxes represent data items. Each arrow indicates that the data item at its head was used in deriving the data item at its tail (from [1]).

3. TECHNICAL APPROACH

Analytic Web: Analytic webs [2] are composite objects that integrate PDGs and DDGs that differ from one another, for example, because of different input datasets, different handling of different contingencies that may have arisen, or different choices of gap-filling models made by scientists.

Data Derivation Graph (DDG): A DDG documents the provenance of data items and datasets generated in executing a scientific process. We view a scientific process as an integrated system in which tools and humans function as operators, and data items and datasets are their operands. Intermediate and final outputs result from execution of this process. Each trace that produces a data item or dataset supports the creation of *process provenance metadata*. This, in turn, can support the reproduction and analysis of the data item or dataset, and provide evidence of its suitability for use in further scientific processes.

The DDG in Figure 1 depicts how data items have been derived from each other. Each box represents a data item, and each arrow indicates that the data item at the head was used to develop the data item at the tail. Thus, for example, the arrow from “Data to Post-Process 1” to “Selection Criteria 2” indicates that the latter was used to create the former. The arrow from “Data to Post-Process 1” to “Revised Data 1” indicates, further, that the derivation of “Data to Post-Process 1” required both “Selection Criteria 2” and “Revised Data 1” as inputs. Figure 1 also indicates that “Revised Data 1” was derived from three other data items, and one of them, “Real Time Data 1”, was itself derived from three additional data items. Figure 1 does not depict a correspondingly precise specification of the exact tool or process represented by each of the arrows, but this additional information is needed to provide a complete specification of the derivation of the data items. With these additional annotations the resulting structure becomes a DDG. In particular the DAG rooted at each box in Figure 1 provides the information needed to document the provenance of the data item represented by that box. Note that a DDG can incorporate different instances of a single type of a data item. Thus, for example, the DDG in Figure 1 incorporates “Revised Data 1” and “Revised Data 2”, and “Selection Criteria 1” as well as “Selection Criteria 2”[1]. Different instances of a single data item type occur frequently, for example, as a result of iteration of scientific processing loops. Accurate and unambiguous documentation of data item and dataset provenance requires that the data items generated by each iteration be clearly identified and distinguished from one another. Analytic web DDGs do this by indicating the way in which subsequent instances are derived from prior instances and from other data items.

Process Derivation Graph (PDG): While the DDG is a powerful device for recording the way in which data items and dataset instances have been derived, it only captures history and does not specify the general process and activity types by which data items and datasets are created. To do this, an analytic web uses a PDG, a definition of all possible executions of the scientific process. The DDG and PDG complement each other. The DDG provides a retrospective history of the development of all datasets and data items. It is a structure of instances—data and dataset instances and instances of the tools and processes used to create them. The PDG is prospective, indicating all possible ways that a process can be performed. Moreover, the PDG is a structure of types, the types of tools that can be applied to types of data items and datasets in order to create new instances of (possibly different) types.

Defining real scientific processes in a way that is both clear, concise, and precise raised some challenges that seem to us to be of interest and importance to software engineers. As noted above, Scientific Workflow systems currently address the problem of defining such processes. Most of these systems have suggested the use of data-flow graphs (DFGs) as vehicles for doing this. Indeed the Kepler project [5], in particular, has demonstrated that DFGs can be used to specify how data items can be developed through sequences of tool applications. The DFG representations are clear and easy to grasp in some cases. In addition, Kepler provides an

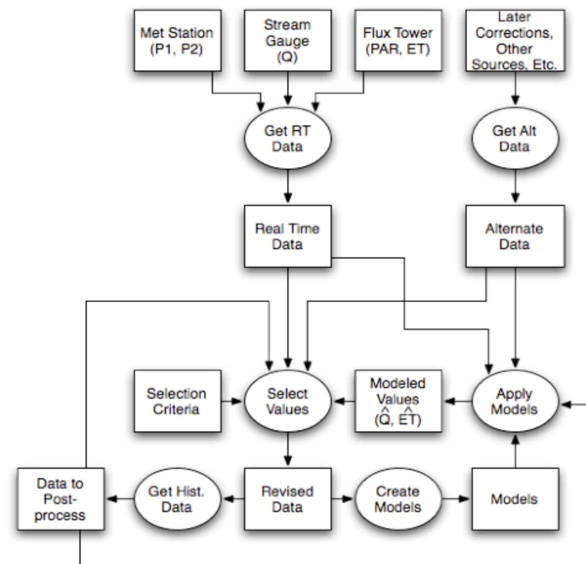

Figure 2. A data-flow graph for the water flux system.

[1] The actual instance annotation generated by an automated system such as we envisage would be system-generated and might contain additional useful information such as a timestamp.

impressive set of tools that support the development of DFGs and their viewing, editing, and evolution. On the other hand, our experience has indicated that the relatively restricted DFG semantics can make it a struggle to represent real scientific processes with the clarity, conciseness, and completeness that are often desired. Our experience has indicated that important process control flow aspects such as step coordination, parallel processing, multiple interrelated loops, exception handling, and loop iteration and termination conditions can help support the desired clarity, conciseness, and completeness. DFGs can represent these semantic features, but these representations can be cumbersome, thereby compromising clarity, and posing obstacles for process developers and scientists. Software engineering experience suggests that complete and precise definitions of complex systems can be facilitated by appropriately powerful language semantics. This suggests that software engineering concepts and technologies might find an additional vehicle for evaluation through their application in this domain. A specific example is provided by Figure 2's DFG depiction of the process described in Section 2.

The icons represent dataset types (boxes) or process types (ovals) and arrows indicate the flow of datasets into and out of processes (from [3]).

The four subprocesses described in section 2 can all be identified as loops in the DFG shown in Figure 2, but the nature of their interactions with each other is hard to determine precisely. Thus, for example, the DFG suggests that many different instances of "Revised Data" can be generated by executing this process, but the diagram makes it hard to determine whether or not the scientist may wish to restrict which selection criteria are to be applied to which "Revised Data" under which circumstances. In general, the DFG does not specify how the various processes are initiated, controlled, or coordinated. For example: how is model-building coordinated with realtime data processing? When alternate measurements are substituted for original measurements, how is the required updating of neighboring modeled values initiated and controlled? What happens if a process (e.g., retrieval of a realtime measurement) cannot complete successfully? Do the four major subsystems operate independently of one another? Such critical questions cannot be answered easily, if at all, from the information provided by this DFG. While it is certainly possible to define a far more elaborate DFG that could provide answers to such questions, our experience has indicated that such a DFG would be quite complex and confusing, and that it would risk denying scientists the clarity and conciseness they need.

Our approach to such problems is to use a process definition language with semantic features such as abstraction, concurrency, and exception management. Specifically, we used the Little-JIL process language to define PDGs, using our experiences to help evaluate the importance of various language semantic features.

Little-JIL: Little-JIL is a graphical language for the coordination of agents [6, 7]. Its semantics are precisely defined using finite-state automata. A Little-JIL process is defined as a hierarchical decomposition of steps, where a *step* represents a task to be done by an assigned agent. A step is represented iconically by a step bar, and is best thought of as the definition of an abstract procedural module. Each step definition specifies its input and output artifact types, best thought of as parameters to the step. During process execution, artifact instances are bound to the step parameters as arguments to create step instances. A step also specifies the exceptions it handles, flow of control between its children, etc. A step with no sub-steps is called a leaf step, and represents an activity performed by an agent without process guidance. An agent may be a human or a computational tool executed with the designated input when the leaf step is encountered. In scientific processes, leaf steps may represent the performance of such computational activities as matrix inversion or curve-fitting by tools or proprietary packages, perhaps performed on different platforms or at remote sites. The role of the Little-JIL process definition in these circumstances is to define the way artifacts flow to and from such steps, and how their processing is coordinated with other needed scientific processing, and with the activities of humans. The example process in Figure 3 illustrates many Little-JIL features. Some of these features are:

Step sequencing. Every non-leaf step has a sequencing badge (an icon in the left of the step bar), which defines the order in which sub-steps execute. A sequential step (right arrow) indicates that its sub-steps are executed sequentially from left to right. A parallel step (equal sign) indicates that its sub-steps can be executed in any (possibly interleaved) order. A choice step (line through circle) indicates that the human executing the step can choose among sub-steps, while a try step (right arrow through X) mandates the sequence in which sub-steps are to be tried until a successful outcome occurs. A child step is connected to its parent by an edge that can carry a cardinality annotation. Kleene * and + annotations indicate unbounded multiple instantiation of the child step.

Artifacts and artifact flows. Artifacts are entities (e.g., data items or datasets) that are used or produced by the step. The artifacts used by the step (IN parameters) or produced by the step (OUT parameters) are declared in the step interface (circle atop the step bar). In addition, the flow of artifacts between parent and child steps is indicated by attaching artifact annotations and directional arrows to parent-child edges. These annotations define the arguments that are to be bound to the formal parameters defined as part of the step interface. By specifying different bindings of arguments to different instances of steps, Little-JIL can create different contexts for different step instances. Software engineers might expect that such a facility for creating instances from step abstractions should improve process definition clarity and expedite step reuse. Our experience with this work offered the chance to evaluate this expectation.

Requisites. A Little-JIL step optionally can be preceded and/or succeeded by a step executed before and/or after (respectively) the execution of the step's main body. A prerequisite is represented by a down arrow to the left of the step bar, and a post-requisite is represented by an up arrow to the right of the step bar. Requisites enable checking a specified condition either as a precondition for step execution or as a post-execution check to assure acceptable execution. If a requisite fails, an exception is triggered.

Channels. Artifacts can also flow between steps along channels, essentially buffers that hold queues of artifacts produced by the step(s) at the tail of the channel, and used by the step(s) at the head of the channel. Steps accessing the channel typically block until needed artifacts are delivered by the channel. Thus, channels can also be used for synchronization.

Exception Handling. A Little-JIL step can signal the occurrence of exceptional conditions when a requisite fails or when other aspects of the step's execution fail. This triggers execution of a matching exception handler associated with the parent or other ancestor of the step that throws the exception. The handler is represented as a step attached to an X on the right of the step bar. Exception handlers can receive arguments that can be used to help clarify the nature of the exception. This capability facilitates creating a descriptive context that can help the handling of the

exception to be more precise and effective. Little-JIL also supports the specification of four alternatives for how execution proceeds after completion of exception handling. For example execution could continue by repeating the step to which the exception handler was attached, with that step's parent, or with its sequential sibling. Although DFGs can be used to represent how exceptions are handled and how execution is to resume, our experience suggests that such DFGs quickly become complex and impenetrable, especially when exceptions can originate in multiple ways and from multiple process locations. Our experience suggests that suitable language constructs help with this problem.

Scoping. A parent step and all of its descendants represent a Little-JIL scope, enabling specification that certain data items and datasets are specific to that scope. Little-JIL also supports recursive specifications of a step within its own scope. This language construct can greatly clarify the iterative application of a process step to specifically defined data items and datasets.

A PDG of the hydrological modeling process. The step depicted in Figure 3, "Process Water Budget Datasets", defines the four major subprocesses described in Section 2 and indicates that they can be carried out in parallel with each other. Note that the two left subprocesses have a Kleene + on the edges connecting them to the parent step, indicating that each can be instantiated one or more times. Each instance is created upon the availability of an execution agent for the step, and such an agent is created in response to a daily timer interrupt. The Kleene * on the rightmost two steps indicates that these steps may be instantiated zero or more times. In these cases, a new instance of the step is instantiated at a scientist's request. Full definition of the last of the four substeps, "Create New Models" is not shown for lack of space, but we now sketch the details of the other three.

"Realtime Data Processing" is the subprocess that creates a dataset for each full 24-hour day. The left-pointing arrow indicates that this is done by first executing the left substep, "Initialize Dataset", and then the right substep "Augment Realtime Dataset". The Kleene + on the right substep edge indicates that this substep is executed at least once, and indeed this step is instantiated anew every 30 minutes when a new set of measurements is to be gathered. Recall that step parameter declarations and edge argument bindings are integral parts of Little-JIL process definitions. Most such details are not shown in this figure, however, where they seem obvious, to reduce diagram clutter. Some edge annotations are shown for emphasis and clarity, however. Thus, for example, the edge connecting "Initialize Dataset" to its parent has the annotation "Dataset for *today*", with an upward arrow, which indicates that the result of performing this substep is the creation of a dataset describing readings for the day on which this step was instantiated, and that this dataset is passed onwards for use by its right sibling, "Augment Realtime Dataset". "Augment Realtime Dataset" consists of the sequential execution of three substeps that result in augmenting the set of daily readings by the readings from the most recent time period. The first substep, "Get RT Readings" acquires the raw data from actual sensors and instruments. In Figure 3 we provide some elaboration only on the definition of the process for obtaining a precipitation reading, done by "Read Precip". The "Read Others" step is not elaborated here, but can be expected to look quite similar to the elaboration of "Read Precip", which consists of obtaining a reading from each of two (redundant) precipitation gauges (steps "Read Precip 1" and "Read Precip 2"), and also specifying how to handle the exceptional situation where a precipitation reading is missing (step "Handle No Precip"). Shortly, with the aid of Figure 4, we describe how this exception is to be handled, indicating how powerful exception handling semantics facilitate creating clear and concise process definitions.

Once a complete set of readings has been gathered by "Get RT Readings", they are passed to "Apply Models" where their suitability is considered. As noted above, scientists routinely scrutinize raw data to determine whether there seems to be a need to modify or replace it. Some data may be suspect because of environmental conditions (e.g. wind direction, ice buildup), and some may be completely missing (e.g. because of sensor failure). In such cases replacement data is generated and substituted for the data that are suspect or missing. Models are used to determine

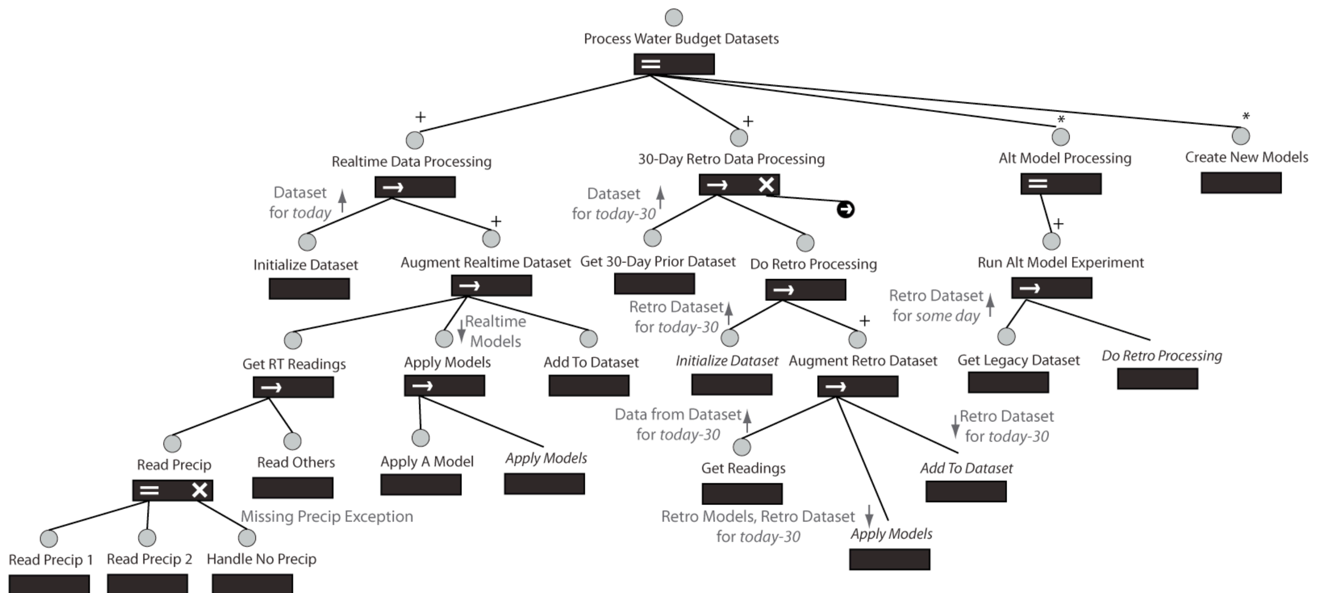

Figure 3. Example Little-JIL PDG for the hydrological modeling process described in Section 2.

whether and how such data are to be replaced. Figure 3 shows that a set of such models, "Realtime Models" is passed to the step "Apply Models" for use in examining the data produced by "Get RT Readings" and for suggesting replacements. Note that "Apply Models" is defined by the sequential execution of "Apply A Model" (which selects one model and applies it to relevant data items), followed by the recursive execution of "Apply Models" (note that the name of this step is italicized to indicate that it is the reinvocation of a step that has previously been defined elsewhere). Thus this step demonstrates how iteration can be specified. Indeed, in this case, the iteration is more than just the repetition of a step "Apply A Model", but is iteration over a set, namely the set of models. In this example, appropriate definition of the parameters to "Apply Models" and the edges connecting the instances to their parents can assure that the set of candidate models is reduced by the deletion of the model just used by the immediately prior execution of "Apply A Model". Thus the scientist performing "Apply Models" is free to select the order in which models are to be considered, but will not be able to make the mistake of reapplying a model that had been applied previously. The details of how this is actually specified by Little-JIL are less important than the observation that this recursive execution of a well-designed step abstraction seems to be a particularly useful feature in a language used to describe this important, and common, scientific data processing activity.

"Augment Realtime Dataset" concludes with the execution of "Add to Dataset", a step whose job is to take the data items that have resulted from the application of the appropriate models to raw data, and appending these data items to the growing dataset that represents the data gathered in realtime for the current day. To do this, "Add to Dataset" must incorporate subprocesses for considering the results of applying the various models to the various raw data items, selecting those that seem most appropriate, and annotating the resulting data items with specifications of which models were applied to which raw data items. The details of this subprocess are not shown here for lack of space. Subsequently, however, we describe how the annotation of the resulting data items is facilitated using a DDG, which provides a specification of the provenance of these data items.

The second substep of "Process Water Budget Datasets", "30-Day Retro Data Processing", defines how realtime datasets generated by "Realtime Data Processing" are systematically reconsidered, as described in Section 2. "30-Day Retro Data Processing" is the sequential execution of "Get 30-Day Prior Dataset" and "Do Retro Processing". The details of "Get 30-Day Prior Dataset" are not provided here, but note that the process defines how to handle the exceptional situation that arises should this dataset not be available. If thrown, the exception is handled by "30-Day Retro Data Processing" (note the X on the right of the step bar). The right-pointing arrow in the circle attached to the X indicates that the handling consists simply of terminating the execution of "30-Day Retro Data Processing". If the dataset is accessed successfully, then the annotation on the edge connecting "Get 30-Day Prior Dataset" to its parent indicates that this dataset is then made available to be passed to "Do Retro Processing".

From the description of the Water Budget process given in section 2, it should be expected that the "Do Retro Processing" process should bear strong resemblance to the "Realtime Data Processing" process. Figure 3 shows that appropriate language semantics emphasize this, supporting clarity and reuse. Indeed "Do Retro Processing" consists of the sequential execution of (another instance of) "Initialize Dataset" and then one or more executions (Kleene +) of "Augment Retro Dataset", in strong analogy to the definition of "Realtime Data Processing". In the case of "Do Retro Processing", the output argument is annotated as being "Retro Dataset for *today*-30", emphasizing that the dataset to be created has an appropriate name. "Augment Retro Dataset" consists of the sequential execution of "Get Readings", followed by new instances of "Apply Models", and "Add to Dataset", again analogously to the decomposition of "Augment Realtime Dataset". In the case of the decomposition of "Augment Retro Dataset", however, the arguments bound to these steps are those relevant for retrospective processing. Thus, "Get Readings" obtains readings from "Retro Dataset for *today*-30", the dataset found and passed as an output from "Get 30-Day Prior Dataset" rather than from a realtime data stream. These data items are then input to "Apply Models". Note that this instance of "Apply Models" takes as input "Retro Models", a set of models appropriate for modifying values in retrospective datasets. As noted above, retrospective datasets afford the opportunity to use later data items to help in deciding which raw data is to be modified or replaced and which data items are to be substituted. Thus, the set of models passed to this instance of "Apply Models" may be different from the set passed to the previous instance of this step. In addition, note that these models may require that a neighborhood of data items, both prior and subsequent, may have to be passed from "Get Readings" in order to support the use of these more complex models. "Add to Dataset" is an instance of a step previously instantiated, but in this context, it is adding instances of retrospectively modeled data items to a retrospective dataset. The complete parameter specifications and bindings are not shown here as they seem obvious and would create additional diagrammatic clutter.

The third substep of "Process Water Budget Datasets", "Alt Model Processing", may be instantiated, for example, when a scientist wishes to evaluate one or more new models by applying them retrospectively to previously created datasets. The process for doing this is another example of how abstraction facilities can help improve reuse and clarity. Note that this process is defined to be the execution of one or more (Kleene +) instances of "Run Alt Model Experiment", each of which is an evaluation against a different retrospective dataset. The process for doing one such evaluation is simply the execution of "Get Legacy Dataset", which fetches a previously developed dataset, "Retro Dataset for *some day*", followed sequentially by "Do Retro Processing", which was previously defined as the principal substep of "30-Day Retro Data Processing".

To provide elaboration of the need for powerful exception handling capabilities we now provide further details of how the "Handle No Precip" step handles the "Missing Precip" exception thrown from one of the substeps of "Read Precip" shown in Figure 3. Figure 4 depicts the "Handle No Precip" step. This step is an exception handler that is called when no data has been received from one of the two precipitation gauges. The first step performed is "Choose Source", a choice step (note the choice icon in the step bar) that chooses between the left substep that rereads from gauge 1 and the right substep that rereads from gauge 2. If the reread fails to produce a value, the step throws an exception that is handled by the "Sensor Down" handler that belongs to the "Handle MS Sensor Timeout" step. This handler seeks the precipitation measurement through the "Get Airport" step. That step may also throw an exception in case the airport is also not able to provide the requested measurement. The exception is handled by the "Put Null Reading" step, which is the exception handler for "Sensor Down". Thus Figure 4 demonstrates how the

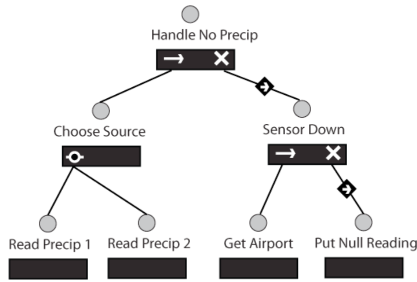

Figure 4. Exception handling in step "Handle No Precip".

context in which an exception arises can have a strong effect upon how the exception is handled. In this example an exception may arise during the handling of another exception, leading to the need for a different reaction. The example illustrates the benefits of using scope to determine context, which is commonplace in programming languages but not in data-flow based models.

Generating DDGs from Executions of PDGs: We demonstrate how DDGs can be generated from executions of a PDG by incrementally showing how a DDG can be used to document the provenance of a specific data item that is produced and passed as an output from the "Add to Dataset" step. Our previous discussion noted that this output results from a scientist's consideration of alternative values generated by alternative models, each applied by a different instance of the "Apply A Model" step. Figure 5 illustrates a portion of a visualization of a DDG that could be generated by the steps leading up to "Add to Dataset". In this visualization, process steps are depicted as ovals annotated with the name of the corresponding step in bold face. Process artifacts are depicted as rectangles annotated in gray type to emphasize that these are descriptive names, most usefully provided by the scientist. An actual DDG would incorporate actual values (or references to actual values) in place of the visualized rectangles. The annotations shown here are intended only to be explanatory of the semantic role of the artifact in the DDG. Edges whose tails originate from an oval indicate that the artifact at the head was taken as an input to the step represented by the oval. Edges whose tails originate from a rectangle indicate that the oval at the head represents the step that generated the artifact. Thus, note that Figure 5 documents that "Add to Dataset" produces an artifact, annotated "P", by taking as inputs two artifacts, each the output of a different model (a DDG documenting the application and consideration of more than two models would be analogous). Each of these artifacts is shown as the output of an instance of the "Apply A Model" step, each of which uses the same two artifacts (annotated "P1" and "P2") as inputs. But one instance of "Apply A Model" also uses as input a model annotated "Model 1", while the other uses a model annotated "Model 2". The details of how the scientist chooses between the two inputs to "Add to Dataset" are not documented here, as this step is a leaf step, and thus the PDG provides no information about this. A scientist inspecting this DDG would note that the artifact annotated by "P" has the same value as one of the two inputs, which then indicates the model that had been chosen by the scientist. If further process details are desirable, these details would be provided as decompositions of the "Add to Dataset" step or as documentation provided by the scientist.

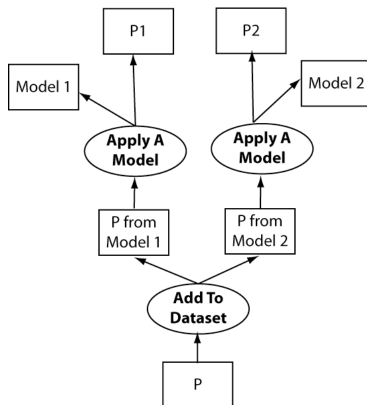

Figure 5: A DDG showing the derivation of the value p resulting after applying different precipitation models.

Additional information about the provenance of "P1" and "P2" is provided by additional parts of the DDG shown in Figure 6. This information is passed successively through the steps, "Read Precip", "Get RT Readings", and "Apply Models". The DDG is a correct depiction of the fact that these artifacts are passed unchanged through these steps. DDG segment documents how "P1" and "P2" were generated initially by the "Read Precip 1" and "Read Precip 2" steps, respectively, and how these artifacts were then passed unchanged This is readily inferable because Little-JIL mandates that the relations among the parameters of Little-JIL steps must be documented. The DDG in Figure 6 is, however, more cluttered than is desirable, suggesting that it might be preferable to depict it as shown in Figure 7, where the steps that merely pass values through unchanged are shown in gray.

Note that Figures 6 and 7 represent an execution of the PDG in which both "Read Precip 1" and "Read Precip 2" execute successfully. But the process allows for the handling of exceptional situations as well. Figure 8 depicts the DDG that would result in the case that "Read Precip 1" fails initially, triggering execution of the "Handle No Precip" exception handler, resulting in a retry of "Read Precip 1". In this case, the retry also fails triggering an additional exception handled by the "Get Airport" step that produces a precipitation reading. The different executions of the PDG result in different DDGs, documented by Figures 7 and 8, each providing precise provenance of the eventual value "P", that is produced by the "Add to Dataset" step.

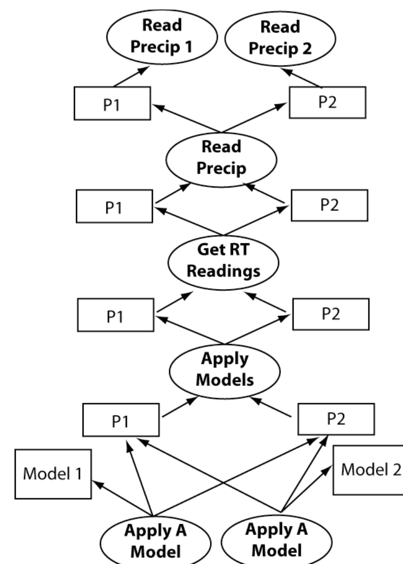

Figure 6: DDG depicting the derivation of precipitation readings from two different precipitation gauges.

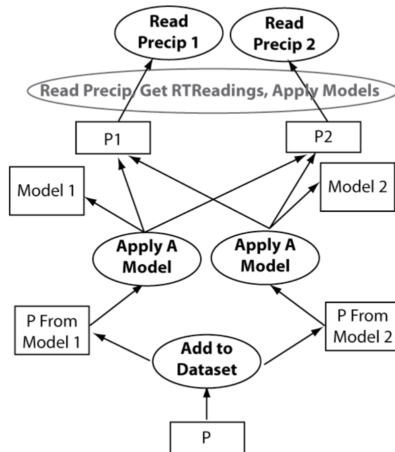

Figure 7: Another depiction of the DDG in Figure 6, which is intended to reduce visual clutter.

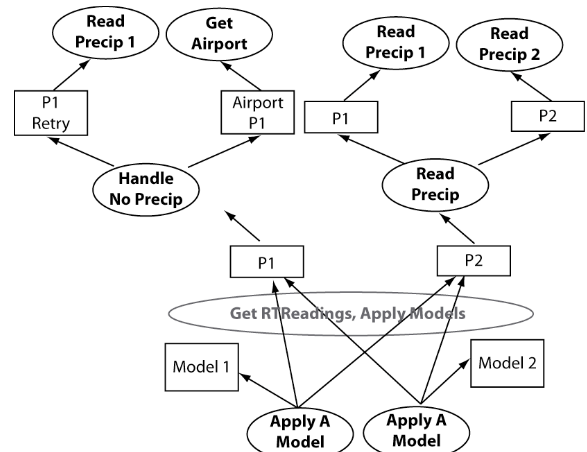

Figure 8: DDG resulting from execution of exceptional flow caused by failure of one precipitation gauge.

4. EXPERIENCE AND EVALUATION

The entire Water Budget PDG is defined as a hierarchical structure of approximately 12 Little-JIL diagrams, where most diagrams are significantly smaller than the diagram in Figure 3. Creating this process definition required the use of a broad range of semantic features. We now present a more detailed analysis of what this experience suggested about the desiderata for a language to be used as the basis for defining an analytic web's PDG.

Decomposition and Abstraction

The example made good use of Little-JIL's abstraction facilities, as shown in the four parallel substeps of "Process Data" in Figure 3. Note that the first two contain instantiations of the steps, "Initialize Dataset" and "Add to Dataset", and the third also uses these steps as part of its reuse of the "Do Retro Processing" step. Further, all three of these substeps use the "Apply Models" step, although the sets of models used by the substeps may be different. This use of abstraction indeed seems to help in achieving clarity and reuse, which should be no surprise to software engineers.

Concurrency specification and control

Additional complexity in the example process is attributable to process parallelism. Data streams from various sensors must be processed in parallel and in realtime as data are gathered and transmitted concurrently. Other parallelism occurs in that the activity of generating and evaluating streams of data items (in "Realtime Data Processing") must occur concurrently with the creation and evaluation of retrospective datasets (in "30-Day Retro Processing" and "Alt Model Processing") and with new model generation ("Create New Models"), which tends to occur much more infrequently. In the example, Little-JIL channels are used to represent how models built by the "Create New Models" step are then used by all three of its sibling steps, although space does not permit showing the details of how this is done.

Exception specification and management

Little-JIL's exception management facilities enabled us to define complex exceptional situations, including need to deal with exceptions that occur during the handling of exceptions themselves, relatively easily. Figure 4 shows only one set of contingencies that must be handled to deal with missing data items. The complete definition of this process required more extensive use of exception management, and this seems to be an important feature of scientific processes. Such exception management semantics as scoped exception handling and different exception resumption semantics proved useful in supporting clarity and completeness in defining this process.

Late-binding of resources

This example only hinted at the importance of supporting the late-binding of resources to steps to permit flexible reactions to contingencies. Figure 4 specified an airport precipitation gauge as the agent used to obtain a precipitation reading. But it is reasonable to leave the choice of agent to an external resource manager (human or automated) charged with making a late-binding decision about the agent deemed most likely to be successful in delivering the needed measurement. This points to the desirability of a process definition facility able to specify only the needed capability and a runtime system able to support realtime selection of an agent best able to provide that capability.

PDG feature summary

Our use of Little-JIL to specify the PDG for a realistically complex scientific process enabled us to suggest language features that should be useful in supporting the definition of the PDG in an analytic web. We are struck by the similarity between these needs of the scientific community and what is generally provided by modern programming languages. This supports our intuition that scientific processes have strong similarities to other systems traditionally implemented or modeled with computer software. Thus, the challenges of defining them have strong parallels with the challenges of programming complex software systems, and it is not surprising to find that a process language should incorporate the salient features of modern programming languages.

DDG Evaluation

The example process also shows how DDGs can be built incrementally as the execution of a PDG proceeds and can be defined as traces through the PDG. DDGs grow as DAGs, increasing in depth as PDG execution proceeds; iteration of processing steps is manifest as additional levels in the DDG DAG. This emphasizes the role of these steps in establishing scopes, and the DDG is capable of clearly illustrating this role. DDGs derived from executions of lengthy processes may seem cumbersome. But it is the pictorial depiction of an entire DDG that is large. Their internal representations are typical DAG-like structures that should be amenable to relatively terse internal representation. In addition, the representation of an entire DDG may not be of interest in many cases; elided versions might often suffice. Thus

tools that allow users to tailor DDGs through elision seem useful. A number of tools of this kind already exist. The presentation of DDGs is an important area that requires future research.

5. RELATED WORK

There has been a great deal of prior work on scientific workflow. Most of this work is based on the use of data flow graphs (DFGs) to specify process flow. Chimera [8-10], one of the earliest scientific workflow systems, used pictorial visualizations of DFGs to represent scientific processes. Taverna [11, 12], a more recent effort, focuses on the integration of web services, particularly for bioinformatics applications. Taverna's integration mechanism is a workflow notation also based upon a DFG formalism. JOpera [13, 14] has suggested using XML to specify scientific workflows as plug-ins integrated using Eclipse [15]. JOpera workflows also use a DFG formalism to represent scientific processes. Teuta [16, 17] represents scientific processes through UML diagrams that offer some features, such as limited forms of concurrency, that go beyond the semantic features of a basic DFG. Kepler [5, 18, 19] is perhaps the most advanced of these projects. It is based upon Ptolemy II [20, 21], which uses a powerful and flexible DFG structure to specify how datasets move between processing capabilities. Kepler integrates a broad range of tools to support specification, execution, and visualization of scientific processes. It seems particularly effective in supporting processes for realtime streaming sensor data.

It seems important to note that most of these scientific workflow representations incorporate semantic features that are sufficient to represent scientific processes. Our view is that all too often, however, these representations are too cumbersome and confusing. As noted above, software engineers should readily recognize that languages with weaker semantics may be sufficient to represent complex systems, but that there is important value in seeking languages whose more powerful notations support greater clarity and conciseness without sacrificing completeness. Scientific Workflow technology seems to be at a stage where the search for such languages is clearly indicated.

To be more specific, Kepler's reliance upon the use of a hierarchical structure of DFGs can complicate the clear specification of such features as alternative semantics for continuation after the handling of exceptions. Indeed our work indicates that such powerful exception management semantics seem necessary in defining complex scientific processes. Kepler represents a scientific process by means of a hierarchy of pairs. The pair consists of a DFG and a Director, which defines the semantics of the DFG. Several different Directors are defined as part of the Ptolemy II system, upon which Kepler is built [22]. Different Directors define different DFG semantics (e.g. Synchronous Dataflow (SDF), Finite State Machine (FSM), Heterochronous Dataflow (HDF), Discrete Events (DE), and Process Networks (PN)), and SDF, FSM, and DE seem to be frequently used. The process definer may use different Directors at different hierarchical levels to specify the semantics needed to facilitate a process definition. The fact that execution semantics may change between process hierarchical levels, however, seems to us to jeopardize the clarity of such process definitions. Thus, one may have to look at two or three diagrams at different hierarchical levels simultaneously (a DFG and the Directors at its level and its parent's level) in order to adequately understand a process definition. In addition, certain nestings of Directors at different levels in a process hierarchy may be semantically problematic, and some are indeed prohibited. DFGs using appropriate Directors can precisely define conditionals and iteration. But such constructs are often cumbersome to define, and the Kepler technology suggests no uniform way in which to define them. Further, the semantics of loop executions defined through various layered Directors is sometimes not immediately clear.

Among the mechanisms that augment Kepler to support representing procedure invocation and exception management are "dynamic embedding" and "collection aware actors" [23]. Dynamic embedding (part of a project that extends Kepler) is a mechanism that can be used to introduce a placeholder step into a DFG process definition, and then allow for the use of any of several processes that match that placeholder's interface. Other scientific workflow systems such as Taverna offer somewhat analogous capabilities. An agent assigned to the placeholder can then decide at runtime which of the actual process definitions is executed when the placeholder is to run. Thus the logic behind choosing a procedure or an exception handler is hidden in the programming language used to define the agent (Java is generally used). As is the case when an agent is late-bound to a Little-JIL leaf step, this complicates comprehension of process behavior and analysis of the process based on its definition. Moreover, when this construct is used to support exception handling, execution may not always return to exactly the point where the exception was raised.

The lack of strong facilities for exploiting abstraction also leads to duplication, complexity, and lack of clarity. Kepler's "composite actor" is an example of a construct that can be used to support a modest form of abstraction. A Kepler composite actor is a subprocess that can be assigned as the agent for a step. It is possible to re-use the composite actor as the actor for any of a number of steps. But composite actors seem to have limitations in their ability to reference context information that might impact the way in which the composite actor performs. Thus, for example, when a composite actor is used to define a subprocess that delivers its result by choosing among a set of lower level tools, the tools must have identical parameter lists. Our experience suggests the need for more adaptability in a process language's abstraction capability. Other approaches to supporting abstraction seem to be under consideration by the process and scientific workflow communities. This line of work seems to us to be particularly important, and of particular interest to software engineering.

Other Scientific Workflow systems offer somewhat different approaches to supporting these semantic features. Space limitations prevent detailed critiques of all of them. We suggest that such critiquing, and suggestions for improvement, fit nicely into the tradition of software engineering research. Little-JIL suggests some approaches to improvement, but this general line of research would seem to be of interest and value both to Scientific Workflow and Software Engineering.

There is also other work that supports documentation of the provenance of scientific datasets (e.g., [24, 25]). There are generally two approaches to this work. In one approach (e.g., [26-28], each data artifact is stored in a database, and annotated with information about the tool or system used to create it and its input artifacts. Artifact provenance documentation can then be obtained by querying the database. The other approach entails building a derivation graph on the fly as execution of the scientific process proceeds. The Chimera [9] and Kepler [19] projects have adopted this latter approach. Our proposed approach falls into the latter category as well, entailing on-the-fly construction of a derivation structure, the DDG. Our DDG depicts the progress of execution through a PDG that incorporates powerful semantic structures such as concurrency, exception handling, non-trivial iteration, and

abstraction. The presence of these powerful semantic features in the PDG definition language makes it possible for the DDGs generated from such PDGs to incorporate these semantic features. Thus, for example, artifacts produced on different iterations through a given activity are shown as the roots of lower level subgraphs of a DDG, where the context of each activity execution is provided by nodes of the DDG that are specified as loop abstractions. Our experience suggests that these features can help to make DDGs clearer. Other scientific workflow systems seem to also value incorporation of such higher level semantic features. Continued investigation of how to incorporate these features to best effect is indicated.

Our work is reminiscent of the Odin project [29] that documented how collections of software tools are used to produce software products. As in this work, Odin maintained a type structure, showing which software tools generate which types of software objects, and an instance structure recording the specific software artifacts generated by applications of specific tools. Indeed, Odin itself improved on such pioneering efforts as Make [30] and SCCS [31] and, in turn, contributed to the genesis of Software Configuration Management (SCM). Documenting scientific data provenance does indeed bear a resemblance to SCM, and we note that other recent work in scientific data provenance has also started to recognize the problem of documenting provenance in situations where the scientific process is evolving [19] [32].

6. FUTURE WORK

Analytic Tools and Their Roles: This project suggests the value of analytic tools to support reasoning about scientific processes, data, and datasets. Thus an analytic web should eventually incorporate tools such as parsers, cross-referencers, and semantic checkers for analyzing PDGs. These tools will help identify such process errors as incomplete or incorrect exception handling. We also suggest evaluating more powerful analyzers such as model checkers (e.g., [33, 34]) in analytic webs. Model checking should determine whether process definitions contain errors such as datasets that are generated but never used. These defects can be caused by incorrect process design or just misspellings. More importantly, such analyses might identify defects that could lead to unsound scientific results. Applying certain statistical techniques can yield unreliable results when they follow the application of certain other techniques [35]. Typically this occurs when subsequent data processing is done without knowledge of earlier processing. Model checkers can scan all execution sequences in a process defined using a language, such as Little-JIL, detect the possibility of invalid sequences, and guide process modification that prevents these sequences. There is also a role for dynamic event sequence analysis in analytic webs, where DDGs can be examined for faulty analysis sequences as they are generated.
DDG Optimization: Previously we noted that DDGs generated by long process executions can be quite large. Optimization might reduce the sizes of such DDGs. In addition, we plan to consider regenerating parts of a DDG as an alternative to storing all of its nodes. Odin demonstrated that complete, precise documentation of the artifacts and tools used to generate a DAG such as the DDG could be used to regenerate it. Doing so costs execution time, but saves storage space. We plan to evaluate this time-space tradeoff to decide whether some parts of a generated DDG might be regenerated rather than stored.

A Full Environment: Finally, there are interesting issues concerning how the various tools suggested here might be integrated. We suggest the need for tools to create and execute PDGs, tools to construct DDGs, tools to generate process metadata, tools for viewing DDGs and PDGs, and tools for analysis. It is not clear how best to integrate these tools so that their existence and utility will be sufficiently clear to the scientists who are to use them (user integration), nor how best to implement them to make best use of each other's capabilities and implementations (backend integration). Prior research on tool integration should be useful in providing guidance on these issues.

7. ACKNOWLEDGEMENTS

This paper is based upon work supported by the National Science Foundation under Award No. CCR-0205575. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We are grateful to many colleagues who contributed ideas that led to the analytic web concept. We thank E. Riseman, A. Hanson, D. Jensen, P. Kuzeja, D. Foster, H. Schultz, G. Avrunin, and M. Raunak for their conversations, and Anne Ngu for her views on Kepler.

8. REFERENCES

1. Ellison, A.M., Osterweil, L.J., Hadley, J.L., Wise, A., et al. 2006. Analytic Webs Support the Synthesis of Ecological Data Sets. Ecology, 87, 6. June 2006, 1345-1358.

2. Osterweil, L.J., Wise, A., Clarke, L.A., Ellison, A.M., et al. 2005. Process Technology To Facilitate the Conduct of Science. In Proceedings of the Software Process Workshop, (Beijing, China, May 2005), Springer-Verlag, 403-415.

3. Boose, E.R., Ellison, A.M., Osterweil, L.J., Podorozhny, R., et al. 2007. Ensuring Reliable Datasets for Environmental Models and Forecasts. Ecological Informatics 2, 237-247.

4. Dingman, S.L. 2002. Physical Hydrology. 2nd Ed. Prentice Hall, NJ.

5. Altintas, I., Berkeley, C., Jaeger, E., Jones, M., et al. 2004. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In Proceedings of the 16th International Conference on Scientific and Statistical Database Management, (Santorini Island, Greece), 423-424.

6. Cass, A.G., Lerner, B.S., Mccall, E.K., Osterweil, L.J., et al. 2000. Little-JIL/Juliette: A Process Definition Language and Interpreter. In Proceedings of the 22nd International Conference on Software Engineering, Demonstration Paper, (Limerick, Ireland, 4-11 June), 754-758.

7. Wise, A. 2006. Little-JIL 1.5 Language Report. Department of Computer Science, University of Massachusetts, UM-CS-2006-51.

8. Foster, I., Vöckler, J., Wilde, M. and Zhao, Y. 2003. the Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration. In Proceedings of the 15th International Conference on Scientific and Statistical Database Management, IEEE Computer Society, 1-11.

9. Foster, I., Vöckler, J.S., Wilde, M. and Zhao, Y. 2002. Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation. In Proceedings of the 14th International Conference on Scientific and Statistical Database Management, 37-46.

10. Deelman, E., Blythe, J., Gil, Y. and Kesselman, C. 2004. Workflow Management In Griphyn. In Grid Resource Management: State of the Art and Future Trends, Kluwer Academic Publishers, 99-116.

11. Wolstencroft, K., Oinn, T., Goble, C., Ferris, J., et al. 2005. Panoply of Utilities In Taverna. In Proceedings of the First International Conference on E-Science and Grid Computing, IEEE Computer Society 156-162.

12. Oinn, T., Addis, M., Ferris, J., Marvin, D., et al. 2004. Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows. Bioinformatics, 20, 17, 3045-3054.

13. Heinis, T., Pautasso, C. and Alonso, G. 2006. Mirroring Resources or Mapping Requests: Implementing WS-RF for Grid Workflows. In Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid, IEEE Computer Society, 497-504.

14. Pautasso, C. and Alonso, G. 2005. The Jopera Visual Composition Language. Journal of Visual Languages & Computing, 16, 1-2, 119-152.

15. Eclipse.Org 2007. Eclipse-An Open Development Platform, 2007.

16. Fahringer, T., Jugravu, A., Pllana, S., Prodan, R., et al. 2005. ASKALON: A Tool Set for Cluster and Grid Computing: Research Articles. Concurrency and Computation: Practice and Experience, 17, 2-4, 143-169.

17. Fahringer, T., Prodan, R., Duan, R., Nerieri, F., et al. 2005. ASKALON: A Grid Application Development and Computing Environment. In Proceedings of the Sixth IEEE/ACM International Workshop on Grid Computing, IEEE Computer Society, 122-131.

18. Ludäscher, B., Altintas, I., Berkeley, C., Higgins, D., et al. 2006. Scientific Workflow Management and the Kepler System: Research Articles. Concurrency and Computation: Practice & Experience, 18, 10, 1039-1065.

19. Altintas, I., Barney, O. and Jaeger-Frank, E. 2006. Provenance Collection Support In the Kepler Scientific Workflow System. In Proceedings of the International Provenance and Annotation Workshop (Revised Selected Papers), (Chicago, IL,, May 3-5, 2006), Springer Verlag 118-132.

20. Edwards, S.A. and Lee, E.A. 2003. The Semantics and Execution of A Synchronous Block-Diagram Language. Science of Computer Programming, 48, 1, 21-42.

21. Baldwin, P., Kohli, S., Lee, E.A., Liu, X., et al. 2004. Modeling of Sensor Nets In Ptolemy II. In Proceedings of the Third International Symposium on Information Processing In Sensor Networks (Berkeley, California), ACM, 359 - 368.

22. Girault, A., Lee, B. and Lee, E.A. 1999. Hierarchical Finite State Machines with Multiple Concurrency Models. IEEE Transactions on CAD of Integrated Circuits and Systems, 18, 6, 742-760.

23. McPhillips, T.M. and Bowers, S. 2005. An Approach for Pipelining Nested Collections In Scientific Workflows. SIGMOD Record, 34, 3, 12-17.

24. Simmhan, Y.L., Plale, B. and Gannon, D. 2005. A Survey of Data Provenance In E-Science. ACM SIGMOD Record, 34, 3, 31-36.

25. Moreau, L., Ludäscher, B., Altintas, I., Barga, R.S., et al. 2008. The First Provenance Challenge. Concurrency and Computation: Practice & Experience, 20, 5. April, 2008, 409-418.

26. Buneman, P., Khanna, S. and Tan, W.-C. 2001. Why and Where: A Characterization of Data Provenance. In Proceedings of the Eighth International Conference on Database Theory, (London, UK, January 2001), Lecture Notes In Computer Science 1973, Springer Verlag, 316-330.

27. Lanter, D.P. 1991. Design of A Lineage-Based Meta-Data Base for GIS. Cartography and Geographic Information Systems, 18, 4, 255-261.

28. Aiken, A., Chen, J., Stonebraker, M. and Woodruff, A. 1996. Tioga-2: A Direct Manipulation Database Visualization Environment. In Proceedings of the Twelfth International Conference on Data Engineering, IEEE Computer Society, 208 - 217

29. Clemm, G.M. and Osterweil, L.J. 1990. A Mechanism for Environment Integration. ACM Transactions on Programming Languages and Systems, 12, 1. January, 1-25.

30. Feldman, S.I. 1979. Make—A Program for Maintaining Computer Programs. Software—Practice and Experience, 9, 3. March, 255–265.

31. Rochkind, M.J. 1975. The Source Code Control System. IEEE Transactions on Software Engineering, SE-1. December 1975, 364-370.

32. Callahan, S.P., Freire, J., Santos, E., Scheidegger, C.E., et al. 2006. Vistrails: Visualization Meets Data Management. In Proceedings of the International Conference on Management of Data, (Chicago, IL, June 2006), ACM SIGMOD, 745-747.

33. Dwyer, M.B., Clarke, L.A., Cobleigh, J.M. and Naumovich, G. 2004. Flow Analysis for Verifying Properties of Concurrent Software Systems. ACM Transactions on Software Engineering and Methodology, 13, 4. October 2004, 359-430.

34. Cobleigh, J.M., Clarke, L.A. and Osterweil, L.J. 2002. FLAVERS: A Finite State Verification Technique for Software Systems. IBM Systems Journal, 41, 1. 2002, 140-165.

35. Oates, T. and Jensen, D. 1999. Toward A Theoretical Understanding of Why and When Decision Tree Pruning Algorithms Fail. In Proceedings of the Sixteenth National Conference on Artificial Intelligence (Orlando, Florida.), 372-378.