Provenance and Quality Control in Sensor Networks

Barbara Lerner[1], Emery Boose[2], Leon Osterweil[3], Aaron Ellison[2], Lori Clarke[3]
[1] Mount Holyoke College
[2] Harvard University
[3] University of Massachusetts Amherst
blerner@mtholyoke.edu, boose@fas.harvard.edu, ljo@cs.umass.edu,
aellison@fas.harvard.edu, clarke@cs.umass.edu

***Abstract*—Scientists and society increasingly rely on streaming data from electronic sensors to assess, model, and forecast environmental changes. Because analyses of time-series data require uninterrupted data streams or datasets, scientists regularly fill gaps in the data by substituting modeled values. As modeling increases in complexity, the provenance metadata needed to describe and define processes used to model data and create derived datasets quickly exceeds the capacity of individual flags or groups of flags to annotate individual data values. In theory, necessary provenance metadata could be captured in narrative form, but the time and effort required to do so are prohibitive. A system that can capture provenance metadata automatically and allow scientists to query them for useful details is what scientists really need. In this paper we describe a system that uses Little-JIL, a process programming language, to rigorously define modeling and data-derivation processes, and a mathematical graph structure – a Data Derivation Graph (DDG) – that precisely describes execution histories. Our system and approach support understanding the (potentially) different processes used to create data values, reasoning about the soundness of these processes, and helping to ensure that the data processing in sensor networks is reliable and reproducible.**

Keywords*—provenance metadata, scientific workflow, sensor network, Little-JIL

I. INTRODUCTION

Scientists and society increasingly rely on streaming data from electronic sensors to assess current environmental states and to forecast future environmental changes. Because analyses of time-series data require uninterrupted data streams or datasets (i.e., there must be a reliable observation for each time slot), scientists regularly fill gaps or correct “problems” in data streams by substituting modeled values for missing, out-of-range, or suspect observations. Different scientists substitute, model, or gap-fill data differently, and some approaches can be inconsistent with subsequent analyses. Such inconsistencies can undermine the quality and reduce the reliability of derived datasets, but these changes in quality and reliability often are invisible to subsequent users of the derived datasets. Therefore, it is critically important to be able to identify which data values represent actual observations and which have been modeled, and how modeled values have been computed. Furthermore, even observed values may undergo subsequent revision; e.g., to compensate for sensor drift that is discovered at a later time. Finally, a given data value may have been adjusted more than once. All of this suggests that the different data items in a dataset should be annotated with information (*metadata*) about exactly how their values were derived. A full history of all of the adjustments to a given datum is referred to as the data item’s *provenance*; the annotation is referred to as *provenance metadata*.

Often scientists “flag” values in a dataset using schemes that identify special conditions attendant to the data. At the Harvard Forest Long Term Ecological Research (LTER) site, current practice is to flag estimated values (including modeled values) with the single letter "E.” But a simple flag (or even several flags) is insufficient to answer all of the questions that may arise with regard to data provenance. For example, if a precipitation datum in a dataset actually originated at another site (e.g. due to sensor failure), it may be important to know which site was the origin of the datum, especially if it turns out that the second site was also experiencing sensor reliability problems on that date. Or if measurements are corrected *post-hoc* (e.g. to compensate for sensor drift), we may need to know how the data were corrected and over what range of dates, in order to correctly update derivative data products (e.g. monthly or annual summaries). Finally, if a datum was computed (not actually observed) using a model, it is important to track software and modeling tools used, as there can be variation in precision and accuracy, for example, among the different versions of the tools and algorithms used in model computation.

As data modeling increases in complexity, the provenance metadata needed to describe and define the processes, models, and associated derived data rapidly exceeds the expressive power of modest numbers of individual flags or groups of flags. Provenance metadata can be captured in narrative form, but the considerable effort required to capture these metadata accurately and then to decipher them correctly renders narratives and their analysis error-prone, especially since narratives are rarely machine readable. A system that can capture provenance metadata automatically and allow scientists to query them for useful details is what scientists really need. Our solution is to continually record comprehensive metadata as the data are collected and processed so that scientists can (re)examine the data, perhaps in ways that were not anticipated, or not possible, initially. In this paper we describe our experience in treating scientific data values to be the outputs of the execution of a (scientific data processing) process where the provenance metadata of the generated data is a summary of the execution history of the process. Our work uses Little-JIL, a process programming language, to define such processes, and a graph structure, called a Data Derivation Graph (DDG), to summarize their execution histories. The rigorous definitions and semantics of Little-JIL, and of the derived DDGs, support reasoning about the processes used to build data and datasets. This can build confidence in, and ensure the quality of, scientific data and derived data products [1].

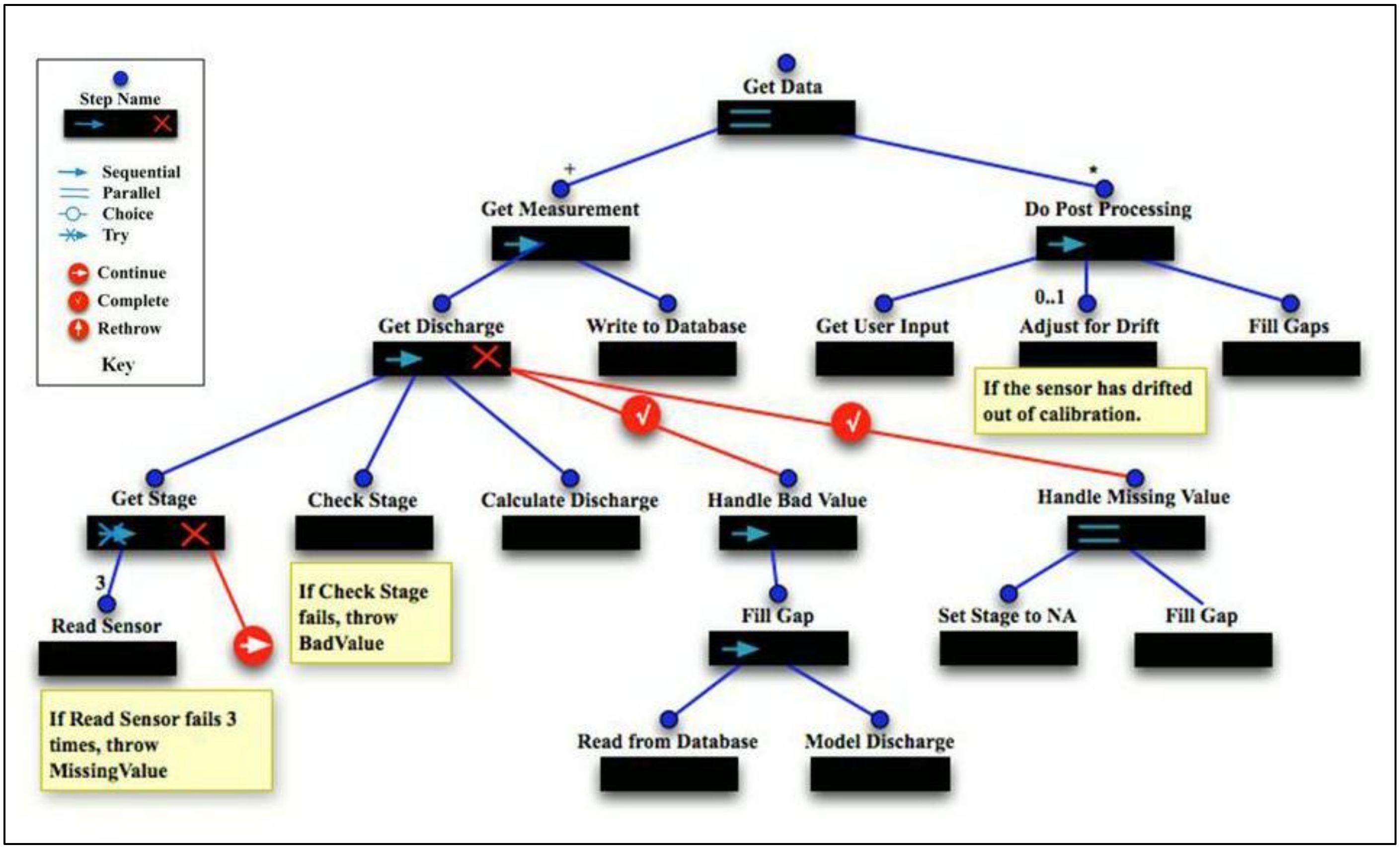

Figure 1. Little-JIL diagram for the stream discharge process.

II. STREAM GAGE EXAMPLE

Our example is an ongoing study of water movement through small forested watersheds at the Harvard Forest. The study relies on automated measurements of stream discharge (rate of flow) at a series of stream gages. At each gage, a pressure sensor is used to measure the stage or height of the water at the gage. A datalogger samples the sensor every 10 seconds, then calculates and retains 15-minute averages. The 15-minute values are retrieved from the datalogger, checked to see if they are within range, and (if they are) used to calculate stream discharge based on empirical flow equations for the particular gage. The resulting time-stamped 15-minute values of discharge are then posted online (http://harvardforest.fas.harvard.edu:8080/exist/xquery/data.xq?id=hf070).

In this paper we propose an extension of the current approach that will combine (1) automated processing of real-time measurements, along with gap filling for missing or out-of-range values, and (2) user-initiated post-processing to correct for sensor drift and update modeled values using both preceding and subsequent measurements.

III. PROVENANCE AND LITTLE-JIL

Little-JIL [2,3,4] is a graphical process programming language that supports the representation and execution of processes that may involve the interaction of multiple agents to accomplish a task (note: our terminology differs somewhat from that used in the Open Provenance Model [5]; in particular, the OPM concept of "process" corresponds more closely to the Little-JIL concept of "step"). Little-JIL processes are defined using a hierarchical decomposition of steps and substeps. This hierarchical decomposition allows a process to be viewed at various levels of abstraction, with a step's substep structure defining the way in which the step is to be carried out. A leaf step is carried out by assigning it to an "agent", an entity that is responsible for assuring the acceptable performance of the step, but in a way that is outside of the direct control of Little-JIL. Agents may be either humans or automated devices (e.g. software systems or sensors).

Artifacts flow through a Little-JIL process by being passed as parameters between steps and substeps. Each edge in a Little-JIL diagram carries a specification of the artifacts that are being passed between parent and child, along with binding information needed to relate the data flowing along an edge to the parameter specifications of the steps that are connected by the edge. Little-JIL edges can also carry cardinality information that specifies the number of instances of the substep that are to be instantiated for execution. The cardinality specification may be an integer or a Boolean expression used to determine the circumstances under which the substep is to be generated for execution. To simplify the depiction, the Little-JIL diagram does not directly show the artifacts, but a user can see this information by clicking on an edge in the Little-JIL editor.

Each step also specifies the resources required for the step to execute (the step's agent is considered to be a resource, but additional resources may also be specified), any exceptions that may be thrown by the step, and any provisions that the step may make for handling exceptions that could be thrown by any of the step's descendants.

The graphical representation of a Little-JIL step with its different badges and possible connections to other steps is shown in the key to Figure 1. The interface badge is a circle on the top of the step name that connects a step to its parent. The interface badge contains the specification of any artifacts that are either required for, or generated by, the step's execution as well as the type of the agent required to execute the step. Below the circle is the step's name. The icon at the left of the black rectangle identifies the sequencing construct that controls how the step's substeps are executed. There are four possibilities: sequential (all substeps in order from left to right), parallel (all substeps in any order or concurrently), choice (choose one substep at runtime), and try (execute substeps from left to right until one succeeds). The red X at the right edge of the black rectangle attaches a step to its exception handlers. Exceptions may be "thrown" by any of the descendants of a step. Control flow then goes to the nearest ancestor with a handler for that exception. After completing execution, the handler determines where execution should resume. There are three possibilities: continue (continue the step following the substep that threw the exception), complete (treat the parent step of the handler as having completed its execution and continue from there), and rethrow (throw the same exception thereby passing the exception up the step hierarchy to the next ancestor with a handler for that exception).

Figure 1 shows the Little-JIL diagram for the stream discharge process. The parallel root step (Get Data) builds and updates a database of sensor data through the concurrent operations of its two substeps, Get Measurement and Do Post Processing. Get Measurement collects and processes sensor data in real time and adds a record to the database for each measurement. Under normal conditions Read Sensor returns a measured value, Check Stage checks to see that the value is in range, Calculate Discharge calculates stream discharge, and the resulting values are added to the database. Exceptional conditions are handled by the corresponding exception handler. For example, if Check Stage determines that the measured value is out of range, the Handle Bad Value step generates a modeled discharge value based on preceding measurements read from the database. Similarly, if Read Sensor fails on three attempts, the Handle Missing Value step assigns a value of NA to stage and concurrently generates a modeled value for discharge.

Meanwhile Do Post Processing (shown here in abbreviated form) runs concurrently with Get Measurement. In contrast to Get Measurement, which runs continuously to process streaming data in real-time, Do Post Processing only executes infrequently, when a scientist determines that post processing is required. Do Post Processing first gets input from the user (including the range of dates and adjustment and modeling parameters), optionally adjusts a block of measurements for sensor drift, and then updates all modeled values in that block using both preceding and subsequent data.

We attach cardinality to substep edges to control the number of times that a step is repeated. In this example, the edge to Get Measurement has a cardinality labeled "+", meaning that the step is done one or more times. The edge to the Do Post Processing step has a cardinality labeled "*", meaning that the step is done zero or more times. The edge to Adjust for Drift has a cardinality label "0..1", meaning the step is done either 0 or 1 times, thereby making this activity optional. Finally, the edge to the Read Sensor step is labeled with a cardinality of 3, meaning that we will try to read the sensor 3 times before deciding that the sensor is unreachable. Due to the semantics of the Try step, Get Stage is complete as soon as Read Sensor successfully gets a value. If Read Sensor fails 3 times consecutively, it will throw an exception that will be handled by the Handle Missing Value exception handler attached to the Get Discharge step.

One of the strengths of Little-JIL is the ability to represent processes at any desired level of detail or abstraction. In our example, each of the leaf steps could be decomposed into its constituent substeps to show (for example) the equations used to calculate discharge from stage (Calculate Discharge) or the more complex series of calculations used to model discharge based on recent precipitation and discharge (Model Discharge). At the same time, the entire process shown here might be embedded in a much larger process that calculates water flux in a watershed by integrating measurements such as precipitation, evapotranspiration, stream discharge, water content of snow pack, soil moisture, and height of the water table.

The Little-JIL diagram provides a rigorous specification of the process but does not tell us what actually happened in any particular execution of the process. For that, we need the information contained in the DDG that is produced when a Little-JIL process is executed. Figure 2 provides examples, in the form of four DDG fragments, of different ways in which the process shown in Figure 1 can be executed, leading to the creation of a single stream discharge value. A DDG consists of two kinds of nodes and two kinds of edges. In Figure 2, rounded nodes represent process steps that have been executed, while rectangular nodes represent values that have been used and generated by these steps. Different colors are used to denote different kinds of steps and different kinds of values. Green edges represent the flow of control between steps while red edges show the flow of data that is generated by one step and then used as input by others.

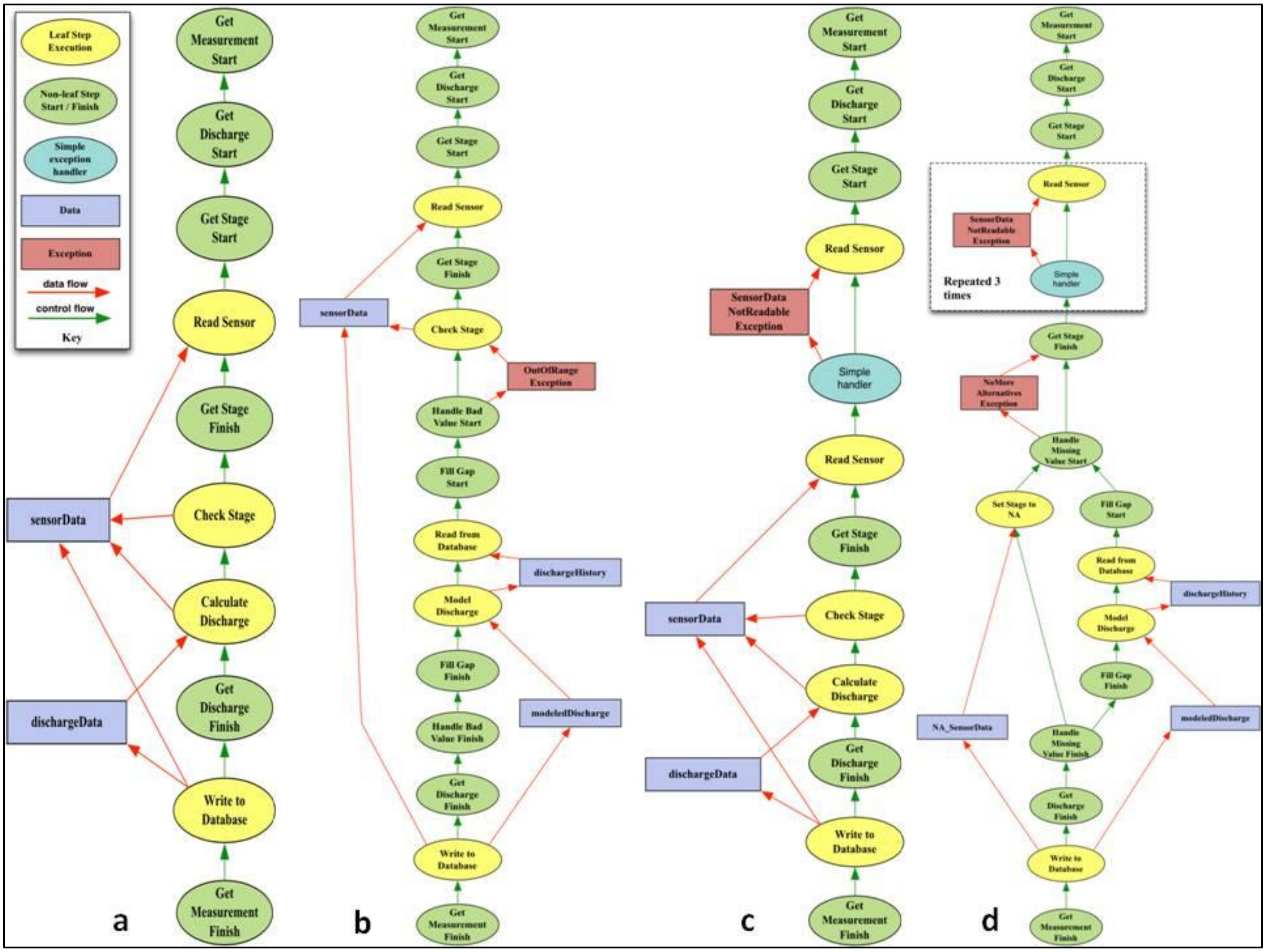

Figure 2. Four possible DDGs resulting from a single execution of the Get Measurement step: (a) normal sensor reading, (b) out-of-range value, (c) retry of Read Sensor, (d) missing value after three successive failures of Read Sensor.

The graphical representations in Figure 2 show the flow of data and control under four scenarios: (a) an in-range value is returned by the sensor and used to calculate stream discharge, (b) the Check Stage step determines that the sensor value is out of range and so a modeled value of stream discharge is generated, (c) the first attempt to read the sensor fails so the Read Sensor step is tried again, successfully returning a value on the second try, (d) Read Sensor is tried three times and fails to return a value on any of the three tries, resulting in assignment of a missing value for stage and a modeled value for stream discharge. The last three scenarios take advantage of Little-JIL's ability to precisely describe and handle exceptions. In each case the DDG shows the exact derivation of the final stream discharge value. In particular, the bottom yellow oval in each figure represents execution of the step that writes the sensor data and discharge data to the archival database. By following the red arrows up from this oval, the scientist can observe the origin or provenance of each value that is saved in the database. In the first and third cases, the observed sensor value and corresponding calculated discharge value are saved. In the second case, the observed sensor value is saved and a modeled discharge value is generated and saved since the observed sensor value is not usable. In the fourth case, a special NA (missing) value is recorded for the sensor value along with the modeled discharge value.

Most of the processing demonstrated in this example is sequential, leading to a single, straight control flow path through the process. The fourth case, however, demonstrates parallel control flow that occurs during the execution of the Handle Missing Value step. Here the recording of NA for the stage value happens concurrently with the calculation of the Fill Gap step. Note that the Fill Gap step under Handle Missing Value is a reference to the same collection of steps that is rooted at Fill Gap under the Handle Bad Value exception handler. This ability to refer to steps defined elsewhere in the process provides the ability to duplicate the same behavior in different contexts throughout a process, where the context is determined by the parameter values passed in for use by the step.

IV. RELATED WORK

Scientific data provenance is receiving increased attention [6, 7]. The Open Provenance Model [5] defines a graph representation of provenance metadata, similar in many respects to the DDGs presented here. One area of future work is to map DDGs into OPM to allow interoperability with other provenance repositories.

One significant difference between Little-JIL and other scientific workflow approaches is in exception handling. Exception handling constructs were introduced into programming languages, such as C++ and Java, to help deal with erroneous or unlikely situations where the appropriate response is often best determined in the calling scope of where the exceptional situation arose. In Little-JIL, the hierarchical levels of the process definition serve as scopes that are searched upward for an exception handler. This provides the benefits that normally come from exception handling mechanisms, most importantly, the ability to cleanly separate exception handling code from code describing the computations to be carried out in nominal (usually expected) cases, avoiding the spaghetti code that otherwise frequently arises when code to handle exceptional cases is interleaved with the processing of nominal cases.

Some workflow management systems provide support for detecting failures during execution, such as the failure of a web service, and offer a limited number of ways to manage those failures [8,9]. Kepler [10] provides the ability to annotate a collection with an exception, which an actor can then use to filter out collections that contain exceptions. User-defined exception handling is just beginning to appear in scientific workflow languages [11,12,13].

In addition to the ability to define complex exception handling, the provenance recorded in DDGs distinguishes exception objects from other types of data. We expect that a common concern among scientists is to be able to easily identify when the execution of a process encountered problems. By explicitly capturing this information in a DDG, it will be easier for scientists to perform queries that will identify the problems encountered during process execution. In the sample DDGs shown in this paper, we distinguish exception nodes by their color. As we develop the query mechanisms to access information from DDGs, we plan to give the scientist the ability to develop queries that can distinguish exceptional situations from expected situations as well.

Provenance metadata has previously been used to track changes made as sensor data is republished [14]. The emphasis in that work has been on linking together sites on the Internet that are using each other's data in order to track how the data are republished and to control access to the data. Thus, provenance metadata are used to track how sensor data are accessed and updated even though they may be distributed widely. The focus of our work is on using provenance information to support reasoning aimed at assuring that processes have the desired properties of correctness, robustness, and access control, and also to allow processes to be used directly in computing the data itself, as in the post-processing work described earlier.

V. DISCUSSION AND FUTURE WORK

Our experience to date suggests that our approach is effective in capturing detailed and accurate provenance information. Moreover, our approach supports the capture of execution details down to low levels, if those low level details are incorporated into the Little-JIL process definition. However, DDGs quickly can become large and unwieldy, as can be seen even in our simple example. We are now investigating ways to store DDGs using various database technologies that support querying and visualization. Such databases will allow scientists to focus on particular areas of interest, such as data collected from a specific instrument at a specific site on a specific date. Because many data items follow the same path through the process, we are exploring database representations that allow us to compress the stored representation considerably, yet allow us to extract provenance metadata of an individual datum without paying the storage cost of the complete DDG. Even in our simple example (e.g. Figure 2d), a repeating node pattern is easily identified. Other kinds of repetition can arise in a DDG that represents identical derivation paths of different individual discharge values. However, in more complex processes, individual paths may diverge, especially if different data values use different computations, if there is parallelism in computation, or if data values often require special error handling. A similar compression approach has been pursued by Anand et al. [15].

We are also investigating visualization mechanisms [16, 17, 18] that build upon queries of the provenance metadata to streamline the amount of data presented to the scientist. As mentioned earlier, one of the strengths of Little-JIL is the way in which the hierarchical decomposition of processes allows processes to be viewed at varying levels of abstraction. The DDGs that we produce capture the complete data flow, via the red edges, but also maintain information about the hierarchy expressed in the process, via the non-leaf start and finish nodes. We plan to take advantage of this information in visualization, to allow the scientist to zoom in and out on provenance detail, and also allow the scientist to express queries at varying levels of abstraction, again as reflected in the process. For example, the substeps rooted at Get Discharge could be collapsed into a single node showing only the stage and discharge values output by the step or fully expanded to show intervening details (Get Discharge Start to Get Discharge Finish, as shown in Figure 2).

ACKNOWLEDGMENTS

This work was supported by NSF grants DBI-1003938, CCF-0905530, CCR-0205575 and IIS-0705772, and is a contribution from the Harvard Forest Long-Term Ecological Research (LTER) program. We would like to thank Margo Seltzer for her contributions to the preliminary database work. We would also like to thank the students and programmers who have contributed to the creation of various versions of the stream discharge process and who have worked on the development of the software to capture DDGs: Alexander Wise, Cori Teshera-Sterne, Morgan Vigil and Sofiya Taskova.

REFERENCES

[1] E. R. Boose, A. M. Ellison, L. J. Osterweil, R. Podorozhny, L. Clarke, A. Wise, J. L. Hadley, and D. R. Foster. 2007. Ensuring reliable datasets for environmental models and forecasts. Ecological Informatics 2: 237-247.

[2] A. Wise. Little-JIL 1.5 language report. Technical report, Department of Computer Science, University of Massachusetts, Amherst, MA 01003, October 2006.

[3] A. Wise, A. G. Cass, B. S. Lerner, E. K. McCall and L. J. Osterweil, Using Little-JIL to coordinate agents in software engineering. In Proceedings of the Automated Software Engineering Conference, pages 155-163, Grenoble, France, September 2000.

[4] B. S. Lerner. Verifying process models built using parameterized state machines. In 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04), pages 274–284, Boston, MA, July 2004.

[5] L. Moreau, B. Plale, S. Miles, C. Goble, P. Missier, R. Barga, Y. Simmhan, J. Futrelle, R. E. McGrath, J. Myers, P. Paulson, S. Bowers, B. Ludäscher, N. Kwasnikowska, J. V. den Bussche, T. Elkvist, J. Freire, and P. Groth. The open provenance model (v1.01, http://eprints.ecs.soton.ac.uk/16148/1/opm-v1.01.pdf).

[6] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pages 1345–1350, Vancouver, June 2008. ACM.

[7] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. SIGMOD Record, 34:31–36, September 2005.

[8] A. Azimi and S. Parsa. A reliable framework for adaptive scientific workflow management systems based on SOA. In Proceedings of the 13th International Conference on Advanced Communication Technology (ICACT 2011), pages 1358–1363, Seoul, 2011.

[9] Q. L, W. Lin, W. Dou, J. Jiang, and J. Chen. A QoS-aware exception handling method in scientific workflow execution. Concurrency and Computation: Practice and Experience, 23, in press.

[10] T. M. McPhillips and S. Bowers. An approach for pipelining nested collections in scientific workflows. SIGMOD Record, 34:12–17, September 2005.

[11] J. Li, Y. Mai, and G. Butler. Implementing exception handling policies for workflow management system. In Proceedings of the Tenth Asia-Pacific Software Engineering Conference (APSEC '03), Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[12] R. Tolosana-Calasanz, J. A. Bañares, O. F. Rana, P. Álvarez, J. Ezpeleta, and A. Hoheisel. Adaptive exception handling for scientific workflows. Concurrency and Computation: Practice and Experience.,., 22:617–642, April 2010.

[13] X. Fei and S. Lu. A dataflow-based scientific workflow composition framework. IEEE Transactions on Services Computing, 99(PrePrints), 2010.[13

[14] U. Park and J. Heidemann. Provenance in sensornet republishing. In IPAW, pages 280–292, 2008.

[15] M. K. Anand, S. Bowers, T. McPhillips, and B. Ludäscher. Efficient provenance storage over nested data collections. In Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, EDBT '09, pages 958–969, New York, NY, USA, 2009. ACM.

[16] M. K. Anand, S. Bowers, and B. Ludäscher. A navigation model for exploring scientific workflow provenance graphs. In Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science, WORKS '09, pages 2:1–2:10, New York, NY, USA, 2009. ACM.

[17] P. Macko and M. Seltzer. Provenance Map Orbiter: Interactive exploration of large provenance graphs. In Proceedings of TaPP '11, 3rd Usenix Workshop on the Theory and Practice of Provenance, Crete, Greece, June 2011.

[18] O. Biton, S. Cohen-Boulakia, S. B. Davidson, and C. S. Hara. Querying and managing provenance through user views in scientific workflows. In ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering, pages 1072–1081, Cancun, Mexico, April 2008. IEEE Computer Society.