I give permission for public access to my thesis for any copying to be done at the discretion of the archives librarian and/or the College librarian.

Signature: **Date:**

ABSTRACT

Scientists use technology ubiquitously to collect and process data. They often use software to handle massive datasets and produce scientific results, which they post on the web, making them readily available to the public. Flaws and differences in the way data is collected and processed can impair its usefulness for interpretation. To ensure the authenticity and reproducibility of that result, as well as to improve the result by incorporating corrections in its processing, it is essential to be able to trace the provenance, or history, of the results.

Data provenance is defined as the information describing all entities - procedures and data - that were involved in producing a result. We aim to create a software tool that provides provenance for scientific data analyses. The major issues in this research are collecting, persisting, querying, and visualizing the provenance. The amount of data provenance is usually massive and challenging to present in a meaningful way.

The focus of my work is on persisting provenance and developing the interface for interesting queries so that they can be made by a non-programmer. We are using the example of a hydrological study at the Harvard Forest which measures stream discharge as a function of other quantities. We use a definition of

the process written in the graphical programming language Little-JIL to generate a graph (Data Derivation Graph or DDG) documenting the provenance of the data for each process execution. We store the DDG into an RDF (Resource Description Framework) database, making it available for querying. We provide a GUI that allows the scientist to query the provenance data without becoming an expert in database technology.

Capturing, persisting, and querying the provenance of scientific data

Senior Thesis Project

Sofiya Taskova

Mount Holyoke College Class of 2012

ACKNOWLEDGEMENTS

I would like to extend my thanks to my senior thesis advisor Professor Barbara Lerner, my summer REU advisor Dr. Emery Boose, my REU project partners Garrett Rosenblatt, Andrew Kaldunski, Morgan Vigil, the student researcher Cory Teshera-Sterne who set up the project in the summer of 2009, Professor Lee Osterweil, Professor Lori Clarke, Sr. Software Engineer at UMass Amherst Alexander Wise, the UMass Laboratory for Advanced Software Engineering Research (LASER), the Miller Worley Center for the Environment, Mount Holyoke College for funding, and the Harvard Forest for the opportunity to participate in the Harvard Forest Summer REU Program.

Table of Contents

List of Figures

I. INTRODUCTION

Technology is pervasive in scientific research in all stages from the collection to the analysis of scientific data. Electronic sensors have allowed for automated collection of data with sampling rates tens, hundreds, or even thousands of times higher than previously possible. There is also a massive increase in the rate and complexity of data processing thanks to computer software. Lastly, computer networks have facilitated greatly the collaboration between scientists. There are multiple websites of organizations and universities which make their data sets available for other scientists and the general public. The quality of the different data sets varies, which can potentially compromise the quality of the conclusions made from the data analysis. Any flaws and differences in the way data are collected and processed can make the results less useful for interpretation.

To produce good trustworthy data, the scientist needs to supply a method of verifying that her results are not accidental but rather the product of a well-defined process.

The recording of all procedures and all other data involved in the production of results helps to establish the authenticity of results. It also makes possible the reproducing or reenacting of the experiment or analysis, which are essential for verifying or improving the results, for testing the effectiveness of the

process, and for comparing alternative manipulation techniques. Due to the speed of data collection, it is no longer possible for manual documentation of all the details of how the data is being manipulated.

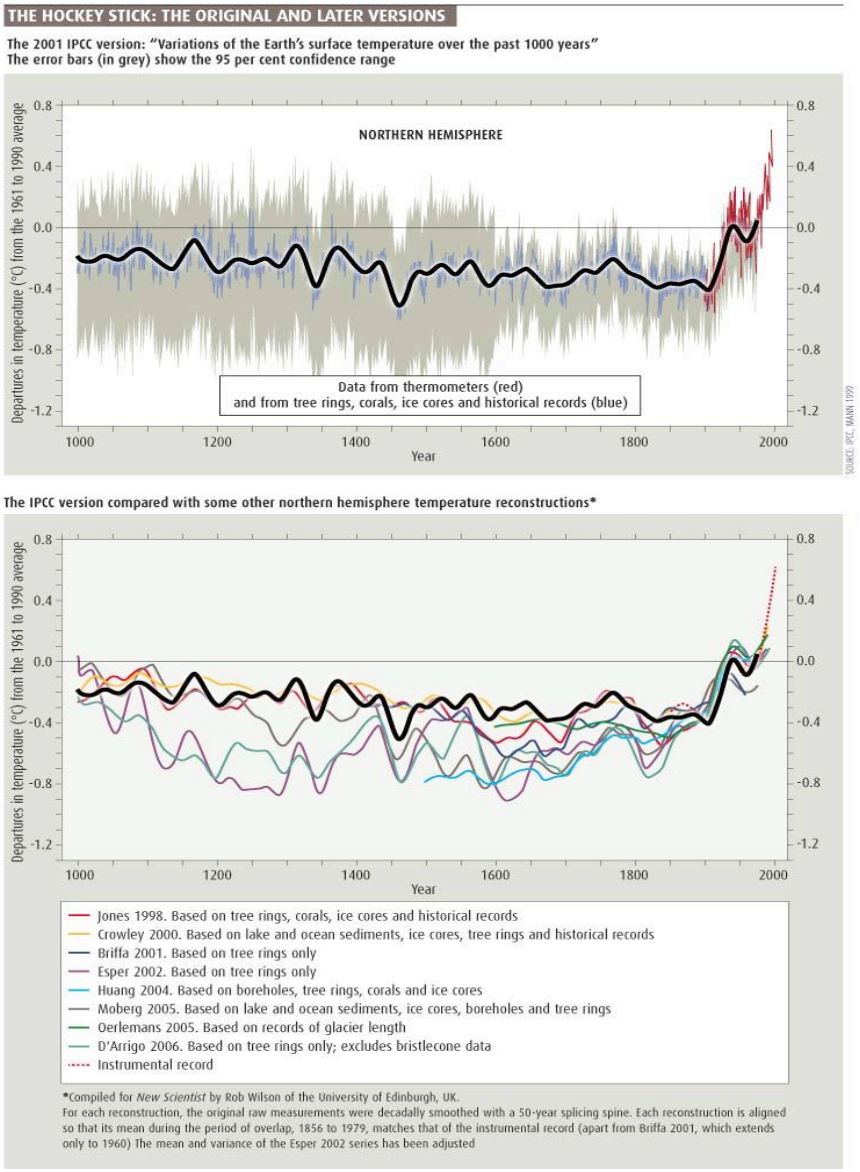

Figure 1. The "hockey stick" graph. [11]

Data provenance is defined as the information describing all entities, procedures and data, which were involved in producing a result. It is synonymous

with the ancestry of data, or history of how a piece of data came to exist. Provenance is critical for establishing the authenticity and reliability of data, reproducing the results, comparing procedures, and correcting the procedure that output the results “after the fact”.

An example of a study where provenance is crucial is the attempt to measure the average temperature of the Northern Hemisphere for the past 1000 years. The results are known by the “hockey stick” graph (Figure 1). The blade of the hockey stick describes the unprecedented slope at which the temperature has been rising in the past century. It is important to note that in the lower graph in Figure 1, the different colored curves show the results obtained from different sources. Because the results from this study may dictate many economic and policy decisions, different parties can choose which data to use to serve their interests. While *Science* publishes that “Global Warming Is Changing the World” [12], *Business World* claims that climate change is a “myth” [13]. It is difficult to choose which sources to trust, and being informed on the provenance of the data shown in the publications can help make good choices and, in this case, take proper care of the planet.

In order to be reliable, provenance needs to be collected concurrently with the processing of the data. The collected provenance needs to provide a handle to the entire process so that it is possible to analyze the results and to experiment with different processing techniques post-factum. It also needs to provide a way to check the quality of the data and to determine resource usage, to attribute

procedures to the agents who performed them so as to allow for accountability, and to give contextual information on the analysis or experiment.

We are interested in building software that targets all the above requirements, making provenance readily available for the user to explore. We expect the user group of the provenance software to be made up of mostly non-programmers, so using our software must not require as little programming experience as possible.

In this paper I will show that it is possible for a non-programmer to access meaningful data provenance on any data set derived from any abstract process. I will present some background information on the problem of collecting, storing, and querying data provenance with the use of the specific example of a hydrological study at the Harvard Forest. I will support my main claim with a demonstration of a program which collects and stores data provenance on the fly – as data is being processed – and allows the user to obtain, using only a Graphical User Interface (GUI), a meaningful response to a data provenance query that has been deemed interesting for a wide group of processes.

I.1. A Motivating Example: Harvard Forest Hydrological Study

The example being used in the project is the hydrological study conducted by ecologist Emery Boose and other scientists at the Harvard Forest Long-Term Ecological Research (LTER) site in Petersham, Massachusetts. The study attempts to measure the movement of water within an ecosystem by measuring precipitation, evapotranspiration (the sum of evaporation and plant transpiration from the Earth's land surface to atmosphere), sap flow in trees, groundwater (water located beneath the ground surface), and stream discharge (the volume rate of water flow). From data collected with the respective sensors for each of these measurements, the scientists are looking to derive conclusions about variations in the flow of water in the system caused by factors such as climate change, differences in topography, soils and vegetation. [1]

The study measures water flux at several different frequency rates from every 15-60 minutes, with daily and annual summaries depending on the nature of the measurement (e.g. evapotranspiration, groundwater response to snow melt, and ecosystem response to climate). The example I used for my thesis is the measurement of stream discharge with a value collected every 15 minutes.

The data collection is done with electronic sensors, some of which are connected via a wireless network, from where the data can be processed and made

public (posted on the Internet) in real time for use by other scientists and the general public.

A simple example of a study to demonstrate the data requirements and analysis involved is a study of the water balance using three measurements from one single watershed. This simple system is governed by the formula

$$P - ET - Q = dS$$

where P = precipitation, ET = evapotranspiration, Q = stream discharge, and dS = change in water storage. Each term in the equation could be an instantaneous rate or an amount integrated over a time interval [1]. The equation states that the change in water storage in the system is equal to the precipitation (water coming into the system) minus the sum of evapotranspiration and stream discharge (water leaving the system).

The measurements are made with different sensors: a meteorological station measures precipitation and an eddy flux tower measures evapotranspiration. Both are transmitting data wirelessly. A stream gauge measures stream discharge. It requires manual collection of the data using a Palm Pilot and transferring the data into spreadsheets (see Figure 2). For some of the quantities, multiple independent gauges are used to make sure that the measurements are accurate. Rules are established to trigger an error signal in case the different gauges do not agree within an acceptable range. Evapotranspiration is measured at the eddy flux tower and modeled using linear regression on recently measured related quantities. Stream discharge is measured as follows: a

pressure sensor is placed at the bottom of the stream in order to measure the height of the water next to a barrier that only allows water to pass through an opening with a certain shape (see Figure 3). The geometry of the shape allows to determine the amount of discharge. Fifteen minute averages are taken by the on-site data logger (an electronic device which records the data over time) via an external sensor, which in turn is placed at the bottom of the stream. Outliers (as defined by the ecologist) are removed for quality control purposes. Then, the measurements are converted into the meaningful quantity, e.g. "amount of stream discharge" (using parameters and calculations also provided by the ecologist).

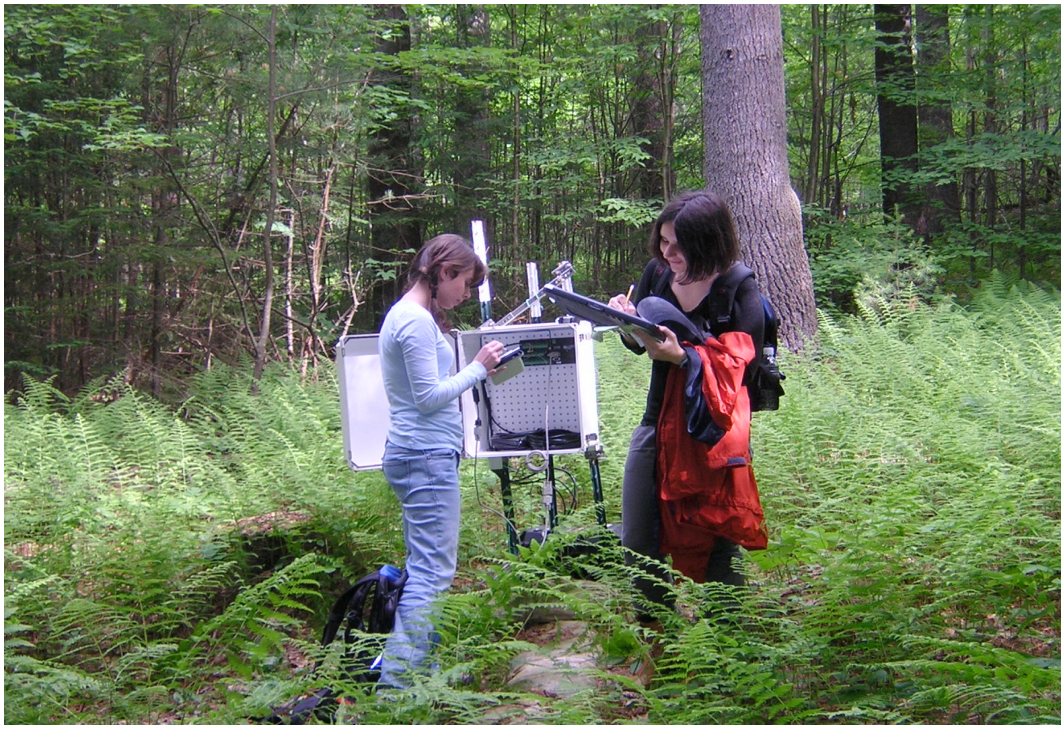

Figure 2. Manual data collection to be used for calibrating the sensors. Photo: Aleta Wiley

Figure 3. Big Weir in Nelson Brook at the Harvard Forest. At the bottom of the weir behind the barrier are placed pressure sensors. The shape through which the water flows is restricted by the barrier to be regular, allowing for the discharge to be calculated solely based on the input data for pressure. Photo by Emery Boose

If the measured value is not within the acceptable range defined by the scientist, the value recorded by an alternative gauge may be used. If all measured values are out of range, a modeled value may be substituted to avoid holes in the data. The latter are the most undesirable from the point of view of processing and analysis, which require data for each point in time.

Post-processing of the data is necessary in order to introduce a model with which to replace missing values or “fill the gaps”. Gap filling needs to be done

either because data was not recorded by the sensor or because the recorded value from the sensor was found unsuitable or not meaningful by the scientist and therefore removed. A frequently encountered example of post-processing in the hydrological study consists of a readjustment of a number of outputs over a period of time for which sensors are discovered to have drifted out of calibration.

To allow for real-time quality control and to enable gap filling when measured values are unreliable or when sensors fail, modeling the measured quantities is pivotal.

In the hydrological study the data may be modeled with a simple linear reservoir model which uses recent measured values of precipitation and discharge. Essentially, the value to be inserted is chosen by extrapolating from the curve of the most recent existing values. Because new measurements are constantly incoming, post-processing is also done daily or monthly for each quantity. Provenance here is the only tool that would allow whoever is using the results to know whether a particular value is obtained from processing a measured value or from inserting a modeled value. If the value is modeled, provenance should also be able to answer the questions whether the original value was missing or considered an outlier, and what criteria for determining outliers were used.

The resulting data points are then posted on the web for public use. The provenance that needs to be posted online includes original measurements, rules on how quality control is performed, original and updated models with model IDs

and update times, and a dataset of all quantities of interest after all post-processing, along with the data source (e.g. real-time measurement or modeled value), and processing history.

Several challenges arise when collecting the provenance of the data in the study described in this chapter. The collection of provenance would ideally be simultaneous with the execution of all procedures starting from the sensors collecting data in a stream and ending with the most recent version of the data posted on the web. The storing of the provenance into a database should ultimately also be done immediately after the collecting to enable queries to be performed on that database. The size of the provenance database has to be minimized without compromising any of the provenance that is deemed by the scientists and the users of the data to be interesting and important. A number of additional considerations regarding the collecting, storing and querying of provenance are discussed in Chapter IV.

II. BACKGROUND AND RELATED WORK

Definitions:

Provenance of scientific data – the meta-data that describes all the procedures and data involved in producing the results [2]

Process – a system in which actors – computer software or humans – interact and operate on data [2]

Workflow – a sequence of connected steps connected via data flow between the steps

In order to make data provenance available to the scientist, we need to start by defining strictly the process that is used for obtaining the results. For that purpose we use workflow, which, when executed, allows us to create a graph of the provenance. In this chapter, I will provide more detail about how a workflow is defined and a provenance graph created, focusing specifically on the process language Little-JIL.

Osterweil et al. [2] propose a solution to making available the provenance of scientific data that allows the data to be reproduced by any other independent entity, verified, and analyzed for defects such as errors in how the result was produced from the raw input to the statistical analysis. We are using the approach

proposed in the paper to write a program for collecting, storing, and querying data provenance.

We adopt the latter solution by following the steps:

1. Define the scientific process used on the hydrology project using a Process Definition Graph in Little-JIL
2. Execute the process creating a Data Derivation Graph to record the provenance
3. Store the provenance in a database
4. Provide a tool to allow the ecologist to query the provenance.

The approach involves the use of two types of graphs: a Process Definition Graph (PDG) and a Data Derivation Graph (DDG). The PDG is built using the graphical process language Little-JIL [2], details on which can be found in the next section. The PDG describes all possible ways in which the data can be correctly processed. It provides blueprints of the artifacts that can potentially be handled by the process, as well as blueprints of the procedures or actions that constitute the process. We will refer to those procedures as steps, as they are called in Little-JIL. Each step is represented by a node in the PDG.

The DDG on the other hand documents the steps and data that were involved in each particular execution of the process. It represents a trace of the execution of the PDG on some specific data input. Its nodes can represent either step or data instances. As an action defined in the PDG is executed for a particular input, that execution is represented by a node in the DDG. Similarly, as a piece of

data is plugged in for a Little-JIL input or output blueprint, a data instance node is created in the DDG to represent that data at that stage of the execution of the process. We will go into detail on how the PDG and the DDG relate to each other at the end of this chapter.

It is important to note that the approach meets important requirements in terms of making data more reliable: it provides full detail on all possible executions of the data's processing, as well as a complete description of all the procedures (code, calculations), additional data, and models used for each execution of the process.

II.1. Little-JIL

The language Little-JIL is a graphical process programming language with rigorously defined semantics. It has been created at the Laboratory for Advanced Software Engineering Research (LASER) at the University of Massachusetts, Amherst. It can be used to define the scientific process.

The purpose of Little-JIL is to allow a formal, rigorous description of any process, however complex the interactions of software and humans. Little-JIL relies on the assumption that each agent (piece of software or human) within the process knows how to perform its task, and so the language itself only supplies coordination of the activities. An activity performed by a single agent is the basic unit of the Little-JIL process – the step. And so each agent is responsible for

completing its step and reporting back with a result, or reporting that the step could not be completed successfully.

While developing a process requires programming expertise, Little-JIL was built to allow non-programmers to understand processes, using several abstractions of the kinds of components that can comprise a process, such as an agent, an event, a resource, an exception etc.

Little-JIL allows the user to express a range of detail on conditions in which the step should be executed and different ways the data could be handled depending on the properties of the data. The clear and rigorous process description in Little-JIL makes it possible to analyze the process and make statements about its correctness, efficiency, and reliability.

Little-JIL assumes that a coordinated process includes the following elements:

- agents, each of which is assigned one or more tasks to perform
- communication paths between the agents
- a coordination mechanism allowing the agents to work in a distributed environment

Each agent has an agenda which allows it to communicate with the central process and can migrate between different machines. The information the agent returns to the interpreter is whether the task was done successfully and what data needs to be updated.

As mentioned earlier, the step is the basic unit of the Little-JIL diagram, representing a unit of work in the process. The control flow is described by

decomposing steps into substeps which are added as their children, and by specifying the sequencing of execution of the children with a special badge on the parent. Steps are essentially blueprints of the processing that the data needs to undergo, and at runtime those blueprints are instantiated.

A program in Little-JIL is a tree whose nodes are the basic process units – the steps. In the nodes and the edges between the nodes are incorporated ways to specify control flow, scope of variables (known as parameters), requisites, exception handling, messages, and resources.

Non-leaf steps are steps that are only responsible for control flow. There are four types of non-leaf steps: sequential, parallel, try, and choice. The type of non-leaf step determines the order in which the children of the step are to be executed. The children of sequential steps are executed one by one from left to right (each child only starts executing after the preceding child has completed). The children of a parallel step can (but do not have to) be executed at the same time. The try step requires that exactly one of its children executes successfully: each child executes consecutively from left to right until the first child to return a success. The choice step allows agents to decide, while the process is running, which one child of the choice step to execute (no other children are executed).

The fact that agents are free to choose which child of choice non-leaf steps to execute adds to the need for recording provenance to trace exactly which path on the Little-JIL graph was taken by a given run of the process.

Steps also support the option of adding requisites, or tests, which allow the action of the step to be performed only when the input data meets certain conditions.

There are prerequisites, which perform that test before the step executes, and post-requisites, which perform a test immediately after the execution, possibly on the result returned by the step. Requisites are also steps themselves.

Another attribute of the step can be the exception and the handler for the exception. The exception allows the user to specify what types of errors they can expect to occur during the execution, what agents they want to handle the exception, and how the exception should be indicated and handled. The search for an appropriate handler, should an exception occur, is done by walking up toward the root of the tree. The actions to be taken after the exception is handled, such as retrying to execute the step, continuing to the next step, or terminating the whole process, can also be specified in the language.

Parameters are the variables that hold the data being manipulated by the process. They are specified as attributes of the step. The direction in which they are passed to other steps is also specified - local parameters only get passed to and from the children of a step; in parameters are only passed from the parent of the step; out parameters are only passed to the parent of the step; in-out parameters are passed from and returned to the parent of the step.

Figure 4 shows the visual representation of the language's features. The control flow badge shows the step type - sequential, parallel, choice, or try (legend also on Figure 4). The remaining badges have identical names with the features they stand for.

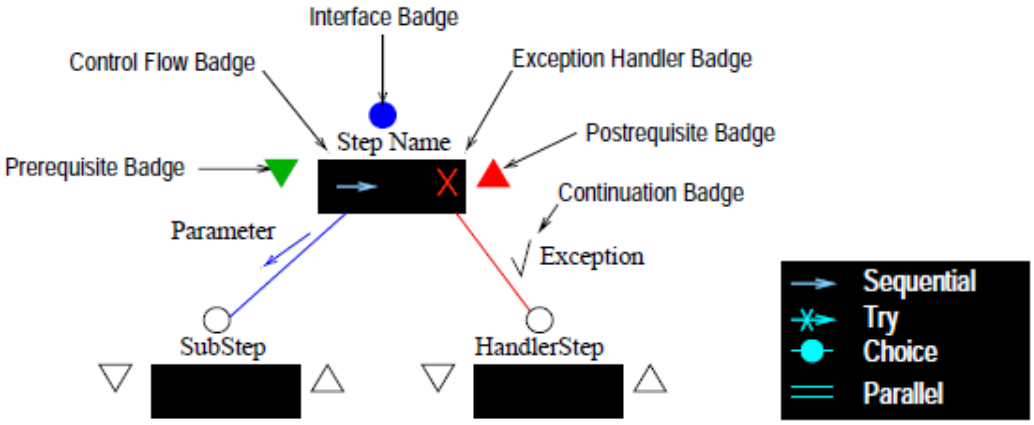

Figure 4. Legend of the Little-JIL syntax. [14]

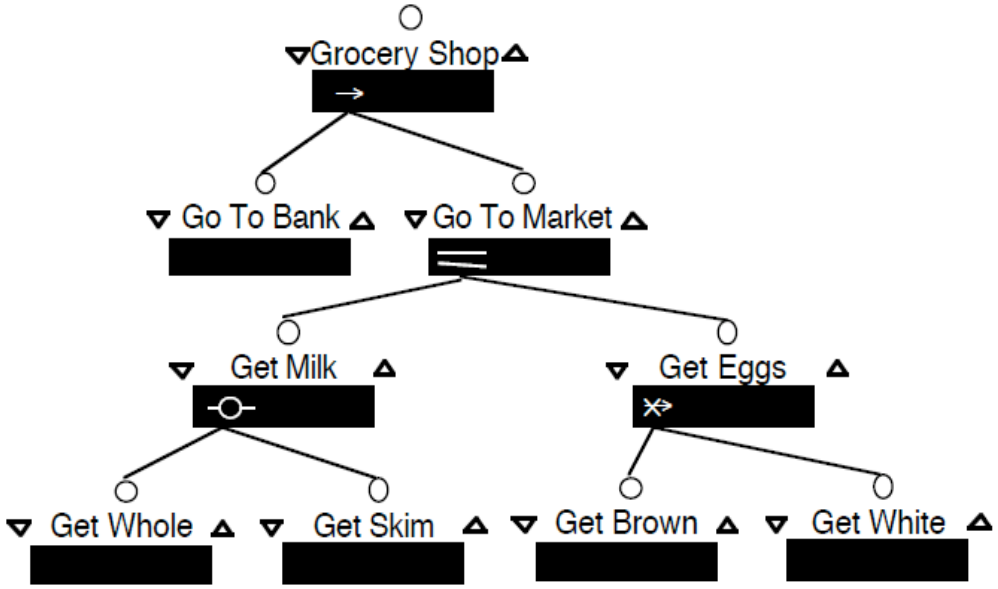

Figure 5. An example process defined in Little-JIL. [15]

Figure 5 shows a sample Little-JIL process. The badges shown illustrate all types of possible sequencing of substeps of a step: step Grocery Shop has a "sequential" badge (performs all substeps in order from left to right), Go To Market has a "parallel" badge (all substeps are started at the same time), Get Milk has a "choice" badge (choose one substep to perform), and Get Eggs has a "try" badge (try to complete substeps in order from left to right until one substep completes).

II.2. Related work

There are other systems used by scientists to describe scientific workflow. Many of those also support the collection of provenance data. Best known of those are Taverna [3] (UK), Kepler [4] (US), and VisTrails [5] (US). Provenance Map Orbiter [6], Provenance Browser [7], Zoom* UserViews [8], and The Open Provenance Model (US) [9] are also relevant to my thesis as projects focusing on visualizing and communicating provenance.

Taverna, Kepler, and VisTrails cater to the needs of the scientist and are already being used. The main distinction between those systems and Little-JIL is that they are primarily data flow languages, whereas Little-JIL is primarily based on step abstraction and decomposition. We believe that the step decomposition approach scales better as processes increase in complexity. Step abstraction allows for the ability to study and understand a process at varying levels of

abstraction. Taverna and Kepler both display provenance graphs and Kepler shows a log of provenance events. VisTrails is designed to support browsing provenance and data mining and has highly developed visualization features. It differs slightly in objective from Little-JIL in that it also traces changes to the workflow itself.

Provenance Map Orbiter is a visualization tool developed at Harvard University and has the advantage of summary nodes which are used for semantically zooming in and out, allows for nesting of summary nodes, and supports filtering of the provenance information. For example, if a collection of nodes constitutes a procedure, the details of which are not interesting to the user, that collection is represented by a single node which would most likely be named the same as the procedure that the collection of nodes constitutes. This allows for a much clearer visualization of the provenance graph, highlighting only the portion of the graph important to the user. Provenance Browser supports filtering and multiple views (based on context provided by the user about which part of the graph is interesting and at what scale), unlike Orbiter.

Zoom*UserViews is another visualization tool that allows the user to select relevant nodes of the provenance graph but does not support nesting of summary nodes. In summary, each of the multiple visualization tools achieve some success in addressing the problem of provenance querying visualization.

The Open Provenance Model (OPM) uses an approach very similar to the approach of our project: it represents provenance with a directed acyclic graph of

causality with annotations on the graph describing the specifics of the particular execution. The nodes in the provenance graph here are artifacts (physical objects or instances of objects in a computer system), processes (actions performed on the artifacts), and agents (entities executing the processes). The OPM places specific focus on allowing provenance to be collected across different systems that are operating on the same data, and to keep the provenance collecting software agnostic of the technologies used for processing of the data. OPM also allows freedom for any internal representations of provenance in the data processing technology to be kept the same. The OPM does not have as its goal to specify protocols for querying provenance repositories, which we intend to do in our current project.

III. PROJECT

Provenance describes the origin of data and the transformations applied to them. In particular, workflow provenance captures the process and data dependencies of an output, which processes produce the output, and which input those processes have used. The main goal of the software engineering project I worked on is to collect data provenance using the example of the processing in the hydrological study, to make it persistent in a database, and to allow for the user of the software to retrieve answers to interesting provenance queries without having to learn a database query language.

III.1. Data Derivation Graphs

Because the ancestry relationships of data are described by a directed graph, I use a directed graph referred to as a Data Derivation Graph (DDG) [2] to represent provenance.

A DDG describes the specific unique history of how a data set was derived, and so when constructing the DDG we are concerned with the instance of an operation performed on the data (as opposed to the description of the operation

and all the possible ways in which it can be performed) as well as the instance of data rather than the variable which contains it.

The DDG has two types of nodes: procedure instance nodes (PINs) and data instance nodes (DINs). A PIN is created to represent a particular execution of a Little-JIL procedure. Respectively, a DIN stands for an instance of data, and one DIN is created each time a data item is changed by a Little-JIL leaf node. The present convention is to direct edges from successor to predecessor and from product to producer.

The following example demonstrates how a single execution of a process defined in Little-JIL is represented in the DDG.

On the top side of Figure 6 is a simple Little-JIL process of making an ice cream sundae. On the bottom is the DDG that describes an execution of the Little-JIL process. As the DDG is currently visualized, ovals represent PINs, nodes corresponding to non-leaf steps are colored green and nodes corresponding to leaf steps are colored yellow (see legend in Figure 6). DINs are shown as blue rectangles in the DDG.

In the example of Figure 6 data is passed from step Make A Sundae to steps Scoop Ice Cream and Add Toppings, and from step Scoop Ice Cream to step Add Toppings (the directed edges of the DDG point toward the procedures that produced the data).

The data product “Vanilla and strawberry scoops” is created with the information on what the preferred flavors are at the particular instance of the

process execution. Here, the parent step Make A Sundae passes “Vanilla and Strawberry” as favorite flavors to the leaf step Scoop Ice Cream.

There is a directed edge in the DDG pointing from the DIN “Vanilla and Strawberry” to its producer, the PIN “Make A Sundae”; there is also a directed edge from the user "Scoop Ice Cream" of the DIN “Vanilla and Strawberry” to that DIN itself. Note that edges that show control flow (i.e. edges between PINs only) and edges that show data flow (edges between a DIN and a PIN) are visualized differently.

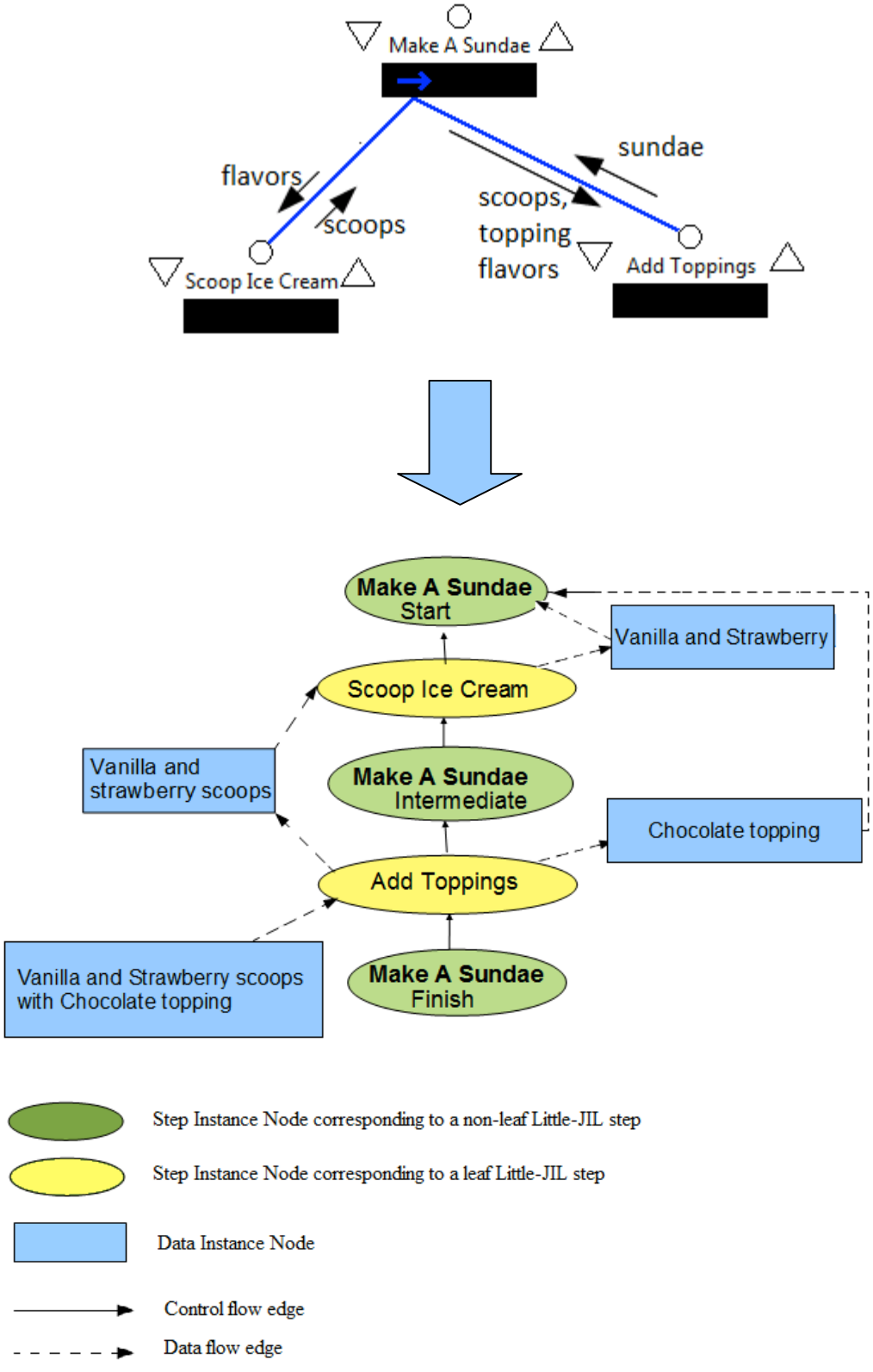

Figure 6. Traversing a Little-JIL tree.

I have been working on the data provenance project under the mentorship of Prof. Barbara Lerner and Emery Boose for two summers as an REU student at the Harvard Forest and in the academic year 2011/2012. The essence of my work in the summer of 2010 was using the hydrological study described in chapter I.1 as a demonstration platform in order to build DDGs that keep all the potentially interesting details of the process execution.

Figure 7 shows the current Little-JIL definition of the data processing for the hydrology project. The root step of the process is “Get Data”, which is decomposed into two main child steps: “Get Measurements” and “Do post processing”. As mentioned earlier, it is necessary to have existing values for each point in time in order to perform analysis on the data. While the left subtree retrieves measurements from the sensors, the right subtree effects the post-processing of the data, which includes substituting missing values and outliers with modeled values, as well as correcting values that have been found to be affected by sensor drift. The two main subtrees are executed in parallel, as

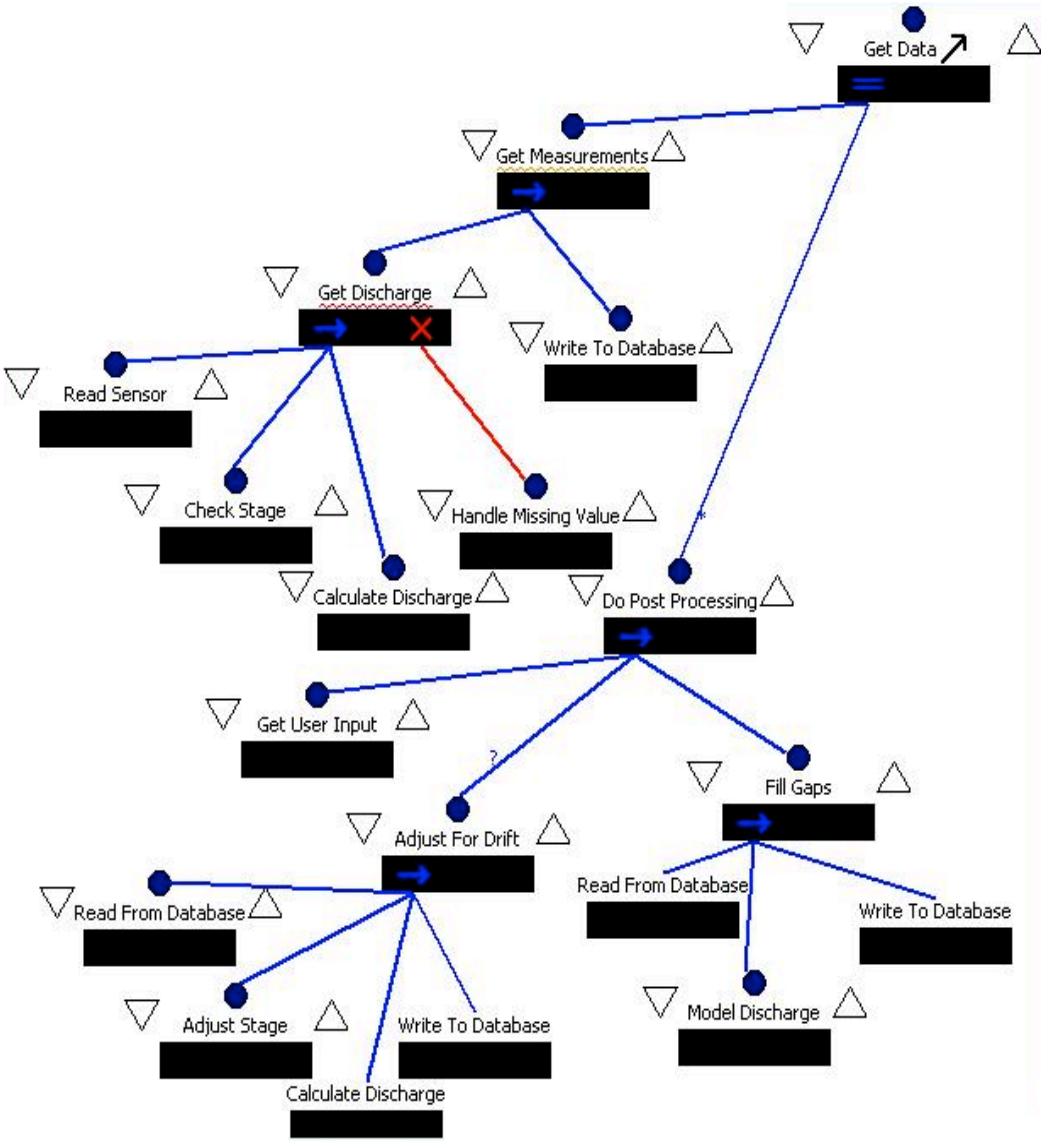

Figure 7. Current Little-JIL process definition for the Harvard Forest hydrology project. Image courtesy: Andrew Kaldunski

dictates the badge on the "Get Data" step – the post-processing should be able to execute whenever necessary, simultaneously with the collection of incoming data.

“Get Measurements” consists of sequentially executing “Get Discharge” and “Write to Database”, and “Get Discharge” consists of reading the output of the sensors (step “Read Sensor”), determining whether the sensor output is an outlier (step “Check Stage”), calculating the desired quantity using the input data and the calculation supplied by the scientist (step “Calculate Discharge”), and propagating the result up to the root of the tree. If the sensor output is an outlier, that causes an exception in the execution of “Get Discharge” and the step “Handle Missing Value” is responsible for making sure that post-processing is done for that value so that a modeled value can be written to the database instead.

In the right subtree, the “Fill Gaps” step is responsible for reading the recent values in the database and performing the calculation (also provided by the scientist) to obtain a modeled value, which is then written to the database in place of the “gap”. Step “Adjust For Drift” takes the values from the database which the scientist has found to be compromised by sensor drift, as well as the numerical quantity describing the drift, and adjusts those values, finally writing the corrected values to the database.

The label on the edge between “Get Data” and “Do Post Processing” indicates that the subtree rooted at “Do Post Processing” may be executed zero or more times, and so the corrections for sensor drift and the replacement of measured values with modeled values may happen repeatedly.

In view of this particular example it is important to note that the data provenance needs to contain information on whether a value was measured or

modeled, whether it has been corrected due to drift, and what it used to be before each correction or replacement with a modeled value. With time, the calculations used for modeling and the criteria for determining outliers may change. Therefore, we also need to store information about the versions of the check-stage and model techniques used for each execution.

It is important to note that while it helps to collect as much provenance as possible on the execution of a process, memory requirements of the DDG rise at a very high rate, making the storage of provenance very expensive.

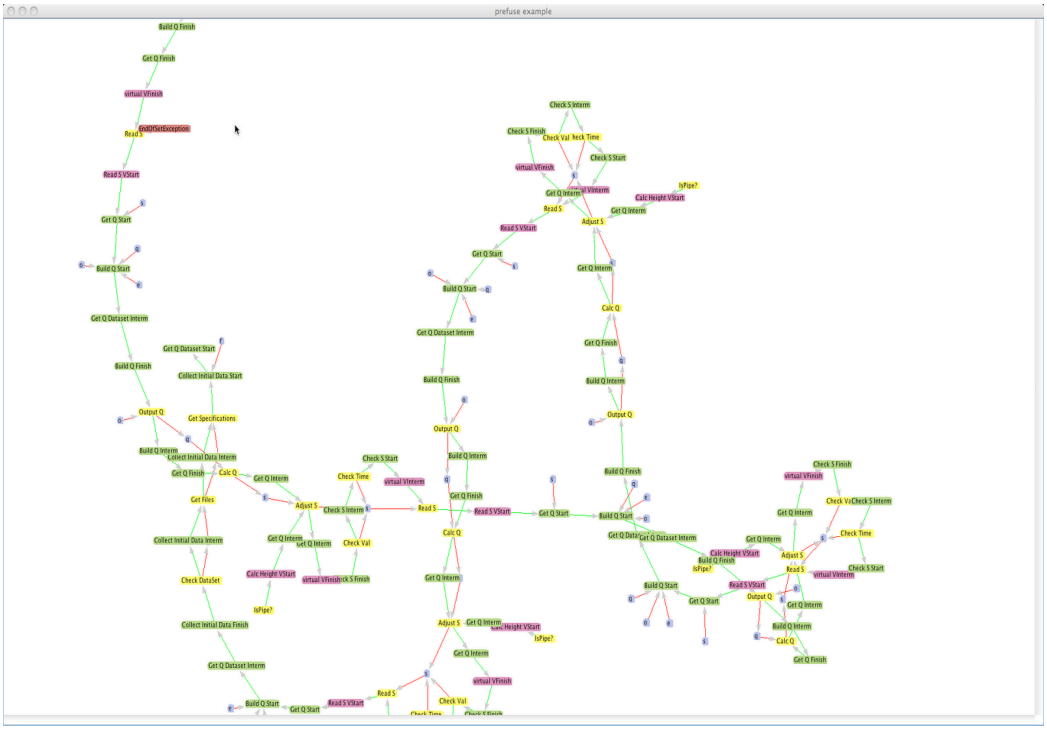

Figure 8. Example process execution DDG from the Harvard Forest hydrology project. Image courtesy of Barbara Lerner

To illustrate how quickly the DDG becomes very large and complex, Figure 8 shows an example DDG produced for an execution of the hydrological

process for only 4 input values. The space constraint is an issue that needs to be addressed in the future.

In my first summer at the Harvard Forest I created the Java representation of DDGs and was involved in determining how to coordinate the DDG-builder with the Little-JIL interpreter to construct a DDG as a process executed. I was also involved in discussions of how DDGs should be built to provide all detail that may be necessary while observing storage and performance limitations. Prof. Barbara Lerner and Alexander Wise modified the Little-JIL interpreter code to build DDGs simultaneously with execution of processes defined in Little-JIL.

III.2. Collecting Provenance

I've created two separate libraries for collecting data provenance – one is abstracted from the Little-JIL process definition language and one is assuming that Little-JIL was used to define the process for which provenance is being collected. I broke down the implementation into a language-independent and a Little-JIL-specific piece which inherits from the language-independent API in order to be able to adapt the code to other process definition languages by simply extending the abstract API.

The library laser.ddg is made to aid in the construction and traversal of a Data Derivation Graph. It is agnostic of the Little-JIL process definition language.

The library provides support for traversing the DDG (and eventually querying and displaying the DDG).

A class diagram of the three most interesting classes is shown in Figure 9. A DDG consists primarily of two types of nodes: data instances (Data Instance Nodes or DINs) and procedures (Procedure Instance Nodes or PINs). Procedure Instance nodes are connected to Data Instance Nodes on input edges and output edges. Similarly, Data Instance Nodes are connected to the Process Instance Nodes that produce and consume the data.

A Procedure Instance Node may have an output edge to a Data Instance Node, which then has a used-by edge to another Procedure Instance Node that uses the data as input. All edges are bidirectional, so we could also traverse the DDG from a Procedure Instance Node, follow an input edge to a Data Instance Node, and then follow a produced-by edge to the Procedure Instance Node that produced the data.

Procedure Instance Nodes also connect directly to each other. These are referred to as predecessors and successors of the Procedure Instance Nodes. The edge between the predecessor and successor is always present, regardless of whether data is passed between the two.

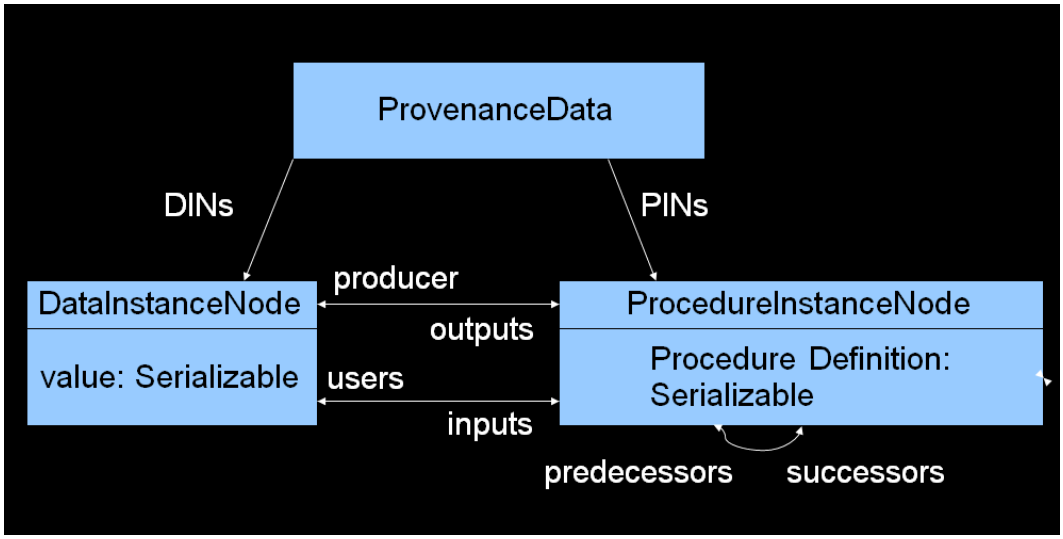

Figure 9. Simplified class diagram of the Procedure Instance Node, Data Instance Node, and Provenance Data classes and their attributes

The data instance node holds a state of a data entity whose processing is being recorded with the use of a DDG. Similarly to the data it wraps, it can only have one producer, but multiple users. The producer and users of the DIN can be used as connections to the respective procedure instance nodes in the graph to provide for the bidirectional traversal of the DDG. (It is considered impossible for two DINs to be immediately adjacent in a DDG since a new DIN is constructed only in the event that the data undergoes a transformation, which is represented as a Procedure Instance Node).

The ProcedureInstanceNode interface is extended by three interfaces: StartProcedureInstanceNode, IntermediateProcedureInstanceNode and FinishProcedureInstanceNode. Those become useful in the cases where a procedure is decomposed into smaller procedures. For example, a method (we

refer to this method as the parent method) may call one or more other methods. When the parent is initially called, a StartProcedureInstanceNode should be created. After a child of the parent method completes execution, control returns to the parent. This would be represented in the DDG as an Intermediate Procedure Instance Node or a Finish Procedure Instance Node depending on whether there are any children of the parent method that have not yet been visited. If the last child of the parent method has been visited, a Finish Procedure Instance Node is instantiated. If all child methods have completed successfully, the Finish Procedure Instance Node should be set to completed. If an exception is thrown by any of the child methods, the exception should be represented with a Data Instance Node that is output from the child Finish Node to the parent. The child Finish Node should be marked as terminated. If the parent propagates the exception rather than handling it, it should also have a Finish Procedure Node that is marked as terminated and that outputs the exception Data Instance Node. For methods that are not decomposed in the DDG, it is sufficient to just use a ProcedureInstanceNode.

The Little-JIL specific library laser.juliette.ddgbuilder inherits all its node types from laser.ddg, only the Procedure Instance Node types (StartProcedureInstanceNode, VirtualProcedureInstanceNode, etc.) are renamed to refer to the Little-JIL terminology, replacing "Procedure" with "Step" (StartStepInstanceNode, VirtualStepInstanceNode, etc.). The implementation of how DDGs are generated is still the same as described in the above examples. A

sample execution of a simple three-step process is shown in Figure 10. The parent A has two children B and C which are executed sequentially. The control returns to the parent after each child is executed, causing an Intermediate and a Finish Step Instance Nodes to be created.

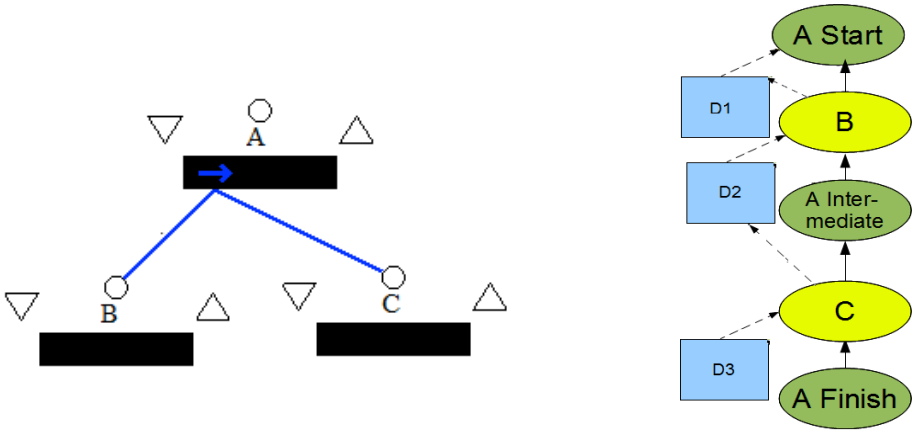

Figure 10. Sequential example. PDG in Little-JIL shown on the left and DDG for one execution shown on the right

For an illustration of how the execution of a parallel step is represented in a DDG, see Figure 11. For parallel steps there is no need to create an Intermediate Step Instance Node for the parent step because control is only returned to that step when all of its children have completed execution.

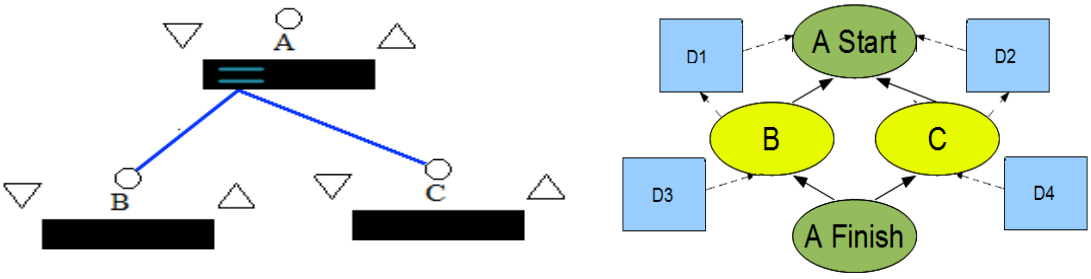

Figure 11. Parallel example. PDG in Little-JIL shown on the left and DDG for one execution shown on the right

Each time that a step produces output, it should create a Data Instance Node with a value. Its value must be serializable in order to allow the actual objects to be recorded in the DDG. In this way the Data Instance Nodes represent the different states of the same object in the process being described by a DDG. Connections should be created between the producer of the data and the data. The Data Instance Node should also be added to the ProvenanceData object, at which point it will be given a unique ID.

Each Step Instance Node has an associated AgentConfiguration object. This object contains information about the agent responsible for executing the procedure. The AgentConfiguration object can record information about the software version, configuration parameters, identify a Web service or provide other information about the execution environment. If a person plays the role of the agent, it could record who the person is.

A ProvenanceData object maintains global information about the entire graph, identifying the starting Step Instance Node, the Data Instance Nodes that represent inputs to the process described by the DDG, and the set of Data Instance Nodes that represent the outputs produced by the process. This information can be used to start forward or backward traversals of the DDG. The ProvenanceData object also maintains the collection of all Data Instance Nodes and all Step Instance Nodes to enable querying the DDG.

Let us consider a PDG slightly altered from the original hydrology project PDG (Figure 12). Assuming the value output from the sensor is deemed bad and requires exception handling, the DDG for the process would be the DDG in Figure 13. Check Stage, which did not produce output when the value was acceptable, produces an Exception Data Instance Node in Figure 13. That Data Instance Node is passed to the exception handling Little-JIL subtree rooted at Handle Bad Value (see Figure 12). The execution of that subtree leads to the creation of the modeledDischarge Data Instance Node (Figure 13), which is written to the database along with the bad value originally produced by the sensor (sensorData).

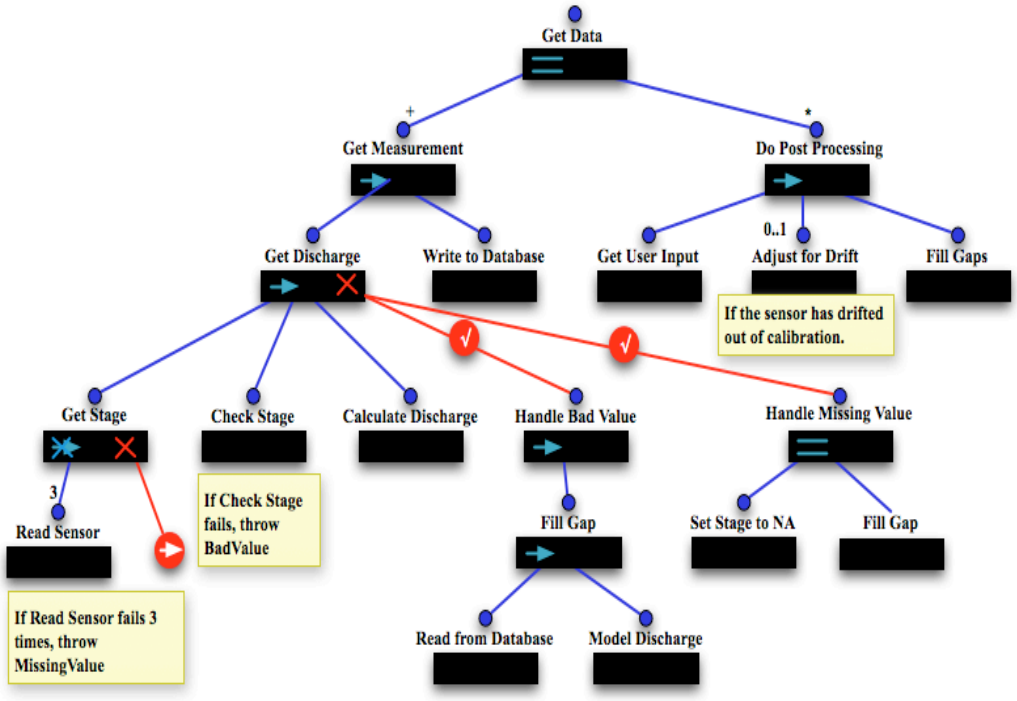

Figure 12. Alternative PDG for the Harvard Forest hydrology project. Image courtesy of Barbara Lerner

As illustrated in Figure 8, and more clearly in Figure 13, the problem that arises when generating DDGs is that the DDG grows to a massive size very quickly. It is not yet clear what parts of the DDG are interesting to the scientist, and so visualization of the large Data Derivation Graphs is hardly intuitive.

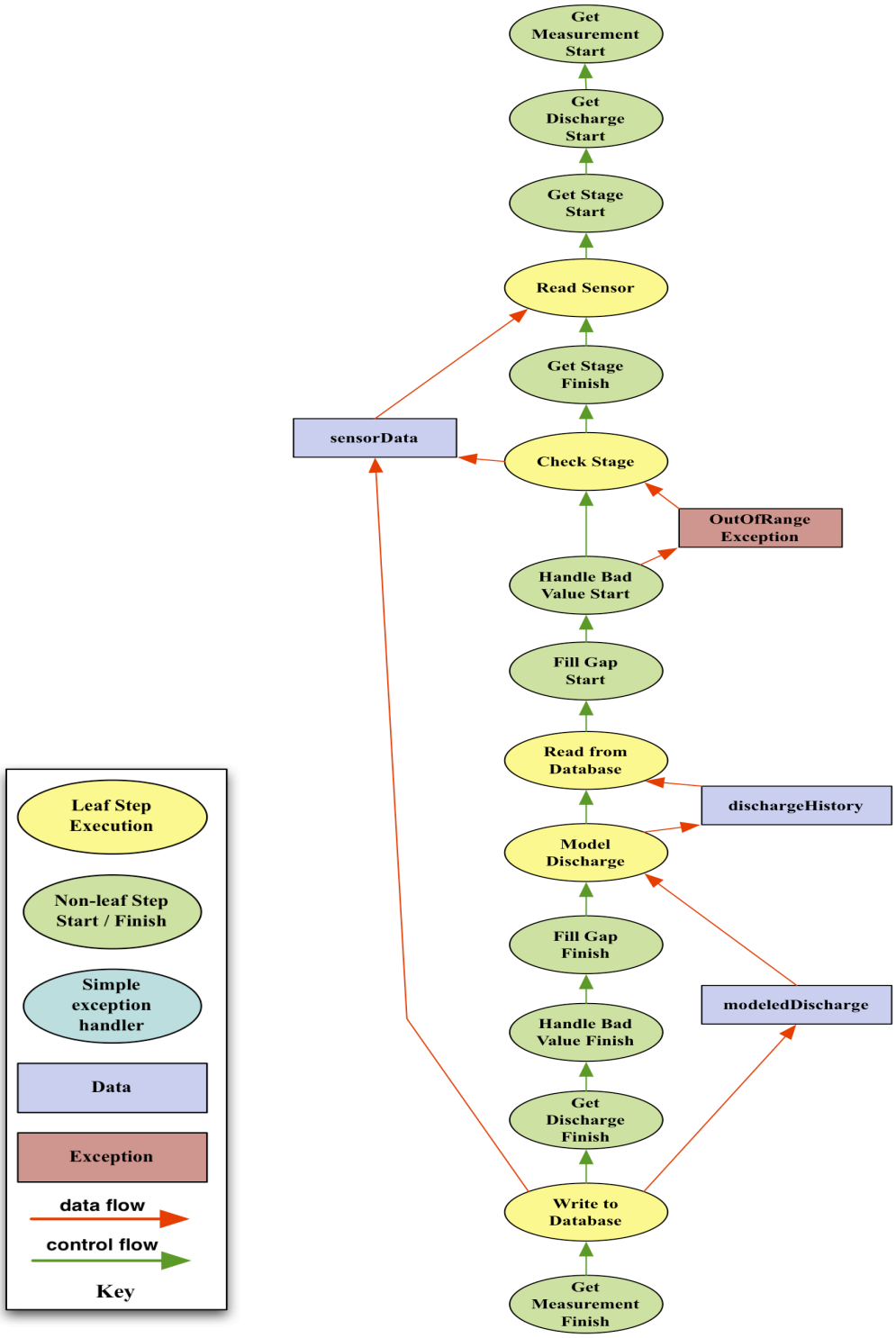

Figure 13. DDG for an execution of the process in Figure 12 given a bad sensor value. Image courtesy of Barbara Lerner

III.3. Storing Provenance

We store the DDG in a database using RDF (Resource Description Framework). RDF is a language which uses what are referred to as "triples" to define the items being stored and their relationships. Each triple consists of a subject, predicate and object. Each subject is an RDF resource and a node marked with a unique URI in the RDF graph. Each object is a property of the subject and represented by another RDF node. The edges leaving the resource node are named the same as the type of the property held by the object node, which is put on the other side of that edge. The "triple" is a statement in RDF such as the following (in English):

"Resource http://www.example.org/index.html has creator http://www.example.org/staffid/85740."

The statement consists of three parts: the URI of the resource (http://www.example.org/index.html), the type of property (http://purl.org/dc/elements/1.1/creator), and the property value ('http://www.example.org/staffid/85740'). See Figure 14 for a visual representation of the statement.

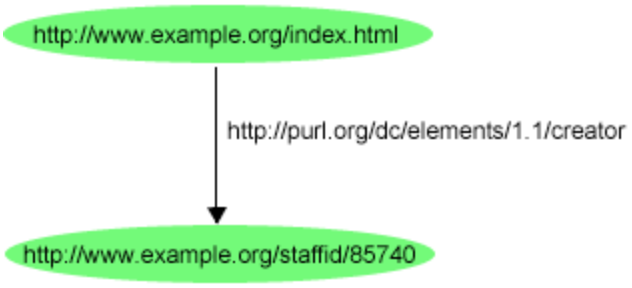

Figure 14. Example RDF statement. [16]

In the academic year 2011/12 I used the Jena toolkit to persist the DDGs created by the DDG-builder into an RDF database. The Jena toolkit is a library that allows for a seamless transition of the provenance graph of Java objects into a database of RDF statements. We chose RDF as the data model for the provenance database because of preliminary information on performance[10] and the model's natural representation of graphs.

I store each execution of a process into a directory in the file system. The provenance database is now available for querying.

III.4. Querying Provenance

The query language of RDF is called SPARQL. The querying capabilities of SPARQL are defined by the mechanism of selecting nodes in the database according to their URI, property type, or property.

The SPARQL query language uses the fact that the RDF database is organized into triples of subject, predicate, and object. The most important query

for the project is the SELECT query, which looks in the database for a given combination of subject and predicate or object and predicate to match all statements in the database with the given properties kept fixed and all other properties left variable. The syntax requires that SELECT be followed by the name of the variable to be retrieved preceded by a '?' character. Next is the clause WHERE followed by the actual pattern we are searching for in the database. We will illustrate how SPARQL queries are made using the real example of some queries that are known as potentially interesting to the scientist.

Some queries that have so far been pinpointed as potentially interesting are the following[10] :

1 Find Producer Query :Which step produced data node d43?

2 Find Data Users Query : Which step(s) used data d43 as input?

3 Find Output Query : What are the outputs of step p11?

4 Provenance-I Query : Which outputs produced by a Read S step does d83 depend on?

5 Provenance-II Query : Which outputs produced by an Output Q step were derived from d80?

6 Traverse-up Query : Which nodes are 5 steps upstream from d17?

7 Traverse-down Query: Which nodes are 7 steps downstream p73?

8 Match Template Query: Were input nodes d80 and d88 handled identically?

9 Find Exception Query: Which steps threw exceptions and what exceptions were thrown?

10 Count Repeated Process Query: How many times was Adjust S step executed?

Some of these queries (e.g. 1, 2, 3, 9, 10) can be expressed with a single SPARQL query, while others require processing of the response of one or more SPARQL queries in Java to return the desired data from the DDG.

I tested several example queries using a simple process of one Little-JIL parent node with two child nodes. Some of the query questions that the software successfully answered were:

1. "What PINs used the DIN DataInstanceNode/2?"

   or, when written in SPARQL:

SELECT ?s WHERE { <http://laser.juliette.ddgbuilder/DataInstanceNode/2> <http://laser.ddg/usedByPIN> ?s };

For this query the variable s stands for the PIN (Procedure Instance Node) to be returned from the database. We want to select all PINs s which are saved as the "http://laser.ddg/usedByPIN" – type property of the resource "http://laser.juliette.ddgbuilder/DataInstanceNode/2".

2. How many times was a particular step executed?",

   or, when written in SPARQL:

SELECT (count(?x) AS ?count) WHERE {?x <http://laser.ddg/hasName ?step_name } ;

The count function in SPARQL simply returns the number of matches in the database to the pattern in the query.

3. “What PIN preceded the PIN ProcedureInstanceNode/2?”

   or, when written in SPARQL:

SELECT ?s WHERE {<http://laser.juliette.ddgbuilder/ProcedureInstanceNode/2> <http://laser.ddg/predecessors> ?s }.

Queries such as 4. “Which outputs produced by a Read S step does d83 depend on?” cannot be performed in SPARQL alone. This particular query requires the following actions:

- finding all executions or instances of the Read S step
- finding the outputs of each instance
- finding the producer of the unique node d83
- iterative querying to find the predecessors of the predecessors and so on of d83
    - checking for each predecessor whether it is using as input one of the outputs of a Read S instance.

These actions are divided according to the restrictions of the SPARQL query language. The first three actions can be performed with SPARQL queries, while the fourth requires additional processing of the result of each query before making the next. The goal of the program is to automate queries that would be interesting

to the user, hiding the details of the processing, and even of the use of the SPARQL query language.

III.5. GUI

We have built a simple Graphical User Interface that allows the user to select a DDG from the file system and to display the entire DDG or snippets of the DDG that are interesting to her.

Figure 15 shows the dialog windows from choosing whether to display an entire DDG or a snippet of the DDG that corresponds to a particular query (top window of Figure 15), where the “Display a DDG...” button will show an entire DDG and “Display the first execution of a Little-JIL step” will show a snippet that represents that first execution of the Little-JIL step to be selected later. “Display the first execution of a Little-JIL step” is simply a sample query. We are looking to hardwire all queries that we know are potentially interesting in that first window. The middle window shows directories in the file system that hold the DDG's databases. After choosing a database and selecting “Display” the user sees a popup of the visual representation of the response to the query being made.

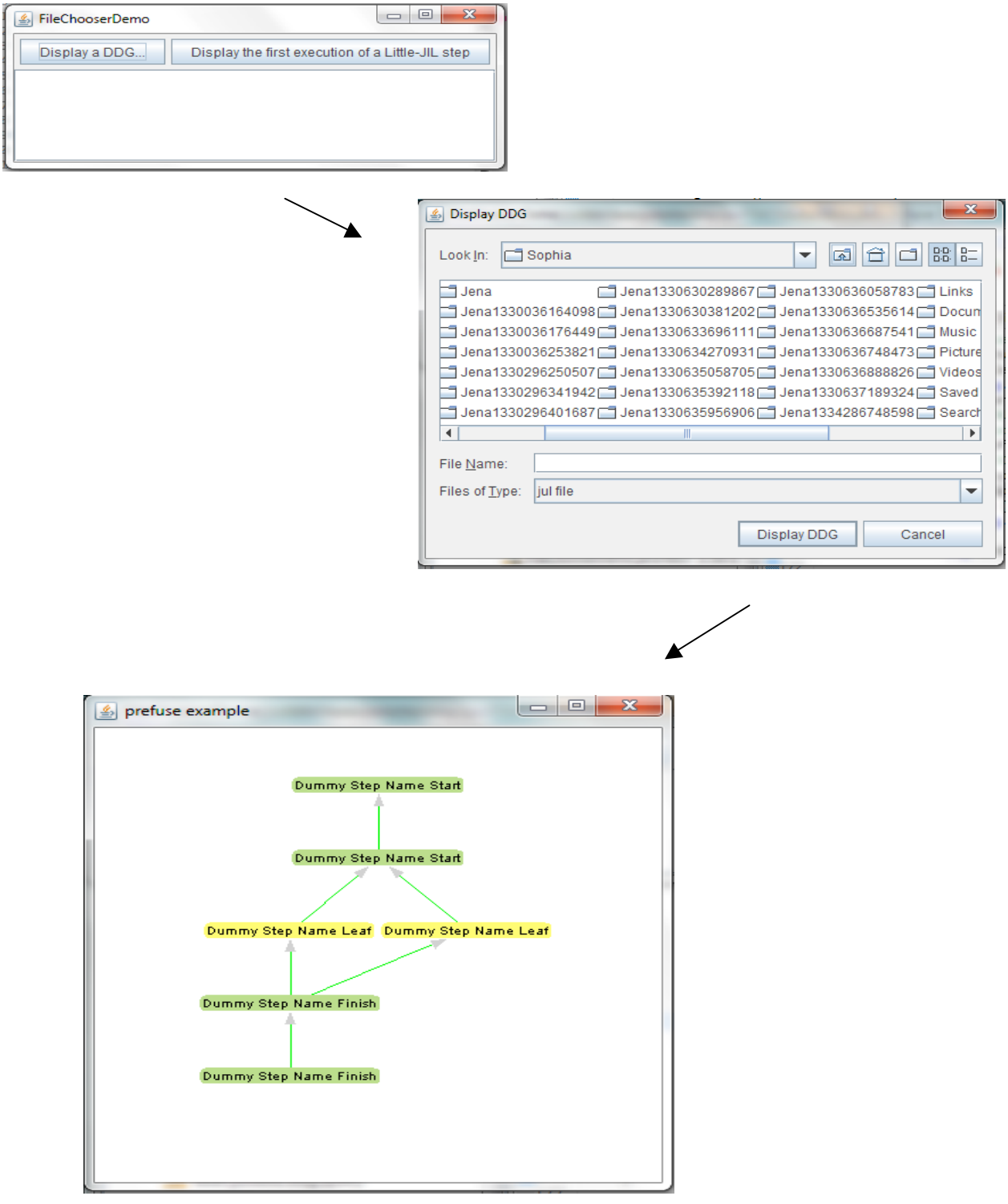

Figure 15. GUI for displaying an entire DDG or a snippet of a DDG

IV. EVALUATION AND DISCUSSION

An important open question on provenance is exactly how much information needs to be stored about a process. There are two main reasons why the question remains open: different processes may have radically different types of provenance information that is important, making it difficult to customize a solution that satisfies all cases. Secondly, even within one single process, the provenance information on the process that is important may change over time in ways that are unexpected to the user. Hence, in order to collect provenance intelligently, we need not only extensive user tests but also statistics on how the important provenance may change.

Different uses of the provenance software may require storing the provenance for different periods of time, from days to months and years, which is an important consideration for future work on the project. Additionally, as mentioned above, the user might not be aware of how far in the future the provenance may become important.

Due to the intensive cooperation of scientists on multiple databases, the software should be able to draw provenance from multiple different databases. It should also be able to interoperate with other provenance collecting tools, and to allow the scientists to use whatever data processing software they choose.

The issue of security will also need to be tackled, as different pieces of the provenance may need to have different visibility depending on what party is looking at the database. For example, if the provenance software is used for medical processes, the provenance must be very secure so as to not be tampered with, and it must also remain private to the physicians and patient and potentially family and friends.

A pressing issue is the size taken by provenance graphs. These graphs are infamous for taking up much more space than the data itself. Here are some numbers to give a more precise idea of the size of the DDG: for the relatively simple process of 19 Little-JIL nodes shown in Fig. 3, which describes almost exactly the process used in the hydrological study at the Harvard Forest, the DDG for one data value has 34 nodes and is of size around 1GB. The empty Jena database is of size 197 MB. The estimate of the provenance database size as the dataset size increases may need to be evaluated experimentally because the files required to set up the RDF database may not increase in size at the rate at which the number of DDG nodes increases.

For persisting DDGs into a database, in my second summer at the Forest I tried using the JenaBean technology. JenaBean is built on top of the Jena technology, which in turn is a toolkit written in Java used for persisting Java objects into a database. JenaBean is made to facilitate the use of Jena and requires very little code for persisting Java objects. However, it requires that the objects are Java beans (meaning that each object must have a nullary constructor and

getter and setter methods for each of the object's properties that will be made persistent). Unfortunately, JenaBean does not support properties of type Map, which is how input and output parameters are represented by the Java DDG-builder API. Thus, the fact that the JenaBean technology is under development itself discouraged me from using it to store DDGs.

We currently store the provenance databases in the file system, with the DDG of each execution of a single process being stored into its own directory. It is debatable whether this configuration is optimal for the types of queries the user may want to perform. It does not impede the comparison of executions (i.e. answering the provenance question "Were results A and B processed identically?") in any way but it may create difficulties in establishing contextual relationships among Procedure and Data instances in different executions (i.e. specifying a particular piece of information for a collection of Procedure or Data Instance Nodes in a set of DDGs would require additional processing in Java and multiple SPARQL queries).

We are currently not storing a reference of each Step Instance Node in the DDG to its corresponding Little-JIL step, and we are not storing a reference from the Data Instance Nodes to the actual data being used in the process. Both of these are drawbacks of the program which need to be mended in the future. A potential issue with storing the data inside the DDG would be that the data itself may or may not be able to be expressed with a number, making it difficult to create a universal handle for that data. Secondly, the DDG might not be the most

convenient medium for bringing the data to the user. From asking potential users we have found that they may be more interested in having the data accessible separately from the DDG.

Several other components that the user may want to be able to reach through the DDG are the process version, agents, and web services. The answer to what provenance information needs to be stored seems largely arbitrary, however there are certain trends that the program will need to consider in the future of the project. Multiple user tests are required to determine those trends.

V. CONCLUSION

Technology will be increasingly pervasive in how scientists do their work. With advancements of technology the sizes of the datasets will increase, and the processing of the data will become more complex. We also expect that the amount of collaboration among scientists will increase, while the importance of consistent quality control will remain high.

The availability of provenance on the results would help significantly in determining their authenticity and usability, and so we believe that our program will be in demand.

We have demonstrated that it is possible to collect "on the fly" the complete provenance based on the Little-JIL Process Definition Graph, and to persist the collected provenance in a conveniently accessible database. We have also been able to perform some queries that have been deemed interesting to the user and we have developed a simple GUI for making those queries without having to learn the query language SPARQL.

Future work on the project may require storing a handle to the PDG that was used for the execution described by a DDG. Interoperability with the software scientists are likely to use for processing and analysis, such as Microsoft Excel and R will also very likely be necessary. Issues may arise with the security of provenance, since many processes have pieces with different visibility and

access for the different participants in them. As mentioned in chapter IV, the most pressing next steps are scaling down the size of the DDGs and performing user tests.

References

[1] Boose, E. R., Ellison, A. M. , Osterweil, L. J. , Podorozhny, R. , Clarke, L., Wise, A. , Hadley, J. L. , Foster, D. R. 2007. Ensuring Reliable Datasets for Environmental Models and Forecasts. Ecological Informatics 2: 237-247.

[2] Leon J. Osterweil , Lori A. Clarke , Aaron M. Ellison , Rodion Podorozhny , Alexander Wise , Emery Boose , Julian Hadley, Experience in using a process language to define scientific workflow and generate dataset provenance, Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, November 09-14, 2008, Atlanta, Georgia [doi>10.1145/1453101.1453147]

[3] Zhao, J., Goble, C., Stevens, R. and Turi, D. (2008), Mining Taverna's semantic web of provenance. Concurrency Computat.: Pract. Exper., 20: 463–472. doi: 10.1002/cpe.1231

[4] Altintas, O. Barney, E. Jaeger-Frank. Provenance collection support in the *Kepler* scientific workflow system. Provenance and Annotation of Data, 4145 (2006), pp. 118–132

[5] Steven P. Callahan , Juliana Freire , Emanuele Santos , Carlos E. Scheidegger , Cláudio T. Silva , Huy T. Vo, VisTrails: visualization meets data management, Proceedings of the 2006 ACM SIGMOD international conference on Management of data, June 27-29, 2006, Chicago, IL, USA [doi>10.1145/1142473.1142574]

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

[7] Anand, M.K., Bowers, S., Lud□scher, B.: Techniques for efficiently querying scientific workflow provenance graphs. In: EDBT 2010: Proceedings of the 13th International Conference on Extending Database Technology, pp. 287-298. ACM, New York (2010)

[8] O. Biton, S. Cohen-Boulakia, and S. Davidson. Zoom*UserViews: Querying relevant provenance in workflow systems (demo). In VLDB, 2007.

[9] Luc Moreau, Juliana Freire, Joe Futrelle, Robert E. Mcgrath. Jim Myers, Patrick Paulson, The Open Provenance Model: An Overview, Provenance and Annotation of Data and Processes: Second International Provenance and Annotation Workshop, IPAW 2008, Salt Lake City, UT, USA, June 17-18, 2008. Revised Selected Papers, Springer-Verlag, Berlin, Heidelberg, 2008 [doi>10.1007/978-3-540-89965-5_31]

[10] D. Cai, Y. Gwon, L. McGinnis, “Storing, Querying, and Compressing the Little-JIL Workow Provenance”
[11] http://www.newscientist.com/data/images/ns/cms/mg18925431.400/mg18925431.400-2_752.jpg
[12] http://www.sciencemag.org/content/316/5822/188.summary
[13] http://www.businessworld.in/businessworld/businessworld/content/Climate-Change-Myth.html
[14] http://laser.cs.umass.edu/techreports/00-66.pdf
[15] http://laser.cs.umass.edu/techreports/06-51.pdf
[16] http://www.w3.org/TR/2004/REC-rdf-primer-20040210/