# Applying Software Engineering Technology to Support the Clear and Precise Specification of Scientific Processes

**Leon J. Osterweil, Lori A. Clarke, Aaron M. Ellison, Emery Boose, Rodion Podorozhny, Alexander Wise**

***Abstract*— With the availability of powerful computational and communication systems, scientists now readily access large, complicated derived datasets and build on those results to produce, through further processing, yet other derived datasets of interest to themselves and others. The scientific processes used to create such datasets must be clearly documented so that scientists can evaluate their soundness, reproduce the results, and build upon them in responsible and appropriate ways. Here, we present the concept of an *analytic web*, which defines the scientific processes employed and details the exact application of those processes in creating derived datasets. The work described here is similar to work often referred to as "scientific workflow", but emphasizes the need for semantically richer, more rigorously defined process definition languages, such as those that were first developed to define software engineering processes. We illustrate the information that comprises an analytic web for a scientific process that measures and analyzes the flux of water through a forested watershed. This is a complex and demanding scientific process that illustrates the benefits of using a semantically rich executable language for defining the process, supporting automatic creation of process provenance metadata, assuring data reproducibility, and supporting analysis of the data's scientific soundness.**

***Index Terms*—Process Programming, Scientific Workflow, Software Engineering.**

## I. Introduction

### A. The Problem

Modern computation and communication systems have dramatically changed the way in which science is done. These systems enable scientists to work with datasets and to create models of their research subjects that are far larger and more detailed than were possible in the past. Faster computing speeds enable far more ambitious analyses of these models, leading to the production of far greater quantities of derived scientific datasets. Ever faster global networks make these datasets accessible to scientists around the world. While these new computational and communications capabilities have opened up the possibility of exciting new research, they have also led to new challenges and problems. When new datasets are generated by extensive processes, they are often promulgated without adequate documentation that describes their creation. If scientists are to make appropriate use of the datasets produced by others and avoid misuse by inappropriate application of subsequent processing, then it is imperative to know how such datasets were produced. Indeed, before scientific results can be accepted, they should be reproduced by other scientists; reproducibility is a fundamental part of science.

An obvious way to address these challenges is to associate with each dataset as an *annotation* a precise description of the dataset. Such annotations, essentially data items that describe data, are called *metadata*. There have already been many calls for the use of metadata, which typically document such details as the date of generation of a dataset, the name of the investigator, and perhaps some specifications of the hardware and software systems used, as well as details of the individual data items (variable name, numerical format, unit of measurement, etc.). We argue that it is necessary to go further. We suggest that a particular type of metadata annotation, *process provenance metadata*, be attached to all datasets, and when necessary to individual data items. The benefits of such process provenance metadata include facilitation of the reproduction of the data by others, expedited identification of data items and datasets of interest, and better understanding of which forms of subsequent processing should, and should not, be applied to data items and datasets.

It is our view that concepts drawn from the domain of software engineering can provide the basis for the generation and association of such process provenance metadata with the derived data items and datasets.

### B. Analytic Webs

Scientific datasets can be viewed as products emerging from a distributed enterprise: input datasets may be stored and retrieved remotely, analytic services may be obtained from external sources, and the datasets that are the products of a scientist's work are immediately accessible by others. The totality of data and capabilities produced and consumed by a working scientific team in pursuit of a particular scientific objective can be thought of as a scientific (usually online) data processing enterprise, and we refer to it by the term *analytic web* [7; 19; 34]. By analogy, the purpose of an analytic web is

Manuscript received March 31, 2008. This material 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.

Leon J. Osterweil is with the University of Massachusetts, Department of Computer Science, Amherst, Massachusetts 01003 USA (413-545- 2186; fax: 413-545-1249; e-mail: ljo@ cs.umass.edu).

Lori A. Clarke is with the University of Massachusetts, Department of Computer Science, Amherst, Massachusetts 01003 USA (413-545- 1328; fax: 413-545-1249; e-mail: clarke@ cs.umass.edu).

Aaron M. Ellison is with Harvard University, Harvard Forest, Petersham, Massachusetts 01366 USA (978-724-3302; fax: 978-724-3595; e-mail: aellison@fas.harvard.edu).

Emery Boose is with Harvard University, Harvard Forest, Petersham, Massachusetts 01366 USA (978-724-3302; fax:978-724-3595; e-mail: boose@fas.harvard.edu).

Rodion Podorozhny is with Texas State University, San Marcos, TX USA (512-245-8472; fax: 512) 245-8750; e-mail: rp31@txstate.edu).

Alexander Wise is with the University of Massachusetts, Department of Computer Science, Amherst, Massachusetts 01003 USA (413-545-4830; fax: 413-545-1249; e-mail: wise@ cs.umass.edu).

to expedite the participation of a scientific team in the marketplaces of scientific investigation. From this view there is then a need to provide strong support for two distinct activities that scientists engage in routinely, namely the *production* of datasets for such marketplaces and the *consumption* of datasets from such marketplaces, for use in subsequent scientific investigation.

**Production**: As raw data are collected by individual investigators or by sensors, they may be pre-processed with data loggers, field computers, etc.; for example, many environmental measurements (*e.g.,* temperature, solar radiation, carbon flux, water flow) are sampled at high frequencies but only hourly averages are stored. Individual researchers take these data and post-process them, entering them into data repositories such as spreadsheets, checking them for errors, and transforming them into datasets used for analysis by the investigators themselves or by others. These datasets are often stored on the investigator's personal computer, where they might be further analyzed (and in the process new datasets may be created) and condensed for publication. The producers of these datasets could use help in managing the execution of these increasingly intricate processes, in documenting exactly what processing was applied to which data items, and in reasoning about the soundness of the resulting data items and datasets. These data items and datasets, on which publications are often based, are archived on institutional servers or within national repositories, where it is expected that they may be accessed and used by others: *i.e.,* dataset consumers.

**Consumption**: Datasets collected by other investigators increasingly are available via the Internet and may be reanalyzed to verify existing models and results or used to generate new models, hypotheses, and scientific insights. We refer to this use of previously existing datasets as dataset consumption. It is not atypical for a consumer who synthesizes datasets to subsequently become a producer of new (synthetic) datasets that are consumed by others (who then may become producers of further datasets).

The conceptualization of dataset production and consumption suggests specific ways in which an analytic web should support a scientific team's activities. Support for production should consist of facilities for generating and storing new data items and datasets. In recognition that others may consume these datasets, the datasets and when necessary the individual data items should be annotated with precise process provenance metadata. An analytic web should provide facilities for accessing such metadata and evaluating its subsequent use. This may entail reproducing the dataset, evaluating its use in further scientific data processing, or using the process generation metadata as a guide to generation of other datasets.

To make this conceptual vision of an analytic web a useful reality, we propose that a specific analytic web be realized by a set of tools aimed at creating, analyzing, and managing two types of closely interrelated graph structures, namely *Dataset Derivation Graphs* (DDGs) and *Process Derivation Graphs* (PDGs). The purpose of a DDG is to organize datasets into a structure based upon the way in which the datasets are derived from each other. The purpose of a PDG is to define precisely the processes by which these derivations are performed. The PDG also serves as the vehicle for executing the process definition and generating subsequent data items and datasets. Moreover, execution of a PDG can result in the automatic creation of the DDG for each of these data items and datasets.

In this paper we provide a concrete example of the way in which an analytic web can support the activities of a specific scientific research team. This example illustrates that DDGs are likely to be large structures that need to be defined in detail and with great care. Our example illustrates the importance of automatically executing the PDG and, during that execution, automatically generating the corresponding DDGs. A major objective of this example is to demonstrate a set of semantic features that seem essential in a process definition language employed to define the PDGs used by actual working scientific teams. An existing process definition language, Little-JIL [44; 43; 10], incorporates many of these semantic features. Key features of Little-JIL are highly effective in defining the PDGs needed to support our example scientific process, but some important semantic features are missing.

## II. Motivating Example

Measuring and forecasting water flux and storage in the ecosystem (including ground water, soils, surface water, snow pack, vegetation, and atmospheric boundary layer) is of tremendous importance to society, of pressing interest to scientists, and a central focus of major scientific investigation efforts, such as NEON (http://www.neoninc.org/) and the Waters Network (http://www.watersnet.org/). Such forecasts require detailed hydrological measurements of natural and human-dominated ecosystems; these measurements come from vast networks of real-time sensors and are subjected to elaborate real-time adjustments and considerable, perhaps iterative, post-processing over ensuing months or years. Producers of such datasets could use help in systematically and correctly applying various processing and analysis tools. Consumers of these datasets will require support to help them understand the datasets, to reproduce them if desired, and to use them appropriately in further processing.

To address this scientific problem, a group of ecologists at the Harvard Forest Long-Term Ecological Research (LTER) site (http://harvardforest.fas.harvard.edu/) is designing a real-time system for estimating water budgets for three small forested watersheds at their site. Their system is being designed to calculate change in water storage using the water balance equation: $dS = P - ET - Q$, where the change in water storage ($dS$) is a linear function of precipitation ($P$), evapotranspiration ($ET$), and surface discharge or streamflow ($Q$). The complete system will include additional measurements of snow pack, soil moisture, ground water, *etc*. Equation inputs come initially from five real-time data streams from three sources:

- **Precipitation ($P$)** – 15-minute precipitation totals ($P1$, $P2$) measured at two rain gauges. Two gauges are used to guard against data missing due to sensor drift and failure.

• **Surface Discharge (*Q*)** – 15-minute average stream flow values measured at a stream gauge. Short gaps in *Q* caused by sensor failure, excessive ice build-up, etc. can be filled by modeling *Q* as a function of preceding measurements of *P* and *Q*.

• **Evapotranspiration (*ET*)** – 30-minute average *ET* values measured at an eddy-flux tower.

• **Photosynthetically active radiation (*PAR*)** – 30-minute average *PAR* values measured at the eddy-flux tower. *PAR* is measured continuously because over short time spans (up to several weeks) *PAR* is the environmental variable most often highly correlated with *ET*. Thus, *PAR* can be used to estimate *ET* when it cannot be accurately measured directly for any of a variety of reasons.

This system will incorporate three features that are typical of virtually any sensor network and raise challenging issues for dataset producers as well as dataset consumers:

1. Real-time quality control entails non-trivial processing, much of it determined on the fly, which may cause different data items in a dataset to have different process provenance. For example such systems may incorporate duplicate sensors (here *P1* and *P2*), real-time modeling (yielding estimates $\hat{Q}$ and $\hat{ET}$ of *Q* and *ET*, respectively), and rules for value selection. Thus, in this case the two precipitation measurements are to be compared and specific actions taken if the values differ by more than a specified amount. To check for instrument problems, *Q* and *ET* are to be compared to their modeled values ($\hat{Q}$ and $\hat{ET}$) using respectively (i) verified runoff models and recent precipitation and flow values and (ii) regressions of recent *ET* and *PAR* values. Modeled values also may be substituted (imputed) for measured values when sensors fail or wind conditions do not support reliable eddy-flux measurements.

2. Regularly scheduled post-processing of data (*e.g.,* after 30 days) is used so that individual imputed data values can be computed using models that take into account values coming from both preceding and subsequent measurements. This feature is necessary because real-time modeling is retrospective only (*i.e.,* based on past data). Thus, post-processing is important especially for times when the ecosystem is undergoing rapid change (*e.g.,* during spring leaf-out or when soils become saturated during heavy or prolonged precipitation).

3. Alternative past measurements also may become available and may then be included in additional datasets. This may be desirable, for example, to make use of recent measurements that did not arrive in time for real-time processing, corrections of earlier measurements as a result of later detection of sensor drift, or replacement or imputation of faulty or missing values with measurements from other sources. Note that the substitution of alternative measurements must be accompanied by post-processing of a sufficient number of preceding and subsequent values to ensure that any "ripple effects" – possible impacts on subsequent modeled values – are accounted for. This indicates another way in which it becomes possible for each data item of a dataset to be the product of a different process.

Fig. 1 is a representation of the water budget process using a data flow graph (DFG). In this DFG, rounded boxes represent types of tools or subprocesses while types of datasets are represented by boxes with a clipped corner. Edges connecting these boxes represent the flow of data or datasets into and out of the tools and subprocesses. In this figure, the Real-time Selection Criteria box represents the on-the-fly processing of the Real-time Data that is used to create new or updated Models and to update the model variables $\hat{Q}$ and $\hat{ET}$. Similarly, the Retrospective Selection Criteria and the Alternative Selection Criteria boxes represent the reprocessing of the Real-time and Retrospective Data, respectively, and can also result in new Models and updated model values. In subsequent sections of this paper we indicate shortcomings of this DFG representation and suggest another graph notation that seems to offer better facilities for representing complex processes such as this one.

## III. Proposed Model of an Analytic Web

The next section of this paper demonstrates how the data management issues indicated in the previous section can be supported by analytic webs. In this section we describe formalisms used to define the graphs comprising such analytic webs.

### *A. Dataset Derivation Graph*

A DDG documents the specific data items or dataset instances created when a producer applies processes (e.g. perhaps defined using a PDG), using specific tools and subprocesses on specific input data items or dataset instances. A DDG thus contains detailed metadata about the process sequences used to build all of its datasets. This metadata is an example of the *process provenance metadata* needed to inform consumers about how a dataset was generated and to support reproduction of the dataset by other scientists. A DDG, as depicted in Fig. 2, uses rectangles to represent specific data items and dataset instances and ovals to represent specific tools or subprocess instances. There is an edge from each data item and dataset instance node to the process instance node from which it was derived (unless the data item or dataset instance represents raw data that was not previously processed). Each oval process instance node is connected by one or more edges to the data item(s) and dataset instance(s) that it used as input(s) to derive the indicated output data item or dataset instance. Each time a process is executed, a new set of data items and dataset instances is created, and these data items and dataset instances, as well as the process instances that created them, must be added as nodes of the evolving DDG. Each DDG node instance can be stored independently with a unique URL for identification.

Data items and dataset instances, such as those represented by DDG nodes, are the usual focus of scientific attention and thus are the objects intended to be documented with metadata such as specified by Ecological Metadata Language (EML; http://knb.ecoinformatics.org/software/eml/). For example,

Transpiration Data For 2/26/06 is a specific instance of type transpiration data and thus is depicted in Fig. 2 by a box. Fig. 2 also shows that Aaron's outlier rejection parameters is another dataset instance. These two instances are taken as inputs by a specific processing tool, namely Reject Outlying Data, to produce the output dataset instance, Cleaned 2/26/06 Data. This dataset instance is, in turn, taken as input to the processing tool Least Squares Interpolation using System R 9/22/06, producing as output the dataset instance Cleaned, Interpolated 2/26/06 Data. Note that the specification of the specific interpolation tool, complete with version number and date of application, is important as it specifies precisely which tool or system was applied to the dataset instance. Different versions of a tool may produce different results. Consumers who want to fully understand the provenance of a dataset, and producers who want to reproduce it exactly, require the documentation provided by a DDG: the specific datasets and tools that were *actually used.* The quantity and intricacy of this documentation is considerable, but it can be produced automatically with the aid of a suitable suite of tools, as described below.

To make this clearer, Fig. 3 illustrates a DDG that could have been generated by executing a sequence of tools and subprocesses represented as a path through the boxes in the diagram shown in Fig. 1. Note that this DDG shows the result of two iterations, represented by two traversals through a loop in the diagram in Fig. 1, resulting in the creation of three instances of Revised Data, denoted here by Revised Data 1, Revised Data 2, and Revised Data 3. Revised Data 1 is simply the set of data items that resulted from filtering the initial real-time data stream by applying some specific filtering criterion. This generally results in a dataset where some data item values are missing, most often due to intentional deletion. Revised Data 2 results from applying initial $\hat{Q}$ and $\hat{ET}$ models to this dataset, thereby filling in missing data item values and replacing others. Revised Data 3 results from creating and applying a second pair of $\hat{Q}$ and $\hat{ET}$ models, (e.g., from regularly scheduled post-processing) and applying them to Revised Data 2. This DDG provides documentation of the exact processing steps that were taken to produce these datasets. It specifies what datasets must be generated to reproduce these processing steps, and thus the dataset that is their final result.

### B. Process Derivation Graphs

A Process Derivation Graph (PDG) is a precise representation of the sequences of steps used to process the data items and datasets in a scientific process. Many different formalisms could be used to define PDGs. But our example suggests that such semantic issues as concurrency, abstraction, exception handling, and agent specification are important to the clear specification of actual scientific processes. The absence from a formalism of capabilities for specifying these semantics restricts the utility of that formalism in supporting specification of PDGs. Thus, while a DFG could, in relatively simple cases, be used to specify a PDG, the DFGs relative paucity of semantic power limits its utility in supporting the clear and precise definition of more realistic scientific processes. Thus, for example, DFGs make it hard to distinguish between those paths that are intended to be part of a scientific process and those that are not, and thus do not reliably specify which paths through the graph can be used safely to generate additional DDG nodes. For example, the DFG in Fig. 1 does not preclude the retroactive processing loop from preceding the real-time processing loop for the same dataset.

Thus, in this paper we use Little-JIL, a visual process definition language originally developed for defining software engineering processes, as to define PDGs. The next section uses Little-JIL to define processes that are part of the example presented in Section II in order to demonstrate how semantic features such as concurrency, exception handling, and abstraction facilitate the specification of PDGs for realistic scientific processes.

### C. Semantics of Little-JIL

A process is defined in Little-JIL using hierarchically decomposable steps [43; 44], where a step represents a task to be done by an assigned agent. Each step has a name and a set of badges that represent control flow among its sub-steps, its interface (a specification of its input and output data), the exceptions it handles, etc. A step with no substeps is called a *leaf step* and represents an activity to be performed by an agent, without any guidance from the process.

**Resources and Agents:** As part of its interface, each Little-JIL step contains a specification of the type of agent that is required to assume responsibility for the step's execution. The agent specification is a specification of a capability. It is assumed that this specification will be considered by a separate Resource Manager that is expected to identify a specific resource instance to be bound as the agent in response to the step's need for a specified capability. Little-JIL agents may be either humans or automated devices, and there are =cases where either might be appropriate. A step may also specify the need for resources other than the agent.

**Substep Decomposition**: Little-JIL steps may be decomposed into substeps of two different kinds, ordinary substeps and exception handlers. The *ordinary substeps* define the details of how the step is to be executed. They are connected to the parent step by edges that may be annotated by specifications of the artifacts that flow between parent and substep, and also by cardinality specifications. Cardinality specifications define the number of times the substep is to be instantiated and may be a fixed number, a Kleene * (for zero or more times), a Kleene + (for one or more times), or a Boolean expression (indicating whether the substep is to be instantiated or not). *Exception handlers* define the way in which exceptions thrown by the step's descendants are to be handled. The edge connecting an exception handler to its parent is annotated with the type of the exception being handled, and with an indication of how execution is to continue after the exception has been handled.

**Step sequencing:** Every non-leaf step has a *sequencing badge* (an icon embedded in the left portion of the step bar; *e.g.,* the right arrow in Fig. 4), which defines the order in which its substeps execute. For example, a sequential step (right arrow) indicates that its substeps are to be executed sequentially from left to right. A parallel step (equal sign)

indicates that its substeps can be executed in any (possibly arbitrarily interleaved) order. A choice step (circle slashed with a horizontal line) indicates that the agent executing the step is to make a choice among any of the step's substeps. A try step (right arrow with an X on its tail) mandates a sequence in which substeps are to be tried in any order until one completes successfully.

**Artifacts and artifact flows:** An *artifact* is an entity (*e.g.,* a datum or dataset) that is used or produced by a step. Parameter declarations are specified in the interface to a step (circle atop the step bar) as lists of the artifacts used by the step (IN parameters) and the artifacts produced by the step (OUT parameters). Artifact flow through steps can be defined to take place in either of two different ways, 1) hierarchically, as the flow of artifacts between parent and child steps, and 2) by means of channels. The flow of artifacts between parent and child steps is (as noted above) indicated by attaching to the edges between parent and child identification of the artifacts as well as arrows indicating the direction of flow of each artifact.

**Channels:** *Channels* are named entities that directly (without the need for hierarchical flow) deliver artifacts produced by specifically identified source step(s) as arguments to specific destination step(s). A Little-JIL channel is defined at present as a FIFO queue. Steps that use the channel either *write* to the end of the channel, *take* data from the front of the channel, or *read*, without removing, data from the front of the channel. The channel construct can be used as a vehicle to coordinate and synchronize steps executing in parallel. On the other hand, as the example below shows, it is restrictive to use only a FIFO queue to model the channel's handling of data.

**Requisites:** *Requisites* are optional and enable the checking of a specified condition either as a precondition for step execution or as a postcondition check to assure that the step execution has been completed acceptably. A downward arrowhead to the left of the step bar represents a prerequisite, and an upward arrowhead to the right of the step bar represents a post-requisite. If a requisite fails, an exception is triggered.

**Exception Handling:** A step in Little-JIL can define *exceptional conditions* when some aspects of the step's execution fail (*e.g.,* one of the step's requisites is violated). This violation triggers the execution of a matching exception handler associated with the parent of the step that throws the exception. An exception handler is represented as a step attached by an edge to an X on the right of the step bar (as shown in Fig. 4. = Little-JIL currently supports specifying four different ways that execution can proceed after execution of an exception handler.

**Scoping and Recursion:** The parent step and all of its descendants represent a *scope*, specifying what data are considered local to that scope. Little-JIL also supports *recursive execution* of steps, which specifies the iterative application of a process step to specified inputs.

## IV. Using Little-JIL to define and execute a PDG and to create a DDG

The purpose of the Water Budget process is to provide estimates of the rate of change of water storage *dS* over various time intervals on the basis of time-ordered sequences of readings from sensors that measure various parameters. While this process may at first glance appear to be relatively straightforward, in Section II we indicated that there are a number of aspects of the fully elaborated process that are challenging to define precisely. We now use a Little-JIL definition of the Water Budget process that illustrate some of these difficulties and some of the semantic features that are useful in a defining them clearly and precisely through the use of an analytic web. The process step definitions are illustrated here by step diagrams that have been created using the Little-JIL editor. In the interest of reducing clutter, the editor does not depict all the details and annotations of all step definitions unless specifically requested by the user. The user may obtain complete detailed information about any aspect of a process definition by moving the cursor over the appropriate icon. To suggest the nature of this additional information we append to each figure a legend that contains such additional explanatory information. Information about a step begins with the name of the step, followed by the step's input and output parameter types, any channels used by the step, and the types of data items carried by these channels. For some steps there is also a specification of exceptions that may be thrown by the step and/or exceptions that may be handled by the step. For edges, the information begins with a specification of the pair of steps that define the edge, followed by information about parameter flow between parent and child steps. There may also be comments that informally describe the general purpose or approach of a step or edge. Much of the information being provided by these legends could be specified through extension of existing annotation schemes, such as EML..

### *A. The Sensor Data Management step: use of decomposition and concurrency*

We begin by describing the step Sensor Data Management (Fig. 5). Sensor Data Management, the root of the Water Budget process, is a parallel step consisting of the execution of three substeps; Get Measurements collects data from the sensors; Model Stream Data Gen creates new predictive models for this data; and Process Data processes the data for publication. The collected data and created models are communicated to Process Data via channels that are declared in Sensor Data Management and are accessible to all of its substeps. The `sensorStream` channel is declared to be a FIFO channel, so that data are removed in the order in which they are put into the channel. The `modelStream` channel is a singleton channel, meaning that at most one item can be in the channel at a time.

The first two substeps are described in Sections B and **E** below. The third substep, Model Stream Data Gen, represents the generation of new models. The annotation indicates that this step is done by human experts, each of whom executes a different instantiation of this step (as indicated by the Kleene * on the edge leading into this step), thereby allowing for the parallel and asynchronous generation of new models. The use of a channel allows for the possibility that new models can be dynamically placed into the channel at any time during the execution of this process. Doing so makes the model available for application to subsequent data items. Note also that there is a different step for the generation of each type of model. Gen New P Model is the step that is responsible for the generation of new P models, while Gen

New Q Model is responsible for the generation of new Q models. The example shows only two such substeps for simplicity, but the complete process definition would require a substep for each model type.

Because each of these steps can be executed in parallel, each can proceed at its own pace. Sensor data streams in as it is generated and new models are derived as needed by human scientists, presumably at a far slower pace. The processing of the data streams (including application of the models as needed) is driven by interval timers and human curiosity. The PDG represents this parallelism and indicates which models are applied to which data items as part of which datasets.

### *B. The Get Measurements step: support for multiple data streams*

The Get Measurements step (Fig. 6) reads and processes the values from the sensors and sends the results to the real time stream. This is done by having different subprocesses, Get Met Station Data, Get Flux Tower Data, and Get Stream Gauge Data, take responsibility for examining the three types of data sources. Each of these substeps can execute independently and in parallel with the other two, and each may throw a different type of exception if difficulties arise with their sensors.

The Get Met Station Data step has two substeps, Get P1 and Get P2, each of which is responsible for dealing with measurements coming from one of the two precipitation measurement gauges. Similarly, the Get Flux Tower Data step has four substeps, each of which is responsible for handling data items coming from each of the various sensors on the flux tower and Get Stream Gauge Data has one substep for handling surface discharge data.

The substeps that access the individual data items also are responsible for annotating each data item with some provenance information. At present this consists of attaching very rudimentary metadata, specifically a date/time stamp and a quality flag. Below we describe how a DDG improves significantly upon the current quality flag.

### *C. The Handle MS Sensor Timeout step: use of exception management*

Each substep of the Get Measurements step is responsible for acquiring data from its sensors, and each also is responsible for defining the subprocess to be employed if exceptional conditions arise. The Get Met Station Data step deals with situations where some, or all, of the expected data items do not arrive by throwing an exception. For example, the substep Get P1 attempts to obtain a precipitation reading from meteorological sensor 1. If this access succeeds, then the value is passed as P1, annotated with the observation date and time and with the *measured* quality attribute. The Get P1 step is also responsible, however, for determining when new P1 data is not available and then, subsequently, for throwing an exception that is to be handled by Handle MS Sensor Timeout step, passing the identifier of this sensor (namely sensor #1) as a parameter. The other two substeps of the Get Measurements step, Get Flux Tower Data and Get Stream Gauge Data, carry out their responsibilities similarly to Get Met Station Data. The final output of the Get Measurements step is exactly one datum of each of the types P1, P2, ET, Q, PAR, VPD, and UStar, each accompanied by provenance annotations.

In this context, the Handle MS Sensor Timeout step (Fig. 7) is specified as the handler for the Sensor Timeout exception that can be thrown in the Get Met Station Data step. The Handle MS Sensor Timeout begins its processing by choosing, based on the value of the input parameter, Reread Precipitation 1 or Reread Precipitation 2. We do not show the details of either Reread Precipitation step here, but include these steps only to illustrate that it is possible for the developers of this process to decide that unusual (and presumably expensive) measures might be taken under these circumstances to attempt to extract the desired data directly from the sensor, rather than from the data stream. If this direct request to the sensor succeeds then output of this step is annotated with the date and time and quality flag. If the selected Reread Precipitation (1 or 2) step fails (e.g. because the sensor is inoperative), a Sensor Down exception is thrown (inside of the Handle MS Sensor Timeout exception handler itself). The Sensor Down handler executes the Get Airport step, which attempts to obtain the desired reading by getting it from a local airport. Here again, nested exception handling capabilities are used to define the handling of still further failure. Thus, if the Get Airport step succeeds, then the resulting datum is annotated with the date and time, source, and the measured quality attribute. If the Get Airport step also fails (e.g. because the airport is also unable to provide the precipitation data), then it must also throw an exception, which would then be caught by the Put Null Reading step of the Get Airport exception handler, to produce as output a null value for P1 (or P2) and a quality attribute "missing". A "missing" value will not take place unless all three exception-handling alternatives have been explored in sequence.

### *D. Using the process definition to create the DDG*

The above process results in the creation of two data items, P1 and P2, each of which might have been arrived at in a number of different ways. Specifically, each of the two measurements might be arrived at by any of the following sequences of process steps:

1. Pi arrives in a timely fashion and is recorded.
2. Pi does not arrive in time, a timeout exception is thrown, Reread Precipitation is executed, and Pi is obtained.
3. Pi does not arrive in time, a timeout exception is thrown, Reread Precipitation is executed, Pi still does not arrive, a Sensor Down exception is thrown, the Airport Read step is executed, and Pi arrives.
4. Pi does not arrive in time, a timeout exception is thrown, Reread Precipitation is executed, Pi still does not arrive, a Sensor Down exception is thrown, the Airport Read step is executed, Pi does not arrive, an Airport Data Read Failure exception is thrown, the Put Null Value step is executed, providing a null value for Pi.

The differences among these four possibilities are important to subsequent process steps. Accordingly, the current process specifies that a quality flag, mentioned previously, be attached to Pi. This quality flag is currently a simple annotation, at this point having the following values:

*missing* - no measured value is available (e.g. because the sensor is down);
*estimated* - a measured value is available from another location (e.g., the airport)
*measured* – the measured value is available, but it is unknown whether or not it is within prespecified bounds;

Unfortunately, the current process specifies that Pi is annotated as *measured* in the first two cases we have just enumerated (making it impossible to distinguish between them), as *estimated* in the third case (but without documenting the measurement location), and as *missing* in the fourth case, again leaving out all of the details describing what alternatives had been tried. We note, however, that each of the four different sequences of process steps can be thought of as a different trace through the process, illustrating the importance of annotating each value with process provenance information, as provided in the DDG.

Examples of the DDGs that represent cases #3 and #4 are shown in Fig. 8. Note that the boxes in this figure represent actual data instances, namely the actual data values that are bound at execution time to the type specifications in the process definition. Thus, for example, one of the boxes at the top of Fig. 8 is annotated by sensor1 null @"try1 time", indicating that this box represents the actual (null) value that was delivered at the specific time, "try 1 time". The adjacent box likewise represents the specific (null) value that was delivered at the specific time "try 2 time".

The ovals in Fig. 8 represent the process step instances that were the actual producers and consumers of the actual data items. Thus, there is an oval labeled Get Airport that indicates that an instance of the Get Airport step was used to generate the data item in the box shown below this oval. This step instance represents the instance of Get Airport that was invoked as the process's response to the two null readings. Two arrows from this oval connect it to two boxes, representing the fact that the values represented by these boxes were used as inputs to the step represented by that oval. In this case, the use that is made of these data items is simply to note that they are both Null, causing the Get Airport step to be executed to produce this output. Other ovals may make more substantive use of their inputs in generating their outputs. Thus, for example, in the left-hand DDG of Fig. 8, the result of the execution of the instance of the Get Airport step is an actual value, annotated with date and time information, which is taken as the final value of Pi. In this case, no actual step is used to generate that value, and instead the DDG indicates that the value is produced as a consequence of the parameter binding operation that occurs as an integral part of the execution of every step. The fact that this oval does not represent an actual step is indicated by the use of italics in its annotation. In the other case, a null reading is obtained, and a null value is then the final value of P1. Thus such DDGs provide more useful information about the provenance of the resulting value of P1 than a mere annotation, *measured, estimated,* or *missing*.

Although the structures shown in Fig. 8 seem to be large annotations, they can be represented efficiently as sequences of pointers to process definition steps, parameter values, and the actual data item instances. Further, these pointers can be generated as side effects of the actual execution of the process represented by the PDG shown in the preceding figures. This suggests that DDGs might be generated at a modest incremental cost during execution of the process and represented relatively efficiently when intermediate datasets are being stored.

### E. *The Process Data step: alternative approaches to processing the data*

The second subprocess, Process Data, in the root Sensor Data Management process, shown in Fig. 5, is described and elaborated in Fig. 9. This step takes input from sensors and investigators, as described above, and produces a time-ordered sequence denoted <dS>, where each dS estimates change in water storage over the previous time interval. Note that <dS> (also dS) is a type and that <dS> instances are generated by each of three different subprocesses of Process Data, namely RT Periodic Processing, PP Periodic Processing, and Alt Model Processing. Process Data indicates that these three subprocesses can be executed in parallel, but whether the actual executions of these steps overlap in time will depend on decisions of the agents bound to perform these substeps and, most importantly, upon the availability of the input data.

RT Periodic Processing produces a <dS> dataset every 24 hours, based on the data that was collected over the proceeding 24 hour period. PP Periodic Processing also produces a <dS> dataset every 24 hours, but it is for a 24-hour period 30 days before. This subprocess has the advantage of doing interpolation that uses data obtained both prior and subsequently to the data in need of interpolation. Alt Model Processing allows investigators to experiment with alternative models and time periods. Here we describe each of these subprocesses in turn. Note that in the DDG, it is important to know which subprocess was responsible for generating each of the different <dS> instances and their different individual data items. As illustrated above, the process definitions provide a basis for generating DDGs that document these differences.

### F. *The RT Periodic Processing step: handling of real-time streaming data*

The RT Periodic Processing step is connected to its parent by an edge annotated with a Kleene +, prescribing that the Little-JIL interpreter will keep instantiating a new instance of this child step indefinitely (the decision to terminate iteration is made by the agent bound to the iterated child step). The agent for this step is an interval timer that initiates step execution every 24 hours. The Get RT Datum step specifies the actual processing of the real-time data and the rate of execution is determined by the availability of data from the sensors and the resources required for the processing of that data.

### G. *The Get RT Datum step: dealing with individual data items*

The Get RT Datum step, shown in Fig. 10, defines the heart of the Water Budget process, specifying how the real-time data items are subjected to cleaning. This step retrieves the readings collected from the sensors as they become available, filters the readings to ensure each reading is within predefined bounds, tags each reading to indicate how it was obtained, and appends readings to corresponding time-ordered sequences. This step is defined as the sequential execution of its substeps and results

in an instance of dS, which has data items of type ET, Q, P, PAR, VPD and UStar as its components. The set dS is then appended to the time ordered sequence <dS> as each new set of data items is processed.

The first and second substeps of Get RT Datum are relatively straightforward, but each adds further useful provenance information to data items. The Check Ranges step takes as input a measurement artifact that was output by a sensor and determines whether it is beyond preset bounds. While the details of the range checking are not particularly difficult or interesting as process features, the specification of the details of the check are important provenance information that should be associated with the data item.

Fig. 11 shows the DDG that results from the application of such a filter. The box labeled Filter Range Data represents the specific values used in applying the filtering to the data item, and the oval labeled Check Ranges represents the application of that step to the value represented by the box labeled P1, using the values specified by Filter Range Data. This results in the creation of a new instance that is bound to the variable P1. Note that this DDG documents the exact quality criterion that was applied in checking the plausibility of the resulting data item. If an investigator wanted to experiment with a different filtering criterion it could be reapplied to the previous data items available from this provenance documentation.

Similarly, the Select Values of P step selects one of the two values P1 and P2 and assigns it as its output artifact, namely P. This example show that this selection is done by an "Expert P Selector" type agent. This specification does not preclude the possibility that the agent might be a human or an automated agent having expertise in selecting the preferred value. The choice is relegated to an automated resource management system, which would presumably use execution state data as the basis for this decision. This choice would then be documented in the DDG, as illustrated in Fig. 12. Note that the range checking of P2 (which would precede Select Values of P) has been omitted here for brevity. Fig. 12 shows the case where Sensor 2 had succeeded in delivering a datum on the first request and where the human agent decided to accept this value as the final value of P. The figure contains an italicized annotation indicating that the chosen value was the one provided by P2, but the actual DDG need only contain the value. The fact that this value had been provided by P2 would be inferable by inspecting the two input values to the Select Values of P step, both of which would be available as nodes in the DDG.

The comparison of measured and modeled values (like the comparison of redundant sensors) also provides a form of real-time quality control and may provide an early warning of a sensor problem. The essence of the Apply Models step, in the Get RT Datum step of Fig. 10 is to effect the use of an empirical formula, called a *model,* to fill in data readings in such cases. As noted above, these models have been created by scientists for the purpose of interpolating values in place of those that are missing or suspect. The models are designed to provide replacements for such values, by using other data items and formulas that are believed to accurately capture the relations of these data items to the missing or suspect data. There are generally a number of alternative models that can be applied in these circumstances and these models are the subjects of considerable research. The model selected and applied to a data item is important provenance information that should be attached to that item. We indicate the way in which this additional provenance information is generated as part of the execution of the Apply Models step of this example.

The models themselves are often structured into sets of models of different types, for example, different types of regression models (*e.g.* linear or second-order) or different probabilistic models. The suitability of a model is then evaluated by a human, perhaps supported by various statistical tools. Eventually a model is chosen and then used to create a data value, which is substituted for the original data value. As this sort of synergy between humans and tools in the evaluation and application of models seems to be at the heart of many scientific activities, it is important to detail it here.

### *H. The Apply Models step: use of abstraction*

The purpose of the Apply Models step is to replace the data readings obtained from the sensors when the quality attribute of the reading is either missing or determined to be out of range (by the Check Ranges step). Although models can be applied to any number of data items, Fig. 13 shows the application of models to only two types of data, ET and Q. The definition of this step illustrates the value of abstraction in defining processes, since here the Apply A Model step is simply instantiated once for each type of data requiring the application of a model. Each Apply A Model substep has a prerequisite (not shown here) whose purpose is to examine the incoming data item and determine whether it has a quality attribute of "missing" or "out of range". Either attribute causes the prerequisite to be satisfied and the main body of the step to be executed.

Also note that the output of this step is a triple of items, namely the original ET value (ET.original), a replacement ET value that has been produced by the model that has been selected (ET.modeled), and an identifier (ETModelID) indicating which model was used to generate the replacement value. In fact, the first and third elements of this triple are redundant with information contained in the DDG, which provides more precise detail about the way in which ET was derived. The triple specified here is included as an indication that such annotations might be derived and included to help users by providing such information as documentation. Thus, this step provides the basis for distinguishing among different data items and different data streams that have been produced by different applications of different models by different scientists at different times. Note, in addition, that all three of the steps are carried out by agents who are required to be of type "HumanExpert". Each of the two substeps is defined to have as "local data" a set of readings that can be used by a selected model in computing the model-generated output of this step. Thus it is important that the language used to define the PDG for this process allows for the possibility of such "local data". Further the example illustrates the value of incorporating scoping semantics into the language that support specification of how such "local data" can be collected and held in the defined local scope, presumably in any way that the model and its agent decides.

### *I. The Apply A Model step: synergizing the efforts of humans and tools and representing a history of human decision-making*

The purpose of this step is to consider iteratively each model selected from the pool of models available, and then to select and apply the model that appears to be most effective in replacing the datum that has been identified for replacement. The step is comprised of two substeps, executed in sequence. The step also incorporates an exception handler to deal with the case in which no model is selected and applied.

The first substep of Apply A Model, Eval Models is shown in Fig. 14 and is the iterative consideration of the set of candidate models. Note that the agent for this step is identified as being of type "Model selector". Here too, this agent might be a human or an automated system capable of performing preliminary evaluations of models. This process does not mandate either, but allows this to be determined at runtime.

Eval Models iteratively evaluates each model in the collection of currently available models by first checking the model's applicability (represented by the pre-requisite) and then consulting an expert to evaluate the model and the results that it gives. Note that Eval Model is to be executed by an agent of type "human expert", indicating that human expertise is required. We do not show the details of this step, but it could entail the use of statistical tools to evaluate model fit. Although a human expert here executes Eval Model, some of its substeps could be executed by automated agents.

The edge leading into Eval Model has a Kleene + annotation, indicating that this step is to be instantiated as many times as there are models. Combined with the pre-requisite, the net effect of this is to instantiate a step for the consideration of only those models that the step's agent deems worthwhile. It is important to note that Eval Models also has access to a cache of data values collected and stored locally to this step that can be used to help in deciding the suitability of each candidate model.

The output of the Eval Models step is a set of pairs, where each pair consists of a model and the output that it produces. Once this stream of sets of ordered pairs has been created the next substep of Apply A Model, Select Best Model, considers all of these candidate models and selects the one deemed best. Here a "human expert" is required to perform the Select Best Model step. This value then becomes a third component of the ordered triple, (sensordatum.original, sensordatum.modeled, Model ID), that is the final output of the Apply A Model step. Again, the first and third elements of this triple are redundant with the more precise derivation information contained in the DDG and should be regarded as documentation intended to be of value to the user.

Fig. 15 provides an example of a DDG that might be generated from one possible execution of this step. In this case, it is assumed that three different models (shown as PModel1, PModel2, and PModel3) are selected from the channel containing the accesses to all possible P Models. Each of the three is subsequently evaluated by the Eval Selected Model step, using Cached Data, the set of data instances that has been cached by the process, thereby producing three instances of Sensor Datum (namely Sensor Datum 1, Sensor Datum 2, and Sensor Datum 3). These three instances of Sensor Datum, and the three models that produced them, are then input to Select Best Model, which then produces as output the ordered triple (sensordatum.original, sensordatum.modeled, Model ID). The original sensor datum, an input parameter to this step, is now referred to as sensordatum.original.

The two DDGs described in Fig. 12 and Fig. 15 are both necessary to provide the complete provenance of SensorDatum.modeled. The DDG in Fig. 15 represents the evolution of SensorDatum.modeled from the datum that was selected as the result of the process depicted in Fig. 14. The combined DDG is shown in Fig. 16. This DDG accurately documents that it is this P value that is bound as the actual datum taken as the subject for consideration for replacement by the alternative models.

After Apply Models has executed, the new values that have been obtained from all of the sensors, or their modeled values, are appended to the end of the data streams that are being accumulated by the RT Periodic Processing step (Fig. 9). Each datum in the data stream is shown as a packet that contains information that is redundant with the DDG. This redundant information should be thought of as documentation intended to be helpful to the user. In this example, the packet consists of SensorDatum.original along with appropriate information about its provenance, such as the date and time of the original measurement, the sensordatum.modeled, the model used to generate sensordatum.modeled, and the date and time that model was applied.

This stream of ordered triples is the real-time output of this process and could be made available in real time. We assume that such data streams also will be labeled by a unique process identifier and the DDG representing the precise trace through the process, and with further annotations indicating the places, dates, and times at which the data streams were collected. As this information consists largely of pointers into the DDG, most of the cost of carrying this provenance information lies in the cost of the DDG itself, which should be relatively efficiently represented as a collection of pointers.

### *J. The PP Periodic Processing and Alt Model Processing steps: abstraction supporting process reuse*

Recall from the Process Data step shown in Fig. 9 that the real-time data stream is not the only data stream that is produced by this process. The Water Budget process calls for automatic post-processing of newly collected data streams exactly 30 days later. As noted above, the correction of missing or out-of-bounds data during real-time processing must rely only upon retrospective data, and we have shown that this can be done by allowing for the caching of such data locally in the Apply A Model step. But post-processing after 30 days allows data gathered both before and after the selected data item to be taken into consideration. This post-processing is done in the PP Periodic Processing substep of Process Data. In addition, consideration of further models is made at irregular intervals, resulting in the creation of new models and their application to various historical data streams. The way this is done is defined in the Alt Model Processing substep of Process Data. Note that the Process Data definition indicates that PP Periodic Processing is invoked every 24 hours, using an interval timer mechanism that works

analogously to the way in which a timer is used to trigger processing in the RT Periodic Processing step.

The Process PP Datum step (Fig. 17) consists of the reinstantiation of the step structure that supported the Process RT Datum step, described previously. In particular, Process PP Datum consists of a first substep, Get Last Month Data, followed sequentially by Get RT Datum. The purpose of Get Last Month Data is to identify the data stream produced exactly 30 days earlier and to output the information needed to access that stream. That having been done, the stream is then taken as the inputs to this new instantiation of Get RT Datum. The process steps required to carry out Process PP Datum are then the same as those needed to carry out Process RT Datum, and differ only in that 1) the data streams to be processed are now historical rather than streaming in real-time, and 2) some additional models, namely models that make use of prospective data, as well as retrospective data, can now be used to deal with missing or out of bounds data items. Thus some different channels are indicated as the inputs to the Process RT Datum step, but the step structure used to process them is identical.

To illustrate this, Fig. 18 depicts the DDG that would result from the execution of the Process PP Datum step, where for simplicity we assume that only one additional model is selected for application to the sensor datum. Note, in particular, that the sensor datum under consideration is actually the triple that has resulted from the execution of the Get RT Datum step shown in Fig. 10. . Thus, Fig. 18 shows that a fourth instance of sensor datum, Sensor Datum 4, has been produced by consideration of an additional model, and that this is now compared with SensorDatum.modeled, which had been the result of the prior comparison of three modeled data items and the original sensor reading. The result shown is the next instance of SensorDatum.modeled (indicated by italics), which is now made available for possible future consideration. Note that subsequent comparisons of SensorDatum.modeled with the results of other models would be represented as successive elaborations of the DDG shown in Fig. 18.

The reused subprocess steps of Get RT Datum must be designed to be applicable in both the RT Periodic Processing and PP Periodic Processing contexts. For example, the data measurements that are cached for use in the Apply A Model steps of Apply Models must include prospective data (acquired by some sort of look-ahead) in the PP Periodic Processing context, whereas they will have only retrospective data in the RT Periodic Processing context.

The reuse of modular capabilities is an inherent abstraction capability in programming languages that usually requires careful design choices. Thus, it should not be a surprise that the reuse of process definitions in a process language also requires some care.

Finally, note that the Alt Model Processing step is defined in a manner that is analogous to the definition of the PP Periodic Processing step. This step defines the way in which scientists can recall historical datasets and apply new models to them, thus offering new ways to replace missing or out of bounds data. The step is to be executed at any time that a scientist wishes to reanalyze a particular dataset. In all other ways, however, this process is virtually identical to the PP Periodic Processing step. Thus, in particular, the first substep of Alt Model Processing is a step whose only role is to identify historical datasets to be reanalyzed and to pass the information needed to access them as handles to channels. The sequentially executed next step is again Get RT Datum.

## V. Evaluation

The Water Budget example has provided a vehicle for demonstrating how an analytic web provides provenance information needed by both producers and consumers of scientific datasets. More specifically this example has demonstrated that a broad range of semantic capabilities, including hierarchical decomposition, abstraction, concurrency, exception management, and complex data handling facilities, are needed to define modern scientific processes. We now present a more detailed analysis of what the need for these capabilities tells us about desiderata for the semantic features of languages such as Little-JIL that are to be used as the basis for defining an analytic web's PDG.

### *A. Strengths and weaknesses of Little-JIL*

Little-JIL is not simply a vehicle for supporting hierarchical decomposition (as is the case in many other process languages), but is better thought of as a vehicle for implementing abstraction. The difference is that a Little-JIL step is accurately thought of as the definition of an abstract concept, capable of being made a concrete specification by its bindings to concrete artifacts and placement in a specific execution context. A step defines a scope, and thus establishes a context. One key mechanism for context definition is the binding of artifacts as a step's inputs and outputs. This capability is tantamount to a capability for passing arguments to a procedure. By varying the argument stream to and from a step, the step is made to perform somewhat differently in different contexts. Steps also provide different contexts by providing different exception handling capabilities. Every step may define a set of handlers for the various types of exceptions that may be raised in its scope. Different instantiations of a step may offer different exception handlers, thereby establishing different execution contexts.

The Water Budget example made interesting use of facilities for abstraction, for example, by its reuse of the Get RT Datum step. Reuse of this step emphasized the strong similarities in the ways in which RT Periodic Processing, PP Periodic Processing and Alt Model Processing perform their work. This reuse shortens the process definitions and clarifies the data consumer's understanding of these processes. Another application of this concept was seen in the Apply Models step, which consisted of two different instances of the Apply A Model step. This step definition emphasized the iterative nature of the step, yet left little doubt about the differences between the two invocations of its substeps, namely the differences in their arguments.

Some of the complexity in the Water Budget process is attributable to the way in which different activities occur in parallel. Data streams from various sensors are processed in parallel, and the data must be processed in real-time as data are gathered and transmitted concurrently. Simultaneously, the much slower activity of generating new models and evaluating

both new and existing models occurs. A parallel step is effective in defining what activities are executed concurrently with each other. Channels were effective in defining data streams between steps that were distant from each other in the architecture of the Water Budget process. Channels also supported synchronization. For example, the parallelism defined by the top step in Sensor Data Management clearly depicted the way in which sensor data was generated and processed.

In this example channel semantics were limited to FIFO queues and parameters passed by copy. But this process requires more semantic power. For example, the Eval Model step accesses a more or less static collection of models but it must retrieve a new copy at each access in case any of the existing models have been modified or new models have been added. Transaction-like semantics might better support the implementation of a step of this kind. Transactions would also support the manipulation of data at different levels of granularity. For example Get RT Datum produces a single result, but is used by PP Periodic Processing to produce a dataset containing all of the results from a 24-hour period. Transaction-like semantics could permit Get RT Datum to release individual results, but allow PP Periodic Processing to control the visibility until a complete dataset has been constructed.

The exception management facilities in Little-JIL enabled us to define features of the Water Budget process that contributed to its reliability and robustness. Thus these features seem important to include in any language used to define PDGs. For example, the Get Met Station Data step indicated both how to identify and how to respond to the lack of needed sensor readings. Requisites are particularly clear devices for showing where missing data can be detected, and exception handlers (such as Handle MS Sensor Timeout) were placed to clearly indicate where responsibility for responding to such contingencies was located. This example also showed the importance of dealing with exceptions that occur during the handling of exceptions themselves. The nested exception handling in the Get Met Station Data step provided an example of this. Such situations also emphasize the importance of providing facilities for specifying how to continue execution upon completion of exception handling.

The example also showed the importance of supporting the late-binding of resources to steps to permit flexible reactions to contingencies (*e.g.* by the run-time selection of agents to retry an execution of a step). For example, in the elaboration of the Handle MS Sensor Timeout step, the process defined the need to execute a step to obtain a data value. The required agent was specified only as an agent having the capability to provide a precipitation reading. The choice of the agent was left to a Resource Manager having a repository of agents, some of which offered this capability. The indicated facility for specifying a needed capability, rather than a specific agent, enables the late-binding of any of a number of possible agents to this step. It therefore enables a separate Resource Manager to keep track of the agents that are able to deal with a request at any given time, and thus supports the real-time selection of one that is actually able to satisfy the request.

In summary, using Little-JIL to support the specification of the PDG for this example helped clarify semantics needed in a language used to define PDGs. Little-JIL offered many specification language features that seemed quite useful and effective, but some important deficiencies were also noted. There is a striking similarity between these needs of the scientific community and what is generally provided by modern programming languages. This supports our intuition that scientific processes bear some strong similarities to computer software, and thus the challenges of defining them have strong parallels with the challenges of programming complex software systems. Thus, it is not surprising to find that a process language needs to incorporate the salient control features of modern programming languages. It is, moreover, not surprising to also find that modern capabilities for dealing with data, such as typing mechanisms, also seem important to the precise specification of processes. Indeed, another weakness of Little-JIL as a language for specifying processes seems to be its relatively weak support for defining data objects.

### *B. DDG Evaluation*

The example of the Water Budget process also showed how DDGs can be built incrementally as the execution of a PDG proceeds and can be defined as traces through the PDG. Dataset Derivation Graphs grow as directed acyclic graphs (DAGs), increasing in depth as PDG execution proceeds; iteration of processing steps is manifest as additional levels in the DDG DAG (e.g., Fig. 18 is an elaboration of the leaves of Fig. 16, which in turn elaborates DDGs shown in earlier figures). Each iteration of the steps of a PDG creates a new scope, and such scopes are root nodes in successive DDGs. This emphasizes the role of these steps in establishing scopes, and the DDG clearly illustrates this role.

DDGs that are derived from the executions of lengthy processes seem large and cumbersome. But it is the pictorial depiction of an entire DDG that is large. Their internal representations are typical tree-like structures that are amenable to terse internal representation. In addition, the depiction of the entire DDG is not likely to be of interest in most cases; simplified versions would probably suffice in many cases. Tools for allowing viewers of DDGs to tailor their views through devices such as elision seem necessary.

Clear depiction of the features of the DDG of greatest interest to dataset producers and consumers may prove to be a challenge. Long and complex process executions will yield DDGs whose depictions are indeed large and potentially confusing. Fig. 18 may seem daunting, certainly at first. But DDGs will be produced automatically from executing PDGs and have the virtues of being precise and accurate. We do worry about their size, as it may be considerable in the case of a long or elaborate process. Especially in view of this, we are also concerned about clarity. Thus efficient internal representation of DDGs, and clear DDG presentation is are an important focus of future research.

## VI. Related Work

There are numerous other scientific workflow projects, many of which have been presented at meetings, such as [14; 38; 41]. Most of these projects (e.g. Kepler [30; 3; 2], Taverna [45; 33], and JOpera [35; 28]) base their specification of process flow

upon the use of various kinds of DFGs (e.g., Fig. 1). For example, Kepler is based upon Ptolemy II [18; 6; 36], which uses a powerful and flexible DFG structure to specify how datasets can be moved between processing capabilities. Kepler integrates a broad range of support tools that help with such key activities as specification, execution, and visualization of scientific data processes. It seems particularly effective in supporting the processing of streaming data, such as data produced by sensors and intended for real-time processing. Chimera [26; 25; 16] was one of the earliest scientific workflow systems. It emphasized the use of pictorial visualizations to represent scientific processes. Chimera's pictorial representations depicted a form of a DFG. Taverna [45; 33] is a more recent system that seems to focus on supporting the integration of web services, particularly for the creation of bioinformatics applications. Taverna's integration mechanism is a workflow notation that is also based upon a DFG formalism. More recently JOpera [35; 28] has suggested the use of XML to specify scientific workflows as plugins that could be integrated using Eclipse. JOpera workflows also are based upon the use of a DFG formalism to represent scientific processes. Teuta [22; 23] 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.

We believe that the reliance of all of these other scientific workflow systems upon the DFG as their basis for the specification of processes is a major drawback. The DFG-based systems described above, for example, make it very difficult to support specification of any but the most straightforward kinds of looping. The example presented in this paper indicates some ways in which complex iterative control is essential to scientific inquiry. Moreover DFG-based process definitions complicate the clear depiction of such semantic features as exceptions and abstraction, whose importance was also demonstrated by the example in this paper. Fig. 1 is a DFG of the Water Budget process but it is not a satisfactory PDG as it lacks the ability to specify the details of all the different cases that can arise during execution of a scientific process. On the other hand, a DFG can often provide an intuitive depiction of how information flows through a system. Thus, a DFG might be useful as a complementary representation to the PDG for defining an initial, high-level view of the process. But difficulties arise in trying to maintain consistency between the PDG and DFG, especially at lower levels of hierarchical elaboration, where typical data- or control-flow-oriented graphs have been found to be clumsy for representing more detailed flow. Indeed, we have developed a tool that translates Little-JIL-based PDGs into internal representations that are essentially equivalent to DFGs. Our experimentation with this tool has shown that even modest amounts of use of concurrency and exception handling can cause the number of nodes in the generated DFG to be hundreds or thousands of times as large as the number of steps in the original PDG.

There is also a substantial amount of work aimed at supporting the documentation of the provenance of scientific datasets. Many of the approaches to provenance documentation are summarized in [39; 40]. Indeed, these approaches have been compared to each other more formally in [31]. These approaches seem to fall generally into two categories. In one approach (e.g. [8; 29; 1] each data artifact generated by execution of a scientific process is annotated with detailed information about the tool or system used to create the artifact, along with precise specification of the input artifacts used, and the output artifacts created. Each such annotation is then stored in a database. The complete documentation of the provenance of an artifact can then be obtained by recursively querying the database for the annotations that describe the activities that produced as outputs the artifacts used as inputs to the query. The second approach entails building a derivation graph on the fly as execution of the scientific process proceeds (e.g., [25] and Kepler [4]) .We note that these two approaches are essentially equivalent to each other. Both collect provenance information by documenting the execution trace that has led to the creation of the data artifact being documented. In the former, the provenance structure is stored implicitly and is created upon demand by database queries. In the latter, the derivation structure is built incrementally during execution.

Our own approach falls into the latter category, entailing the on-the-fly construction of a derivation structure, namely the DDG. What distinguishes our work from the prior efforts is that our DDG depicts the progress of execution through our PDG, a process definition structure that can define and depict more complex semantic structures such as concurrency, exception handling, non-trivial iteration, and abstraction. Our DDGs offer depictions of how these semantic features are used to contribute to the development of data artifacts. Thus, for example, artifacts produced on different iterations through a given activity are shown as the roots of distinctly different subgraphs of a DDG in which the context of each activity execution is provided by the DDG. Our work is strongly reminiscent of earlier work on the Odin project [12] that documented the ways in which collections of software tools were applied to produce software products. As in the case of the work described herein, Odin maintained two coordinated structures, a type structure, showing which types of software objects can be generated through the applications of which software tools, and an instance structure that recorded the specific software artifacts generated by a specific sequence of applications of tools.

The Odin Project was aimed at supporting the clear and precise documentation of how various software artifacts resulted from somewhat different applications of somewhat different versions of various tools. It could use that documentation to make smart decisions about what data to store and what data to re-derive, as well as to determine when to automatically do the derivation based on desired outputs. It thus extended earlier work on software configuration management (SCM), such as Make [24] and SCCS [37]. It seems significant, therefore, to note that fundamental problems in documenting scientific data artifact provenance bear a striking resemblance to fundamental problems in SCM [21]. In both cases there is a need to document and communicate a clear and precise understanding of how artifacts of interest

have come into existence. In both cases, the derivation history may be quite complex, and must be maintained despite such complications as changes in versions of tools, reworking of artifacts, concurrent activities by diverse agents, etc. Thus, it is not surprising that the solution approach suggested in this paper is strongly reminiscent of approaches taken in early work in SCM. Indeed, 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 [4; 9] and we suspect that future work in the area of scientific data provenance documentation is likely to follow closely the progress of the SCM field.

## VII. Future Work

We believe that the semantic features in Little-JIL present a useful starting point for considering the features that should be incorporated into languages that are used as the basis for defining the PDG, but we have also noted a number of shortcomings. Further investigation of the essential requirements for the semantics of a PDG is needed. Specific details of DDGs also require further evaluation. For example, we need to evaluate various internal representations of DDGs to determine how to store them efficiently while still supporting efficient creation of needed visual representations. Moreover, datasets represented by the nodes of the DDG might be best regenerated from scratch or they could be cached to expedite generation of subsequent datasets. Specific strategies for determining when and what to cache should be the subject of future research. While DDGs can be used as the basis for the creation and attachment of process metadata to datasets, further research is needed to determine how this is done best. Metadata standards such as EML are starting to appear and process provenance metadata might be most usefully seen as an augmentation of standard annotations of this sort.

The value of an analytic web will be greatly enhanced by the availability of a tool set that supports such capabilities as the creation of the PDG, the execution of the PDG, the automatic creation of the DDG, viewing and querying these graph structures, and reasoning about the soundness of the scientific processes defined. We have begun the creation of such a prototype toolset, called `SciWalker` Our existing prototype provides weak and preliminary support for dataset producers, and virtually no support for dataset consumers. Future work will integrate our existing Little-JIL language support into `SciWalker` and will capture dataset products of Little-JIL execution to support DDG creation and management. Extensive work on the development and evaluation of user interfaces to these tools, particularly emphasizing the sorts of visual depictions of the PDG and DDG, is clearly required as well.

Of particular interest is the possibility of using PDGs as vehicles for integrating other systems that support scientific workflow. Our view is that the basis of most of these systems upon a DFG model of such processes is severely limiting, and that the effective representation of such processes requires more expressive semantic features, such as those supported by the PDG presented here. But we also note that our concept of a PDG supports the idea that PDG steps can be performed by different agents, either human or automated. We suggest that existing scientific workflow systems, such as Kepler, might be used to define lower level scientific processes (e.g., those not entailing complex iteration or exception management), and that those process fragments might then be considered to be the agents responsible for performing PDG steps. Such an approach could make good use of the better developed performance features of established systems such as Kepler and the needed semantic features offered by PDGs.

We also propose to add an important new dimension of support for dataset consumers by integrating powerful analyzers into `SciWalker`. One form of analysis that seems particularly important to dataset consumers is finite-state verification, which is capable of examining a PDG in order to determine whether or not it is possible to execute inappropriate sequences of functional capabilities [5; 17; 13]. For example, faulty scientific inferences can result from the application by dataset consumers of certain types of interpolation models to datasets that already had been smoothed in certain ways by the dataset producer. This inappropriate sequence of events may occur only for certain combinations of executions of the producer's process with the consumer's process. Such potential combinations can be detected by finite-state analysis of PDGs representing both processes. We suggest that it is important to investigate how best to integrate such analysis capabilities into a toolkit such as `SciWalker`

Finally, we believe that the best way to make the progress needed in developing the ideas just outlined is to continue to create analytic webs to represent scientific processes of various kinds. Our work with ecological processes is encouraging, yet preliminary. We hope that there will be much more work of this sort, not just in ecology, but also in the representation of processes in a wide range of other sciences. This work should shed important light on the nature of languages needed to represent such processes, and tools needed to make them accessible to working scientists.

## VIII. Acknowledgements

We are grateful to many colleagues who supported this work and contributed key ideas. In particular, we wish to thank Ed Riseman, Al Hanson, David Jensen, David Foster, Julian Hadley, Paul Kuzeja, Howard Schultz, Bert Rawert, George Avrunin, and Mohammed Raunak for their advice, support, encouragement, and many stimulating conversations.

## References


[1] A. Aiken, J. Chen, M. Stonebraker, A. Woodruff, "Tioga-2: A Direct Manipulation Database Visualization Environment". Intl. Conf. on Data Engineering, 1996.

[2] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, S. Mock, system demonstration, "Kepler: An Extensible System for Design and Execution of Scientific Workflows", 16th Intl. Conf. on Scientific and Statistical Database Management, 2004.

[3] I. Altintas, A. Birnbaum, K. Baldridge, W. Sudholt, M. Miller, C. Amoreira, Y. Potier, B. Ludäscher. "A Framework for the Design and Reuse of Grid Workflows". Intl. Workshop on Scientific Applications on Grid Computing , Springer LNCS 3458, 2005.

[4] I. Altintas, O. Barney, and E. Jaeger-Frank. "Provenance collection support in the Kepler Scientific Workflow System". International Provenance and Annotation Workshop, Springer LNCS, Provenance and Annotation of Data, 4145: 118-132, 2006

[5] G.S. Avrunin, J.C. Corbett, and M.B. Dwyer. "Benchmarking Finite-State Verifiers", *Intl. Journal on Software Tools for Technology Transfer*, 2 (4), 317-320, 2000.

[6] P. Baldwin, S. Kohli, E.A. Lee, X. Liu, and Y. Zhao. "Modeling of Sensor Nets in Ptolemy II. Information Processing in Sensor Networks", 359-368, 2004.

[7] E. R. Boose, A. M. Ellison, L. J. Osterweil, L.A. Clarke, R. Podorozhny, J. L. Hadley, A. Wise, D. R. Foster. "Ensuring Reliable Datasets for Environmental Models and Forecasts", *Ecological Informatics*, , Vol. 2, No. 3, 2007, Elsevier, pp. 237-247.

[8] P. Buneman, S. Khanna, and W.C. Tan. "Why and Where: A Characterization of Data Provenance". J. Van den Bussche and V. Vianu, editors, Intl. Conf. on Database Theory, pp. 316-330. Springer, LNCS 1973, 2001.

[9] S. Callahan, J. Freire, E. Santos, C. Scheidegger, C. Silva, and H. Vo. "Managing the Evolution of Dataflows with VisTrails". IEEE Workshop on Workflow and Data Flow for Scientific Applications.

[10] A. G. Cass, B.S. Lerner, E. K. McCall, L. J. Osterweil, S. M. Sutton, Jr., Alexander Wise. "Little-JIL/Juliette: A Process Definition Language and Interpreter". 22nd Intl. Conf. on Software Engineering, Limerick, Ireland, pp. 754-757, June 2000.

[11] http://cleaner.ncsa.uiuc.edu/home/

[12] G.M. Clemm, L.J. Osterweil. "A Mechanism for Environment Integration." *ACM Transactions on Programming Languages and Systems* (1990) 12(1): 1-25.

[13] J.M. Cobleigh, L.A. Clarke, L.J. Osterweil. "Verifying Properties of Process Definitions". ACM SIGSOFT Intl. Symp. on SW Testing & Analysis. Portland, OR: ACM Press, 2000:96-101.

[14] B.F Cooper, R.S Barga. "Report on SciFlow 2006: IEEE Intl. Workshop on Workflow and Data Flow for Scientific Applications". SIGMOD Record, Volume 35, Number 3, p.54-56 (2006)

[15] http://www.cuahsi.org/

[16] E. Deelman, J. Blythe, Y. Gil, C. Kesselman. "Workflow Management in GriPhyN". 14th Intl. Conf. on Scientific and Statistical Database Management, J. Nabrzyski, J. Schopf, J. Weglarz editors, Kluwer, 2003.

[17] M.B. Dwyer, L.A. Clarke, J.M. Cobleigh, G. Naumovich. "Flow Analysis for Verifying Properties of Concurrent Software Systems". *ACM Trans. on Software Engineering and Methodology* 2004;13(4):359-430.

[18] S.A. Edwards, E.A. Lee. "The Semantics and Execution of a Synchronous Block-Diagram Language". *Science of Computer Programming*, Vol. 48, no. 1, July 2003, pp. 21-42.

[19] A. M. Ellison, L. J. Osterweil, J. L. Hadley, A. Wise, E. Boose, L. A. Clarke, D. R. Foster, A. Hanson, D. Jensen, P. Kuzeja, E. Riseman, H. Schultz. "Analytic Webs Support the Synthesis of Ecological Data Sets". *Ecology* V. 87, No. 6, pp. 1345-1358, June 2006.

[20] http://knb.ecoinformatics.org/software/eml/

[21] J. Estublier, D. Leblang, A. Van der Hoek, R. Conradi, G. Clemm, W. Tichy, D. Wiborg-Weber. "Impact of Software Engineering Research on the Practice of Software Configuration Management", *ACM Trans. on Software Engineering and Methodology*, v. 14, 4 (2005) pp. 383-430.

[22] T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, C. Seragiotto, and H. Truong. "ASKALON: A Tool Set for Cluster and Grid Computing". *Concurrency and Computation: Practice and Experience*, 17(2-4):143-169, 2005.

[23] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, M. Wieczorek. "ASKALON: A Grid Application Development and Computing Environment". 6th IEEE/ACM Intl. Workshop on Grid Computing, November 2005.

[24] S.I. Feldman. "Make- a Program for Maintaining Computer Programs." *Software - Practice and Experience*. (1979) 9(4): 255-265.

[25] I.T. Foster, J.-S. Voeckler, M. Wilde, Y. Zhao. "Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation". 14th Intl. Conf. on Scientific and Statistical Database Management, 2002.

[26] I. T. Foster, J.-S. Vöckler, M. Wilde, Y. Zhao, "The Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration". Conf. on Innovative Data Systems Research, 2003.

[27] http://harvardforest.fas.harvard.edu/neon/neon.html

[28] T. Heinis, C. Pautasso, G. Alonso. "Mirroring Resources or Mapping Requests: Implementing WS-RF for Grid Workflows". pp. 497-504.

[29] D.P. Lanter, "Design of a Lineage-based Meta-data Base for GIS", *Cartography and Geographic Information Systems,* 18(4) pp. 255-261, 1991

[30] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, Y. Zhao, "Scientific Workflow Management and the Kepler System", *Concurrency and Computation: Practice & Experience*, 18(10), pp. 1039-1065, 2006.

[31] L. Moreau, B. Ludäscher, I. Altintas, R.S. Barga, S. Bowers, S. Callahan, B. Clifford, S. Cohen, S. Cohen-Boulakia, S. Davidson, E. Deelman, L. Digiampietri, I. Foster, J. Freire, J. Frew, J. Futrelle, T. Gibson, Y. Gil, C. Goble, J. Golbeck, P. Groth, D.A. Holland, S. Jiang, J. Kim, D. Koop, A. Krenek, T. McPhillips, G. Mehta, S. Miles, D. Metzger, S. Munroe, J. Myers, B. Plale, N. Podhorszki, V. Ratnakar, E. Santos, C. Scheidegger, K. Schuchardt, M. Seltzer, Y.L. Simmhan, C. Silva, P. Slaughter, E. Stephan, R. Stevens, D. Turi, H. Vo, M. Wilde, J. Zhao, Y. Zhao. "The First Provenance Challenge", *Concurrency and Computation: Practice and Experience,* Wiley InterScience , 2007.

[32] http://www.neoninc.org/

[33] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, P. Li. "Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows". *Bioinformatics Journal* 20(17) pp 3045-3054, 2004.

[34] L. J. Osterweil, A. Wise, L.A. Clarke, A. M. Ellison, J. L. Hadley, E. Boose, D. R. Foster. "Process Technology To Facilitate the Conduct of Science". Software Process Workshop, Springer-Verlag Lecture Notes in Computer Science, Vol. 3840, pp. 403-415, 2005.

[35] C. Pautasso, G. Alonso, "The JOpera visual composition language". *Journal of Visual Language Computation* 16, pp. 119-152 (2005).

[36] http://ptolemy.eecs.berkeley.edu/ptolemyII/

[37] M.J. Rochkind. "The Source Code Control System." *IEEE Transactions on Software Engineering* (1975) SE-1: 364-370.

[38] Scientific Applications of Grid Computing: First Intl. Workshop, Beijing, Springer LNCS3458 (3), pp 119-132, 2004.

[39] Y.L. Simmhan, B. Plale, and D. Gannon. "A Survey of Data Provenance Techniques". Technical report 612 , Comp. Sci. Dept., Indiana University, 2005

[40] Y.L. Simmhan, B. Plale, D. Gannon, "A Survey of Data Provenance in e-Science". SIGMOD Rec. 34(3) pp. 31-36, 2005.

[41] 16th Intl. Conf. on Scientific and Statistical Database Management, 2004.

[42] http://www4.nas.edu/webcr.nsf/MeetingDisplay1/WSTB-U-05-0A?OpenDocument&ExpandSection=1

[43] A. Wise, A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, S. M. Sutton, Jr. "Using Little-JIL to Coordinate Agents in Software Engineering" Automated Software Engineering Conf. 2000, 155-163, 2000.

[44] A. Wise. "Little-JIL 1.5 Language Report". Computer Science, U. of Massachusetts, Amherst, October 2006. (UM-CS-2006-51)

[45] K. Wolstencroft, T. Oinn, C. Goble, J. Ferris, C. Wroe, P. Lord, K. Glover, R. Stevens. "Panoply of Utilities in Taverna", in Proc E-Science 2005, 1st IEEE Intl Conf on e-Science and Grid Technologies, 2005

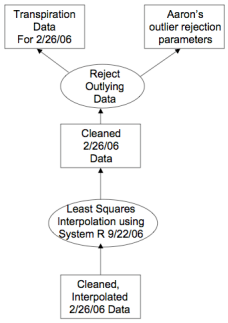



**Fig. 2**. An example of a Dataset Derivation Graph

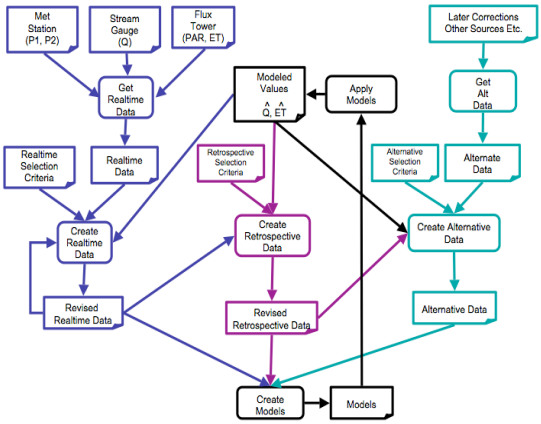



**Fig. 1.** A data flow graph of the water budget process. Each processing loop (real-time, retrospective, and alternative processing as well as model maintenance) is represented in a different color, with each edge colored by the same color as the node it emanates from.)

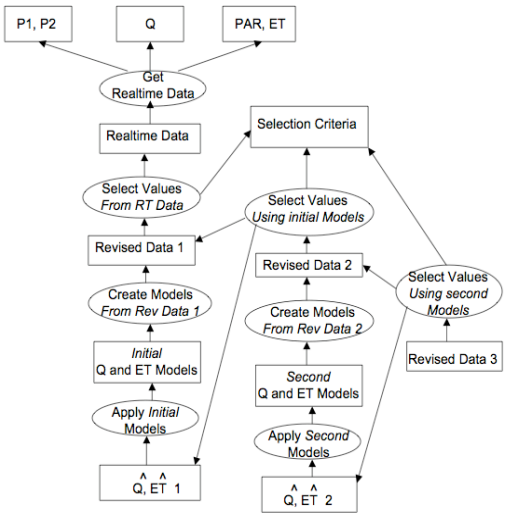



**Fig. 1.** The DDG showing the complete provenance of Revised Data 3

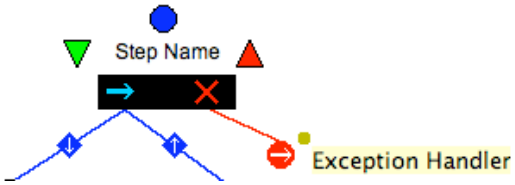
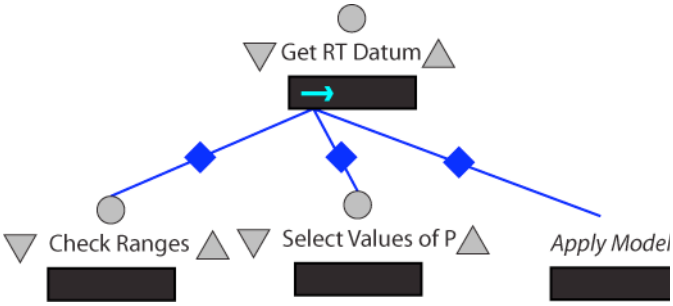


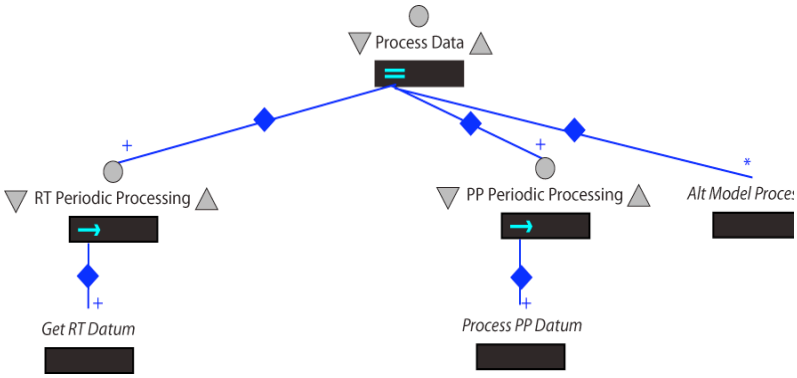

**Fig. 4.** A Little-JIL step icon




**Step: Get RT Datum**
Inputs: SetOfET {et}; SetOfQ {q}; SetOfP {p}; SetOfPAR {par}; SetOfVPD {vpd};SetOfUStar {uStar};ET et; Q q; P p; PAR par; VPD vpd; UStar uStar; // taken from the channel sensorStream
Outputs: SetOfETout {et}; SetOfQout {q}; SetOfPout {p}; SetOfPARout {par};SetOfVPDout {vpd}; SetOfUStarOut {uStar};

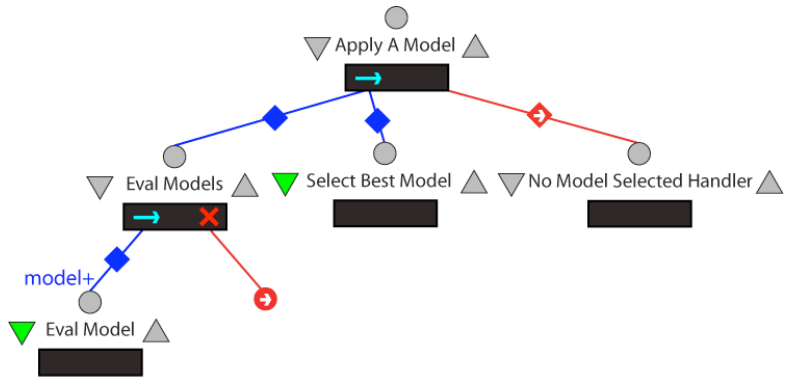
**Step: Select Values of P**
Inputs: PrecReading p1, p2;
Outputs: PrecReading p;

**Fig. 10.** The Get RT Datum step




**Step: RT Periodic Processing**
Comments: the channels sensorStream and modelStream are accessed from an enclosing scope
Outputs: seqofdS <dS>;

**Fig. 9.** The Process Data step




**Step: Apply a Model**
Outputs: ModeledDatumTriple modeledDatum =
new ModeledDatumTriple (sensorDatum.original, sensorDatum.modeled, modelID);
Exceptions caught: "No Model Selected" with "No Model Selected Handler"

**Step: Eval Models**
Inputs: SensorReading sensorDatum; Cache readingsCache;
Outputs: SetOfModelIDandDatumPairs {datumPairs}; //
where datumPairs=new DatumPairTuple( ModelID, SensorDatum.modeled);

**Step: Select Best Model**
Agent: human expert
Inputs: Cache readingsCache; SensorReading sensorDatum.original; SetOfModelIDandDatumPairs datumPairs //
where datumPairs=new DatumPairTuple(ModelID, SensorDatum.modeled);
Outputs: ModeledDatumTriple modeledDatumTriple =
new ModeledDatumTriple(sensorDatum.original, sensorDatum.modeled, modelID);
Exceptions raised: No Model Selected

**Step: Eval Model**
Inputs: SensorReading sensorDatum; Cache readingsCache; ModelType model;
Outputs: DatumPairTuple = new DatumPairTuple(ModelID, SensorDatum.modeled);

**Fig. 14.** The Apply a Model step

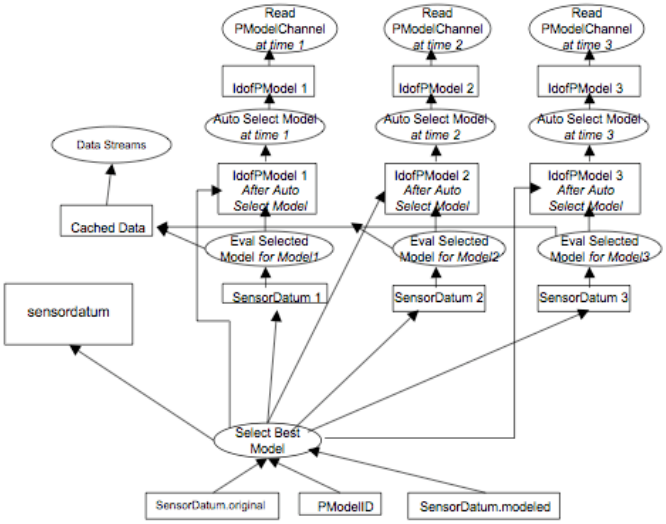



**Fig. 15.** The DDG representing the selection and application of a model to generate a modeled value for P

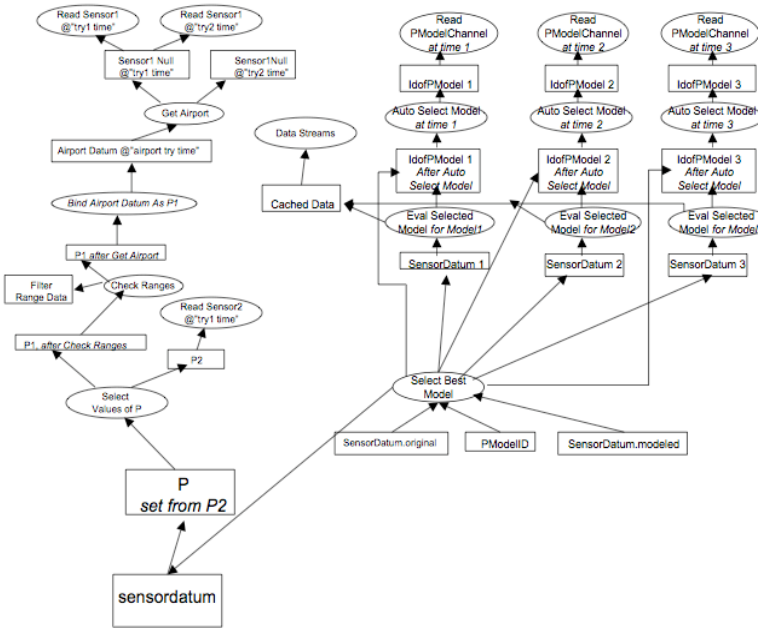
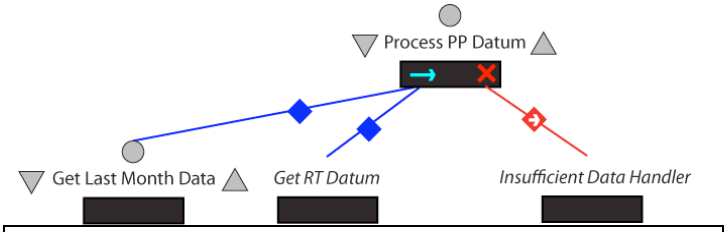



**Fig. 16.** The DDG formed by combining the DDGs of Fig. 12 and Fig. 15




**Step: Process PP Datum**
Outputs: SetOfET {et}; SetOfQ {q}; SetOfP {p}; SetOfPAR {par};
SetOfVPD {vpd};SetOfUStar {uStar}

**Step: Get Last Month Data**
FIFO Channel SensorStream sensorStream = new SensorStream();
// to be populated with"last month" data items

**Step: Get RT Datum**
Comments: "Augmented model set" is a set of models that also includes models that can benefit from prospective data, now available, as well as retrospective data

**Fig. 17.** The Process PP Datum step

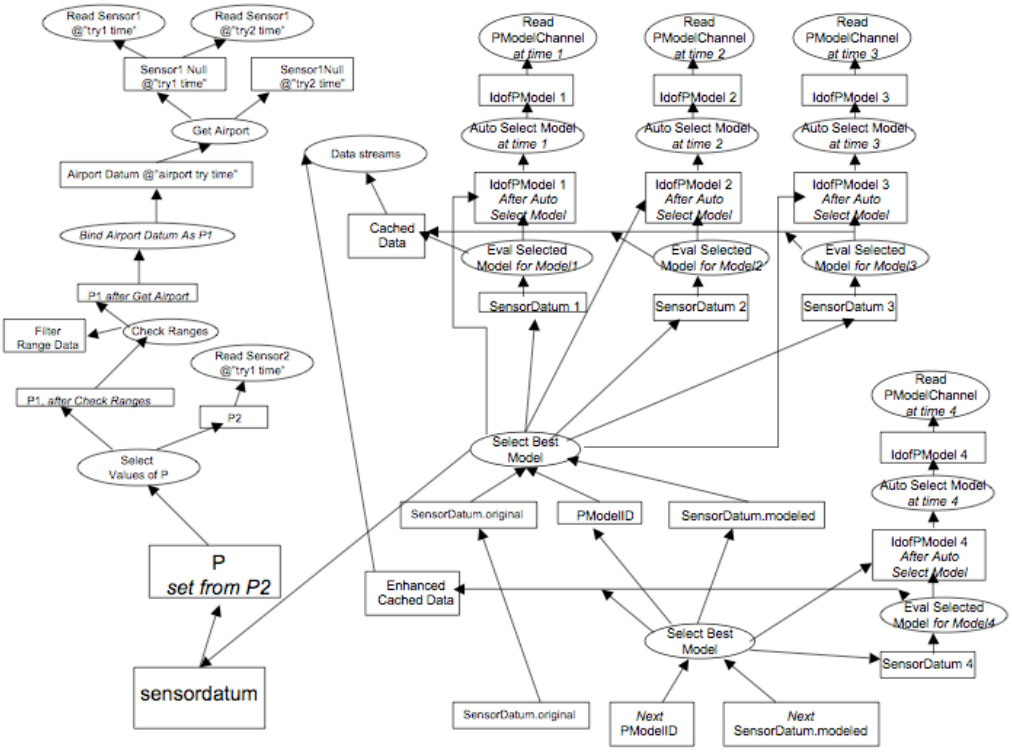



**Fig. 18.** The DDG showing the results of the execution of Process PP Datum