Provenance Support for Rework

Xiang Zhao
University of Massachusetts Amherst

Barbara Staudt Lerner
Mount Holyoke College

Leon J. Osterweil
University of Massachusetts Amherst

Emery R. Boose
Harvard University

Aaron M. Ellison
Harvard University

Abstract

Rework occurs commonly in software development. This paper describes a simple rework example, namely the code refactoring process. We show that contextual information is central to supporting such rework, and we present an artifact provenance support approach that can help developers keep track of previous decisions to improve their effectiveness in rework.

1 Introduction

Rework [5, 6] is a pervasive activity in creative processes such as software development and scientific data analysis. Our notion of rework is that it is the repeating of activities in new contexts when new information indicates that revising the work is desirable. Such situations arise quite often in software development. For example, a design that responds to a requirement specification may suggest that the requirement specification was inconsistent or incomplete, leading to revision of the requirement specification. This reconsideration, elucidated by new understandings derived from design considerations, is a simple example of rework. Further, modifying the requirement specification may then trigger further rework to deal with the effects of the modifications on design and perhaps code as well, possibly involving multiple rounds of rework. Indeed it is widely believed that developers typically spend much of their time doing rework. It is important to note that rework is inevitable, since, as work progresses, the problems being addressed become better understood and actions taken with earlier, less complete knowledge often need to be reviewed and revised. Since rework is inevitable it is important to find ways to make it more efficient and effective.

This paper uses articulate descriptions of artifact provenance to create *context* information that can improve the effectiveness of rework. Section 2 presents an example based on refactoring of an Object-Oriented (OO) program and discusses the role of software artifact provenance in creating context information that supports rework. Section 3 describes how we capture and use provenance through a structure that we call a Data Derivation Graph (DDG). Section 4 describes some related work. Appendix A presents a second rework example based on scientific processes.

2 Modeling Rework in Code Refactoring

Refactoring is an important activity that is carried out frequently in the course of OO software development. Refactoring an OO software product changes the product's internal structure without changing its external behaviors. Its goal is to improve such program characteristics as efficiency, readability, or evolvability. While there are many different kinds of refactoring (e.g. see [7]), in this paper we use the refactoring technique called **separating query from modifier**, that improves a badly designed method that is supposed to be used to query an object but has undesired side effects on the object state. The technique splits the method into one query and one modifier to eliminate the side effects, providing a query that is safer. We will demonstrate how this refactoring process incorporates multiple instances of rework, and indicate how using appropriate provenance data can support the creation of context information that helps users to be more effective with this kind of rework.

The **separating query from modifier** form of rework is described in [7] as follows: To begin, the (human) refactorer creates a query method that returns the same value as the original method. Next, the refactorer modifies the original method to return the result of a call to the query. Then for each reference to the original method, the refactorer replaces that reference with a call to the query preceded by a call to the modified method. Finally, the original method is assigned a void return type.

To accommodate the possibility of errors, compilations and unit tests are interspersed between the major phases of this refactoring process to check that each phase has been done correctly. A more complete and realistic refactoring process further indicates the rework that must be done if a compilation or unit test fails. This typically involves revisiting the work that has been done using an understanding of why that work failed to come up with another attempt that will hopefully succeed. The process specification must also accommodate the possibility that additional errors may be made in attempting to fix earlier errors, requiring more rework which may entail examining a lengthy history of previous attempts to fix the error. In addition, multiple errors may need to be addressed in parallel, etc. This brief explanation should suggest how provenance data can be useful as the source of relevant history, and how presentation of this data could comprise context information that could help guide the efforts of the refactorer.

We now provide a detailed specification of some key parts of this refactoring process, indicating where and how rework occurs, and how provenance data can facilitate these parts. We use Little-JIL, a process definition language. The salient features of Little-JIL are described in Appendix B and in [17, 18].

Figure 1 shows a high level Little-JIL definition of the second step of **separating query from modifier**, namely **Modify Original Method**. The step is decomposed into three substeps: making the change, compiling the changed code and rerunning a regression test set. Each of the last two steps throws a typed exception if the step uncovers an error, with each exception handled by a child of the **Modify Original Method** step. Yellow "post-it" notes document the flow of process artifacts (e.g. **sourcefilename**, the source file being modified) between process steps. Thus, for example, Figure 1 shows that after changes are made in **Change return statement**, **sourcefilecontent** is sent to the parent step, which passes it to the compile and unit test steps, which could then throw either the **CompilationFailureException** or **UnitTestFailureException** exception. Figure 2 shows the **UnitTestFailureException** handler, which has previously performed substeps (e.g. **Change return statement** and **Compile**) that can throw exceptions. These will be exception instances that are different from those thrown before, necessitating different rounds of rework aimed at fixing different aspects of the artifacts. Because of this refactorers have to make decisions in ever-deeper *contexts* as these artifacts evolve, making their correction increasingly difficult to understand. For example, Figure 2 shows how **Change return statement** could be executed several times but each time in a different *context*. The refactorer will then be faced with questions like: How did I get here? Why did previous fixes not work? How will my changes affect other artifacts? Appropriate contextual information can help answer these

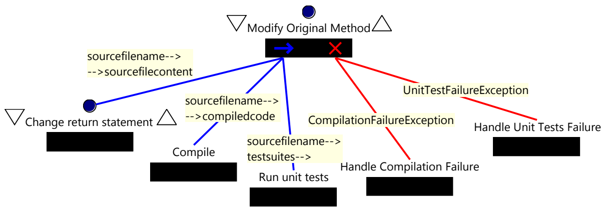

Figure 1: Method Original Method Step Definition

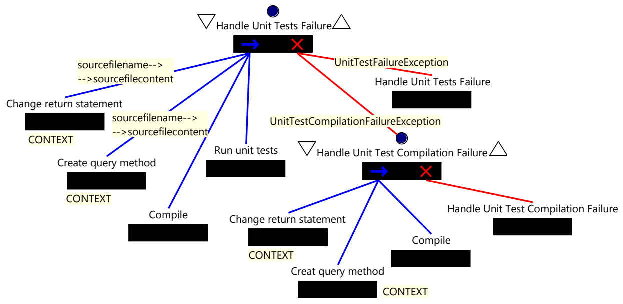

Figure 2: Handle Unit Tests Failure Step Definition

questions and support better decision-making. For example, the evolution history of the **sourcefilecontent** artifact could remind the refactorer of previous changes, thus helping the refactorer to avoid repeating a previous mistake, and suggesting a more suitable correction.

3 Provenance Support for Rework

We consider *context* to be the collection of all information about previous and current process execution states. We collect and store this information in a **Data Derivation Graph** (DDG) [9], which is an execution trace that records the data-flows and control-flows in a Little-JIL process as the process executes. Specifically, it records the step by which each artifact instance (including exceptions) is produced and consumed, the sequence of steps executed, the artifact values associated with each step execution, and the scopes within which step instance was executed. Figure 3 shows the DDG generated by executing a small portion of the refactoring process described in Section 2. Ovals represent step instance execution stages and rectangles represent artifact instances. A step's start and finish stages are separated to show how parent steps create scopes for their descendants (if any). Exception objects are shown in brown to distinguish them from other data objects. There are three types of edges, depicting data derivation, control flow and artifact versions. An arrow from an artifact instance to a step stage instance represents the derivation of that artifact instance from the execution of that step instance. An arrow from a step stage instance to an artifact instance indicates that

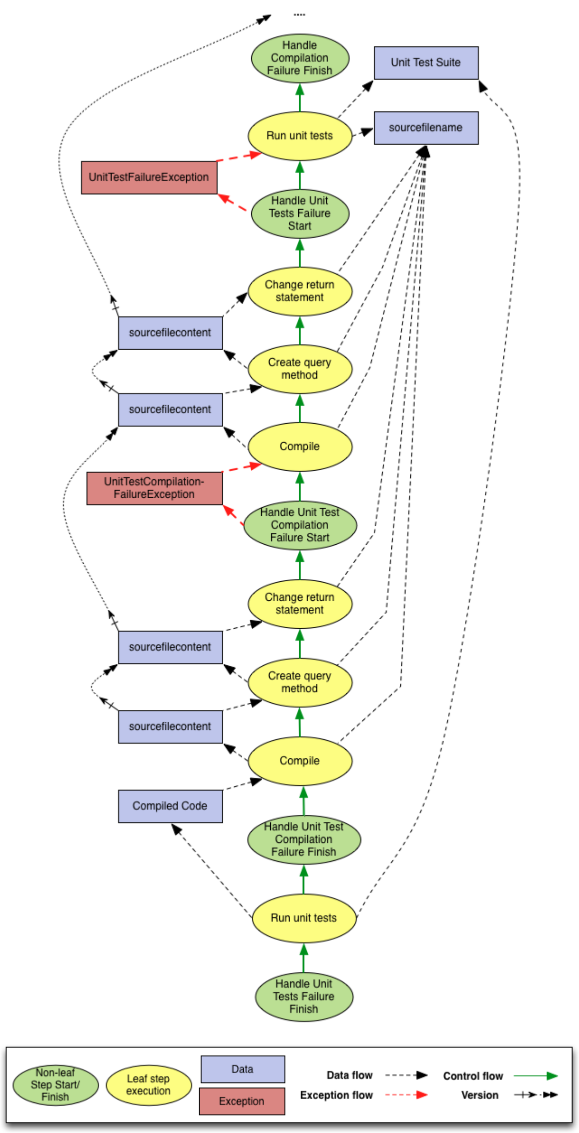

Figure 3: A DDG Example

the step derived its output artifact(s) using the artifact instance(s) being pointed to. For example, the fact that a **sourcefilecontent** instance points to the **Change return statement** step instance indicates that **sourcefilecontent** is derived from the step that modifies the source method to return a call to the created query method. Derivation edges related to exceptions are shown in red to distinguish them more clearly. The DDG also contains control flow edges, which represent the execution order between two steps. Version edges indicate the update series for some particular artifact. They can be traversed to provide a sense of the artifact's derivation history. Version edges are not generated in the DDG currently, but will be incorporated in future work.

Figure 3 corresponds to part of the rework process starting from the completion of the **Handle Compilation Failure** step in Figure 1. This example illustrates the result of running unit tests after the **UnitTestFailureException** is thrown, and as defined in Figure 2, the results from the refactorer's reconsideration of previous decisions and repeating of the **Change return statement** and **Create query method** steps. In the scope of **Handle Unit Test Failure**, compilation fails because the change made most recently fails triggering the **UnitTestCompilationFailureException** and causing another round of rework. The process definition indicates how the exception handlers are nested and thus how the rework activities are as well, thereby providing the basis for an accurate presentation of the histories of derivations of all variables comprising current context.

Our provenance support for rework automatically generates and maintains the DDG dynamically, making it accessible from all steps. Data objects in the DDG are linked to their actual values. These links and values are omitted from Figure 3 for simplicity. To suggest additional ways the DDG can aid rework, we incorporated a text-diff tool that records differences between DDG artifacts, which is particularly useful for comparing different versions of an artifact (found by traversing the *version* edges in Figure 3).

Our experience in modeling and executing this simple refactoring process suggests that the kind of provenance support we propose here provides useful artifact management and context information assistance to reworkers. This assistance becomes increasingly useful as the rework activity and associated contextual information become more complex, in particular in supporting rework processes in which modifications result in conflicts with each other, creating complex ripple effects that propagate through the artifact space.

4 Related Work

Rework in the form of iterative artifact development is central in the Spiral Model [1] and the Incremental Commitment Model [2]. Cass et al. [6] proposed initial approaches to formalizing rework processes, and later characterized a rework pattern [5] as being triggered by exception instances and fixed by revisiting previous steps.

Other technologies exist for capturing data provenance during workflow execution. VisTrails[8] tracks changes to data and constructs a history tree to capture provenance. Callahan et al. [4] incorporated this approach in a process setting and proposed a uniform environment. Kepler [3] provides a mechanism for integrating a broad range of supporting tools for specification, execution, and visualization of scientific data processes, and builds a provenance data store incrementally as is done by our

DDG. Some of the other approaches to provenance are summarized in [16]. We argue that exception handling and recursion are key features of Little-JIL that are missing from workflow languages and that enable creation of data provenance structures with semantic features essential to the effective support of rework.

5 Future Work

We will continue to improve our provenance support for the software refactoring processes. For example, we will consider how to properly place and show the version edges shown in Figure 3 in the actual DDG to help the developers to better understand the derivation history of the specific artifacts they are interested in. We are also building an interface in the Little-JIL step definition to invoke filter mechanisms for the DDG in order to provide more fine-grained contextual information per users' queries, which at the same time could be used to deal with the privacy issues related to the process by hiding sensitive private data.

6 Acknowledgments

The authors thank Sandy Wise for many conversations leading to important insights and ideas about how to design our refactoring process, Lori Clarke for numerous discussions about DDGs and their relationship to Little-JIL, and the students who have worked on the Harvard Forest project over the years: Cori Teshera Sterne, Morgan Vigil, Sofiya Taskova, Garrett Rosenblatt and Andy Kaldunski. We also thank the National Science Foundation for its support of this research through grants #CCF-0905530, CCF-0905530, DBI 04-52254 and DBI 10-03938. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] BOEHM, B. A spiral model of software development and enhancement. *SIGSOFT SW. Eng. Notes 11*, 4 (Aug. 1986), 14–24.

[2] BOEHM, B., AND LANE, J. A. New processes for new horizons: the incremental commitment model. In *Proceedings of the 32nd ICSE* (New York, USA, 2010), ICSE '10, ACM, pp. 501–502.

[3] BOWERS, S., MCPHILLIPS, T. M., AND LUDÄSCHER, B. Provenance in collection-oriented scientific workflows. *Concurrency and Computation: Practice and Experience 20*, 5 (2008), 519–529.

[4] CALLAHAN, S. P., FREIRE, J., SCHEIDEGGER, C. E., SILVA, C. T., VO, H. T., AND INC, V. Towards process provenance for existing applications. In *Proc. of IPAW* (2008), pp. 120–127.

[5] CASS, A. G., OSTERWEIL, L. J., AND WISE, A. A pattern for modeling rework in software development processes. In *Proceedings of the International Conference on Software Process* (Berlin, Heidelberg, 2009), ICSP '09, Springer-Verlag, pp. 305–316.

[6] CASS, A. G., SUTTON, S. M., AND OSTERWEIL, L. J. Formalizing rework in software processes. In *EWSPT* (2003), F. Oquendo, Ed., vol. 2786 of *LNCS*, Springer, pp. 16–31.

[7] FOWLER, M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[8] HOWE, B., LAWSON, P., BELLINGER, R., FREIRE, J., ANDERSON, E., SANTOS, E., SCHEIDEGGER, C. E., BAPTISTA, A., AND SILVA, C. T. End-to-end escience: Integrating workflow, query, visualization, and provenance at an ocean observatory. In *Proc. of the IEEE Intl. Conf. on e-Science* (2008).

[9] LERNER, B., BOOSE, E., OSTERWEIL, L. J., ELLISON, A., AND CLARKE, L. Provenance and quality control in sensor networks. In *Proceedings of the Environmental Information Management Conference* (Santa Barbara, CA, USA, 2011).

[10] MISSIER, P. Incremental workflow improvement through analysis of its data provenance. In *Proceedings of TaPP '11, 3rd Usenix Workshop on the Theory and Practice of Provenance* (Crete, Greece, June 2011).

[11] MISSIER, P., BELHAJJAME, K., ZHAO, J., ROOS, M., AND GOBLE, C. Data lineage model for Taverna workflows with lightweight annotation requirements. In *Provenance and Annotation of Data and Processes* (Salt Lake City, Utah, June 2008), no. 5272 in Lecture Notes in Computer Science, Springer, pp. 17–30.

[12] MISSIER, P., EMBURY, S., AND STAPENHURST, R. Exploiting provenance to make sense of automated decisions in scientific workflows. In *Provenance and Annotation of Data and Processes: Second Intl. Provenance and Annotation Workshop, IPAW 2008* (Salt Lake City, Utah, June 2008), no. 5272 in Lecture Notes in Computer Science, Springer-Verlag, pp. 174–185.

[13] MOREAU, L., PLALE, B., MILES, S., GOBLE, C., MISSIER, P., BARGA, R., SIMMHAN, Y., FUTRELLE, J., MCGRATH, R. E., MYERS, J., PAULSON, P., BOWERS, S., LUDÄSCHER, B., KWASNIKOWSKA, N., DEN BUSSCHE, J. V., ELKVIST, T., FREIRE, J., AND GROTH, P. The open provenance model (v1.01).

[14] MUNISWAMY-REDDY, K.-K., AND SELTZER, M. Provenance as first-class cloud data. In *3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS'09)* (2009).

[15] OLIVEIRA, F. T., MURTA, L., WERNER, C., AND MATTOSO, M. Using provenance to improve workflow design. In *Provenance and Annotation of Data and Processes: Second International Provenance and Annotation Workshop, IPAW 2008* (Salt Lake City, Utah, June 2008), no. 5272 in Lecture Notes in Computer Science, Springer-Verlag, pp. 136–143.

[16] SIMMHAN, Y. L., PLALE, B., AND GANNON, D. A survey of data provenance in e-science. *SIGMOD Rec. 34* (September 2005), 31–36.

[17] WISE, A. Little-JIL 1.5 language report. Tech. Rep. (UM-CS-2006-51), Department of Computer Science, U. of Massachusetts, Amherst, 2006.

[18] WISE, A., CASS, A., LERNER, B., MCCALL, E., OSTERWEIL, L., AND SUTTON, S.M., J. Using Little-JIL to coordinate agents in software engineering. In *Proceedings of the Fifteenth IEEE International Conference on ASE* (2000), pp. 155 –163.

[19] ZENG, R., HE, X., LI, J., LIU, Z., AND VAN DER AALST, W. A method to build and analyze scientific workflows from provenance through process mining. In *Proceedings of TaPP '11, 3rd Usenix Workshop on the Theory and Practice of Provenance* (Crete, Greece, June 2011).

A A Scientific Dataset Rework Example

The use of provenance can also be of central importance in the development of datasets that represent current or previous states of the world. Datasets include data collected by trained observers and remote, unsupervised sensors, both of which have varying degrees of accuracy or precision. Datasets are more than simply records of observations. Invariably, some observations will appear to be anomalous; on further inspection they may turn out to be accurate or inaccurate. Other observations may be completely missing due to such problems as sensor failure or communications difficulties. Inaccurate measurements may be adjusted based on auxiliary information or rejected outright (and converted to missing values). Missing data, whether screened as outliers or missing because of instrument or observer error may be replaced by modeled values. In sum, different values in a typical scientific dataset will have been arrived at by different means: observed, adjusted, or modeled. Scientists who access and use such datasets typically need to know their provenance, namely the ways in which the different values have been obtained. Many other investigators have made these observations and developed a wide variety of approaches to documenting provenance [3, 8, 12, 13].

Scientists typically regard the development of datasets as an ongoing evolutionary process. Often processes are applied to datasets iteratively and over long time periods. For example, the ways in which initial data values were screened and the modeled values used to replace them were calculated may be reviewed and reassessed many times, not only by the originator of the dataset but also by other individuals or groups. Such reassessment and reanalysis often result in the replacement of an earlier version of the dataset with a newer one. Revised datasets are common. Their associated provenances may be large and complex, reflecting not only variations in the ways in which initial data values were created but also the history of how individual data values have evolved as a result of multiple revisions.

The replacement of one dataset by another is typically determined, at least in part, by careful consideration of the factors that drove the generation of previous versions of the dataset. Thus it seems important to make available to dataset evolvers the provenance information that documents how previous datasets have been generated and have been replaced by newer versions. We think of provenance information as first-class data that is part of the rework process undertaken by scientists who examine both the data and their provenance when making decisions about rework. The result of the rework consists of a new version of the dataset and also an extension to the provenance reflecting the rework process itself.

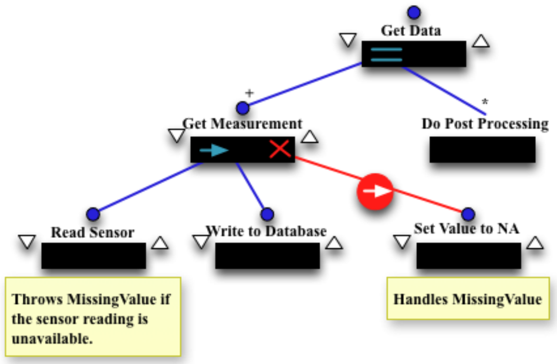

Figure 4: Collecting Sensor Data

A.1 Modeling the Scientific Rework Process

Figure 4 shows a simple scientific process written in Little-JIL to collect data from a sensor. This process repeatedly gets sensor readings and saves the values in a database. If a sensor reading is unavailable, an NA value is written to the database.

The Get Data process is completely automated. Later, the raw data are reviewed, either by a scientist or a software system, who (which) replaces missing values (NAs) or outliers with modeled values. Figure 5 shows these activities as the Fill Gaps and Replace with Modeled Value steps and their substeps. Of particular interest is the Insert Modeled Value substep of the Fill Gaps step. This step first evaluates the available models, noting what has previously been tried, and selects a model to apply. When applying the model, the scientist may determine that the model yields unsatisfactory results, leading to creation and application of new models.

Updating the Modeling Technique is the third substep of Do Post-Processing in Figure 5. This step first finds the values that were modeled with the technique the scientist wishes to replace and then repeats the Insert Modeled Value activity. This recursive use of Insert Modeled Value and Update Modeling Technique captures the notion of rework in the scientific process.

A.2 Provenance as First-Class Data

Figure 6 shows a portion of a DDG created during the execution of the Fill Gaps process. In this example, the Find Gaps step takes Sensor Data as input and produces Gap Locations (values = NA) as output. The Analyze History step takes the history of sensor values from the DDG as input and finds the models that were used to create the current version of the data. In this simplified example, the dataset consists only of sensor readings and

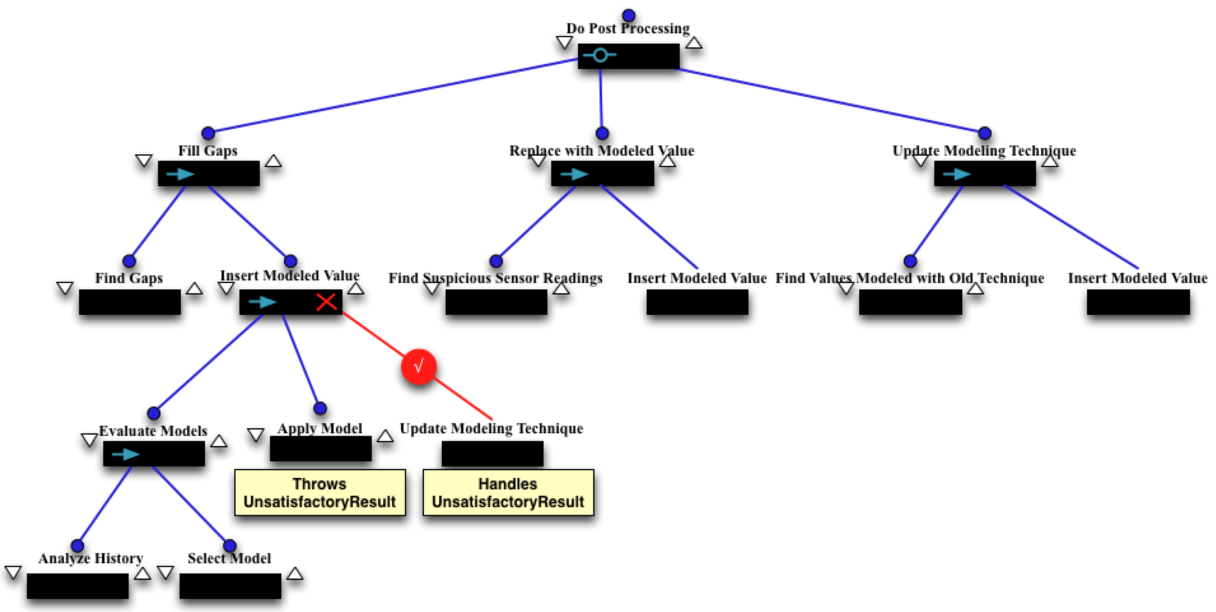
Do Post Processing
Fill Gaps
Replace with Modeled Value
Update Modeling Technique
Find Gaps
Insert Modeled Value
Find Suspicious Sensor Readings
Insert Modeled Value
Find Values Modeled with Old Technique
Insert Modeled Value
Evaluate Models
Apply Model
Update Modeling Technique
Throws UnsatisfactoryResult
Handles UnsatisfactoryResult
Analyze History
Select Model

Figure 5: Scientific Rework Process

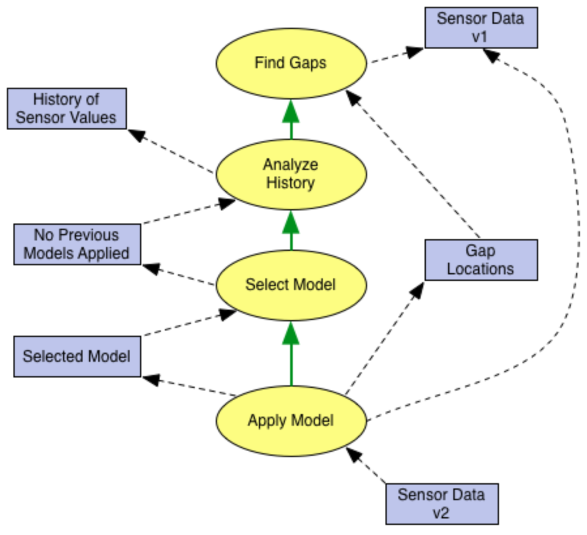

Figure 6: DDG of the Gap-Filling Process

NAs, so the output is that there are no previous models applied to the data. Also note that we have used an alternative view that omits the non-leaf nodes in order to present a more compact representation.

Figure 7 shows how the DDG can be used and enhanced during rework. This DDG shows an Unsatisfactory Result exception being thrown when the model is applied. This leads to the rework step of Updating Modeling Technique. The Find Values Modeled with Old Technique and Analyze History both use the History of Sensor Values extracted from the DDG as their input. This history also includes the result of Apply Model that just failed. As the process executes, the DDG is continuously updated and immediately available for examination within the process itself.

Figure 7 shows additional features associated with DDGs that describe rework. In addition to the control flow and data flow edges in Figure 6, other edges represent versioning and object equivalence. Specifically, the Sensor Data that is initially contained in the dataset may be replaced with new values when the Apply Model step is executed. One of the inputs of Apply Model is a Sensor Data object; it outputs a modified Sensor Data object. A version number on the node label distinguishes these objects. In Figure 7, different versions of Sensor Data are connected with double-headed edges; we can follow an edge from Sensor Data v3 to v2 to v1. Models used to produce those values also are connected with double-headed versioning edges.

Equivalence edges illustrate that multiple nodes can correspond to the same data. Data can enter the process either by being generated by the process directly or by being looked up in a repository. If a data value is calculated during the execution of the process, it will appear as an output from the step that calculated it. If it is passed as a parameter to another step, a data flow edge represents that. If the value becomes persistent, either because it is written to a repository or becomes part of the DDG, it can re-enter the process as the result of a repository

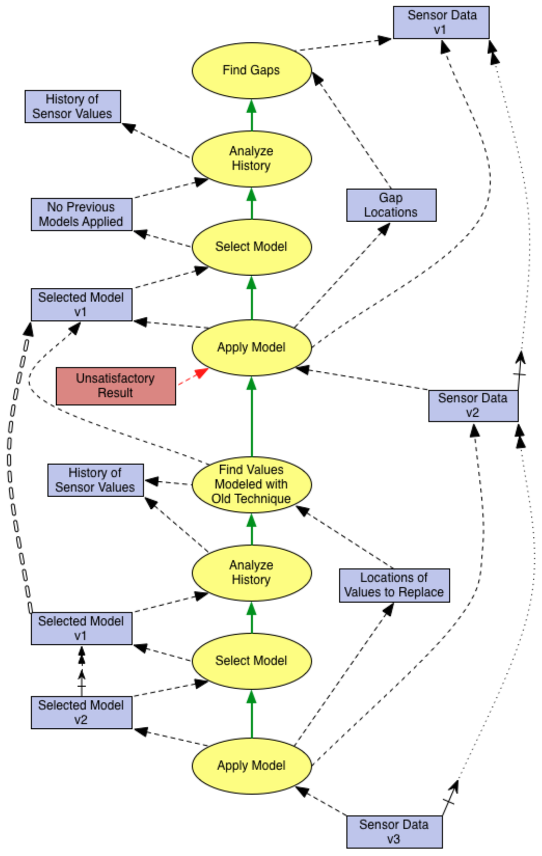

Figure 7: DDG of the Scientific Rework Process

query or a DDG query. The DDG shows this by connecting such nodes with an edge consisting of double lines to indicate that the data values are equivalent, but that the data did not flow directly from the step that output them to the later step that uses them. In Figure 7 an equivalence edge connects the Selected Model v1 node that is retrieved from the persistent DDG by the Analyze History step and the Selected Model v1 node that is the output of the Select Model in the top of the DDG to indicate that those represent the same model, even though the model did not flow directly between the steps involved.

While scientists generally acknowledge that the type of rework we describe here is common, there has also been little work in modeling these larger scientific process, particularly those that incorporate rework. Our work shows that doing this well is facilitated by appropriately powerful process language features, and by the inclusion of edges such as version and equivalence edges in the provenance structure. Oliveira et al. [15] organize related, perhaps reworked, scientific processes using process families. In contrast, we focus on making provenance first-class data, using it to provide to the working scientist the provenance of individual data and datasets (rather than whole processes) as they have evolved through previous process executions. This use of provenance data as first-class data is beginning to appear in other aspects of provenance research as well. Zeng et al. [19] mine provenance data and event logs to create more complex workflows. Missier [10] uses provenance data to learn and guide automated decision making in workflows that require thousands of iterations. Muniswamy-Reddy and Seltzer [14] use provenance data to optimize cloud storage.

B Little-JIL

Little-JIL is a graphical process definition language particularly suited for defining processes that require the coordination of multiple human and computational agents. Its semantics are precisely defined using finite-state automata. Among its distinguishing features are its use of scoping to make clear the identity of input and output datasets, its facilities for specifying parallel processing and for defining the handling of exceptional conditions, and the clarity with which iteration can be specified and controlled. A process is defined in Little-JIL using hierarchically decomposed steps.

A Little-JIL process definition consists of three main components: **artifact space**, **resource repository**, and **coordination definitions**. The coordination definitions include a collection of steps or activities that different agents are assigned to perform during process execution, and describe the coordination among the artifacts, activities, resources, and agents (which are treated as special kinds of resource). A Little-JIL coordination definition has a visual representation that is comprised of steps, which are hierarchically decomposed to the level of details (leaf steps) as users desire. Figure 8 shows the iconic representation of a single step. A Little-JIL step represents a task to be done by an assigned agent, and it can communicate with its parent steps and substeps through copy-in and copy-out parameter bindings of the artifacts. Each step has a sequencing badge to represent the type of control flow among its substeps, an interface to specify its input/output artifacts and resources, a prerequesite to be checked against before the step starts, a postrequesite to be checked against before the step reaches successful completion, and handlers for exceptions. A Little-JIL step also specifies how it should respond to events that may occur during execution and other features such as cardinality.

The rigorous and articulate data-flow and control-flow specifications in Little-JIL set up the basis for our provenance support. The complete specifications of Little-JIL can be obtained in [17]; we highlight some important features here.

- **Step sequencing**. Every non-leaf step has a sequencing badge (an icon embedded in the left portion of the step bar), which defines the order in which its sub-steps execute. Besides the sequential step shown in Figure 8, Little-JIL also supports concurrency, ordered choices, and unordered choices.

- **Data artifacts and data flows**. Each step declares the data that it creates and uses, while annotations on the edges (not shown in Figure 8) indicate how the data flows from one activity to another. As is shown in Figure 1, the **Change return statement** step declares **sourcefilename** as the input parameter and **sourceefilecontent** as the output parameter. The **sourcefilename** will be passed into its scope when the step is posted and ready to start, and the **sourcefilecontent** will be copied out once the step completes.

- **Requisites**. A Little-JIL step optionally can be preceded and/or succeeded by a step executed before and/or after (respectively) the execution of the step's main body. Requisites enable the checking of a specified condition either as a pre-condition for step execution or as a post-condition to assure that the execution has been acceptable. If a requisite fails, an exception is triggered to allow the error to be handled. The compilation and unit testing steps can also be implemented as post-requisites in our refactoring process definitions.

- **Exception Handling**. A step in Little-JIL can signal the occurrence of exceptional conditions when there are aspects of its execution that fail (such as violation of one of the step's requisites). These are important to allow for deviations in the execution of the process due to errors or unusual conditions. Our current process model treats the rework process as being triggered by exception instances, and the exception handler, as is defined in Figure 2, elaborates the rework process. In our refactoring process definitions after the exceptions are handled, the process will resume execution from the point where it was interrupted with **continue** semantics. Little-JIL also offers other exception handling semantics such as rethrowing the exception, restarting the step, and completing the step.

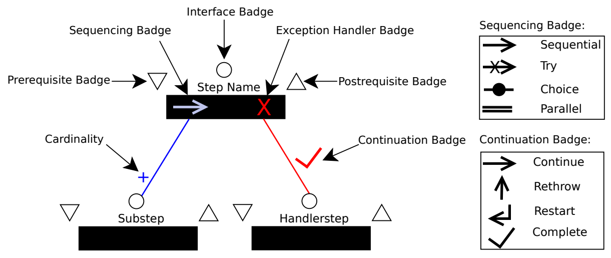

Figure 8: Little-JIL Step