PDiffView: Viewing the Difference in Provenance of Workflow Results

Zhuowei Bao [#1], Sarah Cohen-Boulakia [*2], Susan B. Davidson [#3], Pierrick Girard[#4]

[#] *Department of Computer and Information Science, University of Pennsylvania, USA*
[1]zhuowei, [3]susan, [4]pierrick@cis.upenn.edu

[*] *Laboratoire de Recherche en Informatique, Université Paris-Sud, France*
[2]cohen@lri.fr

1. MOTIVATION

Scientific workflow systems are becoming increasingly important for managing in-silico experiments. Such experiments are typically specified as directed flow graphs, in which the nodes represent *modules* and edges represent *data flow* between the modules. Each execution (a.k.a. run) of an experiment may vary the parameters and data inputs to the modules in the specification; furthermore, alternative paths of the workflow may be followed. In this process, the scientist's goal is to identify parameter settings and approaches which lead to good final results. Comparing workflow executions of the same specification and understanding the difference between them is thus of paramount importance for understanding the provenance of final results [4].

Although the problem of differencing directed acyclic graphs is NP-hard, an analysis of most real workflows shows that their structure can be captured as a **s**eries-**p**arallel graph overlaid with well-nested **f**orking and **l**ooping (SPFL). For this natural restriction, we have developed efficient, polynomial time differencing algorithms [1]. **P**rovenance **Diff**erence **View**er (PDiffView) is a prototype based on these algorithms for differencing runs of SPFL specifications (comparison with related works available in [1]).

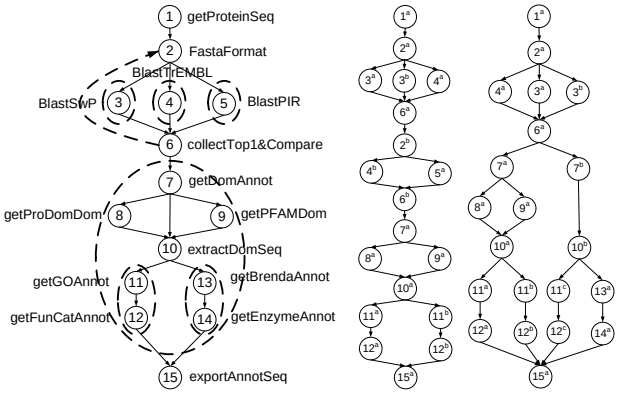

Figure 1: Protein annotation workflow

VLDB '09, August 24-28, 2009, Lyon, France

As an example, consider Figure 1(a) which represents a classical scientific analysis involving protein annotation. The aim of this analysis is to infer the biological function of a new sequence from other sequences. While the details of the scientific analysis are not important, the structure of the workflow is, and is shown using a modified dataflow notation annotated with control flow information for forks and loops. A loop is indicated by a dotted backarrow, *e.g.*, from module 6 (collectTop1&Compare) to module 2 (FastaFormat), and forking is indicated by a dotted oblong, *e.g.*, the oblong around module 3 (BlastSwP) indicates that similar proteins can be searched for simultaneously. Multiple outgoing edges from a node indicate a non-exclusive choice.

In a run, loops are unrolled and the number of fork executions is given explicitly. In addition, a nonempty subset of outgoing edges may be taken for each node. For example, two runs of the protein annotation workflow specification are shown in Figure 1(b) and (c). Observe that run R_1 has two executions of the loop from module 6 to module 2, while run R_2 has two fork executions between modules 6 and 15.[1]

The *difference* between a pair of runs is defined as a minimum cost sequence of path edit operations (*path edit script*) that transforms one run to the other.

PDiffView allows users to view, store, generate and import/export SPFL specifications and their associated runs. The user may then see the difference between two runs by stepping through the edit script, or by seeing a graphical overview of the difference.

Since runs may be very large, scientists may wish to see the difference between runs at a more abstract level. We therefore allow users to *group* modules of the specification into composite modules. By applying grouping recursively, a *hierarchy* of composite modules can be formed. When the differencing is performed, composite steps within which changes occur are highlighted. The user can then expand composite steps to see details of the difference.

An equally important difference in the provenance of two data products are the parameter settings and input datasets: Two executions could have exactly the same control flow but produce very different results due to the data used. We capture this in PDiffView by annotating nodes with parameter settings, and edges with the data flowing between steps. Annotations are revealed by clicking on nodes or edges.

[1]To avoid confusion, in the runs we annotate each node label with a superscript to generate unique identifiers for the module executions. For example, 3^a indicates that the step is an execution of module 3, BlastSwP.

2. PDIFFVIEW OVERVIEW

We have implemented the prototype system PDiffView in Java, whose architecture is shown in Figure 2. The process of comparing two workflow runs of the same specification is described as follows: The user starts by specifying two runs and optionally a well-nested hierarchy over the underlying specification through the graphical user interface (GUI). The differencing algorithm (implemented in the *Diff* engine) is then called to compute the minimum-cost edit script between two given runs at the lowest level of the workflow. If a user-specified hierarchy is present, the projection algorithm (implemented in the *Group* engine) is also applied to produce the corresponding hierarchy projected over the runs. Finally, combining these results by hiding internal changes inside composite modules, we present the user with an induced edit script at a high level view of the workflow as a compact description of the difference between two runs. A workflow generator is also provided to randomly generate a synthetic specification, or to simulate the execution of a given specification. In the remainder of this section, we will describe each of these building blocks in more detail.

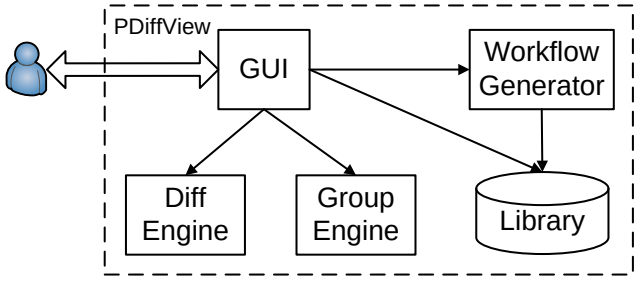

Figure 2: PDiffView system architecture

2.1 SPFL Workflow Model

An SPFL workflow specification is defined by a triple $(G, \mathcal{F}, \mathcal{L})$, where G is a series-parallel graph with unique labels on the nodes, and $\mathcal{F}$ and $\mathcal{L}$ are two sets of subgraphs of G describing the well-nested set of allowed forks and loops respectively. A workflow run R is then produced by applying a sequence of series, parallel, fork and loop executions recursively on the given specification. Intuitively, a series execution executes its sequential components in series; a parallel execution chooses a nonempty subset of all branches and executes them in parallel; a loop execution unfolds the cycle and executes all iterations of the loop in series; and a fork execution replicates one or more copies of the subgraph and executes them in parallel.

EXAMPLE 2.1. *Figure 1 shows two runs, R_1 and R_2, that are produced from the specification, $(G, \mathcal{F}, \mathcal{L})$, in which the loop is defined over the subgraph $(2, 3, 4, 5, 6)$, while the forks are defined over the subgraphs $(2, 3, 6)$, $(2, 4, 6)$, $(2, 5, 6)$ and etc. Note that only nodes inside the dotted oblong are replicated by an execution of the corresponding fork.*

SNAPSHOT 2.1. *Figure 3 shows a snapshot of viewing the specification depicted in Figure 1 using PDiffView. The big pane in the middle shows the specification graph G on which allowed forks and loops are highlighted in user-specified colors; this is done using the pane to the immediate right which lists the forks and loops. For instance, F3 has been selected on the small pane (in yellow). Consequently, the modules involved in F3 are highlighted in the big pane: getGOAnnot and getFunCatAnnot can have multiple parallel executions taking inputs from extractDomSeq and producing outputs sent to exportAnnotSeq. Similarly, the loop from FastaFormat to collectTop1Compare involving other three Blast modules has also been highlighted (in red). The small pane on the bottom left corner gives a miniature of this specification, and the small panes on the bottom right corner show the miniatures of two most recently visited runs of this specification. The users can view the detailed description of a run in a new window by double-clicking the corresponding miniature.*

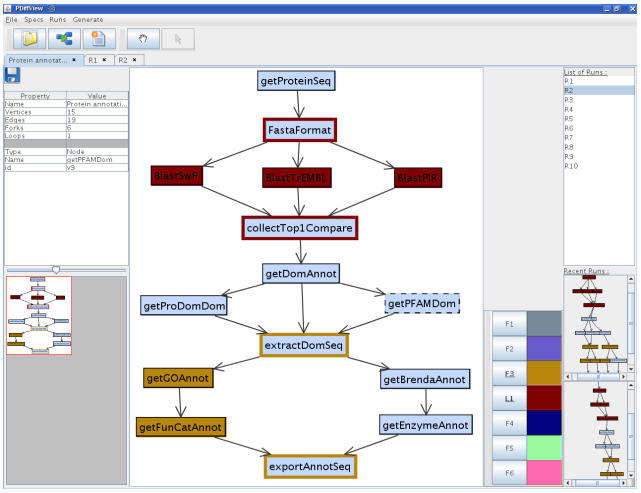

Figure 3: Viewing the specification $(G, \mathcal{F}, \mathcal{L})$ using PDiffView

2.2 Differencing Workflow Runs

The goal of differencing two runs is to find the minimum changes that transforms the first run to the second. We consider four kinds of *path edit operations*: (1) *Path Insertion:* Create a new path between two existing nodes; (2) *Path Deletion:* Remove a path (inverse of path insertion); (3) *Path Expansion:* Create a new iteration of a loop by inserting a path between two existing consecutive iterations; and (4) *Path Contraction:* Remove an iteration of a loop by contracting the last path (inverse of path expansion). Note that edit paths must be *elementary* such that each internal node has exactly one incoming edge and one outgoing edge, and must transform one valid run to another valid run.

EXAMPLE 2.2. *Consider the runs R_1 and R_2 in Figure 1. A path edit script that transforms R_1 to R_2 is shown in Figure 4. Note that each intermediate run is valid with respect to the specification in Figure 1(a).*

The *Diff* engine computes the minimum-cost edit script using the algorithm in [1]. This polynomial-time algorithm relies on a well-known tree representation of SP-graphs [5] with extra annotations for forks and loops, and adopts a very general cost model: Each path edit operation is assigned a cost of l^{ϵ}, where ϵ is a user-specified real number

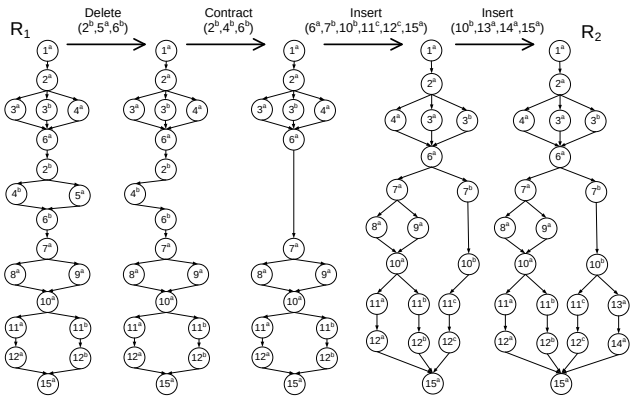

Figure 4: A path edit script from R_1 to R_2

no greater than one and l is the length of path to be edited. This feature allows users to capture a variety of application-specific notions of edit distance. For example, by setting ϵ to 0 (*unit cost model*) users will get an edit script with the minimum number of edit operations, and by setting ϵ to 1 (*length cost model*) users will get an edit script with the minimum number of inserted or deleted edges.

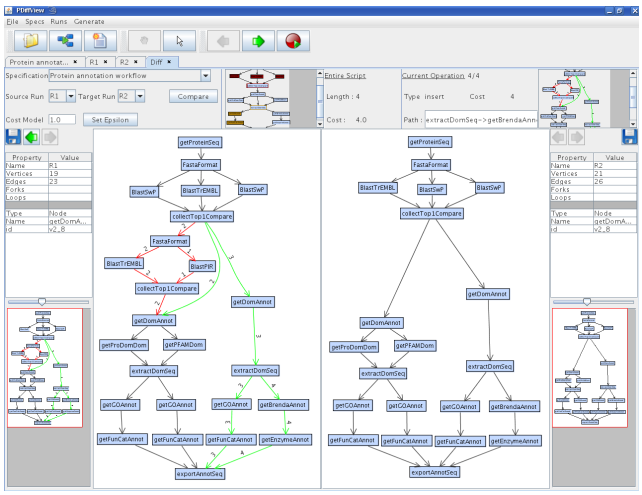

Figure 5: Viewing the difference between R_1 and R_2 using PDiffView

SNAPSHOT 2.2. *Figure 5 shows a snapshot comparing R_1 and R_2 using PDiffView. Users may see the difference by stepping through the minimum-cost edit script, or by seeing an overview. The edit script is the same as that given in Figure 4. The big pane on the left-hand side shows the source run R_1, with green edges indicating inserted paths and red edges indicating deleted paths in the edit script. The target run R_2 is shown in the big pane on the right-hand side. The small pane on top shows the specification, and the small pane on the top right gives the context for the edit operation being applied. The small panes on the bottom right and left corners display miniatures of the respective runs, and brief summaries of their statistics are listed above.*

2.3 Forming a Hierarchical View

Modules in a specification can be grouped together by the user to form *composite* modules. By recursively grouping modules, the user can create a *well-nested hierarchy* of composite modules over the specification, defining the levels at which differences between runs can be viewed.

While users are free to define composite modules over any set of modules, some groupings may give counter-intuitive results. Returning to the example in Figure 1(a), if nodes 3 and 8 were grouped together, the induced specification would have the cycle $(3,8) \rightarrow 6 \rightarrow 7 \rightarrow (3,8)$. This grouping would also break the loop defined over $(2,3,4,5,6)$ by merging a node (*i.e.*, 3) inside of this loop with a node (*i.e.*, 8) outside of the loop. To avoid such bad groupings, we suggest that users utilize the given forks and loops to form a more intuitive hierarchy.

EXAMPLE 2.3. *Using the forks and loops associated with the specification $(G,\mathcal{F},\mathcal{L})$ shown in Figure 1(a), we create a well-nested hierarchy $H = (1, A, B, 15)$, where $A = (2,3,4,5,6)$ and $B = (7,8,9,10,(11,12),(13,14))$ are two composite modules that represent the highest-level loop and fork respectively. Note that $(11,12)$ and $(13,14)$ are grouped together due to the inner forks. The induced specification $(G,\mathcal{F},\mathcal{L})^H$ is shown in Figure 6(a). Observe that the well-nested hierarchy constructed from forks and loops always gives a reasonable and meaningful partition of modules, resulting in a simplified specification graph that helps users better understand the differencing results.*

The user-defined hierarchy over the specification is automatically projected onto the runs so that the data and modules involved within any composite modules are hidden. A naive approach for doing this is to group together all instances of the same module in the run. Although this approach is straightforward, it suffers from two disadvantages: 1) it may introduce cycles or change the reachability between modules in the induced run graph; and 2) it may break the forking and looping semantics by partially merging different execution copies. The *Group* engine therefore uses an *SPFL-aware* hierarchy projection algorithm, which is a best effort approach to avoid the above two problems. The key idea is to find the minimum fork or loop which contains all nodes of a group, and then project this group onto the run by grouping together all module instances only within the same copy of this fork or loop.

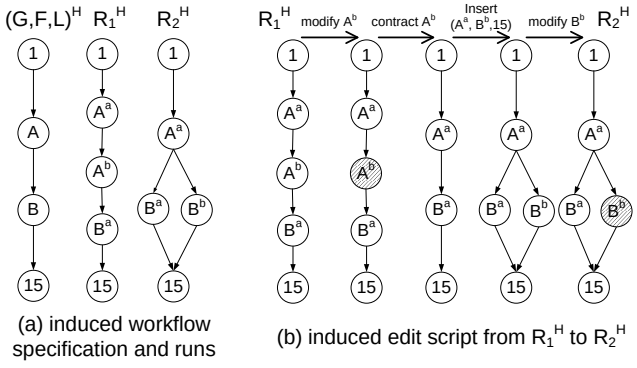

Figure 6: Induced workflow and edit script

EXAMPLE 2.4. *The induced workflow runs R_1^H and R_2^H produced by our projection algorithm are shown in Figure 6(a). We can easily observe that R_1 executes two iterations of the loop A in series, while R_2 executes two copies of the fork B in parallel. Figure 6(b) then shows an abstract edit script from R_1^H to R_2^H induced by the hierarchy H defined above. Note that the first operation edits a path inside the composite module A^b. We denote this internal operation by shading A^b in the resulting run. Comparing Figure 4 with Figure 6(b) shows that the induced edit script under a high-level view hides a large amount of details of internal edit operations that are irrelevant to the users, thus providing them with a compact description of provenance difference that can later be explored by expanding composite modules in the hierarchy.*

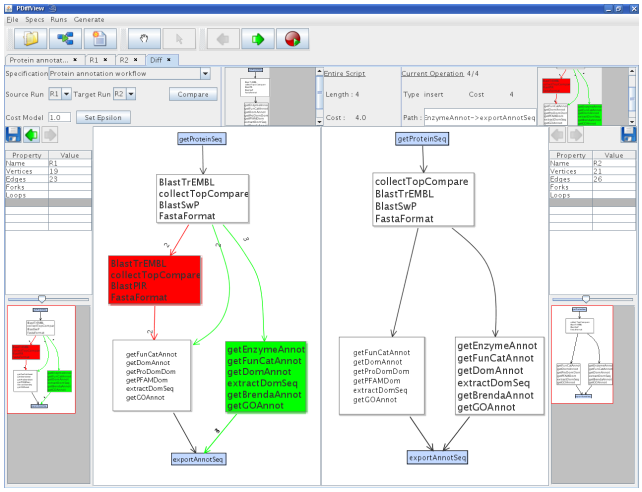

Figure 7: Viewing the difference at a high-level view using PDiffView

SNAPSHOT 2.3. *Figure 7 shows a snapshot comparing R_1 and R_2 under a high-level view H using PDiffView. The induced minimum-cost edit script shown here is the same as that given in Figure 6(b). We color a composite module green if an internal path insertion or expansion operation has been applied somewhere inside. In addition, if two incident edges are both colored green, then the entire composite module is newly inserted. Similarly, we use red to indicate internal path deletion and contraction. Since the grouping is performed interactively on the specification, the user may zoom in or zoom out of the current view by opening or grouping the modules in the specification. In this way, the user can easily adjust the level of granularity at which he wishes to view the differencing results.*

3. DEMONSTRATION

Our demo will be done using several selected workflows inspired from real workflows available on repositories such as myExperiments [1]. The demo will highlight the following features: 1) *Loading/Saving Data:* Users may load/save workflow specifications and runs to/from the local library, or import/export them to/from external XML files. We will also show how to generate synthetic specifications and runs given user-specified parameters; 2) *Viewing SPFL Workflows:* Users may zoom in and out of the workflow, and highlight forks and loops in different colors. Coloring helps users see the correspondence between the modules in the specification and those in the run; 3) *Differencing Workflow Runs:* Users may step through the minimum-cost edit script using the cost function of their choice; and 4) *Forming a Hierarchical View:* Users may recursively group modules in the specification to form a well-nested hierarchy that defines the level at which differences can be viewed. The hierarchy is then projected onto the runs, providing users with a compact representation of differencing results in which irrelevant changes are hidden inside the composite modules.

Our prototype system PDiffView and a demonstratation video are available at:

`http://www.cis.upenn.edu/~zhuowei/diff`

Why this demo is of interest to the db community? Provenance is of great interest to the database community (*e.g.*, the PODS 2008 keynote talk by Peter Buneman [2], and the SIGMOD 2008 tutorial by Davidson and Freire [3]). Since scientific workflows move data into and out of databases, understanding the difference in provenance between two data items in a database entails understanding their provenance through workflows.

4. ACKNOWLEDGEMENTS

This work was supported by NSF grant number IIS 0513778, SEII 0612177 and IIS 0803524.

5. REFERENCES

[1] Z. Bao, S. Cohen-Boulakia, S. B. Davidson, A. Eyal, and S. Khanna. Differencing provenance in scientific workflows. In *ICDE*, pages 808–819, 2009.

[2] P. Buneman, J. Cheney, W. C. Tan, and S. Vansummeren. Curated databases. In *PODS*, pages 1–12, 2008.

[3] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD*, pages 1345–1350, 2008.

[4] L. Moreau and B. Ludäscher, editors. *Concurrency and Computation: Practice & Experience, Special Issue on the First Provenance Challenge*, volume 20. Wiley, 2007. http://twiki.ipaw.info/bin/view/Challenge/.

[5] J. Valdes, R. E. Tarjan, and E. L. Lawler. The recognition of series parallel digraphs. In *STOC*, pages 1–12, 1979.