WOLVES: Achieving Correct Provenance Analysis by Detecting and Resolving Unsound Workflow Views

Peng Sun[1], Ziyang Liu[1], Sivaramakrishnan Natarajan[1]
Susan B. Davidson[2], Yi Chen[1]
Arizona State University[1], University of Pennsylvania[2]
{peng.sun, ziyang.liu, snatara5}@asu.edu[1], susan@cis.upenn.edu[2], yi@asu.edu[1]

ABSTRACT

Workflow views abstract groups of tasks in a workflow into composite tasks, and are used for simplifying provenance analysis, workflow sharing and reuse. An *unsound* view does not preserve the dataflow between tasks in the workflow, and can therefore cause incorrect provenance analysis. In this demo we present WOLVES, a system that efficiently identifies and corrects unsound workflow views with minimal changes (*view correction*). Since the view correction problem is NP-hard, WOLVES allows the user to choose between two forms of local optimality, strong and weak. Efficient time algorithms achieving these optimalities are implemented in WOLVES.

1. INTRODUCTION

Technological advances have enabled the capture of massive amounts of data in many different domains, taking us a step closer to solving complex problems such as global climate change and uncovering the secrets hidden in genes. Workflow management systems are therefore increasingly used for managing and analyzing this data, allowing users to specify complex, multi-step, "in-silico" experiments or analyses. To ensure reproducibility and verifiability of results, many workflow systems are now providing support for provenance [3].

The *provenance* of a data item is the sequence of steps used to produce the data, together with the intermediate data and parameters used as input to those steps. In general, it can be thought of as a graph which captures the causal dependencies between entities such as data and processes (a *provenance graph* [6]), and queries of provenance as calculating transitive closures of dependencies. As workflows and the associated data obtained from execution become large and complex, the size of the provenance graph as well as the cost of answering transitive closure queries become problematic.

For efficient provenance computation and feasible provenance analysis by users, we explore the use of workflow *views*. By abstracting groups of tasks in a workflow into high level composite tasks, a view can hide irrelevant details and be much smaller than the original workflow. Thus analyzing provenance queries that involve transitive closures at the view level can be more efficient than that at the workflow level.

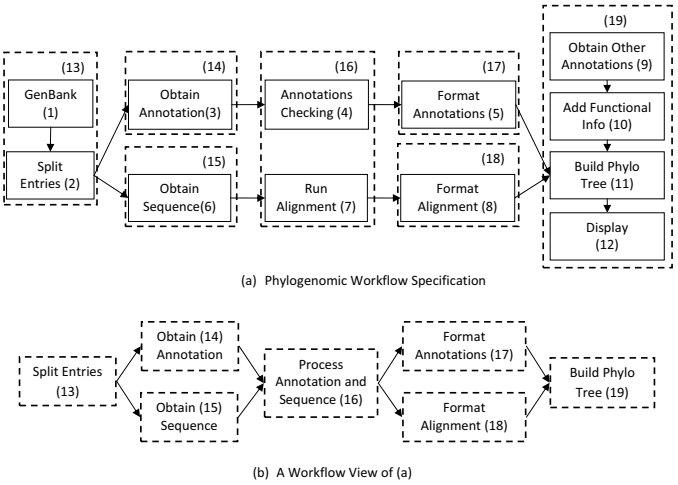


Figure 1: Sample Workflow Specification and View: Phylogenomic Inference of Protein Biological Functions

As an example, consider the workflow in Figure 1 (a) which describes a common analysis in molecular biology: *Phylogenomic inference of protein biological function*. Tasks are modeled as nodes in a directed graph, where edges represent data dependencies between the tasks. First, users select a set of entries from a database, such as GenBank (1), and split the entries (2) to extract a set of sequences (6), and annotations (3). The retrieved annotations are then curated (4) and formatted (5) to build a Phylogenomic tree (11). For the extracted sequences, an alignment is created (7) and formatted (8). Other annotations may also be considered (9) and processed (10). A Phylogenomic tree will then be built (11) and displayed (12). Note that the graph itself is the provenance graph for the final output – the Phylogenomic tree – and that the data items flowing between tasks have been omitted for simplicity.

A view is constructed by abstracting the tasks in each dotted box into a *composite task* and preserving all the inter-composite task edges, as shown in Figure 1(b). For instance, the composite task *Build Phylo Tree* (19) consists of four atomic tasks, and simplifies the provenance graph for users who are not interested in details of checking additional annotations.

However, unless a view is carefully designed, it may not preserve the dataflow between tasks in the workflow, and thus can be misleading and convey incorrect provenance analysis. For example, suppose the user found that the formatted sequence output by task (18) in a workflow execution is not good, and would like to check

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

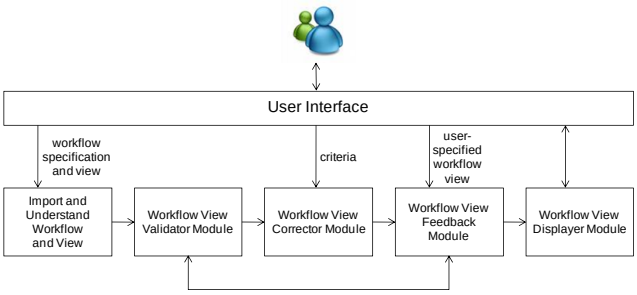


Figure 2: Architecture of WOLVES

its provenance with respect to the view in Figure 1 (b). Based on the view, all the outputs of tasks (13), (14), (15) and (16) will be considered as the provenance of the output of task (18), since there are paths from each of them to task (18). Nevertheless, this is wrong! There is no path between node (3) (contained in (14)) and (8) (contained in (18)) in the workflow in Figure 1(a), i.e. the output of task (14) is not part of the provenance of the output of task (18) according to the original workflow. Provenance analysis based on this view would therefore be incorrect.

Ideally, a view should *preserve all the data dependencies* between tasks in the workflow, without adding or removing paths. We call such a view *sound* with respect to provenance.

Although it would seem natural to design views that are sound, our survey of workflow designs in a well-curated workflow repository [1] revealed unsound views. Furthermore, existing tools to construct views (e.g. [2]) do not guarantee this soundness property.

To address this challenge, in this demo we present a system WOLVES (WOrkfLow ViEwS), which *detects* and *resolves* unsound workflow views. Soundness diagnosis and correction can be done either by making suggestions while users are creating a view, or by correcting unsound views after the view is created. The technical contributions of our work include: (1) We introduce the concept of soundness of workflow views, which is crucial for the correctness of view based provenance analysis. We identify a new research problem: diagnosis and resolution of unsound views. (2) WOLVES efficiently validates workflow views. (3) We prove that the problem of refining an unsound view to a sound one with minimal changes is NP-hard. (4) WOLVES tackles this problem efficiently by providing polynomial time algorithms to achieve two forms of local optimality.

2. SYSTEM OVERVIEW

In this section we present an overview of the WOLVES system. Two main functions of WOLVES are 1) detecting unsound views and 2) resolving unsound views.

The architecture of WOLVES is shown in Figure 2. A user can *Import* a workflow specification and a view into WOLVES or she can construct a workflow view using WOLVES directly. The workflow specification and view can then be visualized as graphs with the help of GUI interface so that the user can *Understand* them in an intuitive way. After that, both the workflow specification and view are sent to *Workflow View Validator*, which checks the soundness of the view. A view that is unsound will be corrected with minimal change to the view by *Workflow View Corrector*, in which one of the three correctors can be selected by the user, each of which implements one different optimality *criteria*. An unsound view will be resolved with minimal cost using the specified corrector and shown to the user, who then decides whether the view is acceptable, or whether further changes should be made using *Workflow View Feedback*. The user-specified new workflow view will then be sent back to *Workflow View Validator* to start another iteration until the user is satisfied with the new workflow view. Finally, the view is shown to the user by *Workflow View Displayer*.

In the next subsections we discuss two important modules of WOLVES: *workflow view validator*, and *unsound view corrector*.

2.1 Workflow View Validator

We start by defining a *sound view* [1].

Definition 2.1: [Sound View] A workflow view is *sound* if it preserves the data dependencies in the workflow specification. Specifically, there is a directed path between two composite tasks T_1 and T_2 in the view if and only if $\exists t_1 \in T_1, \exists t_2 \in T_2$, such that there is a directed path between t_1 and t_2 in the workflow specification. ■

For example, the view in Figure 1(b) is unsound. As we can see, there is a path (data dependency) between composite tasks (14) and (18) in the view; however, there is no such path between the corresponding tasks (3) and (8) in the workflow specification in Figure 1(a).

Although soundness is a desirable property for a view, checking whether a view is sound can take exponential time, if Definition 2.1 is directly applied by checking all possible paths in a graph.

Interestingly, it is sufficient to check the paths between the input and output of each composite task to determine the soundness of the view, and this can be done in polynomial time. Thus we introduce the concept of a *sound composite task*.

Definition 2.2: Given a composite task T, let $T.in$ denote the set of atomic tasks in T that receive input from some atomic task $t \notin T$, and $T.out$ denotes the set of atomic tasks in T that send output to some atomic task $t \notin T$. ■

Definition 2.3: [Sound Composite Task] A composite task T in a workflow view is *sound* if and only if $\forall t_i \in T.in$ and $\forall t_o \in T.out$, there is a directed path from t_i to t_o in the workflow specification. ■

As an example, the composite task (16) in Figure 1(b) is unsound, since there is no path from atomic task $(4) \in (16).in$ to $(7) \in (16).out$ in Figure 1(a).

Proposition 2.1: A view V of a workflow specification W is sound if and only if all composite tasks in V are sound. ■

According to Proposition 2.1, the *Workflow View Validator* in WOLVES can efficiently check whether a view is sound without enumerating all possible paths in the workflow.

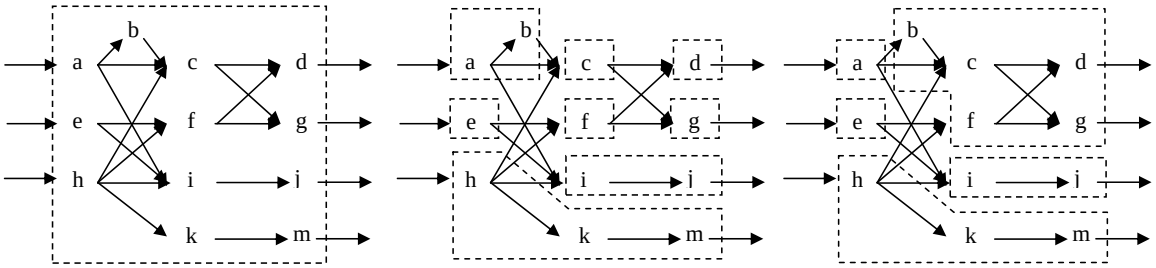
In the remaining part of this section, we focus on an even more challenging work: resolving unsound workflow views.

2.2 Unsound View Corrector

According to Proposition 2.1, a view is sound if and only if every composite task is sound. Thus, to resolve an unsound workflow view, we focus on the problem of *resolving unsound composite tasks*.

Two alternatives can be pursued for correcting an unsound task: Splitting an unsound composite task into multiple *smaller composite tasks* (i.e. the union of the set of atomic tasks corresponding to each resulting composite task is equal to the set of atomic tasks corresponding to the original composite task), or merging it with other composite tasks. Note that splitting composite tasks re-

[1]Due to space limit, some formal definitions, proofs of theorems and algorithms are omitted. For details, please refer to the full paper of this work [7].



(a) Unsound Composite Task (b) Weakly Local Optimal Correction (c) Strongly Local Optimal Correction

Figure 3: Local Optimal Corrections of an Unsound Composite Task

fines the initial view and provides more provenance information. In contrast, merging tasks loses information.Therefore, WOLVES resolves an unsound view by splitting unsound composite tasks rather than merging them. However, this problem is shown to be NP-hard.

Theorem 2.2: The problem of resolving an unsound composite task by splitting it to the minimum number of sound composite tasks is NP-hard. Its decision version is NP-complete. ■

To efficiently tackle this problem, we propose two optimality criteria: weak local optimality and strong local optimality. A weak local optimal solution is one in which no two tasks in the resulting view can be merged into a sound task (Definition 2.5), and strongly local optimal solution is one in which no subset of tasks in the view can be merged (Definition 2.6).

Definition 2.4:[Combinable Tasks] If two tasks T_1 and T_2 can be merged so that the resulting composite task is sound, then T_1 and T_2 are *combinable*, denoted as $T_1 \asymp T_2$. If a set of tasks $\mathcal{T}$ can be merged so that the resulting composite task is sound, then $\asymp (\mathcal{T})$. ■

Definition 2.5:[Weak Local Optimality] A split $\mathcal{S} = S_1, S_2, ..., S_n$ of an unsound task T is *weak local optimal* if and only if there does not exist $S_i, S_j \in \mathcal{S}$, $S_i \asymp S_j$. A *weakly local optimal algorithm* can guarantee for any unsound task T to produce a split which is weak local optimal. ■

For example, the weakly local optimal corrector will resolve the unsound task in Figure 3(a) to the result shown in Figure 3(b). Note that no two composite tasks in Figure 3(b) can be merged to form a sound task. For example, if we tentatively merge f and g to form a new composite task T, then T is unsound, since there is no path from $g \in T.in$ to $f \in T.out$.

However, from this example, we can also see that if we merge tasks c, d, f and g in Figure 3(b) to a single task, the resulting task is sound. This is not surprising, as weak local optimality is not optimal. The question is: is it possible to design a polynomial algorithm that can split an unsound composite task to fewer composite tasks than that produced by a weakly local optimal algorithm?

Toward this goal, we define another criteria: strong local optimality. Recall that weak local optimality states that no two output tasks can be merged. In contrast, strong local optimality makes a stronger requirement that no subset of resulting tasks are combinable.

Definition 2.6: A split $\mathcal{S} = S_1, S_2, ..., S_n$ of an unsound task T is *strong local optimal* if and only if there does not exist $\mathcal{S}' \subset \mathcal{S}$, $\asymp (\mathcal{S}')$. A *strong local optimal algorithm* can guarantee for any unsound task T to produce a split which is strong local optimal. ■

According to Definition 2.6, Figure 3(c) is a strongly local optimal split. Now comparing Figure 3(b) and (c), (b) is a split of the unsound tasks in (a) to 8 atomic tasks, while (c) is a split to 5 atomic tasks. Thus (c) is a strictly better correction.

However, achieving strong local optimality is much more challenging than achieving weak local optimality. A straightforward way of realizing strong local optimality is to check whether any subset of atomic tasks are combinable, which takes exponential time. We have designed a more sophisticated polynomial time algorithm to achieve strong local optimality which is incorporated in WOLVES, with time complexity $O(n^3)$, where n is the number of atomic tasks in the composite task.

3. IMPLEMENTATION AND DEMONSTRATION

3.1 Implementation and Evaluation

We have implemented WOLVES in C#. The input of WOLVES is a workflow specification and a workflow view, which can be an existing view or a view which is being constructed. We have tested the performance of the strongly local optimal, weakly local optimal and optimal (but exponential) algorithms implemented in WOLVES. Both the views manually defined by expert users, such as the ones in real workflow repositories, i.e., Kepler [1] and Myexperiment.org [5], and the views automatically constructed by [2] are tested. Experiments show that the strongly local optimal corrector in WOLVES is often able to produce views with similar quality to the one produced by the optimal corrector, but is several orders of magnitude faster. Furthermore, the efficiency of the strongly local optimal corrector is comparable with that of the weakly local optimal corrector.

3.2 Demonstration Outline

What will be shown in this demonstration? Figure 4 shows the GUI of WOLVES. As we can see, the panel is divided into three regions. The top panel, *specification panel*, shows the original workflow specification; the bottom left panel, *view panel*, shows the corresponding workflow view, which could be sound or unsound, and the bottom right panel, *result panel*, displays the correction results. Our demonstration will highlight the following features:

Importing and Understanding Workflow and View. A user may load into the system a workflow specification and a pre-defined workflow view defined in Modeling Markup Language (MOML) [4]. Alternatively, the user can construct a workflow specification and corresponding workflow view using our GUI by selecting *Workflow Builder* in the menu, and the view will be shown in the view panel.

Workflow View Validator Module. The validator automatically checks the soundness of the view wrt the workflow specification, and shows unsound tasks in red.

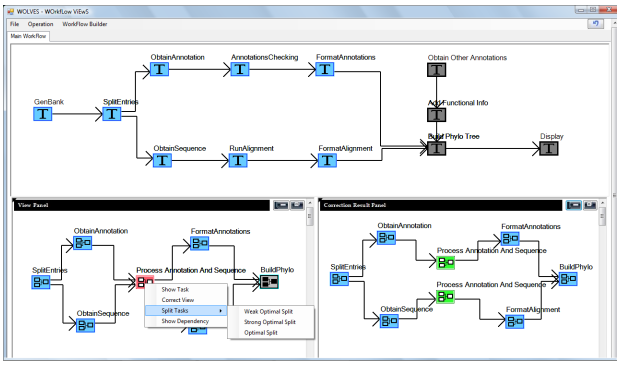


Figure 4: Detecting and Resolving Unsound Workflow View

Workflow View Corrector Module. This module allows users to specify the criterion used to correct the whole view or a selected unsound composite task only. After right clicking on any place of the view panel, the user can select *Correct View* and check one of the correction criteria in the popup window: weak local optimality, strong local optimality or optimality. Besides, users can also right click on an unsound composite task and choose one of the correctors under the popup menu item *Split Task* to resolve the specific task only. The corrected result will be displayed in the result panel, with the resulting sound tasks shown in green.

To assist users in choosing an appropriate correction approach, we provide the estimated time and quality for each approach. The *quality* of an algorithm is measured as the ratio of the number of resulting tasks generated by the optimal algorithm over that generated by the chosen algorithm. Thus, the higher the quality value the better, with the optimal algorithm having quality 1. To make an estimation of the execution time of correcting the current workflow, we group the workflows which have been corrected in the past according to their sizes and substructures, and report the average running time and quality of each approach for the group that the current workflow belongs to.

Workflow View Feedback Module. After the correction is finished, if the user is not satisfied with the refined view, she can modify the view output by WOLVES in the result panel. The user can select multiple tasks, right click on the result panel to show the popup menu, and choose *Create Composite Task* to merge the selected tasks. The result will be shown in the view panel, which will be sent back to *Workflow View Validator Module* for validation. This process continues until users obtain a satisfying workflow view.

Workflow View Displayer Module. All the workflows and views in any panel of the GUI are highly interactive. When the user double clicks on any composite task (or right clicks on it and then chooses *Show Task*), it turns grey and all the atomic tasks corresponding to this composite task will also be changed to grey icons as shown in specification panel (see Figure 4). Clicking *Show Dependency* returns to users the dependency relationship between the other tasks and the selected one. Furthermore, the user can adjust the position of tasks in each panel by dragging and dropping them.

What is the significance of WOLVES in the database community?

The proposed demonstration spans several important subareas in databases, such as provenance, workflows, user interface, as well as graph data management.

Provenance and Workflows. The importance of provenance analysis has been recognized in the database community. While traditional studies on provenance focused on analyzing provenance of SQL query results, there is a growing interest for provenance analysis for general complex data processing, such as business and scientific workflows. However, due to the complexity of workflows and the large data volume, providing correct and efficient provenance analysis is in great demand and yet very challenging. Over the past three years, research findings about workflow provenance have been presented as research papers, demonstrations as well as tutorials in major database conferences like SIGMOD, VLDB and ICDE.

This proposed demonstration introduces a new perspective on the area: view soundness problem for correct and efficient provenance analysis. There are many technical challenges. For instance, the problem of finding the minimal view refinement by task splitting is NP-hard. Allowing view abstraction by task merging, and the interaction between splitting and merging, are open problems.

Views as User Interface. As with database views, workflow views can be thought as an interface for users to issue queries and analyze results for large workflows and associated datasets that are stored in databases. However, unlike database views that are specified in SQL and have been formally analyzed, workflow views can be defined by arbitrary partitions of nodes in a workflow and have not been extensively studied. The proposed demonstration is a step in this direction.

Graph Management. Graph management has been an important topic in the database community, due to its wide application in social networks, computer networks, electronic circuits, metabolic pathways, and chemical structural graphs, to name a few. While we motivate the *sound view* problem in the application of workflow provenance, this problem and our proposed solution are general for diverse types of network data. An unsound view loses the *path information* of a graph, and thus the analysis of node connectivity based on unsound views are incorrect. To the best of our knowledge, our work is the first that identifies the potential problem of view-based graph analysis, and presents some initial solutions.

Since it touches on many subareas of databases and has general applicability to graph data management, WOLVES presents new theoretical challenges as well as practical solutions for database research.

4. ACKNOWLEDGEMENT

This work was supported by NSF grant number IIS-0513778, SEII-0612177, IIS-0803524, IIS-0740129, and NSF CAREER award IIS-0845647.

5. REFERENCES

[1] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock. Kepler: An extensible system for design and execution of scientific workflows. In *SSDBM*, pages 423–424, 2004.

[2] O. Biton, S. C. Boulakia, S. B. Davidson, and C. S. Hara. Querying and managing provenance through user views in scientific workflows. In *ICDE*, pages 1072–1081, 2008.

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

[4] E. Lee and S. Neuendorffer. Moml - a modeling markup language in xml version.

[5] myexperiment website. http://www.myexperiment.org/workflows.

[6] Open provenance model, 2008.

[7] P. Sun, Z. Liu, S. Davidson, and Y. Chen. Detecting and resolving unsound workflow views for efficient provenance analysis. In *SIGMOD*, 2009.