Detecting and Resolving Unsound Workflow Views for Efficient Provenance Analysis

Peng Sun [#1], Ziyang Liu [#2], Susan B. Davidson [*3], Yi Chen [#4]

[#] *Arizona State University, Tempe, USA*
[1] peng.sun@asu.edu
[2] ziyang.liu@asu.edu
[4] yi@asu.edu

[*] *University of Pennsylvania, Philadelphia, USA*
[3] susan@cis.upenn.edu

ABSTRACT

Workflow views abstract groups of tasks in a workflow into high level composite tasks, in order to reuse sub-workflows and to facilitate provenance analysis. However, unless a view is carefully designed it may not preserve the dataflow between tasks in the workflow, i.e. it may not be *sound*. Unsound views can be misleading and cause incorrect provenance analysis.

This paper studies the problem of efficiently identifying and correcting unsound workflow views with minimal changes. In particular, given a workflow view, we wish to split each unsound composite task into the minimal number of groups, such that the resulting view is sound. We prove that this problem is NP-hard by reduction from independent set. We then propose two local optimality conditions (weak and strong), and design polynomial time algorithms for correcting unsound views subject to these conditions. Experiments show that our proposed algorithms are effective and efficient, and that the strong local optimality algorithm produces better solutions than the weak local optimality algorithm with little processing overhead.

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 [1, 2, 3, 4, 5].

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 [7]. As workflows become large and complex, the size of the provenance graph as well as the cost of answering transitive closure queries becomes problematic, and a number of techniques have recently been proposed for reducing the size of the provenance graph and complexity of calculating provenance information [8, 7, 9].

In this paper, we explore the use of *views* for efficient provenance analysis. 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.

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 a set of annotations (3). The retrieved annotations are then curated (4) and formatted (5) to be served as input to building the Phylogenomic tree (11). For the extracted sequences, an alignment is created (7) and then formatted (8). Other annotations for the sequences (9) may also be considered and processed (10) to serve as input to (11). A Phylogenomic tree will then be built 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 has been omitted for simplicity.

By grouping the tasks in each dotted box into a composite task, the provenance graph can be viewed at a higher level 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.

Views are frequently used for purposes of modularization, abstraction and reuse when specifying workflows; examples of views can be found in workflow repositories such as MyExperiment [10]. Existing workflow management systems typically provide a graphical interface that allows users to specify composite tasks, which are the components of a view [11, 12]. The problem of automatically generating views for focussing user attention on "relevant" information was introduced in [9].

However, unless a view is carefully designed, it may not preserve the dataflow between tasks in the workflow, and thus can be misleading and cause incorrect provenance analysis. For example,

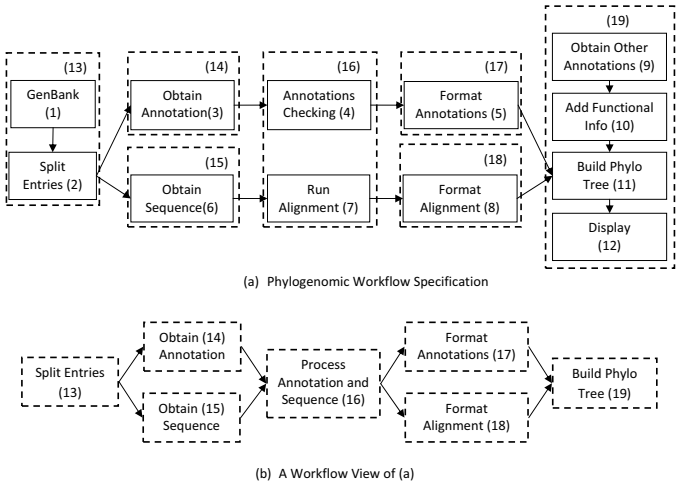


Figure 1: Sample Workflow Specification and View of Building Phylogenomic Tree

consider the view in Figure 1(b), and suppose a user would like to determine the provenance of the output of task (18). Based on the abstracted provenance graph, she would believe that tasks (13), (14), (15) and (16) are all involved since there are paths from each of them to task (18). However, there is no corresponding path between (14) and (18) in the workflow in Figure 1(a). Similarly, the view in (b) shows a path between (15) and (17), which does not have corresponding path in (a).

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 which are sound, our survey of workflow designs in a well-curated workflow repository [11] revealed several unsound views. Furthermore, the views that are automatically generated in [9] are not necessarily sound for composite steps that do not contain a (user-defined) relevant tasks.

Our focus in this paper is therefore on *diagnosing* and *correcting* unsound views. First, we formally define what it means for a view to be sound. However, detecting an unsound view according to its definition is exponential. We thus define an equivalent problem, identifying unsound composite tasks in a view, which can efficiently computed.

Then we design algorithms to correct unsound composite tasks. Two alternatives can be pursued for correcting an unsound task: Splitting it into multiple smaller tasks, or merging it with other tasks. Note that splitting composite tasks refines the initial view to a lower level and provides more provenance information. In contrast, merging tasks loses information, as tasks that are important to the user may be invisible after the merge. Therefore, in this paper we focus on techniques that resolve an unsound view by splitting unsound composite tasks rather than merging them.

Our goal is to correct an unsound view by splitting its unsound composite tasks to a minimal number of tasks, each of which is sound. We show that this problem is NP-hard by reduction from the independent set problem. To efficiently tackle this hard 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, and strong local optimal solution is one in which no any number of tasks in the view can be merged. We show that weak local optimality can be easily achieved with an $O(n^2)$ algorithm. However, achieving strong local optimality is much more challenging, as the straightforward way of doing so takes exponential time. We then present a more elaborate algorithm which achieves strong local optimality in polynomial time $O(n^3)$. The proposed algorithms are much more efficient than the algorithm which produces an optimal solution. The strongly local optimal algorithm often has comparable performance to the weakly local optimal algorithm, and produces solutions that are comparable to the optimal one.

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 contributions of our work include:

- We define the soundness of workflow views for the purpose of efficient provenance computation.
- We design algorithms to efficiently validate workflow views and identify unsound composite tasks.
- We prove that the problem of correcting an unsound task by splitting it into a minimum number of sound tasks is NP-hard.
- We design efficient algorithms for correcting unsound tasks which produce solutions of different forms of local optimality in polynomial time.
- A system for resolving unsound views was developed and verified for its efficiency and effectiveness through experimental studies. This provides an effective tool to guide the design of workflow views.

The rest of paper is organized as follows: Section 2 introduces the general workflow model and the definition of view. Section 2 defines sound views, formulates the View Soundness Problem (VSP) and its equivalent problem: Task Soundness Problem (TSP). Section 4 introduces two optimality criteria, weak local optimality and strong local optimality, and presents algorithms which achieve each criteria for unsound view correction. Experimental results are given in Section 5. Section 6 discusses related work, and Section 7 concludes the paper.

2. PRELIMINARIES

In this section, we introduce the background on the general workflow model and views.

Definition 2.1: A *workflow specification* W is a directed graph where each node corresponds to a task and each edge indicates the dataflow between them. $N(W)$ denotes the node set of W, and $E(W)$ the edge set. ■

A *view* is a directed graph that is an abstraction of a workflow specification by grouping some tasks along with their edges together into a single task as defined in [11, 13, 14, 9]. In other words, each task in a view corresponds to a group of tasks in the workflow specification.

For example, consider the sample workflow specification in Figure 2(a). A view specified by the user is shown in Figure 2(b), where the nodes from a to m in (a) are mapped to a single task M. (c) and (d) are another two views of (a). The formal definition of view is given below.

Definition 2.2: A *view* V of workflow specification W is a directed graph induced by a partition of the nodes $N(W)$: $\{P_1, P_2, ..., P_n\}$ such that $\emptyset \neq P_i \subset N(W)$, $P_i \cap P_j = \emptyset$ for $i \neq j$, and $P_1 \cup P_2 \cup ... \cup P_n = N(W)$.

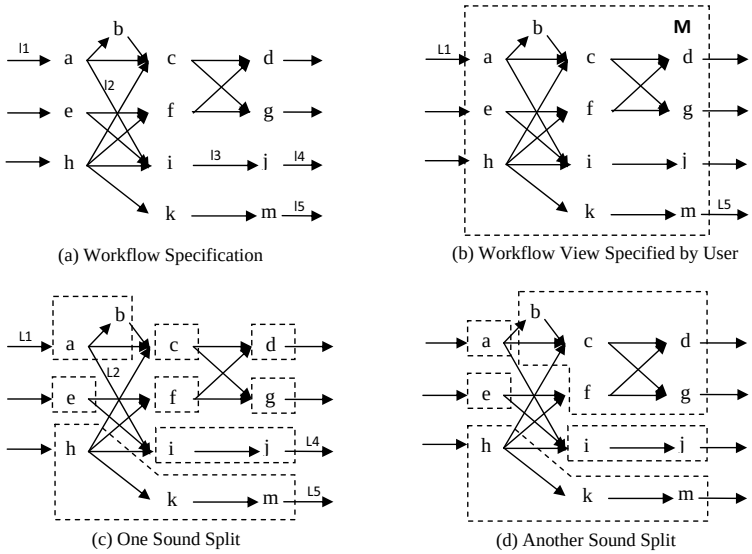


Figure 2: Sound versus Unsound Views

- The node set of V, $N(V)$, is defined by a bijection from the partition to $N(V)$, $\Phi : 2^{N(W)} \longrightarrow N(V)$. $\forall P_i$, $\exists T_i = \Phi(P_i) \in N(V)$. Conversely, $\forall T_i \in N(V)$, $\exists P_i$, such that $\Phi^{-1}(T_i) = P_i$.

- The edge set of V, $E(V)$, is defined by a surjection from $E(W)$ to $E(V)$, $\Psi : E(W) \longrightarrow E(V)$. $\forall l \in E(W)$ from t_m to t_n, $t_m \in P_i$, $t_n \in P_j$. If $i \neq j$, $\Psi(l) = L \in E(V)$, where L is an edge from $\Phi(P_i)$ to $\Phi(P_j)$; conversely, $l = \Psi^{-1}(L)$. Otherwise if $i = j$, $\Psi(l) = \epsilon$.

- Each $T \in N(V)$ is called a *composite task*, and each $t \in N(W)$ an *atomic task*. An edge $l \in E(W)$ is called an *intra-group edge* with respect to V if $\Psi(l) = \epsilon$, and an *inter-group edge* otherwise.

■

Intuitively, Φ maps each group in a partition in the workflow specification to a task in the view. For an edge l in the workflow specification, l is an inter-group edge if $\Psi(l)$ is an edge in the view (i.e., it connects two composite task). Otherwise, l is an intra-group edge.

3. VIEW AND TASK SOUNDNESS PROBLEM

In this section we define sound views, formulate the problems of determining and refining unsound views, and define the view soundness problem and task soundness problem. We then prove that both problems are NP-hard.

3.1 Sound Views

According to Definition 2.2, a view *preserves the edges* among tasks. Specifically, there is an edge from composite task T_i to T_j in the view, if and only if there exist $t_i \in P_i = \Phi^{-1}(T_i)$ and $t_j \in P_j = \Phi^{-1}(T_j)$ in the corresponding workflow specification, such that there is an edge from t_i to t_j. This is a desirable property of view, as such a view preserves the dataflow between adjacent tasks.

Intuitively, we may think that the transitive closure of an edge - a path - in the workflow is also preserved in the view, and thus both immediate (i.e. shallow) provenance and transitive (i.e. deep) provenance are correctly captured in the view. Unfortunately, this is not the case. As discussed in Section 1, a view, which preserves the edges by definition, may not preserve the paths in the workflow specification. In particular, a view may have a path between two tasks which actually does not exist in the workflow specification. Consider the view in Figure 2(b), since M is viewed as a single task, there is a path consisting of edges L_1 and L_5. However, in the workflow specification, there is no path that includes both $l_1 = \Psi^{-1}(L_1)$ and $l_5 = \Psi^{-1}(L_5)$. This view, therefore, does not preserve the paths in the workflow specification. Such a view may convey incorrect dataflow information and thus be misleading and problematic in provenance analysis.

We call a view that preserves the paths in the corresponding workflow specification as a *sound view*, and *unsound view* otherwise, formally defined in the following definition.

Definition 3.1: A view V of a workflow specification W is *sound* if the following conditions hold:

1. $\forall T_i, T_j \in N(V)$, for any path from T_i to T_j consisting of edges $L_1.L_2.\cdots.L_p$, there exists a path in W consisting of edges $\Psi^{-1}(L_1).C_1.\Psi^{-1}(L_2).C_2.\cdots.C_{p-1}.\Psi^{-1}(L_p)$, where $C_m (1 \leq m \leq p-1)$ is a path consisting of intra-group edges only.

2. $\forall t_i, t_j \in N(W)$, for any path from t_i to t_j consisting of edges $l_1.l_2.\cdots.l_p$, there is a path in V consisting of edges $\Psi(l_1).\Psi(l_2).\cdots.\Psi(l_p)$.

■

As we can see, a view is sound if and only if it preserves the paths, that is, if for any path p in the view, there is a corresponding path p' in the workflow specification, and vice versa. A path p in the view corresponds to a path p' in the workflow specification if p' consists of the mapping Ψ of edges in p, with possibly intra-group edges in between. For example, Figure 2(c) is a sound view: for any path in Figure 2(c), e.g., $L_1.L_2.L_4$, there is a path in (a), i.e., $l_1.l_2.l_3.l_4$, such that l_1, l_2 and l_4 maps to L_1, L_2 and L_4, and l_3 is an intra-group edge. Besides, for any path in (a), there is also a corresponding path in (c).

Our focus in this paper is to determine unsound views and refine an unsound view into a sound one, e.g., Figure 2(b) is determined to be an unsound view, and (c), (d) are two possible refinements of

(b). Next we discuss how to determine unsound views, then define a *view soundness problem*, whose goal is to refine an unsound view with minimal cost. We then transform it to an equivalent problem called *task soundness problem*, and prove it to be NP-hard.

3.2 Determining Unsound Views

We first need to check whether a view is sound. However, it is very expensive to do so by directly applying Definition 3.1, as we will need to check if every path in the view satisfies condition 1, and every path in the data satisfies condition 2 in Definition 3.1. Even if we do not count cycles, the number of paths in a workflow specification is $O(2^n)$ in the worst case, where n is the number of tasks. Thus checking view soundness by checking paths is exponential, making it intractable.

Interestingly, we discover that it is sufficient to check the paths between the input and output of each composite tasks to check the view soundness, which can be done in polynomial time. We define the soundness of a task in a view, and show that a view is sound if and only if all its tasks are sound. This property scales the problem down to task-level, such that we can check the soundness of each task to determine the soundness of the view, and later on correct each unsound task individually to resolve an unsound view.

Definition 3.2: Given a composite task T, $T.in$ denotes the atomic tasks in T that receive input from some atomic task $t \notin T$, and $T.out$ denotes the atomic tasks in T that send output to some atomic task $t \notin T$. ■

Definition 3.3: 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 which is composed of nodes in $\Phi^{-1}(T)$ and hence intra-group edges. ■

As an example, the composite task M in Figure 2(b) is unsound because there is no path from $a, e \in M.in$ to $m \in M.out$. All composite tasks in Figure 2 (c) and (d) are sound.

Proposition 3.1: A view V of a workflow specification W is sound (Definition 3.1) if and only if all composite tasks in V are sound.

Proof. $\Rightarrow$: Suppose V is a sound view, but it contains an unsound task T. Since T is unsound, $\exists t_i \in T.in$ and $t_o \in T.out$, such that there is no directed path from t_i to t_o which is composed of intra-cluster edges in T. Assume that in the incoming edge of t_i is l_i, and the outgoing edge of t_o is l_j. Let $L_i = \Phi(l_i)$ and $L_j = \Phi(l_j)$.

In V, there is a path $L_i.L_j$. By Definition 3.1, there should be a path in the workflow specification, such that it consists of l_i, l_j and intra-cluster edges in T. However, there is no such path, which is a contradiction.

$\Leftarrow$: Suppose all the composite tasks in V are sound. Take any path p in V consisting of edges $L_1.L_2.\cdots.L_p$. Let $l_i = \Psi^{-1}(L_i)(1 \leq i \leq p)$, and let the endpoints of l_i be n_{i1} and n_{i2}. Consider a path consisting of nodes $n_{11}, n_{12}, n_{21}, n_{22}, \cdots, n_{p1}, n_{p2}$ in the workflow specification. Each n_{i1} and n_{i2} is connected by edge l_i, and each n_{i2} and $n_{i+1,1}$ must be the same task, or belong to the same composite task. Since each composite task in V is sound, there is a path from n_{i2} to $n_{i+1,1}$ for all i consisting of intra-cluster edges. Therefore, the entire path consisting of edges l_i and intra-cluster edges, which means V satisfies condition 1 of Definition 3.1. Using the same idea, it is easy to prove that V satisfies condition 2 as well. Therefore, V is a sound view. ■

According to Proposition 3.1, we can check whether a view is sound by checking whether each composite task in the view is sound. To check the soundness of a composite task, we simply need to check whether there is a directed path composed of intra-group edges from every input to every output of the task, which can be done efficiently. Therefore we have the following proposition.

Proposition 3.2: Checking whether a view is sound can be done in polynomial time with respect to the number of tasks in the workflow specification.

Proof. The proof is omitted. ■

3.3 Refining Unsound Views

To make a view sound, according to According to Proposition 3.1, we can make each composite task in the view sound. Therefore, we define the notion of view refinement as follows.

Definition 3.4: A view V_1 of workflow W is a *refinement* of another view V_0 of W if and only if $\forall T_1 \in N(V_1)$, $\exists T_0 \in N(V_0)$, such that $\Phi_1^{-1}(T_1) \subseteq \Phi_0^{-1}(T_0)$. The set of tasks $\{T_1 | \Phi_1^{-1}(T_1) \subseteq \Phi_0^{-1}(T_0)\}$, which can be viewed as a split of T_0, is called a *group* and denoted as $S(T_0)$. A refinement/split is *sound* if the resulting view is sound. ■

According to Definition 3.4, a refinement of an unsound view V_0 is to split some composite tasks into smaller tasks to create a finer grained view V_1, which is sound. For example, The unsound view in Figure 2(b) is refined into (c) by splitting task M in (b) into eight smaller tasks shown in dotted rectangles in (c).

Definition 3.5: Given a view V_0, the *cost of splitting* a task T_0 in V_0 into a set of tasks $S(T_0)$ in V_1 is defined as $cost(T_0, S(T_0)) = |S(T_0)| - 1$. The *cost of refining* the view V_0 to another view V_1 is given as $cost(V_0, V_1) = \sum_{T_0 \in N(V_0)} cost(T_0, S(T_0))$. ■

In Figure 2, the cost of refining (b) into (c) is 7, while the cost of refining (b) into (d) is only 4. In fact, the view in (d) is a minimal sound refinement of (b).

Given an unsound view, we would like to refine it with minimal cost, thus we define the following view soundness problem.

Definition 3.6: *View Soundness Problem (VSP):* Given an unsound view V_0, find a sound refinement V_1 such that $cost(V_0, V_1)$ is minimized over all sound refinements of V_0. ■

According to Proposition 3.1, VSP is equivalent to the following task soundness problem (TSP). In the rest of this paper, we focus on the discussion of TSP.

Definition 3.7: *Task Soundness Problem (TSP):* Given an unsound task T_0, find a sound split $S(T_0)$ such that $cost(T_0, S(T_0))$ is the minimal among all the sound splits of T_0. ■

Therefore, to refine a view with minimal cost, we should split each unsound task in the view into the smallest number of sound tasks.

3.4 NP-hardness of Task Soundness Problem

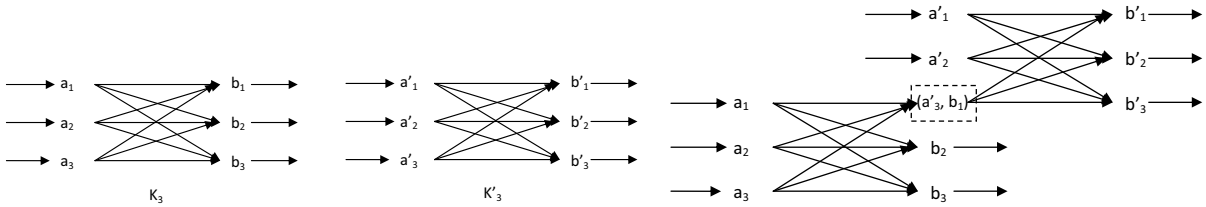
The equivalent decision problem for TSP is defined as: Given a directed graph with M nodes, can we divide it into at most S disjoint subgraphs, such that for each subgraph, all its inputs can reach all its outputs?

We prove the NP-completeness of this problem by reduction from the independent set problem. We first introduce two definitions and a proposition, and then prove the NP-completeness.

Definition 3.8: A complete bipartite task of size p, K_p, is defined as follows:

1. There are two sets of nodes, a_1 - a_p and b_1 - b_p.
2. $a_i \in K_p.in (1 \leq i \leq p)$, $b_i \in K_p.out (1 \leq i \leq p)$.
3. There is an edge from each $a_i (1 \leqslant i \leqslant p)$ to each $b_j (1 \leqslant j \leqslant p)$. ■

For example, K_3 is the composite task shown in Figure 3 (a).



(a) Selecting b_1 and a'_3 (b) Merging b_1 and a'_3

Figure 3: Joining K_3 and K'_3

Definition 3.9: The *join* of two complete bipartite tasks of size p, K_p and K'_p, is defined as follows:

Let the nodes of K_p be a_1 - a_p, b_1 - b_p, and the nodes of K'_p be a'_1 - a'_p, b'_1 - b'_p. Before the join, all nodes in K_p and K'_p are called *open nodes*.

1. Randomly select an open node n_1 in $b_1...b_p$ (or $b'_1...b'_p$), and another open node n_2 in $a'_1...a'_p$ (or $a_1...a_p$).

2. Remove the output of n_1 and the input of n_2, and merge the nodes.

After the join, the merged node (n_1, n_2) is called a *closed node*. ■

For example, consider K_3 and K'_3 in Figure 3(a). If the open nodes b_1 and a'_3 are selected, then the joined task is shown in Figure 3(b). Similarly, we can join multiple complete bipartite tasks by joining every pair of them.

Proposition 3.3: Suppose there a task, consisting of M complete bipartite tasks of size p, $(K_p)_1...(K_p)_M$, several of which are joined. M must be an unsound task. No matter how we refine/split this task, the following statements hold: (1) Each resulting task must be either an entire $(K_p)_i$, or a single node, and (2) If we choose $(K_p)_i$ to be a resulting task, then for each $(K_p)_j$ which is joined with $(K_p)_i$ (i.e., $(K_p)_j$ has a common node with $(K_p)_i$), each task in it must be in a separate resulting task. ■

The idea of proposition 3.3 is that, for a complete bipartite task, either all atomic tasks of it are in a group, or each of its atomic tasks is in a separate group. No other subset of a complete bipartite task can form a sound group, either by itself or with other nodes. In Figure 3(b), for example, nodes $\{a_1, a_2, a_3, b_1, b_2, b_3\}$ either are in the group, or in six separate groups. The formal proof is omitted and can be found in [15].

From Proposition 3.3, a minimal cost refinement of joined bipartite tasks can be obtained. Note that Figure 3(b) is an unsound task. The smallest split is 6: we can either put $\{a_1, a_2, a_3, (a'_3, b_1), b_2, b_3\}$ into one group and the other five nodes into five groups, or put $\{a'_1, a'_2, (a'_3, b_1), b'_1, b'_2, b'_3\}$ into one group and the other five nodes into five groups. Based on this proposition, we have the following proof of NP-completeness of the decision version of TSP.

Theorem 3.4: The decision version of the TSP is NP-complete.

Proof. Consider an arbitrary instance of the independent set problem. Let $G(N, E)$ be an undirected graph. Let $m = |N|$. The independent set problem is: Can we find a subset N' of N, such that $|N'| \geq c$ and there is no edge between any two nodes in $|N'|$?

Now we construct an instance of TSP. For each node n in N, we prepare a complete bipartite task K_m. For every pair of nodes n_i, $n_j \in N$ that have an edge, we join the corresponding bipartite tasks, $(K_m)_i$ and $(K_m)_j$. Now, we get a big composite task T, which is unsound. Suppose T has M nodes. The following question is asked: Can we split T into at most $(c + M - 2cm)$ groups, such that every group is sound? It is easy to see that this transformation takes polynomial time.

Given a solution N' of the independent set problem, suppose N' has c' nodes ($c' \geq c$), $n_1...n_{c'}$. Then in the corresponding TSP, we let each of $(K_m)_1$, $(K_m)_2...(K_m)_{c'}$ form a group. Since no two of them are joined, this is legal. Each other node has to be in one group alone. The total number of groups is thus $c' + M - 2c'm \leq c + M - 2cm$.

Now the other direction. Suppose there is a solution for TSP. Note that according to Proposition 3.3, each group either contains a K_m or contains a single node. Therefore, if the number of groups is no more than $c + M - 2cm$, then it is easy to see that there must be at least c such K_ms, each of which forms a group. This means no two of such K_ms are joined. Then, we can get a solution N' to the independent set problem, which consists of the corresponding nodes in N. ■

4. ALGORITHMS

In this section, we first discuss how to discover unsound tasks in a workflow view efficiently, then propose algorithms for splitting an unsound composite task into a set of groups, each of which is a sound task.

We proved in Section 2 that the task soundness problem is NP-hard. Therefore, instead of aiming for the optimal solution, we propose two other criteria which, as will be shown later, can be achieved in polynomial time: *weak local optimality* and *strong local optimality*. An algorithm that satisfies weak or strong local optimality does not necessarily produce the optimal solution for an unsound task, but one that is "good" in a local sense. We show that weak local optimality is easy to achieve with a straightforward algorithm (Section 4.2). However, achieving strong local optimality is much more challenging. We present in Section 4.3 an algorithm which is strong local optimal, which proved in Section 4.4. Both algorithms are illustrated by the same running example as that in Section 2.

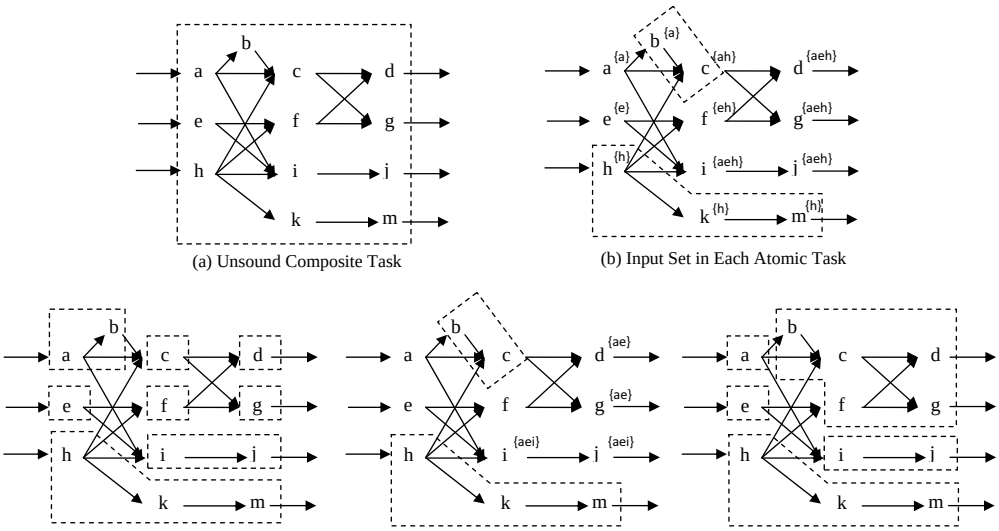
4.1 Detecting Unsound Tasks in a Workflow View

We begin by discussing how to determine whether a composite task is sound. First, we introduce the notion of an *input set*.

Definition 4.1: In a composite task T, the *input set* of a task $t \in N(T)$ with respect to T, denoted as $t.inSet(T)$, is a set of nodes $N \subseteq T.in$, such that $\forall n \in N$, n can reach t through directed edges. ■

For example, for the composite task T in Figure 4(a), the $inSet(T)$ of each atomic task is annotated in Figure 4(b). According to Definition 4.1, a task T is sound if $\forall t \in T.out$, $t.inSet(T) = T.in$.

The algorithm for detecting unsound tasks is presented in Algo-



(a) Unsound Composite Task

(b) Input Set in Each Atomic Task

(c) Result of (a) in Algorithm 2

(d) Re-split Unsound Cluster

(e) Result of (a) in Algorithm 3

Figure 4: Running Example

Algorithm 1 Unsound Task Detection Algorithm

DetectAndCorrect (V)

1: **for** each $T \in N(V)$ **do**
2: *CalcInset*(T)
3: **if** $!IsSound(T)$ **then**
4: $/ * The\ detaield\ implementation\ of\ SPLIT()$
5: $will\ be\ presented\ in\ the\ following\ sections. * /$
6: $SPLIT(T)$
7: **end if**
8: **end for**

CalcInset (T)

1: **for** each $t \in T.out$ **do**
2: go backwards along dataflow from t to traverse each task t'
3: **if** $t' \in T.in$ **then**
4: $t.inSet(T) = t.inSet(T) \cup \{t'\}$
5: **end if**
6: **end for**

IsSound (T)

1: **for** each $t \in T.out$ **do**
2: **if** $t.inset(T) \neq T.in$ **then**
3: return $false$
4: **end if**
5: **end for**
6: return *true*

rithm 1. For each composite task T in a view V, procedure *DetectAndCorrect* calls *CalcInset*(V) to compute $inset(T)$ for each task in $T.out$, which can be obtained by traversing T. Then it calls $IsSound(T)$ to check the soundness of T, and if T is unsound, it calls $SPLIT(T)$ to split it (for which it can use Algorithm 2 for weak local optimality or Algorithm 3 for strong local optimality).

For the composite task in Figure 4(a), we determine that it is unsound because $m.inset(T) = \{h\}$ while $j.inset(T) = \{a, e, h\}$.

For each task $t \in T.out$, Algorithm 1 finds $t.inset(T)$ by traversing the task starting from t, which takes $O(n^2)$ time, where n is the number of atomic tasks in T. Therefore, if T has m output tasks, then the complexity of Algorithm 1 is $O(mn^2)$.

4.2 Weakly Local Optimal Algorithm

In this subsection we introduce the *weak local optimality* criterion for judging the quality of correcting task soundness problem. Then a polynomial time algorithm satisfying weak local optimality is presented with examples.

Definition 4.2: 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 we say $\asymp(\mathcal{T})$. ■

Definition 4.3: 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 $S_i, S_j \in \mathcal{S}$, $S_i \asymp S_j$. An algorithm for the task soundness problem is called a *weakly local optimal algorithm* if for any unsound task T, it guarantees to produce a split which is weak local optimal. ■

In short, weak local optimality indicates that any two groups in the split are not combinable. Weak local optimality can be achieved by a straightforward algorithm which works as follows:

1. Split the given unsound task, such that each group contains one atomic task.
2. Test whether any two groups are combinable. If so, merge them.
3. Repeat 2) until no two groups are combinable.

The pseudo code of this algorithm is shown in Algorithm 2. In our running example, Figure 4(a), during the first iteration we find that $a \asymp b$, $i \asymp j$ and $h \asymp k$, and we merge these three pairs. During the next iteration, we find $(h, k) \asymp m$, and put h, k, m in one group. During the third iteration, no two groups are found to be combinable, and therefore we stop and output the current split. The result of splitting the task in Figure 4(a) using Algorithm 2 is shown in Figure 4(c).

Algorithm 2 obviously gives a split of an unsound task which is guaranteed to be weak local optimal. The formal proof is omitted.

Now we analyze the time complexity of Algorithm 2. Let n denote the number of atomic tasks in T, thus we initially have n groups. Each time we execute line 8 to merge two groups, the total number of groups decreases by 1, thus line 8 is executed at most n times. During each "while" loop, suppose there are currently p

Algorithm 2 Weakly Local Optimal Algorithm

SPLIT (T)

1: break T into pieces, i.e., each single task forms a group
2: *changed* = true
3: **while** *changed* = true **do**
4: *changed* = false
5: **for** every pair of new groups (a, b) and every pair of one old group and one new group (a, b) **do**
6: {Initially, all groups are new groups. From the 2nd iteration, a group is a new group if it is merged from two groups in the last iteration.}
7: **if** $CanMerge(a, b)$ **then**
8: merge a and b
9: *changed* = true
10: **end if**
11: **end for**
12: **end while**

CanMerge (A, B)

1: $A2B = B2A = Ain = Aout = Bin = Bout$ = false
2: **if** A has an outgoing edge to B **then**
3: $A2B$ = true
4: **end if**
5: **if** A has an incoming edge from B **then**
6: $B2A$ = true
7: **end if**
8: **if** A has more outgoing edges other than the one to B **then**
9: $Aout$ = true
10: **end if**
11: **if** A has more incoming edges other than the one from B **then**
12: Ain = true
13: **end if**
14: **if** B has more outgoing edges other than the one to A **then**
15: $Bout$ = true
16: **end if**
17: **if** B has more incoming edges other than the one from A **then**
18: Bin = true
19: **end if**
20: **if** $A2B$=true AND $Aout$=true AND Bin=true **then**
21: return false
22: **end if**
23: **if** $B2A$=true AND $Bout$=true AND Ain=true **then**
24: return false
25: **end if**
26: return true

groups ($p \leq n$), and line 8 is executed q times to merge $2q$ groups into q groups ($2q \leq p$). Then in the next iteration of the "while" loop, we have $p - 2q$ old groups and q new groups, thus the "for" loop will be executed $q^2 + q(p - 2q) < pq \leq nq$ times. This means that, each time line 8 is executed in the current "while" loop, it causes procedure *CanMerge* (line 7) to execute at most n times in the next iteration of the "while" look. In addition, *CanMerge* is executed n^2 times in the first iteration of the "while" loop. Therefore, *CanMerge* is executed $O(n^2)$ times altogether. *CanMerge* requires $O(1)$ time to run with the help of a hash table, which can be pre-built in $O(n^2)$ time. Therefore, the total time complexity of Algorithm 2 is $O(n^2)$.

4.3 Strongly Local Optimal Algorithm

Running Algorithm 2, which is weak local optimal, on the task in Figure 4(a) produces the split shown in Figure 4(c). As we can see, tasks c, d, f and g in Figure 4(c) are combinable. However, since no two of them are combinable, Algorithm 2 fails to put them in a single group. In this subsection, we propose and discuss a stronger optimization goal, namely *strong local optimality*, for achieving better splits of unsound tasks.

Definition 4.4: 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 $\mathcal{S}' \subset \mathcal{S}$, $\asymp(\mathcal{S}')$. An algorithm for the task soundness problem is called a *strong local optimal algorithm* if for any unsound task T, it guarantees to produce a split which is strong local optimal. ■

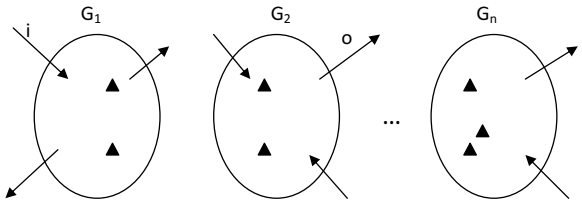


Figure 5: Strongly Connected Component

According to Definition 4.3 and 4.4, Figure 4(c) is a weak local optimal split.

Achieving strong local optimality is much more challenging than achieving weak local optimality. A straightforward way of achieving strong local optimality is to check whether any subset of atomic tasks are combinable. Since the number of subsets are exponential in the number of atomic tasks, this takes exponential time. Before presenting a clever algorithm that achieves strong local optimality in polynomial time, we introduce a lemma and a definition as necessary background.

Lemma 4.1: Given an unsound task T, if there is a strongly connected component $S \subseteq T$, then in any strong local optimal split of T (hence any optimal split of T as well), all tasks in S must belong to the same group.

Proof. In Figure 5, an unsound task is split into some number groups. The triangle nodes comprise a strongly connected component, which is not placed in one group, but instead in n groups $G_1, G_2, ..., G_n$. Assume that the lemma is not true, and this split is strong local optimal.

Since the triangle nodes are placed in multiple groups, in any such group, e.g., G_1, at least one triangle node, e.g., a, belongs to $G_1.out$, as it needs to connect to triangle nodes in another group. For the same reason, in any other such group, e.g., G_2, at least one triangle node b belongs to $G_2.in$. This indicates that $\asymp(G_1, G_2, ..., G_n)$. The reason is that for any input i of G_1 and any output o of another group G_2, i can reach a through edges in G_1 (because G_1 is sound), a can reach b through edges among the triangle nodes, and b can reach o through edges in G_2. This means that each input of $\bigcup(G_1, G_2, ..., G_n)$ can reach each of its output through edges in it, and thus $\asymp\{G_1, G_2, ..., G_n\}$, which contradicts with the assumption that the split is strong local optimal. ■

Lemma 4.1 indicates that if an unsound task has a strong connected component within it, the nodes in the strongly connected component can be directly merged into one group. In the rest of this section, we focus on discussion on unsound task which does not have strongly connected components.

Definition 4.5: In a composite task T, the *complete predecessor set* of a set of nodes $U \subseteq T$, denoted as $CPS(U)$, is a set of nodes $P \subseteq T$, such that for every task $p \in P$, each of p's output edges points to a task in U. The *complete predecessor closure* of U, denoted as $CPC(U)$, is equal to $U \cup CPS(U) \cup CPS(CPS(U)) \cup CPS(CPS(CPS(U))) \cup ....$ ■

For example, in Figure 4(a), let $U = \{d, g, i, j\}$. Then $CPS(U) = \{c, f\}$. $h \notin CPS(U)$ because h has an output edge $(h \rightarrow k)$ that does not point to any task in U. $CPC(U) = \{a, b, c, d, e, f, g, i, j\}$.

The concept of *CPC* is crucial for achieving strong local opti-

mality in polynomial time. We will show in Section 4.4 that given a certain set of nodes S, a sound group whose output nodes are exactly those in S must be a subset of $CPC(S)$. Besides, if such a sound group exists, then $CPC(S)$ is sound. Therefore, as long as we can efficiently find the correct set of nodes S which can serve as the output node set for a sound group, we can find the sound group from S easily, by simply computing $CPC(S)$.

Fortunately, finding such S sets is achievable as well. We will prove later that if there exists a sound group S' within a composite task T such that $S'.out = S$, then all nodes in S have the same $inSet(T)$. Therefore, we can cluster the nodes in T according to $inSet(T)$, and iteratively reduce the scope of T until S is found, such that $CPC(S)$ is sound.

Now we present an algorithm which guarantees to produce a strong local optimal split for any unsound task in polynomial time. The pseudo code is shown in Algorithm 3.

Similar as Algorithm 2, Algorithm 3 first puts each atomic task in a unique group (line 3-5 of procedure $SPLIT$). In each "while" loop, it computes $t.inSet(T)$ for each node $t \in T$ (line 9), then cluster the groups in T into $C = \{C_1, C_2, ..., C_c\}$ (line 10). Groups with the same $inSet$ are placed in the same cluster.

In our running example, Figure 4(a), the $inSet(T)$ of each node is annotated using input set near the node in Figure 4(b). There are six clusters: $C_1 = \{a, b\}$, $C_2 = \{e\}$, $C_3 = \{h, k, m\}$, $C_4 = \{c\}$, $C_5 = \{f\}$, $C_6 = \{d, g, i, j\}$.

Then, Algorithm 3 finds the complete predecessor closure (CPC) for each $C_i \in C$. In our running example, the CPC of $\{d, g, i, j\}$ is $\{a, b, c, d, e, f, g, i, j\}$; the CPC of $\{c\}$ is $\{b, c\}$, and the CPC of each other cluster is itself, each of which is a potential sound task. Then the algorithm checks each $C_i \in C$ to see whether $CPC(C_i)$ is sound (line 3 of procedure $FindSoundCluster$). If so, we merge the groups in $CPC(C_i)$, and then repeat the "while" loop in $SPLIT$. Otherwise, we further cluster C_i into a set of clusters, and recursively do so until the CPC of a cluster is sound, or an unsound CPC containing at least two nodes exists.

Continue our running example. There are four CPCs that contains more than one group. $\{a, b\}$, $\{b, c\}$ and $\{h, k, m\}$ are sound, but $\{a, b, c, d, e, f, g, i, j\}$ is not sound, as the input from h to i cannot reach the output from d and g through a directed path within it. In the first two "while" loops, we merge $\{b, c\}$ and $\{h, k, m\}$. In the third "while" loop, there is only one CPC that contains more than one group, which is $CPC(\{d, g, i, j\}) = \{a, \{b, c\}, d, e, f, g, i, j\}$. Therefore, we calculate the $inSet$ of $\{d, g, i, j\}$ with respect to these eight nodes. As shown in Figure 4(d), we group $\{d, g, i, j\}$ into two clusters, $\{d, g\}$ and $\{i, j\}$ in terms of their newly generated input sets. Now we find that $CPC(\{d, g\}) = \{\{b, c\}, d, f, g\}$, which is sound. Therefore, we merge $\{b, c\}$, d, f and g into one group. The procedure is continued until the result is produced as shown in Figure 4(e).

Now we analyze the time complexity of Algorithm 3. Each "while" loop in procedure $SPLIT$ merges at least two groups into one, therefore the "while" loop is executed at most n times, where n is the number of atomic tasks in T. During each "while" loop, to compute $inset(T)$ of the nodes $t \in T$ (line 9), we use a precalculated reachability matrix of the graph (which can be built in $O(n^2)$ time). Therefore line 9-10 takes $O(n)$ time. The "for" loop in line 11-20 recursively processes each cluster $C_j \in C$. If $CPC(C_j)$ is sound and contains more than one groups, then we merge them and enter the next "while" loop. The soundness of $CPC(C_j)$ can be checked in $O(|C_j|$ time, where $|C_j|$ is the number of groups in C_j. Since $\sum_{C_j \in C} |C_j| \leq n$, checking the soundness of all such $CPC(C_j)$ takes $O(n)$ time. Those C_js whose CPC is not sound are then recursively separated into several clusters (line 6-7 of $FindSoundCluster$); checking the soundness of all their CPCs takes $O(n)$ time for the same reason. Since the recursive procedure $FindSoundCluster$ can at most go to depth n which means there are at most n rounds of clustering, the "for" loop in $SPLIT$ takes $O(n^2)$ time. Therefore, the total complexity of Algorithm 3 is $O(n^3)$.

Algorithm 3 Strongly Local Optimal Algorithm

SPLIT (T)

1: {Initially, each atomic task in T is represented by a node. Each atomic task is in a separate group}
2: *changed* = true
3: *group* = $\emptyset$
4: **for** each atomic task $t \in T$ **do**
5: *group* = *group* $\cup \{t\}$
6: **end for**
7: **while** *changed* = true **do**
8: *changed* = false
9: Compute $t.inSet(T)$ for each node $t \in T$
10: Cluster the groups in T according to their $inSet$ into $C = \{C_1, C_2, ..., C_c\}$. Nodes with the same $inSet$ are placed in the same cluster.
11: **for** each $C_j \in C$ **do**
12: **if** (*soundCluster* = $FindSoundCluster(C_j)$) $\neq$ NULL **then**
13: *changed* = true
14: **for** each group $g \in$ *soundCluster* **do**
15: *group* = *group* $- \{g\}$
16: **end for**
17: *group* = *group* $\cup$ *soundCluster*
18: break
19: **end if**
20: **end for**
21: **end while**

FindSoundCluster (*cluster*)

1: **if** CPC(*cluster*) has only 1 node **then**
2: return NULL
3: **else if** $isSound(CPC(cluster))$ **then**
4: return CPC(*cluster*)
5: **end if**
6: Compute $t.inSet(CPC(cluster))$ for each node $t \in$ *cluster*
7: Cluster the nodes in *cluster* according to their $inSet$ into $C = \{C_1, C_2, ..., C_c\}$. Nodes with the same $inSet$ are placed in the same cluster.
8: **for** each $C_j \in C$ **do**
9: **if** (*soundCluster* = $FindSoundCluster(C_j)$) $\neq$ NULL **then**
10: return *soundCluster*
11: **end if**
12: **end for**
13: return NULL

4.4 Proof of Strong Local Optimality

To prove that Algorithm 3 is strong local optimal, we start with some lemmas.

Lemma 4.2: In an unsound composite task T, for any set of tasks $S \subseteq T$ such that S is sound, $\forall t_1, t_2 \in S.out$, $t_1.inSet(T) = t_2.inSet(T)$.

Proof. Since S is sound, $\forall t \in S.out$, $t.inSet(T) = \bigcup_{t' \in S.in} t'.inSet(T)$. Therefore, any two nodes in $S.out$ has the same $inSet$ with respect to T. ∎

Lemma 4.2 indicates that, after we cluster tasks in T into $C = C_1, C_2, ..., C_c$ in line 10 of procedure $SPLIT$, for any sound composite task $S \subseteq T$, $S.out$ must be a subset of a cluster C_i ($1 \leq i \leq c$).

Lemma 4.3: In an unsound composite task T, for any set of tasks $S' \subseteq T$, if there exists $S \subseteq T$ such that S is sound and $S.out = S'$, then $S \subseteq CPC(S')$.

Proof. Suppose there is an $S \subseteq T$, S is sound, $S.out = S'$ but there is a node $s \in S$ such that $s \notin CPC(S')$. Then according

to the definition of CPC, s must have an output edge that goes to somewhere else other than the triangle nodes. By our assumption, every task has at least one input and one output edge, which means that there must be another output of S which does not belong to S'. This contradicts with the condition. ∎

Lemma 4.4: In an unsound composite task T, $\forall S' \subseteq T$, if $CPC(S')$ is not sound, then there does not exist $S \subseteq T$, such that S is sound and $S.out = S'$.

Proof. Suppose there exists an S such that S is sound, $S.out = S'$ and $S \neq CPC(S')$. According to Lemma 4.3, $S \subset CPC(S')$. Then for any node $s \in CPS(S)$, $\{s\} \asymp S$. The reason is that since $s \in CPS(S)$, each of s's output edges serves as an input edge of S, as shown in Figure 6. In addition, because $S' \subset S$ and $S \subset CPC(S')$, therefore, $CPC(S) = CPC(S')$, which means all nodes in $CPC(S')$ can be added into S. Thus, $CPC(S')$ is sound, which contradicts with the condition. ∎

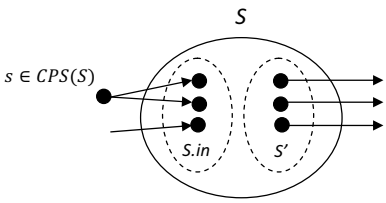


Figure 6: Lemma 4.4

Lemma 4.4 indicates that given a set of nodes S', if we want to find out whether there exists a sound task S whose output node set is exactly S' (i.e., $S.out = S'$), we can simply see if the complete predecessor closure of S' is sound. Besides, according to Lemma 4.3, if $CPC(S')$ is sound, then it is the largest sound composite task whose output node set is exactly S'.

Now we present the proof of strong local optimality of our algorithm.

Theorem 4.5: Algorithm 3 is strong local optimal.

Proof. To show that the algorithm is strong local optimal, we only need to prove that, during any "while" loop between line 7 and 21 of $SPLIT$, if the current split is not strong local optimal, then the algorithm guarantees to find a set of groups that can be merged. If this is true, then it means that the number of groups decreases at each iteration, and the iteration stops when the split reaches strong local optimality.

Suppose that in a "while" loop, the current split is T. Each group is represented by a node in T, and is considered as an atomic task. Suppose there are several sets of atomic tasks that can be merged. Then our algorithm will find one of them, say S, in the following way:

1. Line 10 of $SPLIT$ splits T into $C = C_1, C_2, ..., C_c$. According to Lemma 4.2, $\exists C_j \in C$, $S.out \subseteq C_j$.

2. If $S.out = C_j$, then according to Lemma 4.4, $CPC(C_j)$ is sound. Therefore, when we visit $CPC(C_j)$ during the "for" loop between line 11 and 20, a sound group is found.

3. Otherwise, since $S.out \subseteq C_j$, $CPC(S.out) \subseteq CPC(C_j)$. Line 7 of procedure $FindSoundCluster$ further cluster tasks in C_j into $C' = C'_1, C'_2, ..., C'_{c'}$ based on the $inSet$ of each node in C_j with respect to $CPC(C_j)$. According to Lemma 4.2, $\exists C'_{j'} \in C'$, $S.out \subseteq C_{j'}$.

4. If $S.out = C_{j'}$, then according to Lemma 4.4, $CPC(C_{j'})$ is sound. Therefore, a sound group is found. Otherwise, procedure $FindSoundCluster$ recursively cluster tasks in $C_{j'}$ into a set of groups. The procedure continues in such a way. We can observe that:

   - At each clustering (line 10 in $SPLIT$ and line 7 in $FindSoundCluster$), since the set of tasks being clustered is not sound, we at least produce two clusters. This means that the algorithm will terminate, as the clusters become smaller and smaller.
   - At each clustering, according to Lemma 4.3, we guarantee to produce a cluster whose CPC is a superset of S, and this cluster becomes smaller each time (as the nodes in it are further clustered into at least two clusters each time). So we guarantee to find S eventually.

Therefore, this algorithm will always terminate, and if in the current "while" loop, we find such an S, we merge the nodes in it, treat it as a single task and then enter the next "while" loop. Otherwise, the current split is strong local optimal and we output it. ∎

5. EVALUATION

In our experimental evaluations, we begin by surveying workflow views appearing in the real world (Section 5.1). To verify the effectiveness (Section 5.2) and efficiency (Section 5.3) of our algorithms, we then test them from two perspectives: the number of tasks produced when correcting an unsound view, and the time it takes to process an unsound view.

The experiments were performed on a laptop with Intel Core(TM) 2 CUP 1.66GHZ, 2GB memory, running Windows XP.

5.1 Existence of Unsound Views

To demonstrate the applicability of our approach, we use workflows in the Kepler repository [11]. Kepler is a popular scientific workflow management system, and has a repository of well curated workflows with composite tasks specified by users. After checking the workflow repository in Kepler, we found that 7 out of 170 (4%) workflows contained at least one unsound task. This indicates that sound views are generally desirable in practice, but unsound views exist even in a well-curated workflow repository.

Workflow view construction tools, such as Zoom [9], are designed to help users construct views of workflows. Zoom takes as input a set of user specified relevant tasks, and generates a view that is driven by the user's interest. Here, we choose ten workflows from Kepler repository; for each workflow, we randomly specify 2 sets of relevant tasks, whose sizes are no more than 10% of the workflow size. We then use Zoom to construct a view for it, and count the number of unsound tasks among all the composite tasks created by Zoom.

As shown in Figure 7, only 2 out of 20 of views generated by Zoom are sound. In addition, the percentage of unsound tasks within an unsound view varies from 4% to 28.6%, which is a measurement of the view quality. We also observe there is no obvious relationship between the view quality and the size of the workflow.

This set of experiments suggests that although sound views are generally desirable, unsound views are often created.

5.2 Quality of Unsound View Correction

Now we test the effectiveness and efficiency of our algorithms on synthetic workflows. Different parameters of synthetic workflows are used to test a variety of cases and verify the effectiveness of our approach.

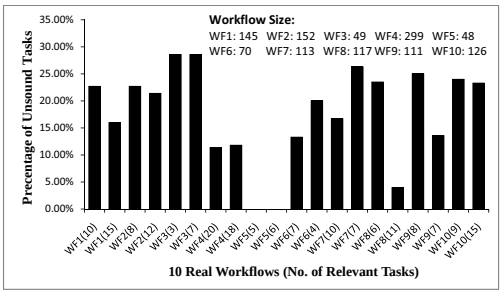


Figure 7: The Percentage of Unsound Views Constructed by Zoom

Table 1: Synthesized Workflows with Different Sizes

Set #	Input Size	Output Size	Workflow Size	Task Size
Set 1	3±1	3±1	10±1	8-11
Set 2	20±2	20±2	100±10	90±10
Set 3	50±2	50±2	200±10	190±10
Set 4	80±5	80±5	300±10	290±10
Set 5	110±5	110±5	400±10	390±10
Set 6	140±10	140±10	500±10	490±10
Set 7	170±10	170±10	600±10	590±10

Table 2: Synthesized Workflows with Different Number of Inputs/Outputs

Set #	Input Size	Output Size	Workflow Size	Task Size
Set 8	10±2	10±2	300±10	290±10
Set 9	20±2	20±2	300±10	290±10
Set 10	30±2	30±2	300±10	290±10
Set 11	40±2	40±2	300±10	290±10
Set 12	50±2	50±2	300±10	290±10
Set 13	60±2	60±2	300±10	290±10

To generate synthetic workflows, we take commonly used workflow patterns[1] (e.g., parallel, sequential, etc.) and randomly combine them, varying two parameters: the number of atomic tasks in the workflow, and the number of inputs/outputs. To combine two workflow patterns, we randomly connect the inputs of one pattern to the outputs of the other. To test the workflow size parameter, we repeatedly combine workflow patterns until the desired size is reached. Seven sets of workflows are generated based on their sizes, as shown in Table 1. Each class has 50 workflows. To test the parameter of the number of inputs/outputs, six sets of workflows are generated, each containing 50 workflows with 300 ± 10 nodes. The numbers of inputs/outputs in the six sets vary from 10 ± 2 to 60 ± 2, as shown in Table 2.

For each synthetic workflow, we create a view by randomly choosing some atomic tasks to create a composite task. The size of the composite task is shown in column "Task Size" in Table 1 and 2. We let each view have only one composite task, as multiple composite tasks in a view are treated independently by our algorithms, and the complexity to refine a view is dominated by the complexity of splitting the largest unsound task.

Since the processing time of the optimal algorithm grows exponentially with the size of the composite task, and becomes unacceptable when the workflow size is only 12 (it can take several hours), we only test the optimal algorithm on workflows in Set 1 of Table 1. The quality of the algorithms is measured as the size of the refined view.

[1]http://www.workflowpatterns.com/

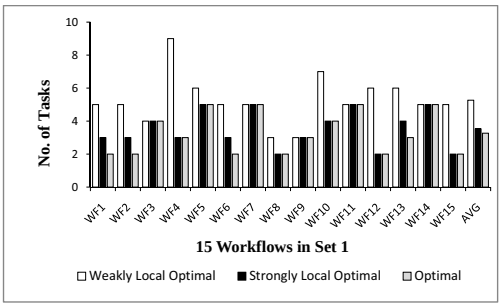


Figure 8: Quality of Weakly, Strongly Local Optimal and Optimized Algorithms

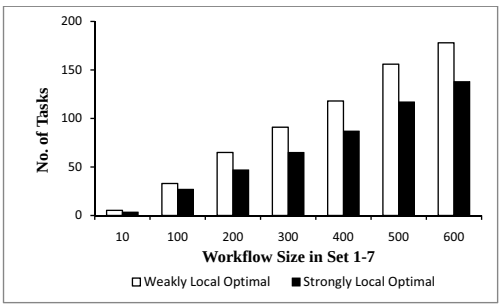


Figure 9: Quality of Weakly, Strongly Local Optimal Algorithms w.r.t Workflow Size

Figure 8 shows the quality of the three algorithms on views of the 15 randomly selected workflows in set 1 of Table 1. As we can see, the number of tasks in the view refined by the strongly local optimal algorithm is the same as that of the optimal algorithm for many views, and is similar to the optimal algorithm for the others. On the other hand, the weakly local optimal algorithm may have many more tasks in the refined view. The last column also shows the average quality of the three algorithms on all the workflows in set 1. It is clear that the optimal algorithm has the best quality; the strongly local optimal gives slightly worse results, and the weakly local optimal generates the worst results.

Figure 9 shows the average number of tasks after the split using the weakly and strongly local optimal algorithms on workflows in sets 1-7 in Table 1. As we can see, the quality of both the weakly and strongly local optimal algorithms increases slowly. Moreover, the strongly local optimal algorithm consistently produces fewer tasks than the weakly local optimal one.

Figure 10 shows that the number of tasks in the refined view produced by the two algorithms grows with increased input/output size. The reason is that for a composite task T, a larger input size means a larger possibility that an output does not depend on an input, and thus T is likely to be split into more groups.

From the quality tests, it is clear that the strongly local optimal algorithm produces better quality refinements (roughly 15% better) than the weakly local optimal one for both synthetic and real workflows. In addition, the quality of the strongly local optimal algorithm is quite close to the quality of the optimal algorithm.

5.3 Processing Time

The average processing time in set 1 of Table 1 for the weakly and strongly local optimal algorithms and for the optimal algorithm are 389 milliseconds, 165 milliseconds and 352×10^3 milliseconds, respectively. Clearly, the processing time of both the weakly and

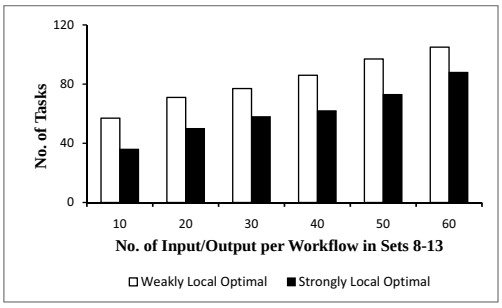


Figure 10: Quality of Weakly, Strongly Local Optimal Algorithms w.r.t Input(Output) Size

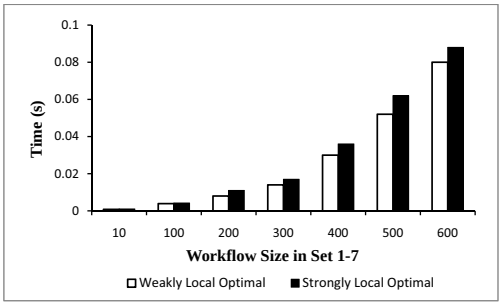


Figure 11: Efficiency of Weakly, Strongly Local Optimal Algorithms w.r.t Workflow Size

strongly local optimal algorithms are dramatically better than that of the optimal algorithm even for small workflows. Note that the strongly local optimal algorithm is faster than the weakly local optimal one for this set of workflows; the reason is that when a workflow is small, it is likely that each complete predecessor closure (CPC) is already sound, and does not need to be recursively split by the function *FindSoundCluster* in Algorithm 3. However, the weakly local optimal algorithm still needs to compare every pair of tasks.

Figure 11 shows the processing time of workflows in sets 1-7 in Table 1. The processing time of both the weakly and strongly local optimal algorithms increases with the size of the workflow. Although the strongly local optimal algorithm takes more time than the weakly local algorithm in each class, when the size of the workflow is less than 600 the extra processing time is no more than 0.01 second. This indicates that the strongly local optimal algorithm provides a more practical solution with good quality improvements and little processing overhead compared with the weakly local optimal one.

To summarize, the strongly local optimal algorithm is the best overall choice regardless of workflow size. It produces views with similar quality as the ones generated by the optimal algorithm with much better efficiency, which is comparable with the efficiency of the weakly local optimal algorithm.

6. RELATED WORK

Provenance on workflows have been much studied in recent works [2, 1, 3, 4, 5]. As workflows become large and complex, the cost of answering transitive closure queries in overwhelming provenance information becomes unacceptable. Therefore, a number of techniques [8, 7, 9] have been proposed to reduce the complexity of provenance graph and improve provenance calculation efficiency. One way to simplify the provenance graph of a workflow is to create views in workflow management system, and the provenance information can then be shown to users w.r.t the specified views.

Many workflow systems allow a user to manually create composite tasks which constitute views. For instance, Kepler [11] and Taverna/myGrid [12] allow users to specify a composite task/view through a graphical interface or using files. [16, 17, 18] allow users to specify views using their proposed view definition languages.

There are also systems [13, 14, 9, 19, 20, 21, 22] that construct views automatically based on user requirements. The *Zoom* system [19, 9, 20] constructs views for focusing user attention on user-specified "relevant tasks" on a workflow, such that (1) a composite task either contains one relevant task (relevant composite task) or none (non-relevant composite task), (2) there is a path between relevant composite tasks in the view if and only if there is path between their contained relevant tasks in the workflow, (3) the number of non-relevant composite tasks are minimized. [13, 21, 14, 22] construct "order-preserving" views for easing execution order analysis on workflows. Specifically, if there is an *edge* between two composite tasks T_i and T_j in a view, then for *any* atomic task t_m in T_i and t_n in T_j, there is a path from t_m to t_n. In contrast, in this paper we propose the definition of *soundness* of a view with respect to provenance analysis. None of existing work can guarantee that the generated views are sound. More comparison of these views can be found in [15].

The term "soundness" of workflow has been used in existing works to evaluate the execution of workflow: a workflow execution is sound if and only if every process will be terminated and there are no dangling references or dead tasks [23, 24, 25, 26, 27, 28]. The notion of soundness in this paper is different from the previous works: it refers to correct provenance analysis at the view level.

Another related field is *Program Slicing* [29], which is a program analysis technique for debugging and understanding programs. It attempts to identify a set of program points whose execution contributes to the value of a variable. Both program slicing and data provenance analyze data dependency [30, 31, 32]. Recent papers [31, 32] identifies the problem of "incorrect dependency provenance" on programs, which shares the same spirit of the "unsound view" problem on workflows. No existing works, however, provide a solution to resolve incorrect dependency provenance in programs.

7. CONCLUSION AND FUTURE WORK

Unsound views, which do not preserve all the paths in the workflow specification, can be misleading and lead to incorrect provenance analysis. In this paper, we formalize and study the problem of identifying and correcting unsound views. A view is sound if and only if it preserves all the data dependencies between any two tasks in the workflow specification. When we identify an unsound view, we generate a sound refinement of the view by splitting each unsound composite task into a minimal set of sound tasks. Consequently, we define the view soundness problem and an equivalent task soundness problem, and prove that they are NP-hard. In order to provide practical solutions, we introduce two criteria: weak local optimality and strong local optimality. Then we design polynomial time algorithms for correcting workflow views with unsound composite tasks that satisfy these criteria. Empirical evaluations show that they efficiently generate sound view refinements with good quality. Furthermore, the strongly local optimal algorithm produces better solutions with small processing overhead compared with the weakly local optimal algorithm.

In the future, we will investigate approximation or randomized algorithms for this problem, and study how to correct unsound

views with minimal cost if tasks are allowed to be either split or merged.

8. REFERENCES

[1] S. B. Davidson and J. Freire, "Provenance and scientific workflows: challenges and opportunities," in *SIGMOD Conference*, 2008, pp. 1345–1350.

[2] S. M. S. da Cruz, P. M. Barros, P. M. Bisch, M. L. M. Campos, and M. Mattoso, "Provenance services for distributed workflows," in *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 526–533.

[3] S. Rajbhandari, O. F. Rana, and I. Wootten, "A fuzzy model for calculating workflow trust using provenance data," in *MG '08: Proceedings of the 15th ACM Mardi Gras conference*. New York, NY, USA: ACM, 2008, pp. 1–8.

[4] Y. L. Simmhan, B. Plale, and D. Gannon, "A framework for collecting provenance in data-centric scientific workflows," in *ICWS '06: Proceedings of the IEEE International Conference on Web Services*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 427–436.

[5] S. Bowers, T. M. McPhillips, and B. Ludäscher, "Provenance in collection-oriented scientific workflows," *Concurr. Comput. : Pract. Exper.*, vol. 20, no. 5, pp. 519–529, 2008.

[6] "Open provenance model," 2008. [Online]. Available: http://twiki.ipaw.info/bin/view/Challenge/OPM

[7] T. Heinis and G. Alonso, "Efficient lineage tracking for scientific workflows," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2008, pp. 1007–1018.

[8] A. P. Chapman, H. V. Jagadish, and P. Ramanan, "Efficient provenance storage," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2008, pp. 993–1006.

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

[10] myexperiment website, "http://www.myexperiment.org/workflows."

[11] 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*, 2004, pp. 423–424.

[12] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li, "Taverna: a tool for the composition and enactment of bioinformatics workflows," in *Bioinformatics*, 2003, pp. 3045–3054.

[13] D.-R. Liu and M. Shen, "Workflow modeling for virtual processes: an order-preserving process-view approach," *Inf. Syst.*, vol. 28, no. 6, pp. 505–532, 2003.

[14] M. R. Ralph Bobrik and T. Bauer, "View-based process visualization," in *Business Process Management*. Springer Berlin / Heidelberg, 2007, pp. 88–95.

[15] "Detecting and Resolving Unsound Workflow Views for Efficient Provenance AnalysisTechnical Report," 2008.

[16] R. Eshuis and P. Grefen, "Constructing customized process views," *Data Knowl. Eng.*, vol. 64, no. 2, pp. 419–438, 2008.

[17] Z. Shan, D. K. W. Chiu, and Q. Li, "Systematic interaction management in a workflow view based business-to-business process engine," in *HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 7*. Washington, DC, USA: IEEE Computer Society, 2005, p. 162.2.

[18] D. K. W. Chiu, S. C. Cheung, S. Till, K. Karlapalem, Q. Li, and E. Kafeza, "Workflow view driven cross-organizational interoperability in a web service environment," *Inf. Technol. and Management*, vol. 5, no. 3-4, pp. 221–250, 2004.

[19] S. Cohen, S. C. Boulakia, and S. B. Davidson, "Towards a model of provenance and user views in scientific workflows." in *DILS*, 2006, pp. 264–279.

[20] Z. Bao, S. Cohen-Boulakia, S. Davidson, A. Eyal, and S. Khanna, "Differencing provenance in scientific workflows," in *Data Engineering, 2009. ICDE 2009. IEEE 25th International Conference on*, 2009.

[21] D.-R. Liu and M. Shen, "Modeling workflows with a process-view approach," in *DASFAA '01: Proceedings of the 7th International Conference on Database Systems for Advanced Applications*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 260–267.

[22] R. Bobrik, M. Reichert, and T. Bauer, "Parameterizable Views for Process Visualization," University of Twente," Technical Report, 2007, http://dbis.eprints.uni-ulm.de/299/1/BRB07.pdf.

[23] R. B. A. Kamel Barkaoui and Z. Sbai, "Workflow soundness verification based on structure theory of petri nets," in *International Journal of Computing and Information Sciences*, 2007, pp. 51–61.

[24] W. M. P. van der Aalst, "Workflow verification: Finding control-flow errors using petri-net-based techniques," in *Business Process Management*, 2000, pp. 161–183. [Online]. Available: citeseer.ist.psu.edu/vanderaalst00workflow.html

[25] E. Kindler, A. Martens, and W. Reisig, "Inter-operability of workflow applications: Local criteria for global soundness," in *Business Process Management*, 2000, pp. 235–253. [Online]. Available: citeseer.ist.psu.edu/kindler00interoperability.html

[26] J. Siegeris and A. Zimmermann, "Workflow model compositions preserving relaxed soundness," in *Business Process Management*, 2006, pp. 177–192.

[27] J. Vanhatalo, H. Völzer, and F. Leymann, "Faster and more focused control-flow analysis for business process models through sese decomposition," 2007, pp. 43–55. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74974-5_4

[28] W. Sadiq and M. E. Orlowska, "Applying graph reduction techniques for identifying structural conflicts in process models," in *CAiSE '99: Proceedings of the 11th International Conference on Advanced Information Systems Engineering*. London, UK: Springer-Verlag, 1999, pp. 195–209.

[29] M. Weiser, "Program slicing," in *ICSE '81: Proceedings of the 5th international conference on Software engineering*. Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449.

[30] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, "A core calculus of dependency," in *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1999, pp. 147–160.

[31] A. A. James Cheney and U. A. Acar, "Provenance as dependency analysis," in *Database Programming Languages*, 2007, pp. 138–152.

[32] J. Cheney, "Program slicing and data provenance," in *IEEE Data Eng. Bull.*, vol. 30(4), 2007, pp. 22– 28.