IEEE International Conference on Data Engineering

Differencing Provenance in Scientific Workflows

Zhuowei Bao #1, Sarah Cohen-Boulakia *2, Susan B. Davidson #3, Anat Eyal#4, Sanjeev Khanna #5

#*Department of Computer and Information Science, University of Pennsylvania, USA*
1zhuowei, 3susan, 4anate, 5sanjeev@seas.upenn.edu

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

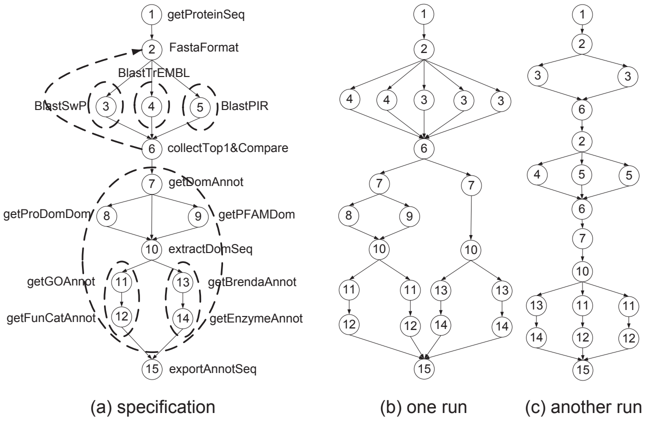
Abstract— **Scientific workflow management systems are increasingly providing the ability to manage and query the provenance of data products. However, the problem of differencing the provenance of two data products produced by executions of the same specification has not been adequately addressed. Although this problem is NP-hard for general workflow specifications, an analysis of real scientific (and business) workflows shows that their specifications can be captured as series-parallel graphs overlaid with well-nested forking and looping. For this natural restriction, we present efficient, polynomial-time algorithms for differencing executions of the same specification and thereby understanding the difference in the provenance of their data products. We then describe a prototype called PDiffView built around our differencing algorithm. Experimental results demonstrate the scalability of our approach using collected, real workflows and increasingly complex runs.**

I. Introduction

Answering scientific questions frequently involves conducting a complex set of analysis or "in-silico" experiments. Such experiments are typically defined as workflows and executed repeatedly. Each execution may vary the parameters and data inputs to the tools used as modules in the workflow; furthermore, alternative paths of the workflow may be followed. In this process, the scientist's goal is to identify executions which lead to "good" biological results. Comparing workflow runs and understanding the difference between them is thus of paramount importance to scientists.

To manage these complex experiments as well as the large number of intermediate and final data products they produce, a number of workflow systems have been developed for scientific applications which provide support to track provenance of derived data products. To understand the similarities and differences of these systems with respect to provenance, a Provenance Challenge Workshop was held [1]. One of the challenge queries was the differencing problem for a dataflow, the execution model commonly supported in scientific workflow systems. While most of the participating systems gave reasonable answers for this simple model, the techniques used do not extend to more complex execution models, *i.e.*, those that support forked executions over an unknown number of elements of an input set (*implicit iteration*), looping until some condition is met (*explicit iteration*), and parallel executions.

As an example of a complex workflow, consider a classical scientific analysis involving protein annotation shown in Fig. 1(a). The aim of this analysis is to infer the biological function of a new sequence from other sequences. While the details of the scientific analysis are not important, the structure of the workflow is, and is shown using a modified dataflow notation annotated with control flow information for forks and loops. A loop is indicated by a dotted backarrow, *e.g.*, from module 6 (collectTop1&Compare) to module 2 (FastaFormat), and forking is indicated by a dotted oblong, *e.g.*, the oblong around module 3 (BlastSwP) indicates that similar proteins can be searched for simultaneously. Multiple outgoing edges from a node indicate (non-exclusive) choice. Note that this workflow could also be expressed using BPEL [2], a standard which is becoming increasingly popular within bioinformatics. However, to simplify the presentation we will use a simpler notation that is also closer to what is used in most scientific workflow systems.



(a) specification (b) one run (c) another run

Fig. 1. Protein annotation workflow specification and runs

In a run, loops are unrolled and the number of fork executions is given explicitly. For example, two runs of the protein annotation workflow specification are shown in Fig. 1(b) and (c). Observe that Run (b) has two fork executions between modules 6 and 15, while Run (c) has two executions of the loop from module 6 to module 2.

In a dataflow execution, module names do not repeat and there is an immediate pairing between nodes in the two executions. Therefore, the naive approach of taking the difference of the nodes and edges in the two runs to calculate their difference works well. However, for the runs in Fig. 1 this approach does not work since node names repeat and hence there are many possible ways of pairing nodes. To determine the best pairing of nodes, a global computation must be performed to match copies that are most similar overall in terms of the control structure and dataflow.

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.103

The difference or edit distance between a pair of valid runs of the same specification is defined as a minimum cost sequence of edit operations that transforms one run to the other. While many edit operations could be considered (*e.g.*, insert or delete a node, and insert or delete an edge), it is important that they *transform a valid run to another valid run*, are *atomic*, and are *complete*. While inserting or deleting a node or an edge are atomic operations that can be used to transform between any two valid runs, they do not guarantee the validity of intermediate results. We therefore use as edit operations the insertion or deletion of elementary paths.

This notion of edit distance has a simple appealing interpretation: It is the shortest path connecting the given pair of runs in the space of all valid runs, where two valid runs are adjacent iff they differ by a single elementary path.

While the differencing problem is NP-hard for general graphs [3], the structure of most workflows can be captured as a series-parallel graph (SP-graph) overlaid with well-nested forks and loops. Such graphs capture the structure of most scientific workflows we have encountered in practice (*e.g.*, that in Fig. 1) as well as well-structured business process and other workflows [4]. For this natural restriction, we present efficient, polynomial-time algorithms for differencing workflow runs of the same specification. The algorithms are based on a well-known tree representation of SP-graphs in which internal nodes are annotated with *series* (in which case the children are ordered) or *parallel* (in which case the children are unordered). We then add annotations to represent loop (ordered) and fork (unordered) executions (*annotated SP-trees*).

In addition to capturing well-structured workflows, SP-graphs are in some sense the most complex graphs that allow efficient differencing algorithms: The simplest graph that is not an SP-graph has four nodes, and the differencing problem already becomes NP-hard on this graph [3].

An equally important difference in the provenance of two data products are parameter settings and input data sets. Two executions could have exactly the same control flow but produce very different results due to the data used. Data affects the differencing problem in two ways: It is a factor in the matching between nodes in the executions; and once the matching is done the data differences can be highlighted as annotations on nodes (for parameter settings) and edges (for data flowing between modules). For simplicity of presentation, however, we will focus solely on control flow in this paper.

A. Contributions and Overview

Our contributions are four-fold: First, we present a model of workflows that is sufficiently general to capture workflows that we have encountered in practice and collected from articles and sample workflows on the web (Section III). Second, for this model of workflows we present efficient, polynomial-time algorithms for differencing workflow executions, first considering forks (Sections IV and V), and then extending the techniques for loops (Section VI). Our algorithms work under fairly general cost models, allowing us to capture a variety of application-specific notions of distance. Third, we describe a prototype called **P**rovenance **Diff**erence **View**er (PDiffView) built around our differencing algorithm (Section VII). Fourth, we provide experimental results showing the scalability of our approach and the effect of the cost model (Section VIII).

II. Related Work

The *tree edit distance* problem has been extensively studied. [5] first formulated this problem for *ordered trees* as a generalization of the string edit distance problem [6]. Their model considers basic edit operations over individual nodes, such as deleting a node in a tree and making the children of this node become the children of its parent. [7] develops a dynamic programming algorithm that solves this problem efficiently. Other edit operations have also been proposed. For instance, [8] and [9] restrict insert and delete operations to leaves, and [10], [11] introduce more complex operations such as subtree move, subtree copy and subtree glue, which are meaningful to describe changes made in structured data. The edit distance problem for *unordered trees* is shown to be NP-hard [12]. However, by constraining the possible mappings between the two trees so that disjoint subtrees are mapped to disjoint subtrees, a polynomial-time algorithm can be given [13].

These techniques do not apply to our workflow difference problem, since we consider series-parallel graphs rather than trees, and use elementary paths in the edit operations. Observe that although our differencing algorithm relies on a tree representation of series-parallel graphs, and the corresponding tree edit distance problem is similar to [13], the transformed tree edit operations are different as is the differencing algorithm.

Other related work includes process mining [14], [15], which develops a notion of quantified process equivalence. Work on *recording* the edit history between workflow versions has also been studied [16] and extended to runs of different specifications [17]. These works compare different models (specifications), whereas we compare different *executions* of the same model.

III. Model and Problem Statements

In this section, we introduce the general workflow model, and formulate the workflow difference problem. We then develop in some detail a natural restriction of the general problem, called the SP-workflow difference problem, that is studied in detail in the remainder of this paper.

A. Definitions and Notation

Given a node-labeled directed graph G, we let $V(G)$ denote the set of all nodes in G and $E(G)$ denote the set of all edges in G. For any node v in $V(G)$, let $\texttt{Label}(v)$ denote the label on v. In addition, let $s(G)$ and $t(G)$ denote the unique source node and unique sink node in G.

Definition 3.1: A **flow network** is a directed graph G in which there exist a single source node $s(G) \in V(G)$ and a single sink source node $t(G) \in V(G)$, and every node $v \in V(G)$ lies on some path from $s(G)$ to $t(G)$.

A sub-class of flow networks that naturally arises when modeling program control and dataflow are *series-parallel graphs*.

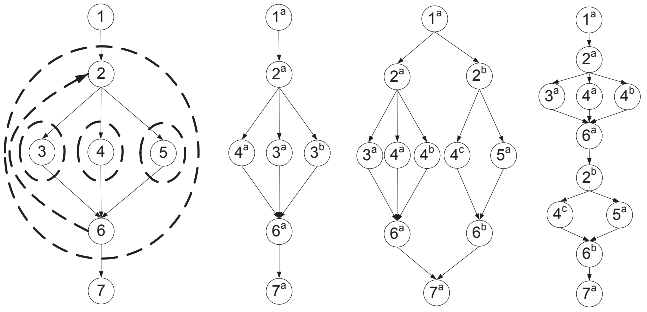
Definition 3.2: A **series-parallel graph** (also called **SP-graph**) is a directed multigraph G with a single source s and a single sink t (two terminals) that can be produced by a sequence of the following operations:

- **Basic SP-graph:** Create a new graph consisting of a single edge directed from node s to node t.
- **Series Composition:** Given two SP-graphs G_1 and G_2 with sources s_1, s_2 and sinks t_1, t_2 respectively, form a new graph $G = S(G_1, G_2)$ by identifying $s = s_1$, $t_1 = s_2$ and $t = t_2$.
- **Parallel Composition:** Given two SP-graphs G_1 and G_2 with sources s_1, s_2 and sinks t_1, t_2 respectively, form a new graph $G = P(G_1, G_2)$ by identifying $s = s_1 = s_2$ and $t = t_1 = t_2$.

In this definition, S and P are two functions that take a pair of SP-graphs as input and produce their series or parallel composition as output. A straightforward induction on the number of operations used to produce the SP-graph shows that every SP-graph is an acyclic flow network.

The inductive definition of SP-graphs given above naturally lends itself to two special classes of subgraphs of SP-graphs.

Definition 3.3: Given an SP-graph G, a subgraph H of G is said to be a **series (parallel) subgraph** if H is the series (parallel) composition of two SP-graphs and G can be constructed from H by applying a sequence of series or parallel compositions with other SP-graphs. In addition, we say any single edge (basic SP-graph) of G is a series subgraph.



(a) specification G (b) run R_1 (c) run R_2 (d) run R_3

Fig. 2. SP-workflow specification and runs

Example 3.1: We will use the simplified SP-workflow example shown in Fig. 2 in the remainder of this paper. All four graphs shown are SP-graphs, ignoring the dotted line and oblongs in (a). The number inside the circle indicates the label on the node. We use a superscript on labels to obtain a unique identifier for each node in a run.

B. General Workflow Model

A workflow model has two components: a *specification* that serves as a template for executions, and the set of valid *runs* for the given specification. Informally, a workflow specification consists of a set of different modules and defines the order in which they can be executed. A workflow run is a partial order of steps where each step is an instance of a module defined in the underlying specification, and the partial order conforms to the ordering constraints in the given specification.

Formally, a **workflow specification** is given by a flow network G with unique labels on the nodes. Given a workflow specification G, a flow network R with labels on the nodes (not necessarily unique) is said to be a **valid run** with respect to G if R is acyclic, and there exists a homomorphism $h : V(R) \to V(G)$ such that 1) $\forall v \in V(R), \texttt{Label}(v) = \texttt{Label}(h(v))$; 2) $h(s(R)) = s(G), h(t(R)) = t(G)$; and 3) $\forall(u, v) \in E(R)$, $(h(u), h(v)) \in E(G)$.

Notice that even if the specification G has cycles, a valid run R is always acyclic, since we unfold the cycles in the specification to capture the sequential order of all iterations in a workflow run. Consequently, the node labels in a run R are not necessarily unique.

C. The Workflow Difference Problem

The goal of the workflow difference problem is to find the edit distance and a path edit script between two valid runs of the same specification. We begin by defining two edit operations, and then propose a cost model for them. Our notion of edit distance has a simple appealing interpretation: It is the shortest path connecting the given pair of valid runs in the space of all valid runs, where two valid runs are adjacent iff one can be transformed into another by a single edit operation and the length of each edge is given by the cost model.

1) Edit Operations and Edit Script: In the following, we assume that R_1 and R_2 are valid runs with respect to the same specification G, and use the notion of an *elementary path*:

Definition 3.4: Given a valid run R with respect to a specification G, a path p is said to be an **elementary path** in R iff 1) each internal node on p has exactly one incoming edge and one outgoing edge; and 2) $s(p)$ has at least two outgoing edges and $t(p)$ has at least two incoming edges.

An edit operation ω applied to a valid run R_1 to produce another valid run R_2 with respect to a specification G is written as $R_1 \xrightarrow[G]{\omega} R_2$. We consider the following two *path edit operations*:

- **Path Insertion:** A path insertion operation creates a new (elementary) path p between two existing nodes and is denoted by $\Lambda \to p$. The restriction we impose on p is that it is an elementary path in R_2.
- **Path Deletion:** This operation is the inverse of the path insertion operation. A path deletion operation is denoted by $p \to \Lambda$, where p is an elementary path to be deleted from a given run.

We define an edit script to be a sequence of zero or more edit operations. Formally, a sequence of path edit operations $\mathcal{E} = \omega_1, \omega_2, \ldots, \omega_k$ is said to be an **edit script** from R_1 to R_2, written as $R_1 \xrightarrow[G]{\mathcal{E}} R_2$, if there exists a sequence of valid runs with respect to G, say $S_0, S_1, \ldots, S_k$, such that $S_0 = R_1, S_k = R_2$ and $S_{i-1} \xrightarrow[G]{\omega_i} S_i$ for $1 \leq i \leq k$.

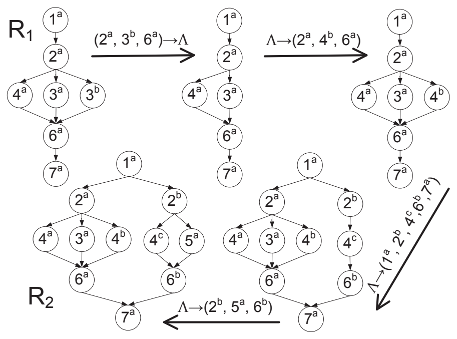


Fig. 3. A path edit script from R_1 to R_2

Example 3.2: A path edit script from R_1 to R_2 (see Fig. 2) is shown in Fig. 3. Note that each intermediate graph is a valid run with respect to the specification G (see Fig. 2).

There are several principles that motivate our choice of edit operations. Firstly, *they preserve the validity of the run.* Other edit operations, such as inserting or deleting a node, may violate the validity of the run, and hence make the notion of distance meaningless with respect to the underlying specification. Secondly, *they are atomic.* More complex operations can be decomposed to a sequence of elementary path edit operations. For example, one could define a path replacement operation that replaces one path by another or a subgraph insertion operation that creates an SP-graph between two nodes in one step. Such operations may be detected by post-processing the output of our algorithm. Finally, *they are complete.* Every pair of valid runs can be transformed from one to another by this set of operations. These two elementary path edit operations are therefore a natural choice.

2) Cost Model and Edit Distance: Given two valid runs, there may be many edit scripts that transform one to another. Among them, we are interested in finding one with the *minimum cost*. To this end, we introduce a cost model for edit operations and edit scripts.

There is a tradeoff between the generality of the cost model and the difficulty in computing a minimum-cost edit script. For example, a simple *unit cost* model would assign each edit operation a cost of *one*, and the cost of an edit script would be the number of its operations. On the other hand, a very general cost model would have a user-defined function to determine the cost of each edit operation, based on the type of the edit operation, as well as the particular path on which it operates. However, since the number of paths in an SP-graph can be exponentially large, we need a cost function with a compact representation that is still general.

The model we will therefore use is that the cost of each edit operation is given by a function γ that is determined by both the length of the elementary path to be edited, and the labels on its two terminals. That is, for all elementary paths p,

$$\gamma(\Lambda \to p) = \gamma(|p|, \texttt{Label}(s(p)), \texttt{Label}(t(p))) \qquad (1)$$

In addition, we constrain γ to be a *distance metric* with respect to elementary path insertions and deletions, which satisfies the following conditions:

1) *non-negativity:* $\gamma(\Lambda \to p) \geq 0$;
2) *identity:* $\gamma(\Lambda \to p) = 0$ iff $|p| = 0$ and $s(p) = t(p)$;
3) *symmetry:* $\gamma(\Lambda \to p) = \gamma(p \to \Lambda)$; and
4) *quadrangle inequality:* for all elementary paths p_1, p_2, p_2', p_3 such that $p_1 \circ p_2 \circ p_3$ and $p_1 \circ p_2' \circ p_3$ are well-defined, $\gamma(\Lambda \to p_1 \circ p_2 \circ p_3) \leq \gamma(\Lambda \to p_1 \circ p_2' \circ p_3) + \gamma(\Lambda \to p_2) + \gamma(p_2' \to \Lambda)$.

The quadrangle inequality essentially says that the cost of inserting an elementary path p directly is never more than the cost of inserting another elementary path p', and then replacing a part of p' to make it identical to p.

Our cost model is general enough to capture a wide spectrum of cost functions. For example, any sublinear function $\gamma(l, A, B) = l^\epsilon$ where $\epsilon \leq 1$ is eligible. When $\epsilon = 0$, this is exactly the unit cost function mentioned above.

Finally, the cost of an edit script is the sum of the costs of its individual operations. To express this, we extend the cost function γ to an edit script $\mathcal{E}$ by letting $\gamma(\mathcal{E}) = \sum_{\omega \in \mathcal{E}} \gamma(\omega)$.

Definition 3.5: Given a cost function γ, the **edit distance** between R_1 and R_2, denoted by $\delta(R_1, R_2)$, is defined as the minimum cost of an edit script from R_1 to R_2. Formally, $\delta(R_1, R_2) = \min\{\gamma(\mathcal{E}) \mid R_1 \xrightarrow[G]{\mathcal{E}} R_2\}$.

Problem Statement: Given two valid runs R_1 and R_2 with respect to a specification G, and a cost function γ, we want to compute the edit distance $\delta(R_1, R_2)$ as well as the corresponding minimum-cost edit script from R_1 to R_2.

D. The SP-Workflow Difference Problem

The problem of computing the workflow difference under a general workflow model is at least as hard as subgraph isomorphism, a well known NP-hard problem. Fortunately, the structure of most scientific workflows in practice – and business process and other workflows, see [4] – can be captured by a restricted model where the specification graphs are SP-graphs overlaid with well-nested forking and looping. We will refer to this model as the *SP-workflow model*. In Sections IV and V, we first discuss in some detail the difference problem under a *basic* SP-workflow model which considers only well-nested forking, and then outline how to extend this framework to handle looping in Section VI.

To define well-nested forking (and eventually, looping), we use the notion of a *laminar family* [18]:

Definition 3.6: Let $\mathcal{F}$ be a collection of subsets over a ground set U. Then $\mathcal{F}$ is a **laminar family** if for any pair of sets H_1, H_2 in $\mathcal{F}$, one of the following is true: (i) $H_1 \subset H_2$; or (ii) $H_2 \subset H_1$; or (iii) $H_1 \cap H_2 = \emptyset$.

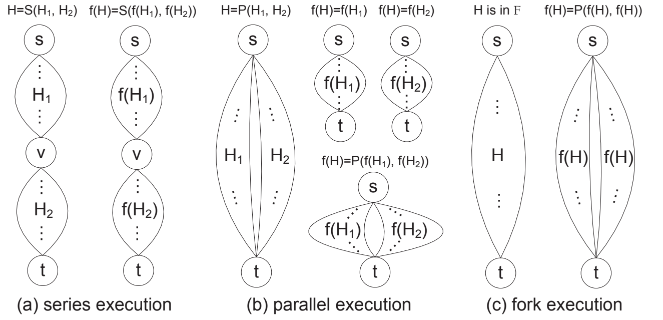
In the basic model, an **SP-workflow specification** is then given by a pair $(G, \mathcal{F})$, where G is an SP-specification graph with unique labels on the nodes, and $\mathcal{F}$ is a laminar family of series subgraphs of G describing the well-nested set of allowed fork executions. Furthermore, we consider three kinds of executions for an SP-workflow specification $(G, \mathcal{F})$:

- **Series Execution:** For any series subgraph H of G, a series execution of H executes its two sequential components in series. In the case where H is a basic SP-graph, it returns H itself as a valid run.

- **Parallel Execution:** For any parallel subgraph H of G, a parallel execution of H executes either one of or both of its two branches in parallel.
- **Fork Execution:** For any series subgraph H of G belonging to $\mathcal{F}$, a fork execution of H replicates one or more copies of H and executes them in parallel: They are split at the forking point (source) $s(H)$ and then joined together at the synchronization point (sink) $t(H)$, generating the parallel composition of one or more valid runs with respect to H. Note that these runs (graphs) may differ from each other as long as they are all valid with respect to the same part of the specification. The fork execution is defined over *series* subgraphs of G, since a forking over a parallel subgraph is equivalent to forking over each of its series component subgraphs.

We may abstract the above three executions by a *nondeterministic recursive* function, called an **execution function** (see Fig. 4), from SP-graphs to SP-graphs:

$$f(H) = \begin{cases} H & \text{if } H = (s(H), t(H)) \\ S(f(H_1), f(H_2)) & \text{if } H = S(H_1, H_2) \\ f(H_1) \text{ or } f(H_2) & \text{if } H = P(H_1, H_2) \\ \ \text{or } P(f(H_1), f(H_2)) & \\ P(f(H), f(H)) & \text{if } H \in \mathcal{F} \end{cases}$$



(a) series execution (b) parallel execution (c) fork execution

Fig. 4. Execution function f

A valid workflow run is now naturally defined as a graph that can be produced by applying a sequence of series, parallel, and fork executions recursively on the given SP-specification. Formally, given an SP-specification $(G, \mathcal{F})$, a node-labeled directed acyclic graph R is said to be a **valid workflow run** with respect to $(G, \mathcal{F})$ if $R = f(G)$ where f is the execution function for $(G, \mathcal{F})$.

Example 3.3: Fig. 2 shows a pair of valid runs R_1 and R_2 that are both produced from the SP-specification $(G, \mathcal{F})$ by applying a sequence of series, parallel and fork executions. Note that in Fig. 2(a) the fork executions are defined over the series subgraphs $(2, 3, 6)$, $(2, 4, 6)$, $(2, 5, 6)$ and the entire graph G. We defer the discussion of the loop implied by the dotted line to Section VI.

One can show by induction that any graph $f(G)$ generated above is an SP-graph and admits a graph homomorphism to the specification graph G. Thus, this new definition of the validity is consistent with our original definition for the general model. However, it further restricts the class of valid runs.

IV. An Equivalent Problem

We now describe a well-known tree representation of SP-graphs [19]. By using SP-trees for both specifications and valid runs, we convert the SP-workflow difference problem into an equivalent edit distance problem on SP-trees. Details of algorithms and proofs of claims can be found in [3].

A. SP-trees

The **SP-tree** representation T (a.k.a. tree decomposition) of an SP-graph G [19] captures the sequence of operations used to construct G as follows:

- If G is a basic SP-graph, then T is a single node v with $\texttt{Type}(v) = Q$.
- If G is the series or parallel composition of G_1 and G_2, then T has a root v with $\texttt{Type}(v) = S$ or P, and its two children are the SP-trees for G_1 and G_2. The children of an S node are ordered while the children of a P node are unordered.

A linear time algorithm for the tree decomposition problem has been given by [19]. We abstract the decomposition as a recursive function g from SP-graphs to SP-trees:

$$g(G) = \begin{cases} Q() & \text{if } G = (s(G), t(G)) \\ S(g(G_1), g(G_2)) & \text{if } G = S(G_1, G_2) \\ P(g(G_1), g(G_2)) & \text{if } G = P(G_1, G_2) \end{cases}$$

In this definition, the Q, S and P functions applied to SP-trees create a new node with the corresponding type as the root, and make all input SP-trees the children of this root. Note that we use the same name as the S and P functions (applied to SP-graphs) defined in Definition 3.2, because they essentially perform the same compositions but on different domains.

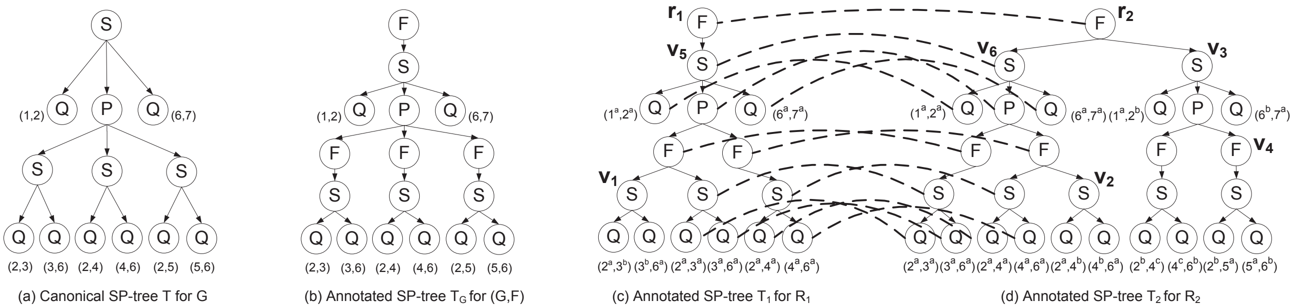
A key observation is that the SP-tree representation of SP-graphs is not unique. We therefore compress a binary SP-tree into a **canonical SP-tree** by repeatedly merging two adjacent nodes with the same type. The canonical SP-tree representation of SP-graphs is unique [19] up to reordering of the children of a P node.

Example 4.1: The canonical SP-tree T for the SP-specification graph G (see Fig. 2(a)) is shown in Fig. 5(a). In this figure, we use a pair of node identifiers to denote the edge represented by each Q node (leaf).

B. Annotated SP-trees for Specifications

Recall that an SP-specification is given by a pair $(G, \mathcal{F})$. The canonical SP-tree for the SP-graph G essentially captures the series and parallel executions implied by this specification. To capture allowed fork executions, we therefore annotate the tree using the given laminar family $\mathcal{F}$.

Given an SP-specification $(G, \mathcal{F})$, the **annotated SP-tree for** $(\mathbf{G}, \mathbf{F})$ is obtained as follows: We first construct the canonical SP-tree for G, and then, for each series subgraph in $\mathcal{F}$, insert an F node as a parent of the root of the subtree which represents this series subgraph.


S
P
Q
F
(1,2)
(6,7)
(2,3) (3,6) (2,4) (4,6) (2,5) (5,6)
r1
r2
v1
v2
v3
v4
v5
v6
$(1^a,2^a)$
$(6^a,7^a)$
$(1^a,2^b)$
$(6^b,7^a)$
$(2^a,3^b)$ $(3^b,6^a)$ $(2^a,3^a)$ $(3^a,6^a)$ $(2^a,4^a)$ $(4^a,6^a)$
$(2^a,3^a)$ $(3^a,6^a)$ $(2^a,4^a)$ $(4^a,6^a)$ $(2^a,4^b)$ $(4^b,6^a)$ $(2^b,4^c)$ $(4^c,6^b)$ $(2^b,5^a)$ $(5^a,6^b)$

(a) Canonical SP-tree T for G (b) Annotated SP-tree T_G for (G,F) (c) Annotated SP-tree T_1 for R_1 (d) Annotated SP-tree T_2 for R_2

Fig. 5. Tree representation of SP-workflow specification and runs

Example 4.2: The annotated SP-tree T_G for the SP-specification $(G, \mathcal{F})$ (see Fig. 2(a)) is shown in Fig. 5(b).

The annotated SP-tree is a semi-ordered tree: For an S node the left-to-right order among its children is significant, but for a P or an F node it is irrelevant.[1] We thus say two annotated SP-trees T and T', are *equivalent*, $T \equiv T'$, if they differ only in the order of children of P or F nodes.

Lemma 4.1: The annotated SP-tree representation of an SP-specification is unique. That is, if two annotated SP-trees T_G and T'_G for the same SP-specification $(G, \mathcal{F})$ are produced, then $T_G \equiv T'_G$.

C. Annotated SP-Trees for Valid Runs

We now define a **tree execution function** f' that takes the annotated SP-tree for a specification as input and produces as output the annotated SP-tree for a valid run. Formally, f' is a *nondeterministic recursive* function from annotated SP-trees to annotated SP-trees:

$$f'(T) = \begin{cases} T & \text{if } T = Q() \\ S(f'(T_1), \ldots, f'(T_k)) & \text{if } T = S(T_1, \ldots, T_k) \\ P(f'(T_{i_1}), \ldots, f'(T_{i_j})) & \text{if } T = P(T_1, \ldots, T_k) \\ F(f'(T_1), \ldots, f'(T_1)) & \text{if } T = F(T_1) \end{cases}$$

where $\{i_1, \ldots, i_j\}$ is a nonempty subset of $\{1, \ldots, k\}$ and F takes one or more copies of $f'(T_1)$ as input.

Given a valid run R with respect to an SP-specification $(G, \mathcal{F})$, **the annotated SP-tree for R** is obtained as follows: We start by constructing the annotated SP-tree T_G for $(G, \mathcal{F})$ and the canonical SP-tree T'_R for R, and then generate the annotated SP-tree T_R for R by a deterministic variant of the tree execution function f'' such that $f''(T_G, T'_R) = T_R$. Intuitively, f'' simulates the original nondeterministic tree execution function f' and leads the tree derivation to the corresponding annotated SP-tree in terms of the given valid run. In each step of the tree derivation described in f', we make the decision (e.g. which subset of children is chosen for a P node, or how many copies are replicated for an F node) by doing a case analysis on the current T_G and T'_R, matching zero or more subtrees in T'_R with each subtree in T_G based on the leaves contained in each subtree. Note that even with a series composition, there may be multiple matches in the subtrees of T'_R due to a fork execution.

Example 4.3: The annotated SP-trees T_1 and T_2 for the runs R_1 and R_2 (see Fig. 2(b) and (c)) are shown in Fig. 5(c) and (d) respectively.

Lemma 4.2: The annotated SP-tree representation of a valid run is unique. That is, if two annotated SP-trees T_R and T'_R for the same valid run R with respect to an SP-specification $(G, \mathcal{F})$ are produced, then $T_R \equiv T'_R$.

D. Edit Distance on Annotated SP-trees

Based on the tree representation of SP-workflows, we now propose an edit distance problem on annotated SP-trees that is equivalent to our SP-workflow difference problem.

In the following, for any annotated SP-tree T, let $\texttt{Graph}(T)$ denote the graph from which T is constructed. In addition, let $T[v]$ denote the subtree rooted at a node v in T and let $p(v)$ denote the parent of node v.

Definition 4.1: Given an annotated SP-tree T, $T[v]$ is said to be an **elementary subtree** in T iff 1) $T[v]$ does not contain any P or F node that has more than one child; and 2) $p(v)$ is a P or an F node that has more than one child.

We consider two *subtree edit operations* over the annotated SP-trees: **Subtree Insertion** and **Subtree Deletion**. Following the notation for path edit operations, we denote a subtree insertion by $\Lambda \to T[v]$ and a subtree deletion by $T[v] \to \Lambda$, where $T[v]$ is an elementary subtree to be edited. The following lemma shows the correspondence between an elementary subtree and an elementary path.

Lemma 4.3: Given the annotated SP-tree T for a valid run R, if $T[v]$ is an elementary subtree in T, then $p = \texttt{Graph}(T[v])$ is an elementary path in R. Conversely, if p is an elementary path in R, then there exists an elementary subtree $T[v]$ in T such that $p = \texttt{Graph}(T[v])$.

Given a cost function γ over path edit operations, we extend γ to subtree edit operations by letting

$$\gamma(\Lambda \to T[v]) = \gamma(\Lambda \to \texttt{Graph}(T[v]))$$

By Lemma 4.3 and Eq. 1 in Section III-C.2, we have the following appealing interpretation of γ on trees: For any elementary subtree $T[v]$, we have

$$\gamma(\Lambda \to T[v]) = \gamma(|T[v]|, \texttt{Label}(s(v)), \texttt{Label}(t(v)))$$

where $|T[v]|$ is the number of leaves (Q nodes) of $T[v]$ and $s(v)$, $t(v)$ are two terminals of $\texttt{Graph}(T[v])$. Note that $s(v)$ and $t(v)$ are two invariants associated with each node v and will not be changed by any subtree edit operation.

[1] In the specification, F nodes can only have one child, however in a run they can have multiple children.

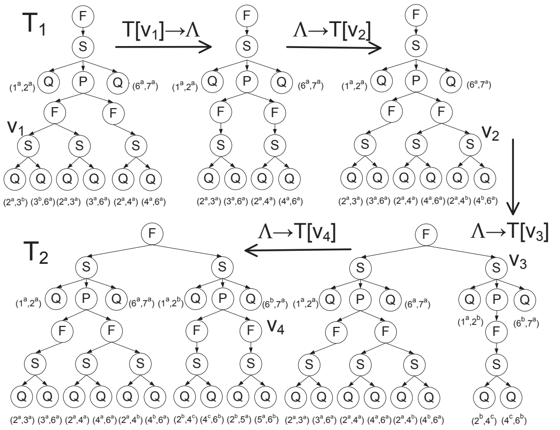


Fig. 6. A subtree edit script from T_1 to T_2

Example 4.4: Fig. 6 shows a subtree edit script from T_1 to T_2 (see Fig. 5) which corresponds to the path edit script (see Fig. 3) between the underlying runs R_1 and R_2.

Definition 4.2: Given a cost function γ, the **edit distance** between T_1 and T_2, denoted by $\delta(T_1, T_2)$, is defined as the minimum cost of a subtree edit script from T_1 to T_2. Formally, $\delta(T_1, T_2) = \min\{\gamma(\mathcal{E}) \mid T_1 \xrightarrow[T_G]{\mathcal{E}} T_2\}$.

The following theorem shows that the two edit distance problems are equivalent.

Theorem 1: Let R_1 and R_2 be a pair of valid runs and let T_1 and T_2 be their annotated SP-trees respectively. Then $\delta(R_1, R_2) = \delta(T_1, T_2)$.

V. Algorithm

The algorithm to compute the edit distance as well as the corresponding minimum-cost edit script between two valid runs R_1 and R_2 with respect to an underlying SP-specification $(G, \mathcal{F})$ has two steps:

1) Generating the annotated SP-trees T_G, T_1 and T_2 for the specification $(G, \mathcal{F})$ and the pair of valid runs R_1 and R_2 respectively.
2) Computing the edit distance $\delta(T_1, T_2)$ as well as the corresponding minimum-cost edit script from T_1 to T_2.

The first subproblem was solved in the previous section. We now study the second subproblem. For clarity of exposition, we focus on computing the edit distance, since the corresponding minimum-cost edit script can be easily produced by bookkeeping.

A. Well-formed Mapping

We now formalize the notion of *mapping* implied by an edit script, and show the correspondence between them. Intuitively, an edit script transforming a tree T_1 to another tree T_2 keeps some of the nodes in T_1 unchanged and inserts and deletes other nodes to create a tree T_2' that is isomorphic to T_2. The bijection between T_2' and T_2 gives rise to a partial one-to-one mapping between the nodes of T_1 and T_2.

Example 5.1: The dashed lines between T_1 and T_2 in Fig. 5 show a mapping that corresponds to the subtree edit script shown in Fig. 6.

For any node v in $T_i (i = 1, 2)$, let $h(v)$ be the node in T_G such that $T_i[v]$ is derived from $T_G[h(v)]$. Formally, $T_i[v] = f'(T_G[h(v)])$ where f' is the tree execution function defined in Section IV-C. In addition, for any pair of nodes (v_1, v_2) in T_1 and T_2, we say v_1 and v_2 are **homologous** if $h(v_1) = h(v_2)$. That is, $T_1[v_1]$ and $T_2[v_2]$ represent two valid runs with respect to the same part of a specification.

Definition 5.1: A set M of pairs of nodes is said to be a **well-formed mapping** from T_1 to T_2 iff

1) *one-to-one:* M is a one-to-one mapping from T_1 to T_2. Formally, for any pair of $(v_1, v_2) \in M$ and $(v_1', v_2') \in M$, $v_1 = v_1'$ iff $v_2 = v_2'$;
2) *root mapped:* The roots of T_1 and T_2 are mapped by M. Formally, $(r_1, r_2) \in M$, where r_1 and r_2 are the roots of T_1 and T_2 respectively;
3) *specification preserved:* If a pair of nodes is mapped by M, then they are homologous. Formally, for any $(v_1, v_2) \in M$, $h(v_1) = h(v_2)$;
4) *parent preserved:* If a pair of nodes is mapped by M, then their parents are also mapped. Formally, for any $(v_1, v_2) \in M$, $(p(v_1), p(v_2)) \in M$; (recall that $p(v)$ denotes the parent of a node v) and
5) *children of an S node preserved:* If a pair of S nodes is mapped by M, then each pair of their children is also mapped. Formally, for any $(v_1, v_2) \in M$ such that $\texttt{Type}(v_1) = \texttt{Type}(v_2) = S$, $(c_i(v_1), c_i(v_2)) \in M$ for all i, where $c_i(v)$ denotes the i'th child of a node v.

Definition 5.2: Let M be a well-formed mapping from T_1 to T_2. A pair of nodes (v_1, v_2) in M is said to be **unstably matched** iff 1) (v_1, v_2) is a pair of P nodes; and 2) both v_1 and v_2 have only one child, and this pair of children is homologous and not mapped by M. A pair in M that is not unstably matched is called **stably matched**.

Given a well-formed mapping M from T_1 to T_2 and a pair of nodes (v_1, v_2) in M, let $M(v_1, v_2)$ be the corresponding mapping from $T_1[v_1]$ to $T_2[v_2]$. We then define its cost $\gamma(M(v_1, v_2))$ as follows:

if (v_1, v_2) is *stably matched* then

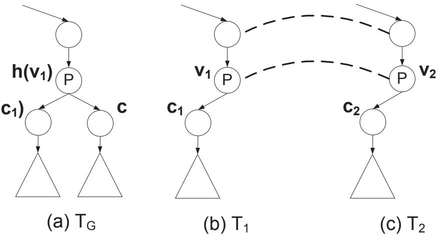
$$\sum_{(c_1,c_2)\in M} \gamma(M(c_1, c_2)) + \sum_{c_1 \notin I_1} X_{T_1}(c_1) + \sum_{c_2 \notin I_2} X_{T_2}(c_2) \quad (2)$$

if (v_1, v_2) is *unstably matched* then

$$X_{T_1}(c_1) + X_{T_2}(c_2) + 2 * W_{T_G}(h(v_1), h(c_1)) \quad (3)$$

where c_1 and c_2 are the children of v_1 and v_2 respectively, I_1 and I_2 are the sets of nodes mapped by M in T_1 and T_2 respectively, $X_T(c)$ is the minimum cost of deleting the subtree $T[c]$, and $W_{T_G}(h(v_1), h(c_1))$ is the minimum cost of deleting an elementary subtree rooted at a child of $h(v_1)$ that is distinct from the subtree rooted at $h(c_1)$ in T_G.

A short explanation follows. For a pair of *stably matched* nodes (v_1, v_2), we sum up the cost of all mappings between their children and the minimum cost of deleting or inserting all unmapped children (see Eq. 2); for a pair of *unstably matched* nodes (v_1, v_2) (see Fig. 7), by Definition 5.2 both of them must have only one child, say c_1 and c_2, and this pair of children is homologous and not mapped by M. Now consider an edit script from $T_1[v_1]$ to $T_2[v_2]$ that deletes $T_1[c_1]$ and inserts $T_2[c_2]$. Observe that v_1 will either have no children (if we delete $T_1[c_1]$ first) or have two homologous children (if we insert $T_2[c_2]$ first) at some intermediate state of the transformation. Both will violate the validity of the tree due to the fact that v_1 is a P node. Hence, the minimum-cost edit script from $T_1[v_1]$ to $T_2[v_2]$ must be constructed as follows: insert $T_G[c]$; delete $T_1[c_1]$; insert $T_2[c_2]$; and delete $T_G[c]$ (inserted in the first step), where $T_G[c]$ is an elementary subtree rooted at a child c of $h(v_1)$ that is distinct from $h(c_1)$ in T_G. This leads to the cost given by Eq. 3.



(a) T_G (b) T_1 (c) T_2

Fig. 7. Unstably matched nodes

Finally, let $\gamma(M)$ denote the cost of a well-formed mapping M from T_1 to T_2, then we have $\gamma(M) = \gamma(M(r_1, r_2))$, where r_1 and r_2 are the roots of T_1 and T_2 respectively.

Theorem 2: Let T_1 and T_2 be the annotated SP-trees for a pair of valid runs. Then $\delta(T_1, T_2) = \min\{\gamma(M) \mid M$ is a well-formed mapping from T_1 to $T_2\}$.

B. Edit Distance on SP-Trees

As a preprocessing step, we sketch an idea for computing the minimum cost of deleting a subtree. Note that the quadrangle inequality of our cost model guarantees that the minimum-cost subtree deletion is always achieved by a sequence of elementary subtree deletions. Recall that $X_T(v)$ denotes the minimum cost of deleting the subtree $T[v]$. We therefore compute $X_T(v)$ for each node v in T in a bottom-up manner: for each Q node, $X_T(v)$ is given by the cost model; for each P or F node, $X_T(v)$ is equal to the sum of $X_T(c)$'s for all children c of v; and for each S node, dynamic programming is used to compute the minimum cost of reducing each of its first i children into a branch-free tree with a total of l leaves.

We are now ready to give the algorithm which computes the edit distance $\delta(T_1, T_2)$. We do a bottom-up computation on T_1 and T_2, and compute the minimum-cost well-formed mapping $M^*(v_1, v_2)$ from $T_1[v_1]$ to $T_2[v_2]$ for each pair of homologous nodes (v_1, v_2). Note that by Definition 5.1 (root mapped) v_1 and v_2 are always mapped by $M^*(v_1, v_2)$ [Line 1]. Based on their types, we have four cases:

Case 1 [Lines 2–4] If (v_1, v_2) is a pair of Q nodes, then $M^*(v_1, v_2)$ consists only of a pair of nodes (v_1, v_2).

Case 2 [Lines 5–9] If (v_1, v_2) is a pair of S nodes, then they agree in the number of children. Moreover, each pair of their children are homologous. By Definition 5.1 (children of an S node preserved) they must be all mapped by $M^*(v_1, v_2)$.

Algorithm 1 *Edit-Distance-on-Trees*

```
Input: $T_1[v_1]$ and $T_2[v_2]$ such that $h(v_1) = h(v_2)$
Output: $M^*(v_1, v_2)$: the minimum-cost well-formed mapping from $T_1[v_1]$ to $T_2[v_2]$
 1: add $(v_1, v_2)$ to $M^*(v_1, v_2)$
 2: if Type$(v_1)$ = Type$(v_2) = Q$ then
 3:    return $M^*(v_1, v_2)$
 4: end if
 5: if Type$(v_1)$ = Type$(v_2) = S$ then
 6:    for each pair of children $(c_1, c_2)$ of $v_1$ and $v_2$ do
 7:       add $M^*(c_1, c_2)$ to $M^*(v_1, v_2)$
 8:    end for
 9: end if
10: if Type$(v_1)$ = Type$(v_2) = P$ then
11:    if both $v_1$ and $v_2$ have only one child, say $c_1$ and $c_2$,
       and $h(c_1) = h(c_2)$ then
12:       if $\gamma(M^*(c_1, c_2)) \leq X_{T_1}(c_1) + X_{T_2}(c_2) + 2 * W_{T_G}(h(v_1), h(c_1))$ then
13:          add $M^*(c_1, c_2)$ to $M^*(v_1, v_2)$
14:       end if
15:    else
16:       for each pair of children $(c_1, c_2)$ of $v_1$ and $v_2$ such
          that $h(c_1) = h(c_2)$ and $\gamma(M^*(c_1, c_2)) \leq X_{T_1}(c_1) + X_{T_2}(c_2)$ do
17:          add $M^*(c_1, c_2)$ to $M^*(v_1, v_2)$
18:       end for
19:    end if
20: end if
21: if Type$(v_1)$ = Type$(v_2) = F$ then
22:    $B^*$ := the minimum-cost bipartite matching
23:    for each pair of children $(c_1, c_2)$ of $v_1$ and $v_2$ such that
       $(c_1, c_2) \in B^*$ do
24:       add $M^*(c_1, c_2)$ to $M^*(v_1, v_2)$
25:    end for
26: end if
27: return $M^*(v_1, v_2)$
```

Case 3 If (v_1, v_2) is a pair of P nodes, we consider the following two subcases: **Case 3a** [Lines 11–14] If both v_1 and v_2 have only one child and they are homologous, say c_1 and c_2, then the pair (c_1, c_2) will be included in $M^*(v_1, v_2)$ only if the minimum cost of a well-formed mapping between them is no greater than the minimum cost of deleting and inserting the corresponding subtrees plus the cost of inserting and deleting a minimum-cost elementary subtree rooted at a child of v_1 that is not homologous to c_1, denoted by $W_{T_G}(h(v_1), h(c_1))$. It is straightforward to compute this using our cost model. **Case 3b** [Lines 15–19] Otherwise, note that for each child of v_1, there exists at most one child of v_2 that is homologous to it, and vice versa. A pair of homologous children will be included in $M^*(v_1, v_2)$ only if the minimum cost of a well-formed mapping between them is no greater than the minimum cost of deleting and inserting the corresponding subtrees.

Case 4 [Lines 21–26] If (v_1, v_2) is a pair of F nodes, then all children of v_1 and v_2 are homologous, and we need to find the minimum-cost matching between them. This is done by setting up the following bipartite graph between children of v_1 and v_2: Each pair of children of v_1 and v_2 are connected by an edge associated with the minimum cost of a well-formed mapping between them; each child of v_1 has an edge to a special node "$-$" associated with the minimum cost of deleting the corresponding subtree; and each child of v_2 has an edge to a special node "$+$" associated with the minimum cost of inserting the corresponding subtree. Now let B^* be the minimum-cost bipartite matching in this graph. Then a pair of children of v_1 and v_2 will be included in $M^*(v_1, v_2)$ only if they are matched by B^*.

Pseudo code of this algorithm is given in Algorithm 1. Note that by Theorem 2 $\delta(T_1, T_2) = \gamma(M^*(r_1, r_2))$, where r_1 and r_2 are the roots of T_1 and T_2 respectively, and the corresponding minimum-cost edit script can be easily derived from $M^*(r_1, r_2)$.

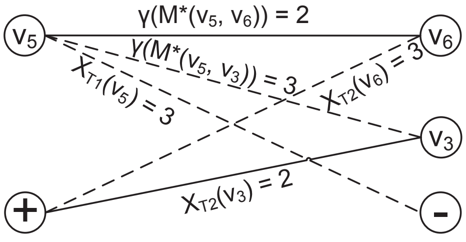


Fig. 8. Bipartite matching for F nodes

Example 5.2: The roots r_1 and r_2 of T_1 and T_2 (see Fig. 5) are a pair of F nodes. The bipartite graph for matching their children is shown in Fig. 8. Observe that r_1 has only one child v_5, while r_2 has two children v_6 and v_3. The minimum-cost bipartite matching is shown in solid lines. The unit cost model is used in this example, and the corresponding edit distance $\delta(T_1, T_2) = \gamma(M^*(r_1, r_2)) = \gamma(M^*(v_5, v_6)) + X_{T_2}(v_3) = 4$. This is confirmed by our edit script shown in Fig. 6.

C. Algorithm Complexity

The main cost of Algorithm 1 is solving the *weighted bipartite matching problem* (a.k.a. the *assignment problem*) for each pair of F nodes. The overall time complexity of this step is bounded by $O(|E|^3)$, using the *Hungarian algorithm* [20], where $|E|$ is the total number of edges in both R_1 and R_2.

VI. Extended SP-Workflow Model

We now outline an *extended SP-workflow model* capable of expressing both fork and loop executions. In the extended model, an SP-workflow specification is given by a triple $(G, \mathcal{F}, \mathcal{L})$ where $\mathcal{F}$ and $\mathcal{L}$ represent well-nested forking and looping imposed over an SP-specification G as a set of subgraphs, such that (1) $\mathcal{F} \cap \mathcal{L} = \emptyset$; and (2) the edge sets of $\mathcal{F} \cup \mathcal{L}$ form a laminar family. As before, elements of $\mathcal{F}$ are series subgraphs. Elements of $\mathcal{L}$ are complete subgraphs. A *complete subgraph* of G is either a series subgraph or a parallel subgraph of G and contains all the paths from its source to its sink. In the canonical SP-tree for G, it corresponds to a nonempty proper subset of consecutive children of an S node.

Intuitively, a loop execution of $H \in \mathcal{L}$ replicates one or more copies of H and executes them in series: They are concatenated by an *implicit* edge from the sink of one copy to the source of the next copy, generating the series composition of one or more valid runs with respect to H along with all implicit edges $(t(H), s(H))$ between them. As before, we may abstract the loop execution by the execution function f below:

$$f(H) = S(f(H), (t(H), s(H)), f(H)) \quad \text{if } H \in \mathcal{L}$$

Example 6.1: Fig. 2(d) shows a run R_3 in which the loop has been executed twice; note the *implicit* edge $(6^a, 2^b)$.

The annotated SP-trees for the specification $(G, \mathcal{F}, \mathcal{L})$ and the valid run R are then constructed as before, adding L nodes to represent allowed loop executions in $\mathcal{L}$. In particular, when generating the annotated SP-tree for R, we may capture the implicit edges from the sink of one iteration to the source of the next iteration by the order of children of an L node.

While subtree insertion and deletion over annotated SP-trees remain the same, the corresponding operations over SP-graphs are more complicated since iterations are connected in series by implicit edges. However, to preserve the atomicity of our edit operations, at most one iteration of a loop should be inserted or deleted by a single edit operation. We therefore introduce two more path edit operations:

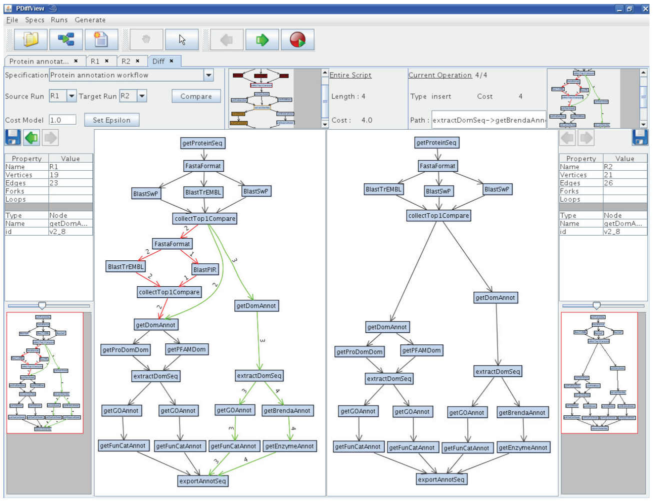
- **Path Expansion:** A path expansion operation creates a new iteration of a loop by inserting an elementary path between two existing consecutive iterations. Note that this operation also involves a set of necessary insertion and deletion of implicit edges.
- **Path Contraction:** This operation is the inverse of the path expansion operation. Intuitively, a path contraction operation removes an iteration of a loop by contracting the last elementary path.

Example 6.2: To delete the second iteration of the loop in R_3 (see Fig. 2(d)), we first delete the path $(2^b, 5^a, 6^b)$, then *contract* the path $(2^b, 4^c, 6^b)$ by replacing the path $(6^a, 2^b, 4^c, 6^b, 7^a)$ with the edge $(6^a, 7^a)$.

To extend our differencing algorithm to handle loops, we do one more case analysis in Algorithm 1: If (v_1, v_2) is a pair of L nodes, we set up the same bipartite graph between children of v_1 and v_2 as described in the case of F nodes. However, instead of finding a minimum-cost bipartite matching for F nodes, we will compute a minimum-cost **non-crossing** bipartite matching for L nodes, since the children of an L node are ordered. This problem can be efficiently solved by dynamic programming in $O(|E|^2)$ time. Therefore, the computation for F nodes still dominates the cost. With this framework and minor changes to our algorithms, we can obtain an efficient, polynomial time differencing algorithm for workflows characterized as SP-graphs with well-nested forking and looping. More details can be found in [3].

VII. Prototype–PDiffView

We have developed a prototype system called **P**rovenance **Diff**erence **View**er (PDiffView)[2] which allows users to view, store, generate and import/export SP-specifications and their associated runs. The user may then see the difference between two runs of the same specification by stepping through the set of edit operations in the minimum-cost edit script, or by seeing an overview. Since the graphs can be large, users may successively cluster modules in the specification to form a hierarchy of composite modules. The difference between two runs of that specification can then be viewed at any level in the defined hierarchy, giving the user the ability to zoom in on composite modules that indicate a large amount of change and ignore others that indicate no change.



Specification: Protein annotation workflow

Source Run: R1 Target Run: R2

Cost Model: 1.0

Entire Script
Length : 4
Cost : 4.0

Current Operation 4/4
Type insert Cost 4
Path : extractDomSeq->getBrendaAnn

Property	Value
Name	R1
Vertices	19
Edges	23
Forks	
Loops	
Type	Node
Name	getDomA...
id	v2_8

Property	Value
Name	R2
Vertices	21
Edges	26
Forks	
Loops	
Type	Node
Name	getDomA...
id	v2_8

Fig. 9. PDiffView snapshot

A snapshot of our prototype system is shown in Fig. 9. The big pane on the left-hand side shows the source run, with green edges indicating inserted paths and red edges indicating deleted paths in the edit script. The target run is shown in the big pane on the right-hand side. The small pane on top shows the specification, and the small pane on the top right gives the context for the edit operation being applied. The small panes on the bottom right and left corners display miniatures of the respective runs, and brief summaries of their statistics are listed above.

VIII. Evaluation

We empirically evaluate our differencing algorithm on both real and synthetic datasets. All experiments were performed on a local Pentium IV 2.8GHZ PC with 2GB memory running Fedora Core 6 with kernel version 2.6.20. The algorithm is implemented in Java 6, and specifications and runs are stored as XML files. In all experiments, the time to parse the XML file is omitted.

A. Real Scientific Workflows

In the first set of experiments, we evaluate the performance of our differencing algorithm over six collected, real scientific workflows[3]. Characteristics of these specifications are listed in Table I. $|\mathcal{F}|$ and $|\mathcal{L}|$ are the number of forks and loops annotated in the specification, respectively, and $||\mathcal{F}||$ and $||\mathcal{L}||$ are the total number of edges in the forks and loops. For each specification, we randomly generate a pair of valid runs, varying their total number of edges from 200 to 2000, and then measure the execution time of computing the minimum-cost edit script under the unit cost model. Each point is an average over 100 sample pairs.

TABLE I
Characteristics of Real Workflow Specifications

WORKFLOW	$\|V\|$	$\|E\|$	$\|\mathcal{F}\|$	$\|\|\mathcal{F}\|\|$	$\|\mathcal{L}\|$	$\|\|\mathcal{L}\|\|$
PA	11	13	3	6	1	6
EMBOSS	17	22	4	10	2	10
SAXPF	27	36	7	18	1	7
MB	17	19	2	6	1	6
PGAQ	37	41	4	22	2	26
BAIDD	29	36	8	17	2	12

Fig. 10 shows that our differencing algorithm performs well even on large runs. In the worst case, we can compute the edit distance between a pair of PGAQ workflow runs with a total of 2000 edges in less than one minute. In practice, most workflow runs have fewer than 200 edges, which can be done in less than one second.

Fig. 10 also shows that the execution time varies between specifications. However, it is hard to understand this variation using only the statistics listed in Table I. In the remainder of this section, we will show the effect of other factors.

B. Series vs Parallel

In the second set of experiments, we compare series specifications with parallel specifications. Let r be the ratio of series compositions to parallel compositions used to construct the specification. For instance, when $r = +\infty$ the (series) specification becomes a single path, and when $r = 0$ the (parallel) specification consists only of two vertices and a set of multi-edges between them. First of all, we randomly generate a synthetic workflow specification with no forks and loops, varying the number of edges from 100 to 1000 and setting r to be 3, 1 and $\frac{1}{3}$ respectively. For each specification, we randomly generate a pair of valid runs with $prob_p = 95\%$, where $prob_p$ is the probability that each parallel branch in the specification is taken by the run. We then measure the execution time and edit distance under the unit cost model. Each point is an average over 200 sample specifications.

Fig. 11 shows that computing the edit distance between a pair of runs of a series specification is expensive: Since there are no forks and loops in the specification, finding the minimum-cost mapping entails computing the minimum cost of deleting a subtree rooted at an S node using dynamic

[2] Available at http://db.cis.upenn.edu/.

[3] Real workflows can be found at http://www.myexperiment.org/.

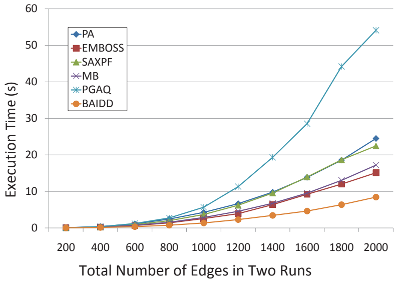


Fig. 10. Real scientific workflows

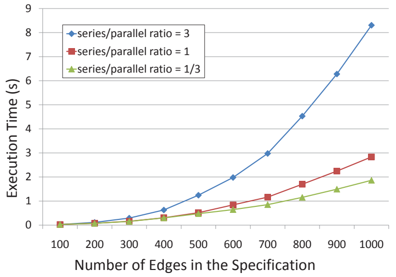


Fig. 11. Series vs Parallel (execution time)

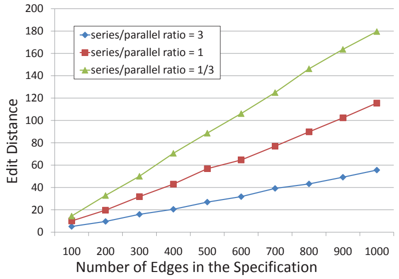


Fig. 12. Series vs Parallel (edit distance)

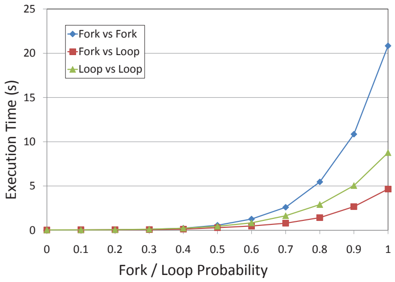


Fig. 13. Fork vs Loop (execution time)

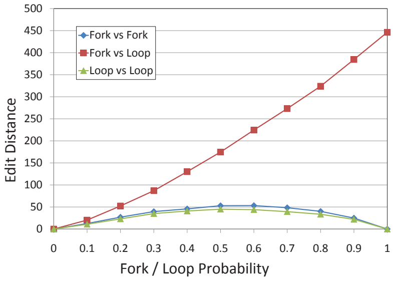


Fig. 14. Fork vs Loop (edit distance)

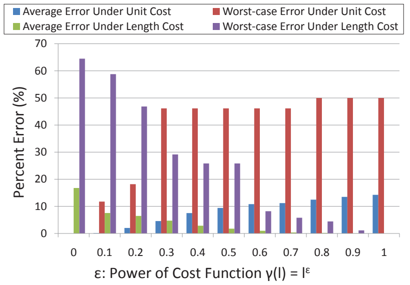


Fig. 15. Varying cost models

programming ($O(|E|^3)$). In contrast, the minimum cost of deleting a subtree rooted at a P node can be easily computed in linear time $O(|E|)$.

Fig. 11 also shows that the execution time increases with the size of specification, confirming our polynomial time complexity result. More interestingly, comparing Fig. 11 with Fig. 10, we observe that forks and loops bring a significant complexity to the differencing problem. Note that each run is randomly generated by taking on average 95% of branches in the specification, thus making Fig. 11 comparable to Fig. 10.

Fig. 12 shows that a pair of runs of a series specification have a smaller edit distance than a pair of runs of a parallel specification. There are two reasons for this: 1) deleting a long path only incurs a cost of one under the unit cost model; and 2) fewer parallel branches in the series specification means that the runs generated will be more similar. Comparing Figs. 12 and 11, we conclude that there is little correlation between the running time of the algorithm and the edit distance.

C. Fork vs Loop

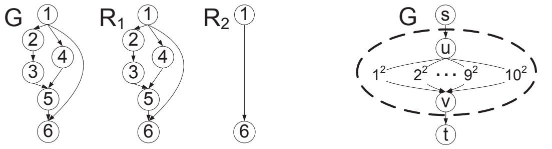
In the third set of experiments, we compare forks with loops. First, we randomly generate a synthetic workflow specification with 100 edges and a series/parallel ratio of 0.5, annotating it with 5 forks and 5 loops. To generate a random valid run, we use the following parameters: 1) max_F and max_L are the maximum number of copies replicated by each fork and loop executions respectively; and 2) $prob_F$ and $prob_L$ are the probabilities that each fork and loop copy is taken by the run, respectively. For instance, the product of max_F and $prob_F$ is the average number of copies in a fork execution. We now fix $prob_p = 1$ and $max_F = max_L = 20$, and randomly generate a run with many forks (and no loops) by varying $prob_F$ from 0 to 1 and setting $prob_L$ to be 0. Similarly, we generate a run with many loops (and no forks). Finally, we measure the execution time and edit distance between different combinations of runs under the unit cost model. Each point is an average over 200 sample specifications.

Fig. 13 shows that computing the edit distance between a pair of runs with many forks is extremely expensive when $prob_F$ is high, and that computing the edit distance between one run with many forks and one run with many loops is cheapest. This is because we do a minimum-cost bipartite matching to pair fork copies across the runs (our implementation uses the *Bellman-Ford algorithm* [21] and therefore takes $O(|E|^4)$ time), whereas to pair loop copies we calculate a minimum-cost non-crossing bipartite matching (it is solved by dynamic programming in $O(|E|^2)$ time). Furthermore, when we pair a run with many forks and a run with many loops, the bipartite matching instances are small because forked copies are never matched with loop copies.

Fig. 14 shows that the edit distance between a pair of runs with many forks (loops) will eventually drop to 0 when the fork (loop) probability approaches 1: Each fork copy will be replicated exactly max_F times, and the runs generated will have the same shape. In contrast, the edit distance between one run with many forks and one run with many loops monotonically increases, since a higher fork and loop probability results in a larger difference between the two runs. Comparing Figs. 14 and 13 again confirms that there is little correlation between running time and edit distance.

D. Influence of Cost Model on Edit Scripts

In the last set of experiments, we evaluate the influence of varying cost models on the minimum-cost edit script produced. Recall that any sublinear function $\gamma(l) = l^\epsilon$, where $\epsilon \leq 1$ and l is the length of path to be edited, can be used.



(a) Synthetic workflow (b) Synthetic specification

Fig. 16. Influence of Cost Model on Edit Scripts

Example 8.1: As shown in Fig. 16(a), two edit scripts that transform R_1 to R_2 are: $\mathcal{E}_1 = \{(1,2,3,5) \to \Lambda, (1,4,5,6) \to \Lambda\}$ and $\mathcal{E}_2 = \{(1,4,5) \to \Lambda, (1,2,3,5,6) \to \Lambda\}$. Which script is better depends on ϵ: 1) when $\epsilon = 0$ (unit cost) or 1 (cost equal to length), $\gamma(\mathcal{E}_1) = \gamma(\mathcal{E}_2)$; 2) when $0 < \epsilon < 1$, $\gamma(\mathcal{E}_1) > \gamma(\mathcal{E}_2)$; and 3) when $\epsilon < 0$, $\gamma(\mathcal{E}_1) < \gamma(\mathcal{E}_2)$. Thus different cost models may lead to different minimum-cost edit scripts between the same pair of valid runs.

To empirically evaluate the effect of different cost models, we use the synthetic workflow specification shown in Fig. 16(b). The specification G contains a fork subgraph connecting a pair of nodes u and v by 10 parallel paths. The length of the ith path is i^2. We now randomly generate a pair of valid runs by setting $max_F = 5$, $prob_F = 1$ and $prob_P = 0.5$. Each random run then contains exactly 5 fork copies, and each copy includes a random subset of roughly 5 parallel paths. We then compute the minimum-cost edit scripts between a pair of runs under different cost models, by varying ϵ from 0 to 1. Finally, we measure the percent error between the edit distance (*i.e.*, the minimum cost) and the cost of these edit scripts under the unit ($\epsilon = 0$) and length ($\epsilon = 1$) cost models. We test for 100 pairs of sample runs and evaluate both average and worst-case errors.

Fig. 15 shows that the minimum-cost edit script produced by one cost model may be suboptimal for another, and that the corresponding cost may be far away from the edit distance (*i.e.*, the minimum cost). As shown in Fig. 15, the average error under the unit cost model monotonically increases, while the average error under the length cost model monotonically decreases. The minimum-cost edit script produced by the length cost model has an average error of 14% and worst case of 50% under the unit cost model; the minimum-cost edit script produced by unit cost model has an average percent error of 16% and worst case of 64% under the length cost model. Not surprisingly, the minimum-cost edit scripts produced by other cost models show a tradeoff between the errors with respect to the unit and length cost models. This is due to the way in which fork copies of H are matched: In the unit cost model, copies which agree on the largest number of paths are matched, ignoring the lengths of unmatched paths. In the length cost model, matched copies may differ in many paths but agree on some of the longer paths.

IX. Conclusions

We show that the problem of differencing workflow runs of the same specification, described by series-parallel graphs overlaid with well-nested forks and loops, can be efficiently solved in $O(|E|^3)$ time, where $|E|$ is the total number of edges in both graphs. The edit distance between a pair of valid runs is naturally defined as a minimum-cost sequence of elementary path insertions and deletions that transforms the first run into the second run, and preserves the validity of each intermediate run. The cost function used for each edit operation is compact yet general, allowing us to capture a variety of application-specific notions of distance, and depends on the start and end nodes as well as the length of the path. Experimental results demonstrate the scalability of our approach.

X. Acknowledgement

We would thank the anonymous reviewers for their valuable comments.

References

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

[2] "Business process execution language for web services." [Online]. Available: http://www.ibm.com/developerworks/library/specification/ws-bpel/

[3] Z. Bao, S. Cohen-Boulakia, S. B. Davidson, A. Eyal, and S. Khanna, "Differencing provenance in scientific workflows," University of Pennsylvania, Tech. Rep. MS-CIS-08-04, 2008. [Online]. Available: http://db.cis.upenn.edu/

[4] P. Wohed, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede, "Analysis of web services composition languages: The case of bpel4ws," in *ER*, 2003, pp. 200–215.

[5] K.-C. Tai, "The tree-to-tree correction problem," *J. ACM*, vol. 26, no. 3, pp. 422–433, 1979.

[6] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *J. ACM*, vol. 21, no. 1, pp. 168–173, 1974.

[7] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM J. Comput.*, vol. 18, no. 6, pp. 1245–1262, 1989.

[8] S. M. Selkow, "The tree-to-tree editing problem," *Inf. Process. Lett.*, vol. 6, no. 6, pp. 184–186, 1977.

[9] Y. Wang, D. J. DeWitt, and J. yi Cai, "An effective change detection algorithm for xml documents," in *ICDE*, 2003, pp. 519–530.

[10] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *SIGMOD*, 1996, pp. 493–504.

[11] S. S. Chawathe and H. Garcia-Molina, "Meaningful change detection in structured data," in *SIGMOD*, 1997, pp. 26–37.

[12] K. Zhang, R. Statman, and D. Shasha, "On the editing distance between unordered labeled trees," *Inf. Process. Lett.*, vol. 42, no. 3, pp. 133–139, 1992.

[13] K. Zhang, "A constrained edit distance between unordered labeled trees," *Algorithmica*, vol. 15, no. 3, pp. 205–222, 1996.

[14] A. K. A. de Medeiros, W. M. P. van der Aalst, and A. J. M. M. Weijters, "Quantifying process equivalence based on observed behavior," *Data Knowl. Eng.*, vol. 64, no. 1, pp. 55–74, 2008.

[15] W. M. P. van der Aalst, A. K. A. de Medeiros, and A. J. M. M. Weijters, "Process equivalence: Comparing two process models based on observed behavior," in *Business Process Management*, 2006, pp. 129–144.

[16] J. Freire, C. T. Silva, S. P. Callahan, E. Santos, C. E. Scheidegger, and H. T. Vo, "Managing rapidly-evolving scientific workflows," in *IPAW*, 2006, pp. 10–18.

[17] C. E. Scheidegger, H. T. Vo, D. Koop, J. Freire, and C. T. Silva, "Querying and creating visualizations by analogy," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1560–1567, 2007.

[18] J. Cheriyan, T. Jordán, and R. Ravi, "On 2-coverings and 2-packings of laminar families," in *ESA*, 1999, pp. 510–520.

[19] J. Valdes, R. E. Tarjan, and E. L. Lawler, "The recognition of series parallel digraphs," in *STOC*, 1979, pp. 1–12.

[20] H. W. Kuhn, "The hungarian method for the assignemnt problem," *Naval Research Logistics Quarterly*, no. 2, pp. 83–97, 1955.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press adnd McGraw-Hill, 2001.