Research Article

Detecting Difference between Process Models Based on the Refined Process Structure Tree

Jing Fan, Jiaxing Wang, Weishi An, Bin Cao, and Tianyang Dong

College of Computer Science and Software Engineering, Zhejiang University of Technology, Hangzhou 310023, China

Correspondence should be addressed to Bin Cao; bincao@zjut.edu.cn

Received 20 January 2017; Accepted 21 February 2017; Published 15 March 2017

A cademic Editor: Jaegeol Yim

Copyright © 2017 Jing Fan et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The development of mobile workflow management systems (mWfMS) leads to large number of business process models. In the meantime, the location restriction embedded in mWfMS may result in different process models for a single business process. In order to help users quickly locate the difference and rebuild the process model, detecting the difference between different process models is needed. Existing detection methods either provide a dissimilarity value to represent the difference or use predefined difference template to generate the result, which cannot reflect the entire composition of the difference. Hence, in this paper, we present a new approach to solve this problem. Firstly, we parse the process models to their corresponding refined process structure trees (PSTs), that is, decomposing a process model into a hierarchy of subprocess models. Then we design a method to convert the PST to its corresponding task based process structure tree (TPST). As a consequence, the problem of detecting difference between two process models is transformed to detect difference between their corresponding TPSTs. Finally, we obtain the difference between two TPSTs based on the divide and conquer strategy, where the difference is described by an edit script and we make the cost of the edit script close to minimum. The extensive experimental evaluation shows that our method can meet the real requirements in terms of precision and efficiency.

1. Introduction

A business process is a series of activities to reach a certain goal, such as approval for vacation, purchase order, or claims for travel expense. It is a workflow if a business process is automated by a supporting software system. Workflow management systems (WfMS) are used to define, execute, and monitor the workflows [1].

Advances in wireless network technology and the widespread use of hand-held terminals enable the realization of mobile workflow management systems (mWfMS), such as Exotica/FMDC [2] and WHAM [3]. A workflow is called mobile if it contains activities that are performed by actors with a mobile device (e.g., mobile phone or PDA). The typical users in mobile workflows are travelling salesman, service technicians, or maintenance engineers [4]. And the mWfMS sometimes have location constraints, which means the location of user should be also considered by mWfMS when allocating activities [5], that is, allocating an activity that has to be performed to the actor with the shortest travel path or at a certain location. Thus, it is necessary for workflow system to know about the current location of mobile users [6].

The development of mWfMS leads to large number of business process models, which are valuable assets. However, different locations for the same business process may result in different execution orders of activities. For example, one company has two offices: Hangzhou and Beijing, and Beijing is the headquarters. For some businesses of Hangzhou, the corresponding materials need to be sent to Beijing. After approval with signature, the materials will be sent back to Hangzhou; in this way, this business process can be successfully executed, while for the same business process of Beijing, there is no need to send the materials. Determining these differences is so meaningful that we can find out the reason of inefficiency during the execution of the business process. That is, detecting difference between process models is helpful for users to quickly locate the difference and rebuild the process model.

Graph edit distance (GED) [7] is a good way to measure the similarity (or dissimilarity) between graphs. Process
model is generally represented as a graph, while GED cannot be directly used to compute the difference between two process models, since GED is applicable for the graphs that only contain one type of node. Thus, it is not suitable for measuring the dissimilarity between the graphs with more than two kinds of nodes, such as Petri net based process model.

Vanhatalo et al. provide a feasible model to detect difference between process models. They parse a workflow graph to its corresponding refined process structure tree (PST), that is, decomposing a workflow graph into a hierarchy of subworkflows that are subgraphs with a single entry and a single exit [8]. But this model cannot be directly used; it is because a leaf node of PST represents an edge of its corresponding process model. In order to process the task nodes, that is, mapping the task nodes or generating the node based edit operations (such as node delete or node insert), we need to parse the edge of PST and get the task nodes. To conveniently obtain and process the nodes, we present a method to parse the PST to its corresponding task based process structure tree (TPST) by referencing the work of Cao et al. [9], where a leaf node of TPST is a task node and a nonleaf node represents a control flow structure of its corresponding process model. Therefore, the problem of detecting difference between process models is transformed to detect difference between their corresponding TPSTs.

Zhang et al. show that the problem of computing the edit distance between labeled trees is NP-complete [10]. In order to efficiently compute the difference between two TPSTs, we present an algorithm that uses the divide and conquer strategy to generate an edit script that we try to make its cost close to minimum. There are three steps to reach this goal: (1) two TPSTs that correspond with two process models are split into several fragments, and the mapped fragment pairs of two TPSTs are found; (2) the mapped nodes in each mapped fragment pair are determined; (3) the edit script of two TPSTs is generated based on the mapped nodes and fragment pairs. In this paper, the process models are modeled by Petri net [11].

Generally, we consider the difference as an edit script that consists of a set of edit operations. In this paper, we consider three kinds of edit operations: node delete, node insert, and fragment move. Node delete and node insert are the basic edit operations, which mean deleting and inserting a node; they are complementary. The reason why we consider fragment move is that it can be represented by a set of node deletes and inserts, while the move of fragment can be more understandable. For example, there are two process models that are modeled by Petri net in Figure 1: Process 1 and Process 2. The following edit script can transform Process 1 to Process 2: deleting nodes I and J, inserting node D, and moving a fragment that consists of E and F in Process 1 to the position where the same fragment in Process 2 is.

The contributions of this paper are highlighted as follows:

1. The implementation of parsing PST to TPST is performed in this paper, and we transformed the problem of computing difference between two process models into computing difference between two TPSTs.
2. The divide and conquer strategy is used to determine the mapped fragment pairs of two TPSTs and then the mapped nodes are determined in each mapped fragment pair, which can narrow the range of node mapping and improve the efficiency of difference detecting.
3. We design an algorithm to generate an edit script between two TPSTs, where we make the cost of this edit script close to minimum.
4. On the basis of the real data, we conduct extensive experiments to evaluate the performance of our algorithm in terms of precision and execution time.

The rest of this paper is organized as follows. The preliminaries are described in Section 2. Section 3 presents the method of parsing the PST to its corresponding TPST. Section 4 introduces the difference detection algorithm. The evaluation in terms of precision and efficiency is performed in Section 5. Section 6 introduces the related work and Section 7 concludes this paper.

2. Preliminaries

In this section, some preliminaries are introduced. Sections 2.1 and 2.2 introduce refined process structure tree (PST) and task based process structure tree (TPST), respectively. Section 2.3 presents some basic notions that are used in our algorithm. The basic edit operations are defined in Section 2.4, and edit script is described in Section 2.5. An edit script consists of a series of edit operations, which can...
transform one TPST to the other. Section 2.6 introduces the costs of edit operations and edit script.

2.1. Refined Process Structure Tree (PST)

**Definition 1** (blocks of a process model). A block of a process model is a nonempty submodel, which is defined as a quadruple $B = (Entry, Exit, V, E)$, where $Entry$ and $Exit$ are the single entry node and the single exit node of the block, respectively, $V$ is the node set, and $E$ is the edge set.

For example, in Figure 2, Process 1 is a process model and it is decomposed into 8 blocks: $F_1 \sim F_8$. The whole process model is block $F_1$, where the leftmost route node XOR is its entry node and the rightmost task node $B$ is its exit node. Thus, Process 1 can be represented by these nested and nonoverlapping blocks: $Process\ 1 = \{F_1(F_2(F_4,F_5),F_3(F_7,F_8))\}$, where $F_1$ contains $F_2$ and $F_3$, the nested blocks of $F_2$ are $F_4$, $F_5$, and $F_6$, and $F_3$ includes $F_7$ and $F_8$.

Generally, there are four kinds of control flow structures in a process model: **Sequence**, **Exclusive**, **Parallel**, and **Loop**. It is hard to compute the difference between two process models since a process model has several structures and these different structures can be assembled in an arbitrary way. Parsing a process model to its corresponding tree model is a good way to simplify this problem. It is because a tree structure is simpler than the structure of a process model and we can easily obtain all nodes and their relationships in a tree. Thus, we use the existing tree model called the refined process structure tree (PST) to represent a process model.

That is, a process model is decomposed into several blocks that consists of a single entry node and a single exit node [8], and these blocks are organized in a hierarchy way. The blocks are organized into a PST in a hierarchy way, and these blocks represent the route nodes of PST and the leaf nodes of PST corresponds with the edges of its corresponding edges.

For example, the process model $Process\ 1$ in Figure 2 can be parsed to its corresponding PST in Figure 3. The route node $F_5$ in PST represents the block named $F_5$ in the process model. The leaf nodes of $F_5$ in PST are $c$, $d$, and $g$, which correspond with three edges in the process model: $c$, $d$, and $g$, respectively. PST can represent the hierarchy relationships of blocks, but its corresponding leaf nodes are unordered. Therefore, it cannot represent the control flow structures of a process model well since some structures are ordered while some structures are unordered. That is, we need a semiordered tree model to represent a process model. Thus, we reference the work of Cao et al. [9], and improve PST to a new semiordered tree model called task based process structure tree (TPST).

2.2. Task Based Process Structure Tree (TPST). The differences between TPST and PST are listed as follows:

1. A leaf node of a TPST is a task node of its corresponding process model.
2. A nonleaf node, that is, a route node, can only be labeled as "**Sequence**, " **Loop," "**XOR," or "**AND," where "**XOR" and "**AND" represent the exclusive and parallel structures, respectively.
(3) TPST is a semiordered tree: if a nonleaf node is labeled as “Sequence” or “Loop,” its child nodes are ordered; otherwise, its child nodes are unordered.

Thus, a TPST can describe a process model that contains ordered and unordered control flow structures. As shown in Figure 4, there are two TPSTs that are parsed from Process 1 and Process 2 of Figure 1. The leaf nodes are the task nodes of the process models and the nonleaf nodes are the route nodes that represent the control flow structures. The labels of route nodes are marked beside the route nodes: for example, the label of the root node “Sequence” in Process 1 is “P0.”

2.3. Basic Notions

Definition 2 (fragment of a TPST). A fragment of a TPST consists of a route node and its adjacent child nodes. It is a tuple $F = (root, node, type)$, where

1. root is the root node of this fragment, which is usually a route node. Root has two kinds of type: (1) ordered, where child nodes of root form a sequence, and (2) unordered, where child nodes of root form a set with no order,
2. node is the node set of this fragment that contains root and its child nodes that are directly connected to root,
3. type is the type of this fragment that is the same as the type of its root node.

For example, Process 1 of Figure 5 can be split into 4 fragments: $F_1(Sequence, \{Sequence, XOR, A, AND, B\}, ordered)$, $F_2(XOR, \{XOR, I, Sequence, C\}, unordered)$, $F_3(AND, \{AND, H, J\}, unordered)$, and $F_4(Sequence, \{Sequence, E, F\}, ordered)$ and there are also 4 fragments in Process 2: $F_5(Sequence, \{Sequence, XOR, A, AND, B\}, ordered)$, $F_6(XOR, \{XOR, D, C\}, unordered)$, $F_7(AND, \{AND, H, Sequence\}, unordered)$, and $F_8(Sequence, \{Sequence, E, F\}, ordered)$.

Definition 3 (node mapping). There are two kinds of nodes in a TPST: task node and route node. Two task nodes can be mapped if their labels are identical. Since a route node is the root node of a fragment, whether two route nodes can be mapped depends on the similarity of their corresponding fragments. The more similar the two fragments are, the more possible the two route nodes can be mapped.

Definition 4 (similarity score of two fragments). Let $F_1$ and $F_2$ be two fragments that are from two different TPSTs; the similarity score of $F_1$ and $F_2$ is the ratio of their mapped nodes to total nodes, which can be computed according to the following equation:

$$\text{Sim}(F_1, F_2) = \frac{2 \times |\text{MapNodes}|}{|node_1| + |node_2|}.$$  \hspace{1cm} (1)

$|\text{MapNodes}|$ is the number of mapped nodes in two fragments $F_1$ and $F_2$. For two unordered fragments, their mapped nodes are the intersection set of their node set. For two ordered fragments, the nodes in their node sequence (Definition 5) meet the longest common node subsequence which are the mapped nodes. $|node_1|$ and $|node_2|$ are the node set size of $F_1$ and $F_2$, respectively.

Definition 5 (node sequence). Let $F_1$ be a fragment, and let $SN_1$ be its sequence of node labels that consists of the label of
root node and the task nodes’ labels from left to right. In the node sequence, we do not consider the route nodes that are leaf nodes.

For example, in Figure 6, the ordered fragments in Process 1 are \(\{F_1, F_4\}\), and \(\{F_2, F_5\}\) are ordered fragments of Process 2. The node sequences of \(F_1\) and \(F_2\) are \(SN_1 = \langle \text{Sequence}, A, B \rangle\) and \(SN_2 = \langle \text{Sequence}, A, B \rangle\), respectively.

Before introducing the longest common node subsequence (LCNS), we describe the related notations: subsequence, common subsequence, and longest common subsequence [12].

Definition 6 (subsequence). Let \(X = (x_1, x_2, \ldots, x_m)\) be a sequence; if there exists \(1 \leq i_1 < i_2 < \cdots < i_k < m\), s.t. \(Z = (z_1, z_2, \ldots, z_k) = (x_{i_1}, x_{i_2}, \ldots, x_{i_k})\), then \(Z\) is regarded as the subsequence of \(X\), which is marked as \(Z < X\).

For example, \(X = (A, B, C, B, D, A, B)\) and \(Z = (B, C, B, A)\), \(Z < X\).

Definition 7 (common subsequence). Let \(X\) and \(Y\) be two sequences, and \(Z < X, Z < Y\); then \(Z\) is the common subsequence of \(X\) and \(Y\).

Definition 8 (longest common subsequence, LCS). Let \(X\) and \(Y\) be two sequences, and \(Z < X, Z < Y\). \(Z\) is the longest common sequence of \(X\) and \(Y\) iff \(\forall\) subsequence \(Z'\) of \(X\) and \(Y\), \(|Z| \geq |Z'|\).

For example, for two ordered fragments \(F_1\) and \(F_5\) in Figure 5, the longest common node subsequence of their corresponding node sequences \(S_1\) and \(S_5\) is LCNS \((S_1, S_5) = \langle \text{Sequence}, XOR, A, AND, B \rangle\). In order to obtain the longest common node subsequence, the dynamic programming method is applied [13].

2.4. Edit Operations. In this paper, we use three kinds of edit operations to describe the difference between two TPSTs: node delete, node insert, and fragment move.

Node Delete: Delete(\(x\)). Node \(x\) can be directly deleted if it is the leaf node; otherwise, before deleting node \(x\), node \(x\)’s child nodes are connected to node \(x\)’s parent node.

For example, as shown in Figure 5, the leaf node \(A\) of Process 1 can be directly deleted: Delete(\(A\)). Before deleting node \(B_1\), its child nodes \(\{I, F_1, C\}\) are connected to its parent \(F_0\).

Node Insert: Insert(\(x, a, position\)). Node \(x\) is inserted as the child node of \(a\). If node \(a\) is unordered, the default position is 0 and \(x\) can be inserted in an arbitrary position. Otherwise, \(x\) is inserted as \(a\)’s position-th child node.

For example, as shown in Figure 5, to insert node \(D\) of Process 2, the first step is to determine the parent node that node \(D\) is going to be inserted in Process 1, that is, node \(B_1\). Since \(B_1\) is unordered, \(D\) can be inserted as \(B_1\)’s child node in an arbitrary position, that is, Insert(\(D, B_1, 0\))

Fragment Move: Move(\(f, a, position\)). A fragment \(f\) is moved as the position-th child fragment of node \(a\). The position is 0 if \(a\) is a unordered route node, which means that \(f\) can be inserted as \(a\)’s child fragment in an arbitrary position.

For example, in Figure 5, \(F_4\) and \(F_8\) are identical but they connect to different parent nodes: \(B_1\) and \(B_4\), respectively. Since \(B_1\) and \(B_4\) are not mapped nodes, \(F_4\) will be removed. The mapped node of \(B_4\) is \(B_2\), so \(F_4\) is removed as the child fragment of \(B_2\) in an arbitrary position since \(B_2\) is unordered, that is, Move(\(F_4, B_2, 0\)).

2.5. Edit Script. In this section, \(\text{TPST}_1\) refers to the TPST that the edit operations are applied, and \(\text{TPST}_2\) is the resulting TPST. Formally, suppose that \(e\) is an edit operation, a sequence \(E\) that consists of a set of operations, \(E = \{e_1, e_2, \ldots, e_m\}\), can transform \(\text{TPST}_1\) into \(\text{TPST}_2\), which is denoted by \(\text{TPST}_1 \rightarrow_\text{ed} \text{TPST}_2\). We call such a sequence an edit script of transforming \(\text{TPST}_1\) to \(\text{TPST}_2\), which is also the difference of \(\text{TPST}_1\) and \(\text{TPST}_2\).

For example, in Figure 5, the edit script of transforming Process 1 into Process 2 is editScript(\{Process 1 \rightarrow Process 2\} = \{Delete(I), Delete(F), Insert(D, B_1, 0), Move(F_4, B_2, 0)\}).

2.6. Cost Model. There exist a large number of edit scripts that can transform one TPST to another TPST. For example, in Figure 5, two kinds of edit scripts can be applied to transform Process 1 into Process 2: (1) Process 1 \rightarrow_{\text{ed}1} Process 2 = \{Delete(I), Delete(F), Insert(D, B_1, 0), Move(F_4, B_2, 0)\}. (2) Process 1 \rightarrow_{\text{ed}2} Process 2 = \{Delete(I), Delete(E), Delete(F),
Delete($p_0$), Delete($f$), Insert($D, B_1, 0$), Insert($P_3, B_2, 0$), Insert($E, P_3, 0$), Insert($F, P_3, 0$)} For these two edit scripts $E_1$ and $E_2$, which one is better? Or there are thousands of edit scripts that can be applied to convert one TPST to another, which edit script is the best? In order to solve this problem, the cost of edit script is proposed to evaluate whether an edit script is good or not. The smaller the cost is, the better the edit script is.

In this paper, we adopt a simple cost model that the cost of each edit operation is equal to 1; for example, Cost_Delete($x$) = Cost_Insert($x$) = Cost_Move($f$) = 1, which represents that deleting a node $x$, inserting a node $x$, and moving a fragment $f$, respectively, have a unit cost.

Then the cost of an edit script $E$ is the sum of all the costs of its corresponding edit operations $e_1, e_2, ..., e_m$; that is, $C(E) = C(e_1) + C(e_2) + ... + C(e_m)$. For example, as mentioned above, $C(E_1) = 4$ and $C(E_2) = 9$, so the edit script $E_1$ is better than $E_2$.

3. Parsing PST to TPST

There exists a method to parse a process model to its corresponding refined process structure tree (PST); that is, a process model is decomposed into a hierarchy of subprocess models with a single entry and a single exit. However, it is inconvenient to compute difference between process models by their corresponding PSTs. It is because a leaf node of PST represents an edge of its corresponding process model. The difference between two process models is described by using the edit operations that include node operations and fragment operation. In order to describe the node operations, we need to further parse out the task nodes through the edges. For convenience, we parse the task nodes from PST in advance and arrange the task nodes and route nodes to form a task based process structure tree (TPST).

TPST is a task based process description, where leaf nodes are task nodes and nonleaf nodes represent control flow structures. The edit operations can be described in the TPST more intuitively by comparing with PST; for example, we can observe which nodes are inserted or deleted or which control flow structures are moved. Besides, TPST is more convenient to design difference detection algorithm because we need to frequently process the task nodes.

Main Idea. The main idea of parsing PST to TPST is parsing the task nodes and route nodes from PST; then all parsed nodes are arranged in a tree, where the structure of TPST is the same as the PST. In terms of implementation, there are three phases to parse a PST to its corresponding TPST: (1) parsing a node of PST to a node of TPST; each node in a PST is converted to be a TPST node, where a leaf node of TPST is a task node and a nonleaf node represents a control flow structure; (2) constructing the TPST: all nodes in a TPST are organized into a tree according to the hierarchy structure of PST; and (3) checking the TPST: the route node needs to be deleted if its type is “Sequence” and it has only one child node. The purpose of this phase is to better understand TPST since it is not necessary to describe a single task node by using a “Sequence.”

```
Input: PST pst
Output: the root node of TPST tpst
(1) Map < PSTNode, TPSTNode > map = 0;
(2) for each node $p$ of pst in level-order do
(3) TPSTNode $f = transToTPSTNode(p);
(4) map.put($p, f$);
(5) end
(6) map.put(pst.exit, transToTPSTNode(pst.exit));
(7) root_tpst = constructTPST(pst, map);
(8) root_tpst = checkTPST(tpst);
(9) return root_tpst;
```

Algorithm 1: Converting PST to TPST.

Algorithm. Algorithm 1 gives the overall pseudo code for parsing PST to TPST, where the input is a PST that corresponds with a process model, and the output is the root node of its corresponding TPST. Firstly, a map named $map$ that records the mapping relationship between PST nodes and TPST nodes is initialized; that is, we can obtain the relationship as to which TPST node is transformed by which PST node from this map (line (1)). Then each node in PST is iterated and transformed to a TPST node by using the function $transToTPSTNode$ and the mapped node pair: PST node and TPST node are saved into $map$ (line (2)–line (5)). Since we parse the task nodes by obtaining the entry nodes of the block in PST and the exit node of the total process model has not been parsed, we parse this exit node in the end (line (6)). Then, a TPST is constructed based on the parsed TPST nodes, where the hierarchy structure is arranged by referencing its corresponding PST. This phase is implemented by the function: $constructTPST$, which outputs the root node of TPST (line (6)–line (7)). In order to make the TPST more understandable, we delete the “Sequence” node that directly connects with its single task node, which is operated by the function: $checkTPST$ (line (8)).

3.1. Phase I: Parsing a Node of PST to a Node of TPST

Main Idea. There are two types of nodes in a PST: leaf node and route node. A leaf node represents an edge, and a route node is a block that contains a set of edges of the original process model. To parse a PST node to its corresponding TPST node, we need to handle different types of nodes in different ways. For a leaf node of PST, we obtain the entry node of its corresponding block and we save it if this entry node is a task node; otherwise, the entry node is discarded. For the route nodes in PST, we just save it.

Algorithm. Algorithm 2 gives the pseudo code for the first phase, where the input is a node of PST and the output is the corresponding TPST node. Firstly, if this PST node is a leaf node, we get the entry node of its corresponding edge (line (1)–line (2)). There are two possibilities of the entry node: route node or task node. The entry node will be abandoned if it is a route node since we just need the task node (line (3)–line (4)). While if the entry node is a task node, we save it as a TPST node by copying its type and label (line (5)–line (9)).
Then, if this PST node is a route node, it is saved by copying its name and type, where there are two types: ordered and unordered (line (10)–line (14)).

Example. Figure 3 is a PST of Process 1 in Figure 1; we can observe the parsing result of this phase in Figure 4. The leaf nodes, $a, b, c, h, j, k$, and $n$, are abandoned since the entry nodes of their corresponding edges are route nodes. While the rest leaf nodes, $d, e, f, g, i, l$, and $m$, are remained because their corresponding entry nodes, $E, I, C, F, A, H$ and $J, B$, are task nodes that correspond with the leaf nodes in the TPST. The nonleaf nodes, $F_1$–$F_5$ in PST, are unchanged, which correspond with the route nodes $P_0, B_1, B_2, P_2, P_1, P_3, P_4, P_5$, respectively.

### 3.2. Phase 2: Constructing the TPST

**Main Idea.** The overall structure between PST and TPST is the same; that is, their corresponding process model is decomposed into the same subprocess models and these subprocess models are organized into the same way. The difference between them is that PST is an edge based process and TPST is a task based process; that is, from the route nodes of PST, we can observe which edges a block contains and we can observe that which task nodes a fragment has from a route node of TPST. However, the organization way between route nodes in PST is the same as in TPST. So after obtain all nodes of TPST, we construct the TPST by referencing that of PST.

**Algorithm.** Algorithm 3 gives the pseudo code for the second phase, where the input is the PST and the map that records the mapping relationship between PST nodes and TPST nodes, and the output is the root node of the TPST. Firstly, for each route node $p$ in PST, its corresponding route node $t$ of TPST is found (line (1)–line (3)). If $p$ is a root node then $t$ is also a root node (line (4)–line (6)). Then, the child nodes of $p$ are obtained, which need to be ordered according to the original process model if the type of $t$ is ordered (line (9)–line (12)). It is because PST cannot reflect the control flow structures that are ordered and unordered. To construct the ordered control flow structures in TPST, we first need to rank the ordered blocks in PST, and then we construct the ordered structure in TPST by referencing that in PST. According to the parent-child relationships in PST, that is, a node has which child nodes and which node is the parent node of this node, the TPST also has this kind of relationships (line (13)–line (18)). After all node relationships in TPST have been constructed, the root node is returned (line (20)).

**Example.** The original process model is Process 1 in Figure 1, Figure 3 is its corresponding PST, and Figure 4 is the resulting TPST by Phase 2. We can observe that the organization way between route nodes in PST is the same as TPST.

### 3.3. Checking the TPST

**Main Idea.** The main idea of this phase is to optimize the structure of TPST and better understand it. In the TPST that we obtain from the first two phases, there exist many sequence route nodes with only one task node, which need to be deleted. The reasons why we delete them are listed as follows. (1) Generally, the sequential relationship is used to describe the relationship between more than two nodes, and it is not suitable for a single node. (2) Once the task node is to be deleted, its corresponding sequence route node is also to be deleted, which leads to more edit operations.

**Algorithm.** Algorithm 4 gives the pseudo code for this phase, where the input is the root node of TPST, and its new root node is output. For each route node in TPST, if its type is “Sequence” and it has only one child then it needs to be deleted (line (1)–line (2)). Firstly, we get the child node and...
parent node of this route node (line (3)-line (4)). Then, this route node is deleted; that is, the only child node of this route node is directly connected with its parent node (line (5)-line (6)).

Example. The parsing result of Phases 1 and 2 is shown in Figure 2, and after Phase 3 we obtain the TPST that is shown in Figure 4. We can observe that all sequence route nodes with only one task node are deleted, for example, $P_2$, $P_3$, $P_3$, and $P_5$, and their parent nodes are directly pointed to their single task nodes, respectively. For example, in Figure 4, $P_2$ is the route node that needs to be deleted, $B_1$ is its parent node, and its child node is $I$. After $P_2$ is deleted, its parent node $B_1$ is the parent node of $I$.

4. Difference Detection

In this paper, the difference between two process models is described by using the node operations and fragment operation, which is shown in Section 2.4. Thus, TPST can reflect the difference in an understandable way, that is, which nodes and fragments are changed. Besides, TPST is more convenient for us to design the difference detection algorithm. Therefore, we transform the problem of detecting difference between process models into detecting difference between their corresponding TPSTs.

Main Idea. The main idea is that we decompose the two TPSTs into several fragments that are defined in Definition 2; the difference is computed based on the mapped fragments and the mapped nodes. In terms of implementation, there are three steps to compute the difference between two TPSTs: (1) fragment mapping: decomposing two TPSTs into several fragments and determine their mapped fragment pairs; (2) node mapping: finding out the mapped nodes based on the mapped pairs of fragments; (3) edit script generation: generating the edit script between two TPSTs based on the mapped fragments and nodes.

Algorithm. The overall pseudo code is shown in Algorithm 5, where the inputs are two TPSTs that correspond with two process models, and the output is an edit script which is regarded as their difference. Firstly, two TPSTs are decomposed into several fragments, respectively; the optimal fragment mapping combination that consists of a series of mapped fragment pairs is found by Fragment_Mapping (line (1)). Then, the mapped nodes in each mapped fragment pairs are found by Node_Mapping (line (2)). Finally, the edit script is generated based on the mapped nodes and fragment pairs by EditScript_Generation (line (3)).

Next we present the implementation of the three phases in Sections 4.1–4.3, and the complexity of difference detection algorithm is given in Section 4.4.

4.1. Phase 1: Fragment Mapping

Main Idea. Directly detecting difference between two trees is complicated; for example, computing the edit distance between two trees is NP-complete. Thus, we adopt the divide and conquer strategy to reduce the complexity of this problem, which can improve the efficiency of the difference detection. We first decompose two TPSTs into several fragments. Then, the similarity scores of all possible mapped fragment pairs from different TPSTs are calculated according to the fragment mapping rules, which are defined in Definitions 9 and 10. Next, a table called Fragment_Mapping_Table (Definition 11) is created based on these similarity scores, and the mapping fragment combination with the highest
sum similarity score is found, that is, the optimal mapping fragment combination.

Definition 9 (unordered fragment mapping). Let $F_1$ and $F_2$ be two unordered fragments; their similarity score is computed according to (1), where the mapped nodes are the intersection nodes of $F_1$ and $F_2$; that is, $\text{MapNodes} = \text{node}_1 \cap \text{node}_2$.

For example, there are two unordered fragments $F_1$ and $F_2$ in Figure 6(a). $|\text{MapNodes}| = |\text{node}_1 \cap \text{node}_2| = |\{\text{XOR}, A, B, C\}| = 4$, $\text{Sim}(F_1, F_2) = 1$. That is to say, $F_1$ and $F_2$ are identical.

Definition 10 (ordered fragment mapping). Let $F_1$ and $F_2$ be two ordered fragments and let $\text{SN}_1$ and $\text{SN}_2$ be their corresponding node sequences, respectively. The similarity score of $F_1$ and $F_2$ is computed according to (1). The mapped nodes are the longest common node subsequence of $\text{SN}_1$ and $\text{SN}_2$; for example, $\text{MapNodes} = \text{LCNS}(\text{SN}_1, \text{SN}_2)$.

In Definition 10, we do not consider the mapping for the leaf nodes that are route nodes. It is because a route node represents its corresponding fragment, so the similarity score of two route nodes is equal to the similarity of their corresponding fragments. In order to compute the similarity score of two route nodes, we need to continue to iterate the TPSTs to find out their corresponding fragments. Once there exists route node in the found fragment, we need to continue to iterate the TPSTs and it stops until the leaf nodes of a fragment are all task nodes. In this way, the computing time increases dramatically. To improve the efficiency, we just consider the mapping for the leaf nodes that are task nodes when mapping two fragments.

For example, in Figure 6(b), $F_3$ is the ordered fragment of Process 1 and $F_5$ and $F_6$ are the ordered fragments of Process 2 in Figure 5. The mapped nodes of $F_1$ and $F_5$ are $\text{MapNodes} = \text{LCNS}(\text{SN}_1, \text{SN}_2) = \{\text{Sequence}, A, B\}$; thus, their similarity score is $\text{Sim}(F_1, F_5) = (3 \times 2)/(5 + 5) = 0.6$. For $F_1$ and $F_6$, their mapped nodes are $\text{MapNodes} = \text{LCNS}(\text{SN}_1, \text{SN}_6) = \{\text{Sequence}\}$ and their similarity score is $\text{Sim}(F_1, F_6) = (1 \times 2)/(5 + 3) = 0.25$.

**Algorithm.** In this phase, the inputs are two TPSTs $t_1$ and $t_2$ and the output is the optimal fragment mapping combination $M_F$. Firstly, the route nodes of two TPSTs are determined by tree traversal. For each route node, it and its adjacent child nodes form a fragment. Accordingly, the type of fragment is determined according to the type of the route node. Then we get two ordered fragment sets and two unordered fragment sets from two TPSTs: $\text{order}_F_{t_1}$ and $\text{Unorder}_F_{t_1}$ and $\text{order}_F_{t_2}$ and $\text{Unorder}_F_{t_2}$. Next, the $\text{Order}_F_{t_1}$ and $\text{Unorder}_F_{t_1}$ are initialized, respectively, where $\text{Order}_F_{t_1}$ records the similarity scores of all possible pairs fragment that one is from $\text{order}_F_{t_1}$ and the other is from $\text{order}_F_{t_2}$. $\text{Unorder}_F_{t_1}$ is created in the same way. Finally, to find the optimal ordered or unordered fragment mapping combinations that have the maximum sum of the similarity score, we turn to the Hungarian algorithm [14, 15]. The
union of ordered and unordered fragment mapping combinations is the final result of mapped fragments between two TPSTs.

**Example.** Figure 7(b) is the initial results of Order_fragmentMT and Unorder_fragmentMT for two TPSTs of Figure 5. After using the Hungarian algorithm twice, we obtain the optimal ordered mapping combination, \{(F_1, F_5), (F_2, F_6), (F_3, F_7)\}, and the optimal unordered mapping combination, \{(F_2, F_6), (F_3, F_7)\}. Finally, the overall optimal fragment mapping combination is \(M_F = \{(F_1, F_5), (F_2, F_6), (F_3, F_7)\}\).

When mapping two fragments, we just need to consider the nodes in the fragments rather than all nodes of the process model. For example, when computing the similarity score of the nodes in the fragments rather than all nodes of the process model, \(\{F_1, F_5\}\), and \(\{F_2, F_6\}\), and \(\{F_3, F_7\}\) are considered rather than considering all the nodes of Process 1 and all the nodes of Process 2. In this way, the computing space is dramatically decreased, and the mapping time is accordingly reduced.

4.2. Phase 2: Node Mapping

**Main Idea.** In this paper, we define two types of node operations: node delete and node insert. Thus, after we determine the mapped nodes, we can judge to which node operation the remaining nodes belong to. The main idea is that we find the mapped nodes in every mapped pair of fragments, and the mapped nodes of two TPSTs are the union of all mapped nodes in all mapped fragment pairs. In terms of implementation, different strategies are adopted to find the mapped nodes in different types of mapped fragments. For unordered fragment, the nodes with the same labels are mapped. For ordered fragment, the nodes that meet the LCNS are mapped. The union of mapped nodes in unordered fragments and ordered fragments are the mapped nodes of two TPSTs.

**Example.** In Figure 5, \((F_1, F_5)\) is the mapped fragment pair and their mapping detail is shown in Figure 6(b), where two nodes are mapped if there exists a line. Firstly, the pair of root node (Sequence, Sequence) is mapped since they have the same type and label. The mapped leaf nodes of \(F_1\) and \(F_5\) are \{(A, A), (B, B)\} since they meet the LCNS. So the mapped nodes of \(F_1\) and \(F_5\) are \{(Sequence, Sequence), (A, A), (B, B)\}. After all mapped nodes in all mapped pairs of fragment, \{(F_1, F_5), (F_2, F_6), (F_3, F_7)\}, are found, we obtain the mapped nodes of Process 1 and Process 2 is \{(P_0, P_0), (B_1, B_1), (B_2, B_2), (P_1, P_0), (A, A), (B, B), (C, C), (E, E), (F, F)\}.

4.3. Phase 3: Edit Script Generation. Computing difference between two process models can be roughly divided into two steps: (1) determining the similar parts, which means that these parts are unchanged between two process models; (2) describing the different parts based on the similar part, where the edit script is used. So far, we have determined the similar parts between two TPSTs, that is, the mapped fragments and nodes. Next, the difference will be computed and described.

**Main Idea.** The goal of this phase is to generate an edit script that can transform the original TPST_1 into the resulting TPST_2. The main idea is that we determine the operation types for the different parts in two TPSTs. For unmapped nodes, the node operation type, node delete or node insert, is determined. For mapped fragments, they need to be moved if they are in different positions. In terms of implementation, there are three steps: (1) deleting nodes: the unmapped nodes in TPST_1 need to be deleted; (2) inserting nodes: the unmapped nodes in TPST_2 need to be inserted; and (3) moving fragments: the mapped fragments with the different positions need to be moved.

The reason why we need to move a fragment is that we have not considered the position of the fragment when mapping the fragments, which may lead to the result that two fragments with a different position can be mapped. In some existing methods, they map the nodes of two trees by using the strategy called top-down [16] or maximum common subtree [9]; for example, a pair of child nodes can be mapped if and only if their corresponding parent nodes have been mapped. In this way, two identical fragments with different position cannot be mapped. So all nodes in one fragment are deleted and all nodes in the other fragment are inserted, which results in more edit operations. Thus, in our paper, we first map two corresponding fragments and then judge whether they have the same position; if their positions are different then the fragment needs to move.

**Algorithm.** The pseudo code of this phase is shown in Algorithm 6, where the inputs are two TPSTs: \(t_1\) and \(t_2\), their mapped node set \(M_N\) and their mapped fragment set \(M_F\). The output is the edit script editScript that can transform \(t_1\) into \(t_2\). There are mainly three steps to generate the edit script for \(t_1\) and \(t_2\). (1) Node deletion: the nodes of \(t_1\) are iterated level by level, the current node \(x\) is deleted once \(x\) does not belong to \(M_N\), and the corresponding operation Delete(x) is added to editScript (line (2)–line (7)). (2) Node insertion: \(t_2\)’s nodes are iterated level by level, and the current node \(y\) is inserted at the same position in \(t_1\) if \(y\) does not appear in \(M_N\). Firstly, the parent node of \(y\) that \(y\) is going to be inserted in \(t_1\): Parent(y) is determined. Then if Parent(y) is unordered, the insert position is default 0, and if it is ordered, the inserted position is to be determined, that is, which position \(y\) is going to insert as the child node of Parent(y). The corresponding edit operation is recorded as Insert(y, Parent(y), position) and added to editScript (line (8)–line (13)). (3) Fragment move: for each mapped fragment pair \((f_1, f_2)\) of \(M_F\), it is not necessary to move if the positions of \(f_1\) and \(f_2\) are identical; for example, the parent node pair of \(f_1\) and \(f_2\), (parent\(_f_1\), parent\(_f_2\)), belongs to \(M_N\). Otherwise, \(f_1\) needs to be moved to the position where \(f_2\) is; for example, \(f_1\)’s new parent node is the mapped node of \(f_2\)’s parent node in \(t_1\). The same as inserting a node, we need to consider the position when moving a fragment (line (14)–line (20)). Thus, Move(f_1, parent(f_1), position) is added to editScript, where parent(f_1) represents the root node that \(f_1\) is going to move.

**Example.** As shown in Figure 5, \(F_4\) and \(F_8\) are two fragments in Process 1 and Process 2, respectively, and they are identical.
but with the different positions. So $F_4$ needs to be moved as the child fragment of $B_2$, because $B_2$ is the mapped node of $F_k$ that is the parent node of $F_k$. That is, the edit operation is $\text{Move}(F_4, B_2, 0)$. The edit script of transforming Process 1 into Process 2 is $\text{editScript}(\text{Process 1, Process 2}) = \{\text{Del}(I), \text{Del}(J), \text{Insert}(D, B_1, 0), \text{Move}(F_4, B_2, 0)\}$. 

4.4. Complexity Analysis. In this section, we analyze the time complexity of our algorithm. Let $n_1$ and $n_2$ be the number of two models' nodes, let $f_1$ and $f_2$ be the size of fragments (i.e., the number of nontask nodes), and let $n$ be the average number of nodes of fragments in two TPSTs. In Phase 1, that is, fragment mapping, we first obtain two fragment sets of two TPSTs by hierarchical traversal, which achieves $O(n_1 + n_2)$ complexity in execution time; then the Hungarian algorithm is used to find the optimum fragment mapping combination, which has the worst time complexity of $O((\min\{f_1, f_2\})^3)$. In Phase 2, that is, node mapping, we first iterate all pairs of mapped fragments, the time of mapping each pair of fragments is $O(n_2)$, and the total time of this phase is $O((\min\{f_1, f_2\}) \times n_2)$. In Phase 3, edit script generation, all nodes of two TPSTs and their mapped fragments are iterated to generate the edit script, which spends $O(n_1 + n_2 + \min\{f_1, f_2\})$. In summary, Phase 2 spends the most time of the overall algorithm, and the total time complexity is $O((\min\{f_1, f_2\}) \times n_2^2)$.

5. Experiment

In this section, we evaluate the performance of our algorithm in terms of precision and efficiency. All experiments were evaluated on a machine with Intel(R) Xeon(R) CPU E5-2637, 3.50 GHz processor and 8 GB RAM, running JDK1.7, and Windows 7.

5.1. Dataset. The dataset that we used consists of two parts. (1) Based on the existing IBM dataset [17] we choose 10 process models as the base process models and modify them to their corresponding 9 variants by removing/inserting some nodes and some edges. In this way, we build a process repository with 100 process models. Table 1 shows the basic information of this process repository: minimum, maximum, and average number of place, task, and edge. (2) We choose 4-process model from the IBM dataset as the base process models, where they contain the following four control flow structures, respectively: Sequence, AND (parallel), XOR (exclusive), or AND + XOR (combining parallel with exclusive structures). For each base, we make some modifications on it to obtain its 5 variants without changing their structure. The modifications consist of deleting, inserting nodes, and moving fragments from the base, which are recorded as the standard edit script (SES). In this way, we create four repositories: Sequence, AND, XOR, and AND + XOR, where each repository contains 6 process models (1 base process model and its 5 variants) with the same structure. Table 2 shows the task node number of every process model in each repository, where the base process model has 160 task nodes.
and its 5 variants contain 140, 120, 100, 80, and 60 task nodes, respectively.

5.2. Quality Study. The second dataset is used to evaluate the precision of our algorithm, where the precision is computed by comparing the result of the algorithm with the standard edit script (SES). At first, we investigate the impact of varying task node size and fixing the structure on precision. Then the average precision is evaluated by fixing the structure.

5 edit scripts of (base, variant_1) (1 ≤ i ≤ 5) are computed, respectively, in each repository: Sequence, AND, XOR, and AND + XOR which are compared to the SESs and the ratios are plotted in Figure 8(a); we observe the impact of varying task node on precision by fixing the structure. The precision of computing difference between Sequence process models is 100%, while it is lower between the process models with other control flow structures. The reason is that there exists only one fragment in a Sequence process model, so the optimal mapped fragment pair between two Sequence process models can be definitely determined, and then the optimal mapped node pairs are found. However, a process model with XOR, AND, or AND + XOR structure has more than one fragment. For one fragment \( f_j \) of a process model Process 1, there exist several fragments in the other process model Process 2 that have the same similar score with \( f_j \), which leads to the Hungarian algorithm randomly choosing one fragment of Process 2 to map with \( f_j \).

Taking Figure 8(c) as an example, the mapped node set of Process 1 and Process 2 is \( M_{F1} = \{(A, A), (B, B), (D, D), (E, E), (F, F), (G, G), (H, H), (I, I), (P_1, P_1), (P_2, P_1), (P_3, P_4), (P_5, P_3), (B_1, B_3), (B_2, B_3)\} \) and \( M_{F2} = \{(F_1, F_3), (F_2, F_3)\} \). Their corresponding mapped node pairs are \( \{(B_1, B_3), (B_2, B_3)\} \) and \( \{(B_4, B_4), (B_5, B_5)\} \), respectively. However, we have not made a strategy to select a better mapping fragment combination between several optimal ones; thus, which fragments are selected to map with \( B_1 \) and \( B_2 \) are unknown.

Overall, in Figure 8(a), the four tests, Sequence, AND, XOR, and AND + XOR, show the similar trends that with the decrease of the task number, the precision increases. It is because the mapped fragment set of two process models becomes smaller, which leads to the lower possibility that more than one optimal mapping fragment combination occurs. However, the precision for detecting difference between Sequence process models remains unchanged. The reason is mentioned above. For variant_2 in XOR repository and variant_3 in AND repository, the reason why their corresponding precisions decrease is that there exist many optimal mapping fragment combinations, and Hungarian algorithm outputs which one is unknown.

Figure 8(b) shows the average precision of four repositories with different complexity of structures and the overall average precision is higher than 70%. In summary, the precision of our algorithm is getting lower with the control flow structure getting more complicated, while it can get a better precision in the general scenario.

5.3. Efficiency Study. In this section, we conduct three kinds of experiments to evaluate the efficiency. (1) The execution time of parsing PST to TPST is evaluated, where we study the impact of changing the number of place, task, and edge, respectively, on the parsing time. (2) The execution time of detecting difference between two process models is studied, where these two process models have the similar complexity. We investigate the impact of changing one element (i.e., place, task, or edge number) of one process model on execution time by fixing the other process model. (3) The execution time of difference detection is evaluated by phases, which can be merged into two phases: node mapping that consists of mapping fragments and finding mapped nodes and edit script generation. We study the impact of varying task number on the execution time of different phases by fixing the structure.

In the first experiment, we first choose 3 sets of process models from the first part dataset. Each set contains 5 candidate process models: \{\{place 1, \ldots, place 5\}\}, \{\{task 1, \ldots, task 5\}\}, \{\{edge 1, \ldots, edge 5\}\}, where their place, task, and edge numbers increase progressively. Then we separately choose three target models for each set, where these three target models have the same element number (place, task, or edge number) to the first, third, and fifth models of each set. In this way, the three sets of process models are \{\{target 1, \ldots, target 3, place 1, \ldots, place 5\}\}, \{\{target 1, \ldots, target 3, task 1, \ldots, task 5\}\}, and \{\{target 1, \ldots, target 3, edge 1, \ldots, edge 5\}\}. In every set, the difference between a target model and a candidate model is computed; that is, \{\{target_i, candidate_j\} (1 ≤ i ≤ 3, 1 ≤ j ≤ 5)\} and the execution time of difference detection is studied.

In Figures 9(a), 9(b), and 9(c), the impacts of varying place number, task number, and edge number are studied, respectively. Overall, these three tests under different varied factors show the similar trends that our method can efficiently parse the PST to TPST in milliseconds.

With the increase of the number of place, task, or edge, the parsing time increases correspondingly. The most significant factor for impacting the parsing time is task number, and the second significant factor is place number. It is because TPST has two kinds of nodes: task node and route node. The task nodes in the process model are still the task nodes of its corresponding TPST, but the place nodes in the process model have been removed or transformed to the route nodes, so varying the number of place has smaller effect on parsing time. Varying edge numbers has the smallest impact on

<table>
<thead>
<tr>
<th>Control flow structure</th>
<th>Base</th>
<th>Variant_1</th>
<th>Variant_2</th>
<th>Variant_3</th>
<th>Variant_4</th>
<th>Variant_5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence/AND/XOR/AND + XOR</td>
<td>160</td>
<td>140</td>
<td>120</td>
<td>100</td>
<td>80</td>
<td>60</td>
</tr>
</tbody>
</table>

Table 2: The second part of dataset.
parsing time. It is because one edge connects two nodes; it does not change the number of nodes, but the complexity of process model also increases with the increase of edge number. In this way, it also leads to the increase of parsing time.

The dataset of the second experiment is the same as the first experiment. Figures 10(a) and 10(b) show the impact of varying place number and task number on the overall execution time, respectively, where the time increases with the increase of place or task number. There are three reasons. (1) The increase of place or task number results in the increase of fragment number, which leads to the increase of times of computing similarity score as well as its execution time. (2) The increase of place or task number results in the increase of fragment size but no new fragments, which leads to the increase of the execution time of computing similarity score between two fragments. (3) The increase of place or task number results in the increase of execution time of generating edit script.

In Figure 10(b), the execution time of computing the difference between the target model with 168 tasks and the candidate model with 99 tasks increases dramatically, while the increase of execution time is not significant for the target model with 20 tasks and the candidate model with 99 tasks. It is because the target model with 20 tasks has few fragments; even though the candidate model contains many tasks and fragments, the time of computing similarity score is small, which will not dramatically increase the execution time.

In the third experiment, we use the second part of dataset to compute the difference between the base and its variants: (base, variant$_i$) ($1 \leq i \leq 5$) in every repository. Then the impact of varying task number on the execution time is investigated by fixing the structure: Sequence, AND, XOR, or AND + XOR. In Figures 11(a), 11(b), 11(c), and 11(d), we observe that the execution time of the second phase (EditScriptGeneration) increases dramatically and the structure is getting more complicated, while it does not increase dramatically for the first phase (Node Mapping). The reason is that the fragment number becomes bigger with the structure getting more complicated, which leads to the increase of computing similarity score times in Node Mapping. The execution time of the second phase is based on the node number and mapped fragment number of two process models. Since the task number of the four base models is the same, and the fragment number is so small that it has few influences on the execution time, the execution time of generating edit script does not change significantly.

In Figures 11(a)–11(d), the execution time of each phase all decreases with the decrease of task node while keeping
the structure. The reason is analyzed in the following: taking Figure 11(a) as an example (the other three results are similar), the times of finding mapped nodes reduce with the decrease of task number. Besides, the decrease of task number results in the decrease of fragment number or fragment size, which can correspondingly lead to the decrease of computing similarity score times.

In conclusion, on the one hand, the execution time increases with the place, task, or edge number is getting larger. In particular, the following case can lead to the significant increase of execution time: changing the place, task, or edge number results in the change of structure. On the other hand, the structure is getting more complicated resulting in the increase of execution time. We can deem that our algorithm can meet the efficiency requirements of the real application scenarios according to the results of the efficiency study.

6. Related Work

The current work of difference detection can be classified into three categories. The first category is to transform the process models into their corresponding tree models, and then the difference detection is based on the tree models. Cao et al. parse the process models into their corresponding process structure tree (PST), and the difference of two process models is obtained by computing the difference between two PSTs, where they use the maximum common subtree to determine the mapped nodes [9]. But this paper does not present the implementation of parsing process models to PSTs, and finding the mapped nodes by maximum common subtree may miss other identical or similar parts of two process models.

The second category of methods performs difference detecting directly based on process models. The most related work is the method of detecting and resolving process model difference in the absence of a change log. Firstly, a process model is decomposed into several fragments with a single entry and a single exit (SESE). Secondly, the mapped nodes and the SESE fragments of two process models are determined. Finally, based on the mapped nodes and fragments, the difference of the fragments is calculated [18]. The difference between this work and our work is that we consider the similar mapping of fragments; in this way, more similar parts of two process models can be determined. Liu et al. present a method to detect the syntactic differences rather than structural differences between process models [19]. Dijkman makes a classification for the differences between process models that frequently occurred [20]. He also proposes a method to diagnose the difference between EPC models, where the exact position and type of the difference are returned [21]. Liu et al. present the definition of the structure difference of process model, and they prove that there exists this kind of differences in reality [22]. Yan et al. design an algorithm to detect the behavior difference between two process models, which achieves higher efficiency compared with the previous work [23]. Li et al. compare two process models by using high
level changes, like “move,” in order to reduce the efforts and make the difference more understandable [24].

The last category is difference detection between structure documents, such as XML documents, where the documents are usually represented in a tree structure. Peters surveys the XML change detection algorithms, where most algorithms only consider three kinds of edit operations: node insertion, deletion, and update, and some certain properties of these algorithms are also described [25]. Al-Ekram et al. propose an algorithm with $O(n^2)$ runtime to detect changes between two versions of an XML document. They use the tree fragment mapping technique to achieve the goal of optimizing the runtime of mapping nodes and minimizing the size of edit script [26]. Cobéna et al. detect difference between XML data by trying to match more nodes. Firstly, the unchanged subtrees are determined. Based on these unchanged subtrees, more mapped nodes are found by considering ancestors and descendants of matched nodes [27]. Wang et al. use XHash and the notion of node signature to compute the difference of two XML documents that are represented to unordered trees [28]. Finis et al. propose the random walks similarity measure to find similar subtree in hierarchical data that can be represented to both ordered trees and unordered trees [29].

7. Conclusion

Nowadays, mobile workflow management system (mWfMS) is popular since the widespread use of mobile devices, which leads to large number of process models. Different locations for one business goal may result in different process models. This paper aims to detect difference between these process models. In order to solve this problem, we parse a process model to its corresponding task based process tree (TPST), and the problem of computing the difference between process models is transformed into detecting difference between TPSTs. Computing the tree edit distance between two labeled trees is NP-complete. So we use the divide and conquer strategy in our algorithm to obtain an edit script of two TPSTs that we make the cost close to minimum, where two TPSTs are decomposed into several fragments and then the corresponding mapped fragments and mapped nodes are determined. In this way, the mapping space is reduced and the mapping efficiency is improved. In experiment, we evaluate
the precision and execution time of our algorithm based on the real and synthetic data. The experimental results show that the precision of our algorithm is acceptable, and the execution time runs in milliseconds.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was partially supported by following foundations: National Natural Science Foundation of China (nos. 61602411 and 61572437), National Key Research & Development Program of China (no. 2016YFB1001403), Key Research and Development Project of Zhejiang Province (nos. 2015C01029, 2015C01034, and 2017C01013), and Major Science and Technology Innovation Project of Hangzhou (no. 20152011A03).

References


Submit your manuscripts at https://www.hindawi.com