published at ICSOC 2015, pp. pp 269-284
available at http://link.springer.com/chapter/10.1007%2F978-3-662-48616-0_17

Mining and Querying Process Change Information based on Change Trees

Georg Kaes and Stefanie Rinderle-Ma

University of Vienna, Faculty of Computer Science, Austria
{georg.kaes, stefanie.rinderle-ma}@univie.ac.at

Abstract Analyzing process change logs provides valuable information about the evolution of process instances. This information can be used to support responsible users in planning and executing future changes. Change mining results in a change process, which represents the dependencies between process changes mined from the change log. However, when it comes to highly adaptive process settings, multiple limitations of the change process representation can be found, i.e., based on change processes it is not possible to provide answers to important analysis questions such as 'How many instances have evolved in a similar way?' or 'Which changes have occurred following a particular change?'. In this paper, change trees and n-gram change trees are introduced to serve as a basis to analyze changes in highly adaptive process instances. Moreover, algorithms for discovering change trees and n-gram change trees from change logs are presented. The applicability of the approach is evaluated based on a systematic comparison with change mining, a proof-of-concept implementation and by analyzing real-world data.

1 Introduction

Process change and evolution is a key concern in many application domains such as care [1], manufacturing [1, 2], logistics [3], and health care [4]. The management of change information in business processes is a relevant challenge for flexible process aware information systems (PAIS) [4]. Change logs are a central asset to log, manage and understand how a process (instance) evolves over time [5]. Approaches such as [6, 7] advocate the exploitation of knowledge on previous changes for supporting users in applying future changes. Whereas [6] analyzes user annotations, [7] presents change mining to discover change processes from change logs. The resulting change processes visualize possible orderings of changes and contain information about possible dependencies between changes. However, as it will be shown, change processes are not suitable for answering the following analysis questions:

Q1: Which process instances have evolved in a similar way?
Q2: Which process instances have evolved in a similar way after a certain change sequence?

Both questions are relevant in practical settings. Analyzing which process instances have evolved in a similar way can be used as a starting point for predicting changes which may become necessary in the future. In the care domain, for example, Q1 can be used to analyze patients' treatment histories in order to predict future changes. Imagine two patients which have received the same treatments over a course of time, whereby the first patient's treatment plan is already further developed than the treatment plan of the second patient. When the question arises which changes may be necessary for the second patient's treatment plan in the future, the change information from the first patient's treatment plan can be used as a starting point.

Q2 provides more focus by asking for, e.g., the development of a treatment after a certain therapy was applied. This information can be used as a basis for analyzing what usually happens after a certain set of changes has been applied. Imagine a therapy in the nursing home setting which requires the nurses to conduct multiple other therapies afterwards, maybe because of possible complications. The question of interest is now, how many treatment plans have evolved in which way after this certain therapy has been conducted. This information can be used to further enhance the planning and conduction of therapies.

The question is how change information of process instances can be represented such that Q1 and Q2 can be sufficiently analyzed. As both Q1 and Q2 refer to the process instance level and as we assume that changes might be applied multiple times (think, for example, of steps such as physical therapy that might become necessary multiple times), a suitable representation must meet the following requirements:

R1: ability to deal with multiple occurrences of (change) instances (Q1, Q2)
R2: ability to deal with multiple occurrences of changes (Q1, Q2)
R3: ability to detect change sequences that follow a certain change pattern (Q2)

Existing approaches such as change mining do not fully meet these requirements. Hence, this paper introduces two new representations for information stored in change logs[1]. At first, the change tree is defined in Section 3 as a basis to meet requirements R1 and R2. In addition to the formal definition an algorithm is presented that mines change trees from change logs. Section 4 introduces the n-gram change tree as a further development of the change tree meeting requirement R3. Based on representing change sequences as n-grams the n-gram change tree offers a projection on those parts of the change instances that evolved after the occurrence of the n-gram. It is shown how n-gram change trees can be constructed from change logs. The feasibility of the change tree and n-gram change tree is evaluated in several ways. Section 5 systematically compares change tree and n-gram change tree to other representations such as change processes and graphs. A proof-of-concept implementation as well as an application of the concepts to a real-world data set from the BPI 2014 challenge is presented in Sect. 6. Section 7 discusses related approaches and Section 8 concludes the paper and gives an overview over future work.

[1] Fundamental definitions of changes and change logs are presented in Section 2.

Overall, change tree and n-gram change tree are novel change log representations that meet requirements R1, R2, and R3 and hence enable the in-depth analysis of highly dynamic process applications.

2 Change Log Definitions

The following definitions of changes and change logs are based on the recommendations from literature. A *change* Δ is defined according to [5] as $\Delta := (type, subject, paramList, S)$

Typical change *types* as summarized in the change patterns collection [8] are INSERT, DELETE, or MOVE. *Subject* refers to the task or activity to be, e.g., inserted or deleted. The *paramList* specifies, for example, the context of a change. Finally, S is the (instance) schema the change is applied to. Change Δ= (INSERT, A, <B, C>, S) inserts activity A between activities B and C in instance schema S.

Making a simplification to [5], a *change log* is defined as

$$cL :=< \Delta_1, \ldots, \Delta_n > \tag{1}$$

Assume the following change log

cL = $< \Delta_1$ = (INSERT, A, <Therapy Fragment C, End>, S_{I1}),
Δ_1 = (INSERT, A, <Therapy Fragment C, End>, S_{I2}),
Δ_2 = (INSERT, B, <Therapy Fragment C, End>, S_{I2})>.

The implicit assumption of an ordering suggests that two times an activity A was inserted between Therapy Fragment C and End, followed by an insertion of activity B between Therapy Fragment C and End.

An MXML-based format for change logs was proposed in the context of change mining [9]. Listing 1.1 shows the MXML representation for cL.

Listing 1.1. Fragment for Change Log Example in MXML

```
<WorkflowLog ...>
  <Process id="OR">
    <ProcessInstance id="1">
      <AuditTrailEntry>
        <Data>
          <Attribute name="CHANGE.postset">End</Attribute>
          <Attribute name="CHANGE.type">INSERT</Attribute>
          <Attribute name="CHANGE.subject">A</Attribute>
          <Attribute name="CHANGE.rationale">Therapy Fragment A required</Attribute>
          <Attribute name="CHANGE.preset">Therapy Fragment C</Attribute>
        </Data>
        <WorkflowModelElement>INSERT.A</WorkflowModelElement>
        <EventType>complete</EventType>
        <Originator>Dr. Ford</Originator>
      </AuditTrailEntry>
    </ProcessInstance>
    <ProcessInstance id="2">
      <AuditTrailEntry>
        <Data>
          <Attribute name="CHANGE.postset">End</Attribute>
          <Attribute name="CHANGE.type">INSERT</Attribute>
          <Attribute name="CHANGE.subject">A</Attribute>
          <Attribute name="CHANGE.rationale">Therapy Fragment A required</Attribute>
          <Attribute name="CHANGE.preset">Therapy Fragment C</Attribute>
        </Data>
        <WorkflowModelElement>INSERT.A</WorkflowModelElement>
        <EventType>complete</EventType>
        <Originator>Dr. Ford</Originator>
      </AuditTrailEntry>
      <AuditTrailEntry>
        <Data>
```

```
          <Attribute name="CHANGE.postset">End</Attribute>
          <Attribute name="CHANGE.type">INSERT</Attribute>
          <Attribute name="CHANGE.subject">B</Attribute>
          <Attribute name="CHANGE.rationale">Therapy Fragment B required</Attribute>
          <Attribute name="CHANGE.preset">Therapy Fragment C</Attribute>
        </Data>
        <WorkflowModelElement>INSERT.B</WorkflowModelElement>
        <EventType>complete</EventType>
        <Originator>Dr. Dent</Originator>
      </AuditTrailEntry>
    </ProcessInstance>
  </Process>
</WorkflowLog>
```

Comparing Equation 1 and the MXML-based representation in Listing 2, the latter collects changes at the instance level whereas the former refers to the schema level. In fact, change mining aggregates the instance-specific change information into an analysis model, i.e., the change process. Figure 1 shows the result of applying change mining to the log in Listing 2 as produced by the Change Miner plugin of the ProM 5.2 framework[2].

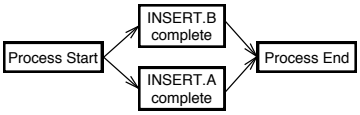


Figure 1. Change Process resulting from Change Mining (Using ProM 5.2)

For the considerations of this paper, a process instance is characterized by the changes that have been applied to the instance[3]. Hence, in the following we only refer to *change instances* (*instances* for short).

Definition 1 (Change Instance, Change Log). *Let $\mathcal{I}$ be a set of process instances and $\mathcal{C}$ be a set of changes. For each $I \in \mathcal{I}$, S_I denotes the instance schema, i.e., the schema the instance is currently running on, and $\Delta_1^I, \ldots, \Delta_n^I$ reflect the changes applied to I (S_I) so far, $\Delta_j^I \in \mathcal{C}, j = 1, \ldots n$. Assume that Δ_i^I was applied before Δ_j^I if $i < j$, $i, j \in \{1, ..., n\}$. Then a change instance I is defined as follows:*

$I : \Delta_1^I \rightarrow \Delta_2^I \rightarrow ... \rightarrow \Delta_n^I$

A change log cL represents a collection of change instances.

For Listing 2 the change log cL with the associated change instances turns out as follows:

`I1:` Δ_1

`I2:` $\Delta_1 \rightarrow \Delta_2$

[2] http://www.promtools.org/doku.php?id=prom52

[3] I.e., any other information such as (instance) schema or execution state will be considered in future work.

3 Change Trees

In order to be able to answer Q1 and meet requirements R1 and R2 (cf. Section 1), a representation for change information shall represent the chronological order of changes made to all instances in one aggregated view. Definition 2 presents the change tree as a representation for instance change information. Intuitively the change tree represents each of the change instances in a change log along with the number of its occurrences along paths from the root to the leafs.

Definition 2 (Change Tree). *Let cL be a change log and C be the changes contained in cL. Then change tree T is defined as a rooted multiway tree T:=(r,V,E) with*

1. $r := \emptyset$ *is the unique root node*
2. $V \subseteq C \times \mathbb{N}_0$
3. $\forall$ *leaf nodes* $v = (\Delta,n) \in V$: $n > 0$
4. $\forall$ *paths p from root r to node* $v = (\Delta, n) \in V$ *with* $n > 0$: *p corresponds to n change instances in cL.*

To complete Def. 2, Alg. 1 sets out how new changes are added to a change tree. Deletion or reorganization on change trees do not become necessary since removing change Δ from the change tree is considered as adding a "compensating" change Δ'.

Algorithm 1: Adding a Change to a Change Tree

Input:
- change Δ (applied to instance I)
- change tree CT (which contains instance I)

```
1  Begin
2  currentnode = root of CT
3  for i = 0; i < I.length; i++ do
4      currentnode = child node where I is stored
5  if there is a child of currentnode containing Δ then
6      Decrement count of currentnode
7      Go to the node containing Δ
8      Increment the count of this node
9  else
10     Decrement count of currentnode
11     Create a new child node for currentnode containing Δ
12     Set the count of this node to 1
13 End
```

Let us illustrate the concept of the change tree along the example depicted in Fig. 2. Figure 2 shows a change log (left side) and its representation as a change tree (right side)[4]. Change instance I1 for example consists of two consecutive

[4] For illustration reasons the change tree is annotated with the number of leafs.

applications of change Δ_1 (*R2*: multiple occurrence of changes). Starting from the root node and going towards *Leaf 2* in the change tree, the change instance can be reproduced. The number *1x* in Leaf 2 indicates that this change instance, i.e., $\Delta_1 \rightarrow \Delta_1$, occurred once in the change log. With this, the change tree offers the possibility to count the number of multiple instance occurrences.

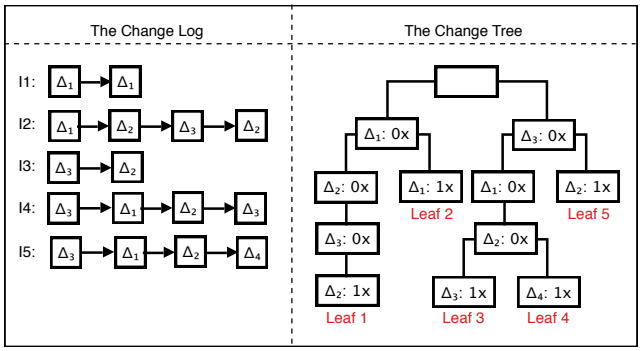


Figure 2. Change Log and Corresponding Change Tree

In order to illustrate how a change is added to a change tree (cf. Alg. 1) consider Fig. 3 which depicts two scenarios based on a change instance $I : \Delta_1$ and a given change tree (left side). In a first scenario change Δ_1 is again applied to I such that I is updated to $I : \Delta_1 \rightarrow \Delta_1$. As a first step I is located in the change tree (lines 2 – 4 in Algorithm 1). As a second step we look if the currently applied change has already been applied to some other change instance I' with $I = I'$ for I before applying Δ_1 the second time (lines 5–9). If such I' exists, the change instance is moved further up the change tree to the next level. This is the case for I in scenario 1. Now assume in a second scenario that not Δ_1 has been applied a second time, but Δ_3 instead. In this case, no I' exists with $I = I'$ (before the new change). Thus a new branch in the change tree is created and the change instance is moved there (change Δ_3, second and third pane).

Overall, the change tree representation covers all instances in the log as well as maintains the number of occurrences of change instances (*R1*) and change occurrences (*R2*). With respect to question *Q1* as set out in the introduction, the change tree offers the possibility to determine how many instances have evolved in which way. This information can be important for predicting and planning future changes.

Algorithm 2 sets out how a change tree can be mined from a change log. The algorithm starts with the first set of changes which has been applied to the process instances in the change log (line 30). For instances I1 and I2 from Fig. 2 this is Δ_1, for I3, I4 and I5 this is Δ_3 (the first row of changes). For each instance in the given set of instances (for the root node this are all instances in the change log) the change at the current position is checked. If a node containing this change does not already exist a new node is generated. For the first iteration

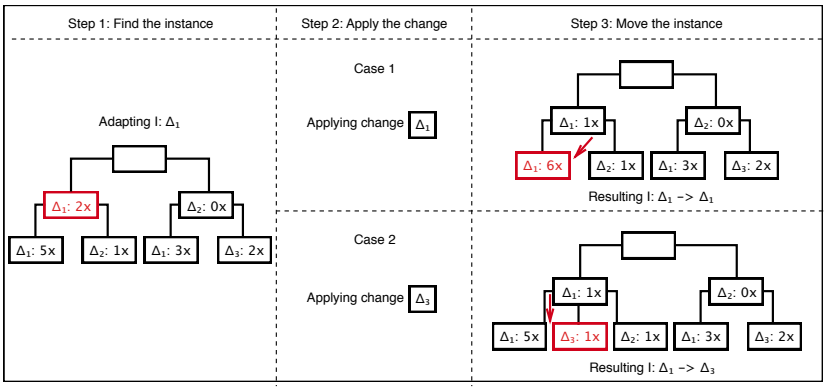


Figure 3. Adding a change to a change tree

two nodes containing Δ_1 and Δ_3 are generated. Next we check if the change we just created the node for is the last change for this process instance or not. If it is, we increment the counter for this node. This means that the instance is finished, we have reached leaf level and we have found one more instance with the given set of changes. If it is not the last instance we add the current instance to the set of instances for this node which will be traversed in later iterations. The set of instances is used to generate the child nodes for our current node, and since there is a following change, there will be a child node. Now we recursively reuse this function to generate all child nodes for each node in our set of nodes. After the first iteration this means that the function is called for the nodes Δ_1 (with instances I1 and I2) and Δ_3 (with instances I3, I4 and I5). Note that the iterator is incremented, thus analyzing the second level of changes (instances I1, I4 and I5: Δ_1, instances I2 and I3: Δ_2).

4 n-gram Change Trees

In order to answer Q1 (cf. Section 1), the change tree represents the chronological order of changes made to all instances in one aggregated view. In order to answer Q2, in addition, all occurrences of a specific change sequence must be detected and "consolidated" in the resulting representation in order to analyze what happened after the change sequence to the process instances of interest.

A change instance as defined in Def. 1 can also be understood as a string and a change sequence as substring. Thus, the problem can be considered as a transformation of the problem of *n-gram models* in language processing where two strings are defined as equivalent *"if they end in the same n - 1 words"* [10]. Hence, the *n-gram* in the change setting will reflect the change pattern (substring) of interest and we will determine the change sequences following the change pattern for each of its occurrences in the log.

Algorithm 2: Create a change tree from a change log

```
   Input: Change log cL
 1 //parent is the parent node for the children which are inserted
 2 //set-of-instances is the set of instances which contains data for child nodes of the
   current parent node
 3 //iterator is the number of the change in the change log instance - iterator=2 means the
   2nd change applied to the instance
 4 function create-children(parent,set-of-instances,iterator)
 5     set-of-nodes = new Array
 6     foreach instance in set-of-instances do
 7         if there is no child node with the current change for this parent then
 8             node = create-node(instance_iterator,parent)
 9             set-of-nodes.add(node)
10         else
11             node = the child node of the parent containing the current change
12         if instance_iterator+1==empty then
13             node.count++
14         else
15             node.nextinstances.add(instance)
16     foreach node in set-of-nodes do
17         create-children(node,node.nextinstances,iterator+1)
19     return;
20 function create-node(label,parent)
21     node.label = label
22     node.count = 0
23     node.nextinstances = new Array
24     node.children = new Array
25     if parent then
26         parent.children.add(node)
27     return node
28 Begin
29 root = create-node(null,null);
30 create-children(root, change log, 0);
```

Assume that we are interested in the n-gram $\Delta_1 \rightarrow \Delta_2$ and consider the example depicted in Fig. 4. For instance I2, the n-gram can be found at the beginning whereas for instances I4 and I5 the n-gram occurs after Δ_3. This results in a change tree where the required sequence of changes can be found in two different branches of the change tree (red nodes in the middle pane).

To see all changes following n-gram $\Delta_1 \rightarrow \Delta_2$, the two subtrees containing the n-gram have to be combined, i.e. restructuring the change tree becomes necessary. This can be achieved by using a suffix tree [11] or, more precisely, a generalized suffix tree which contains more than one string. A suffix tree represents all suffixes for a given string. Suffix trees are commonly used for pattern matching in various scenarios, from string matching to finding common motifs in DNA sequences [12]. In contrast to simple implementations of the suffix tree the algorithm proposed by Ukkonen [13] works in linear time O(n), where n represents the length of the string.

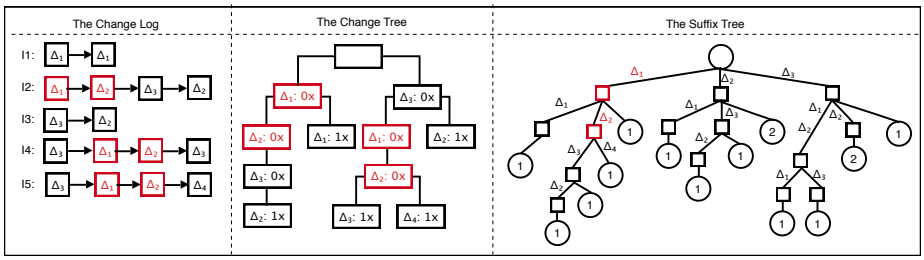


Figure 4. Development of the n-gram Change Tree

In the generalized suffix tree in Fig. 4 the leaf node of path $Root \rightarrow \Delta_2 \rightarrow 2$ means that two instances have change Δ_2 as their suffix. There are also two instances which end with the changes $\Delta_3 \rightarrow \Delta_2$ (instance I2 and I3), but only one instance which ends with $\Delta_2 \rightarrow \Delta_3$ (instance I4).

As depicted in Fig. 4 the suffix tree is constructed over the suffixes for all strings which can be found in the set of change logs for process instances I1 to I5. For n-gram $\Delta_1 \rightarrow \Delta_2$ we can now easily see the combined subtrees since all suffixes which start with the sequence can be found directly at the root node (marked red in Fig. 4, third pane). This information can be represented as the n-gram change tree, depicted in the second pane of Fig. 5. The n-gram change tree contains the n-gram in its root node. The suffixes of the n-gram , i.e., $\Delta_3 \rightarrow \Delta_2$, Δ_3, and Δ_4 have produced two paths from root to leafs. Specifically, $\Delta_3 \rightarrow \Delta_2$ and Δ_3 are aggregated into one such path.

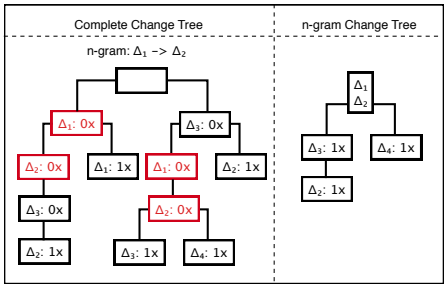


Figure 5. All occurrences of the n-gram in the change tree and the n-gram change tree

The n-gram change tree can be defined similarly to the change tree with some modifications as set out in Def. 3.

Definition 3 (n-gram Change Tree). *Let cL be a change log and C be the changes contained in cL. Then n-gram change tree nT is defined as a change tree (cf. Definition 2) where the following conditions are different:*

1 r :=< Δ_1,...,Δ_l > where $\Delta_i \in C$, i = 1,...,l. is the unique root node*

*4** $\forall$ *paths p from root r to node* $v = (\Delta, n) \in V$ *with* $n > 0$*: p corresponds to a projection of n change instances in cL starting from* $< \Delta_1,\ldots,\Delta_l >$ *as defined in the root node.*

Algorithm 1 also works for n-gram change trees with the only difference that the change is only added to the n-gram change tree if it has been applied after the respective n-gram has been applied to an instance.

Preparing the change logs for queries: In a complex setting such as the nursing domain, often multiple changes are made, which are not necessarily interacting with each other. For example, a patient has problems with his right knee, but at the same time he is on a special diet because of certain allergies. In these cases, we have to **trim the aforementioned data structure, so only the relevant changes are analyzed.** Based on these trimmed change logs, we can analyze the remaining logs with either the change tree or the n-gram change tree.

5 Comparison with Other Representations

This section compares the change tree and the n-gram change tree to other representations such as the change process [9] and graph-based structures. The basis for the evaluation is Listing 2 with an extension by multiple instances[5]. Recall for the following examples that
$<\Delta_1$ `= (INSERT, A, {Therapy Fragment C, End}, S)` and
Δ_2 `= (INSERT, B, {Therapy Fragment C, End}, S)`$>$.

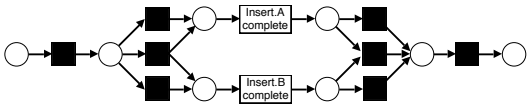
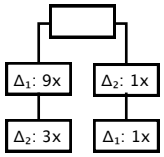
First consider the scenario depicted in Table 1. On the left side the change log is depicted, in the middle the resulting change process after applying change mining, and on the right side the change tree. Note that the resulting change process as for example shown in Fig. 1 has been transformed into a Petri Net in order to reason about the semantics of splits.

For the scenario in Table 1 the change process does not convey the information that for 9 instances change Δ_1 has been applied while change Δ_2 has only been applied for two instances. Moreover, based on the OR-split, it is not possible to see that for 3 instances first Δ_1 and then Δ_2 has been applied while for one scenario the reverse order of change occurred. The reason for this limitation is that change mining abstracts from the number of instance occurrences. In contrast to this the change tree reflects multiple occurrences of change instances, thus fulfilling requirement *R1*. All possible combinations of changes as they can be found in the change logs can be easily detected and interpreted. For example it can be concluded that the probability of having change Δ_1 is nine times the probability of change Δ_2.

In the second scenario (cf. Table 2) the change process cannot correctly reflect the difference between the scenario where only change Δ_1 has been applied, and the two others where Δ_1 respectively Δ_2 followed Δ_1. The multiple occurrence of the same change cannot be reflected correctly in the change process. The

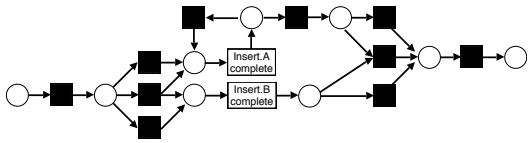
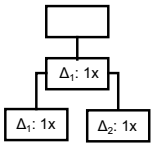
[5] The logs can be found on http://cs.univie.ac.at/project/apes.

Table 1. Multiple Instance Occurrences

Change Log	Change Process	Change Tree
I01-I09: Δ_1 I10: Δ_2 I11-I13: Δ_1 -> Δ_2 I14: Δ_2 -> Δ_1		

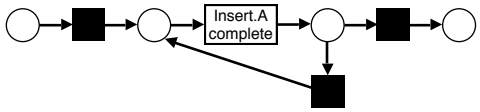
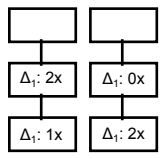
change tree in Table 2 removes the inaccuracy of the change process regarding the occurrence of multiple changes at different points in time. We can now easily see that in the given situation for each case change Δ_1 has been applied first. In half of the cases afterwards change Δ_1 has been applied again - in the other half change Δ_2 was used.

Table 2. Distinction of Change Occurrences

Change Log	Change Process	Change Tree
I1: Δ_1 I2: Δ_1 -> Δ_1 I3: Δ_1 -> Δ_2		

The problem that multiple occurrences of the same change cannot be correctly reflected in the change process is aggravated in the third scenario shown in Table 3. Here, different process scenarios still produce the same change process. The fact that in two cases Δ_1 has been applied only once are not reflected in the resulting change process. The corresponding change trees clearly show a distinction between the instances where Δ_1 has been applied once or multiple times, thus fulfilling requirement *R2*.

Table 3. Multiple Change Occurrences

Change Log	Change Process	Change Trees
I1: Δ_1 I2: Δ_1 I3: Δ_1 -> Δ_1 or I1: Δ_1 -> Δ_1 I2: Δ_1 -> Δ_1		

Representing change logs as graph structure instead of a tree would lead to information loss and is thus not a viable solution. Consider the tree and the graph representation in Figure 6. The change tree on the left has been compressed to a graph structure which requires fewer nodes. Each edge in the graph contains information about how many instances have evolved in the respective direction, so it can be seen that after Δ_1 on the first level four instances have evolved to Δ_3 (and possibly beyond this point). As can be discovered easily in the change tree, only one instance stopped its evolution here at Δ_3 while 2 instances evolved further to Δ_1 and one instance evolved to Δ_2. However, this information is lost in the graph representation since the two Δ_3 nodes would be merged.

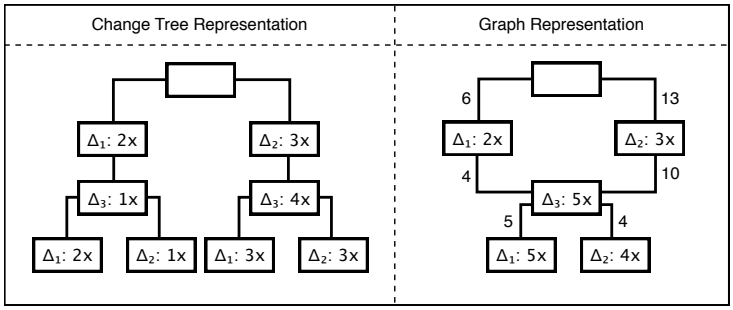


Figure 6. Comparing tree- and graph based models

6 Proof-of-Concept and Real-World Example

In order to provide experts with the possibility to use the change tree and the n-gram change tree for their own projects, we implemented it as a ProM plugin[6]. Based on MXML and XES log files one can build a change tree, select an n-gram and generate the n-gram change tree (cf. Figure 7).

During the Business Process Intelligence (BPI) Challenge real world process logs are analyzed from various points of view. The BPI Challenge 2014[7] was based on data from different processes of Rabobank Group ICT. When analyzing the log files we found that one of these processes is suited to be analyzed by the change tree since its log provides a set of activities which describe what has happened to a specific item in order to solve some problem. These activities are the basic building blocks for our change tree: Each time a new activity is planned for a specific item, the process of this item changes. Thus, the change tree can be used to mine change information from these log files.

[6] The current version of the plugin is available as a nightly build at http://www.promtools.org/prom6/nightly/

[7] http://www.win.tue.nl/bpi/2014/challenge, doi: 10.4121/uuid:c3e5d162-0cfd-4bb0-bd82-af5268819c35

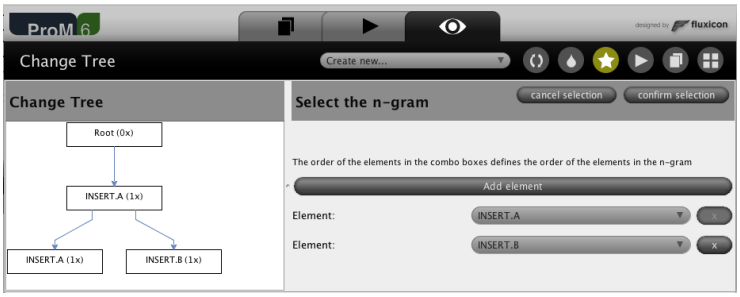


Figure 7. Implementation of the change tree as a ProM plugin

By mining process logs with the change tree we want to find information about what has usually happened after a certain change or a set of changes. For example we found multiple repeating process steps after including "Standard Change Type 88" (SCT 88) into the process. This is reflected by the change tree depicted in Fig. 8. Specifically, the tree is a 1-gram change tree with 1-gram "Standard Change Type 88".

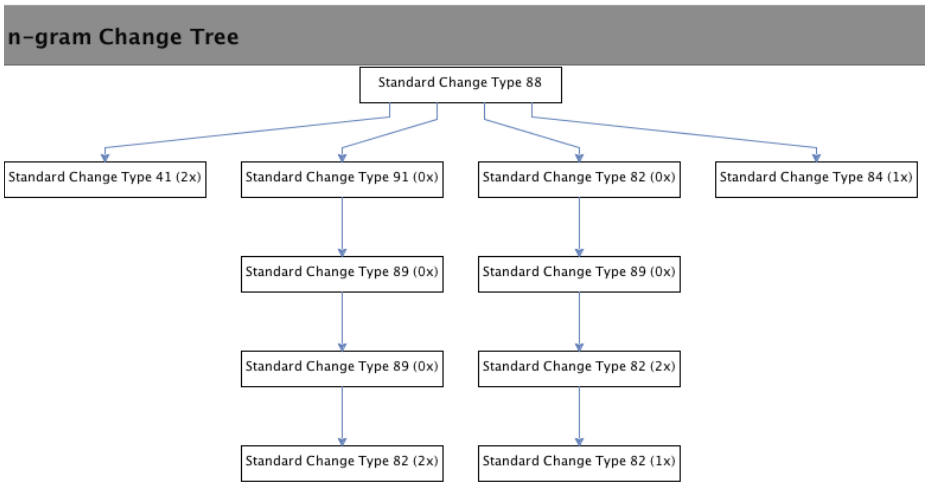


Figure 8. Process steps following Standard Change Type 88

From the 1-gram change tree it can be concluded that change "Standard Change Type 88" was followed in most cases by a change sequence containing "Standard Change Type 82" or "Standard Change Type 41". Since the provided data lacks context information it is not possible to deduce any semantical information from the generated change tree. If more information was available one could predict certain requirements for future tasks based on the change tree.

Think for example of knowledge about resources required for a specific task, such as requiring a technical specialist for executing "Standard Change Type 82". Using the n-gram change tree as a resource planning instrument it can be predicted that after executing change "Standard Change Type 88", with a certain probability "Standard Change Type 82" will become necessary as well and a technical specialist will be required (even if not required for "Standard Change Type 88" in the first place).

7 Related Work

Change of process instances and process schemas have been analyzed from various vantage points.

[7] and [9] describe the generation of a change process based on change mining. This method of analyzing change logs provides valuable information, especially for process instances which are based on a common schema, for example, on how to improve the process schema itself. However, as shown in this paper, change mining is not suited to reflect multiple change instances and multiple occurrences of changes. Moreover, searching for change patterns and their subsequent changes is not supported.

[6] aims at supporting users when applying change operations as well. For this, users can annotate changes with explanations and the systems exploits the changes by their frequencies together with the annotations in order to suggest changes to users. Change trees and n-gram change trees do not consider additional change annotations. However, in the presence of such annotations, [6] can provide complementary information to users.

[14] focuses on analyzing process variants, which are derived from a common process schema. The authors' goal is to find the process schema, where the smallest set of changes has to be applied to in order to obtain the schema of the individual process instances. In a first scenario, the existence of a reference process schema is assumed. This reference schema is adapted such that as few as possible additional changes will be necessary to reflect the process instances. In the second part, a process schema is generated solely based on the process instances and their changes. This approach does not construct analysis models from change logs, but it can serve as valuable complement to the approach presented in this paper. For example, one could find the change with the smallest set of required changes and use it as a basis for future changes.

Changes in the process instances schema cannot only be derived from the change log, but also from the event log. [15] presents methods to detect sudden changes in the process schema solely based on event log entries. For analyzing the effects of a change, such a system would also be of interest: Imagine a doctor who adds a new therapy to a patient where drug X has to be applied each week. It is generally known that drug X cannot be given at the same time as drug Y, which the patient currently receives. However for some reason the administration of drug Y has not been removed from the patient's therapy plan, and the next time the doctor sees that he should administer drug Y he just

skips the corresponding process task. Such effects of changes, which cannot be detected based on the change tree could be analyzed with such a system.

8 Conclusion and Future Work

This paper introduced change trees and n-gram changes together with the associated mining algorithms in order to discover analysis models from change logs that support users in deciding on future change application based on previously applied changes. The benefit of change trees – specifically when compared to existing change mining results – is that they reflect multiple change instances and multiple change occurrences. Both are characteristic to highly adaptive process scenarios. In addition, n-gram change trees enable answers to questions such as *'which changes happened after the occurrence of a certain change pattern?'*. This can be very interesting for users, as they do not have to search possibly complex change tree structures containing all the information in a change log, but a "projection" of the trees to the information of interest. Change trees and n-gram change trees have been evaluated in several ways: we compared them to existing change mining techniques and graph based methods, provided a technical implementation and an application to a real-world log.

Change trees reflect the structural aspect of change logs. Specifically, change instances and patterns are only considered as equal if they contain exactly the same changes. For practical settings it might be also of interest to consider 'similar' change sequences and patterns, i.e., go from a structural point of view to a more semantic one. This also might necessitate the inclusion of additional information such as process instance execution state or other instance parameters.

Data mining techniques such as Generalized Sequential Patterns might be useful to narrow down the number of possible change sequences to those which are statistically significant. Especially in change logs where a large number of different process changes appear, such an approach can significantly increase the usability of the change tree. Additionally, the analysis of scalability and efficiency when it comes to very large process logs is an interesting topic for future work.

Bibliography

1. Kaes, G., Rinderle-Ma, S., Vigne, R., Mangler, J.: Flexibility requirements in real-world process scenarios and prototypical realization in the care domain. In: OTM Workshops. (2014) 55–64
2. Schulte, S., Schuller, D., Steinmetz, R., Abels, S.: Plug-and-play virtual factories. IEEE Internet Computing **16** (2012) 78–82
3. Bassil, S., Keller, R., Kropf, P.: A workflow-oriented system architecture for the management of container transportation. In: Business Process Management. (2004) 116–131
4. Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies. Springer (2012)

5. Rinderle, S., Reichert, M., Jurisch, M., Kreher, U.: On representing, purging, and utilizing change logs in process management systems. In: Business Process Management. (2006) 241–256
6. Weber, B., Reichert, M., Rinderle-Ma, S., Wild, W.: Providing integrated life cycle support in process-aware information systems. Int. J. Cooperative Inf. Syst. **18** (2009) 115–165
7. Günther, C., Rinderle-Ma, S., Reichert, M., van der Aalst, W.: Using process mining to learn from process changes in evolutionary systems. International Journal of Business Process Integration and Management **3** (2008) 61–78
8. Weber, B., Reichert, M., Rinderle-Ma, S.: Change patterns and change support features - enhancing flexibility in process-aware information systems. Data & Knowledge Eng. **66** (2008) 438–466
9. Günther, C., Rinderle, S., Reichert, M., van Der Aalst, W.: Change mining in adaptive process management systems. In: On the Move to Meaningful Internet Systems. (2006) 309–326
10. Brown, P., Desouza, P., Mercer, R., Della Pietra, V., Lai, J.: Class-based n-gram models of natural language. Computational linguistics **18** (1992) 467–479
11. McCreight, E.M.: A space-economical suffix tree construction algorithm. Journal of the ACM **23** (1976) 262–272
12. Sagot, M.F.: Spelling approximate repeated or common motifs using a suffix tree. In: LATIN'98: Theoretical Informatics. (1998) 374–390
13. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14** (1995) 249–260
14. Li, C., Reichert, M., Wombacher, A.: Mining business process variants: Challenges, scenarios, algorithms. DKE **70** (2011) 409 – 434
15. Bose, R., van der Aalst, W., Žliobaitė, I., Pechenizkiy, M.: Handling concept drift in process mining. In: Advanced Information Systems Engineering. (2011) 391–405