Master Thesis – Math Formalization

Kefang Ding

January 8, 2019

Abstract

This article describes the work we have done so far. The following sections are organized in this way. Section 1 describes the problems to solve. Section 2 introduces formal definitions for dfg-method and adding long-term dependency. Section 3 gives details of the algorithm steps.

1 Introduction

The inputs for process model repair include one existing process model, a corresponding event log and a set of KPIs for the data evaluation in event log. After evaluating event log by KPIs, positive and negative labels are assigned on each trace in event log file.

In state-of-the-art technologies in process mining, only positive traces are used to repair model, while negative information is omitted. In this way, the generated models tend to have low precision. In order to increase the precision, we adapt Inductive Miner algorithm on the base of directly-follows relation of activities. This algorithm, called dfg-algorithm, uses both positive and negative information from event log and generates a model with high fitness. Yet after application, dfg-algorithm cannot discover, change the long-term dependency in the model, so that the model still has less precision. With concept long-term dependency, it describes the dependency between activities, where the execution of one activity affects the execution of another activities later. Later, we investigate problem and propose one algorithm to add long-term dependency constraints in model.

In this article, we propose one algorithm which combines dfg-method and adding long-term dependency programs, to incorporate negative information. The architecture of this model repair algorithm is as shown in Figure 1.

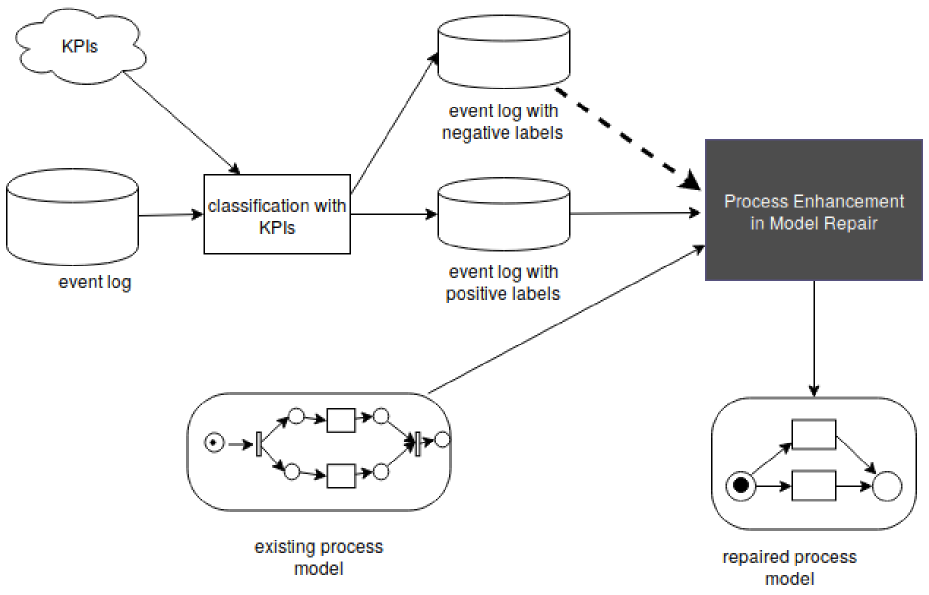
KPIs
event log
classification with KPIs
event log with negative labels
event log with positive labels
Process Enhancement in Model Repair
existing process model
repaired process model

Figure 1: Model Repair Architecture – firstly, event log is divided into positive and negative logs according to KPIs. Later, they are passed as inputs into repair process with existing process model. The output of this process is repaired model.

2 Definitions

2.1 Definitions Related To Dfg-Method

In this part, we provides concepts related to the dfg-method which is based on directly-follows graph. A directly-follows graph as used in [1], represents the directly-follows relation of activities in event log. For instance, if there are traces of $\langle ..., A, B, ... \rangle$ in event log, one edge (A,B) is added into directly-follows graph. By cutting directly-follows graph under different conditions, Inductive Miner[1, 2] discovers a process model. Unlike this process, we adapt Inductive Miner to repair model by using existing model, and event log with labels.

Definition 2.1 (Cardinality in directly-follows graph). *Given a directly-follows graph G(L) derived from an event log L, the cardinality of each directly-follows relation in G(L) is defined as:*

- *$Cardinality(E(A, B))$ is the frequency of traces with $\langle ..., A, B, ... \rangle$.*
- *Start node A cardinality $Cardinality(Start(A))$ is the frequency of traces with begin node A.*
- *End node B cardinality $Cardinality(End(A))$ is the frequency of traces with end node B.*

From the positive and negative event log, we can get directly-follows graphs, respectively $G(L_{pos})$ and $G(L_{neg})$. Each edge of $G(L_{pos})$ and $G(L_{neg})$ has a cardinality to represent the strength of this directly-follows relation. However, when the existing model is transformed into directly-follows graph $G(L_{ext})$, there is no point to assign cardinality on each edge. So we just set 1 to cardinality of each edge.

To incorporate all information from $G(L_{pos})$, $G(L_{neg})$ and $G(L_{ext})$, we define weight for each directly-follows relation in graph.

Definition 2.2 (Weight of directly-follows relation). *Given a directly-follows graph G(L), the weight of each directly-follows relation is defined as*

$$Weight(E(A,B)) = \frac{Cardinality(E(A,B))}{Cardinality(E(A,*))}$$

for start activities A, we have

$$Weight(Start(A)) = \frac{Cardinality(Start(A))}{Cardinality(Start(*))}$$

Similarly for end activities B, we have

$$Weight(End(B)) = \frac{Cardinality(End(B))}{Cardinality(End(*))}$$

E(A,) means all edges with source A, E(*,B) means all edges with target B, Start(*) represents all start nodes, and End(*) represents all end nodes.*

After defining the weights of each directly-follows relation, for each directly-follows relation, there are three weights from G_{pos}, G_{neg} and G_{ext}. The following strategy assigns new weight to directly-follows relation to new generated directly-follows graph G_{new}.

Definition 2.3 (Assign new weights to graph G_{new}). *there are three weights from G_{pos}, G_{neg} and G_{ext}, the new weight is*

- *For one directly-follows relation,*

  $$Weight(E_{G_{new}}(A,B)) = Weight(E_{G_{pos}}(A,B)) + Weight(E_{G_{ext}}(A,B)) - Weight(E_{G_{neg}}(A,B))$$

- *For start activities A, we have*

  $$Weight(Start_{G_{new}}(A)) = Weight(Start_{G_{pos}}(A)) + Weight(Start_{G_{ext}}(A)) - Weight(Start_{G_{neg}}(A))$$

- *For end activities B, we have*

  $$Weight(End_{G_{new}}(A)) = Weight(End_{G_{pos}}(A)) + Weight(End_{G_{ext}}(A)) - Weight(End_{G_{neg}}(A))$$

After assigning all the weight to directly-follows relation in G_{new}, we filter out all directly-follows relation in G_{new} with weight less than 0. Then, we transform the G_{new} into process tree bu using Inductive Miner for the next stage.

2.2 Definitions Related To Add Long-term Dependency

Example 1 Consider event log L with labels

$$L = [\langle A, C, E\rangle^{50,pos}, \langle B, C, D\rangle^{50,pos}, \langle A, C, D\rangle^{50,neg}].$$

$\langle A, C, E\rangle^{50,pos}$ means there are 10 traces $\langle A, C, E\rangle$ labeled as positive in event log. Similarly, $\langle A, C, D\rangle^{50,neg}$ represents there are $\langle A, C, D\rangle$ traces at number 50 in event log which have negative labels.

After applying the dfg-algorithm, a model as shown in Figure 2 is discovered. In event log L, B and D has long-term dependency, and A is expected to support only the execution of E, since $< A, C, D >$ is negative and $< A, C, E >$ is positive. However, the model doesn't express those constraints. Obviously,

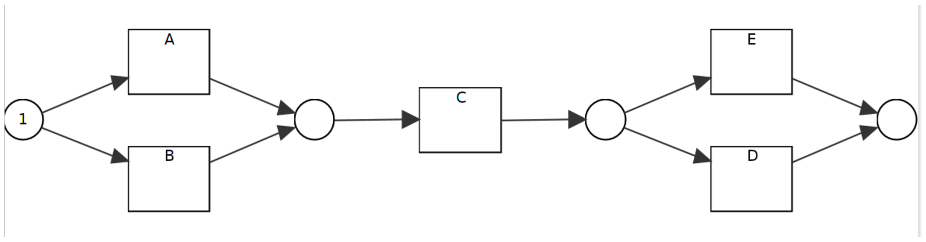

Figure 2: Process model generated from dfg-algorithm

long-term dependency relates the choices structure in process model, such as exclusive choice, loop and or structure. Due to the complexity of or and loop structure, only the long-term dependency in exclusive choice is considered.

The inputs for this algorithm are,

- Repaired model in process tree
- Event log with positive and negative labels

The output of this algorithm is:

- Repaired model in petri net with long-term dependency

Process tree, as one input for the algorithm, is one common model to interpret business process in process mining. It's a block-structured tree. To specify the process tree with respect to long-term dependency, the following definitions are in need. Firstly, the definitions related to tree are reviewed.

Definition 2.4 (Tree). *Let $\mathscr{E}$ be a finite set of entities, a tree is a collection of entities called nodes, which are connected by edges. A tree T is,*

- *t, with $t \in \mathscr{E}$, t has no outgoing edges*
- *$t(T_1, T_2, ..., T_n)$, with $t \in \mathscr{E}, i, n \in \mathbb{N}, i \leq n, T_i$ is a tree.*

T_i is a child or subtree of $t(T_1, T_2, ..., T_n)$, $t(T_1, T_2, ..., T_n)$ is one parent of T_i, which can be expressed in $P(t(T_1, T_2, ..., T_n), T_i)$. The root of tree is the node without any parent; A tree has only one root. A leaf node is the node which has no children nodes.
For a node in a tree, its ancestor and descendant are defined as:

Definition 2.5 (Ancestor Relation Anc(A,t)). *An ancestor of a node t in a tree is a node A, written as $Anc(A, t) \Rightarrow True$, if those conditions hold,*

- *A is a parent of t, written as $P(A, t) \Rightarrow True$, or*
- $\exists t_1, t_2..t_n, n \in \mathscr{E}, i < n, P(A, t_1) \wedge P(t_i, t_{i+1}) \wedge P(t_n, t) \Rightarrow True$

The ancestor of root is empty, while leaf nodes has no descendants. Based on this, we define the ancestors set of a node s.

Definition 2.6 (Ancestors of a node a). *The ancestors set of a node s, Ancestors(s), is defined as:*

$$Ancestors(A) = \{t | Anc(t, s) \Rightarrow True\}$$

Accordingly, descendant relation is given for node t and node s, Des(s,t). If node s is the ancestor of t, then t is a descent of s. $Anc(s, t) \Rightarrow Dec(t, s)$; The set of descendants of node t is Descendants(t).

Definition 2.7 (Least Common Ancestor). *A least common ancestor for node s and node t in a tree is a node n, where*

$$Anc(n, s) \wedge Anc(n, t) \wedge \exists! m Anc(n, m) \wedge Anc(m, s) \wedge Anc(m, t)$$

In process tree, all the leaves are activities in business process, and the middle nodes are operators which represents the relations of all its children nodes[3, 1]. This paper uses four operators in context of long-term dependency. The four relations. $\{\rightarrow, \times, \wedge, \circlearrowleft\}$ are considered.

Next, we only focus on exclusive xor structure on long-term dependency. As known, long-term dependency is associated with choices. In xor block, it means the choices of each xor branch in xor block. For sake of convenience, we define the xor branch.

$Q = \times(Q_1, Q_2, ..Q_n)$, Q_i is one xor branch with respect to Q, rewritten as $XORB_{Q_i}$ to represent one xor branch Q_i in xor block, and record it $XORB_{Q_i} \in XOR_Q$. For each branch, there exists the begin and end nodes to represent the beginning and end execution of this branch, which is written respectively as Begin($XORB_{Q_i}$) and End($XORB_{Q_i}$).

For convenience of analysis, two properties of xor block, purity and nestedness are demonstrated to express the different structures of xor block according to its branches.

Definition 2.8 (XOR Purity and XOR Nestedness). *The xor block purity and nestedness are defined as following:*

- *A xor block XOR_Q is pure if and only $\forall XORB_X \in XOR_Q, XORB_X$ has no xor block descent, which is named as pure xor branch. Else,*
- *A xor block XOR_Q is nested if $\exists XOR_X, Anc(XOR_Q, XOR_X) \rightarrow True$. Similarly, this xor branch with xor block is nested.*

In the Figure3, xor block Xor(c1,c2) are pure and not nested, since all the xor branches are leaf node, but xor block Xor(a,Seq(b,Xor(c1,c2))) is impure and nested with Xor(c1,c2).

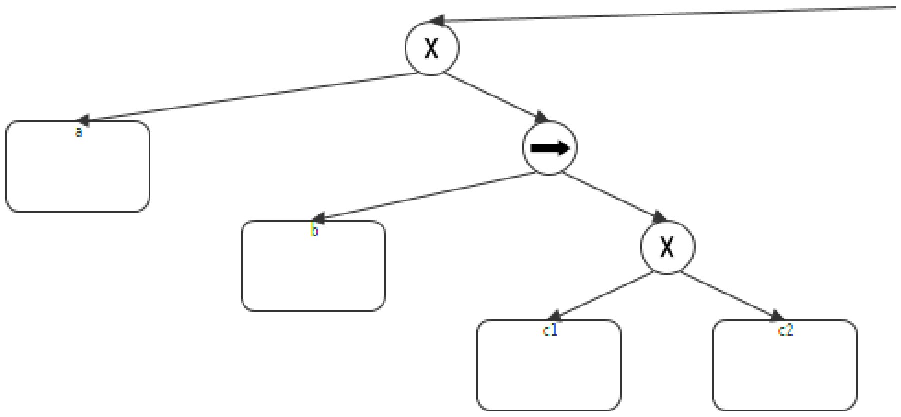

Figure 3: XOR branch variants

Long-term dependency researches on the dependency of choices in xor block, with observation, actually on the pure xor branch, because nested xor branch has multiple choices, which affect the execution of later process. For two arbitrary pure xor branches, to have long-term dependency, they firstly need to satisfy the conditions: (1) they have an order;(2) they have significant correlation. The order of xor branch follows the same rule of node in process tree which is explained in the following.

Definition 2.9 (Order of nodes in process tree)**.** *Node X is before node Y, written in $X \prec Y$, if X is always executed before Y. In the aspect of process tree structure, $X \prec Y$, if the least common ancestor of X and Y is a sequential node, and X positions before Y.*

The correlation of xor branches is significant if they always happen together. To define it, several concepts are listed at first.

Definition 2.10 (Xor branch frequency)**.** *Xor branch $XORB_X$ frequency in event log l is $F_l(XORB_X)$, the count of traces with the execution of $XORB_X$. For multiple xor branches, the frequency in event log l is defined as the count of* *traces with all the execution of xor branches $XORB_(Xi)$, written as*

$$F_l(XORB_(X1), XORB_(X2), ..., XORB_(Xn))$$

.

After calculation of the frequency of the coexistence of multiple xor branches in positive and negative event log, we get the supported connection of those xor branches, and define the correlation.

Definition 2.11 (Correlation of xor branches). *For two pure xor branches, $XORB_X \prec XORB_Y$, the supported connection is given as*

$$SC(XORB_X, XORB_Y) = F_{pos}(XORB_X, XORB_Y) - F_{neg}(XORB_X, XORB_Y)$$

. If $SC(XORB_X, XORB_Y) > lt - threshold$, then we say $XORB_X$ and $XORB_Y$ have significant correlation.

Under above concepts, if $XORB_X \prec XORB_Y$ and they have significant correlation, $XORB_Y$ is potentially dependent on $XORB_X$. In general cases, between $XORB_X$ and $XORB_Y$ there exists one $XORB_Z$, which potentially depends on $XORB_X$; At the same time, $XORB_Y$ is also dependent on $XORB_Z$ with certain potent. In our work, the crossed dependency of $XORB_X$ and $XORB_Y$ is ignored, we only consider the directly potential dependency, which requires $XORB_X$ is directly-followed by $XORB_Y$.

Definition 2.12 (Directly-followed relation of xor branch). *$XORB_X$ is directly-followed by $XORB_Y$, $XORB_X \leq XORB_Y$ if*

$$\exists! XORB_Z, XORB_X \prec XORB_Y \wedge XORB_X \prec XORB_Z \wedge XORB_Z \prec XORB_Y$$

During the execution of a process model in reality, several choices decide one choice later, and one choice decides many choices later. For example, to apply the loan, only if the credit is fine, and the amount of loan is affordable, the loan is possible to give; else it is declined. Therefore, it is not meaningful to only consider the binary long-term dependency of xor branches. We expand the binary long-term dependency into multiple relation.

Definition 2.13 (long-term dependency on set of xor branches). *Two set of xor branches $XORB_{X1}, XORB_{X2}..., XORB_{Xm}$, $XORB_{Y1}, XORB_{Y2}..., XORB_{Yn}$ have long-term dependency, if*

$$\forall XORB_{Xi}, XORB_{Yj}, XORB_{Xi} \leq XORB_{Yj}$$

and

$$SC(XORB_{X1}, XORB_{X2}..., XORB_{Xm}, XORB_{Y1}, XORB_{Y2}..., XORB_{Yn}) > threshold$$

$XORB_{X1}, XORB_{X2}..., XORB_{Xm}$ depend the execution of $XORB_{Y1}, XORB_{Y2}..., XORB_{Yn}$.

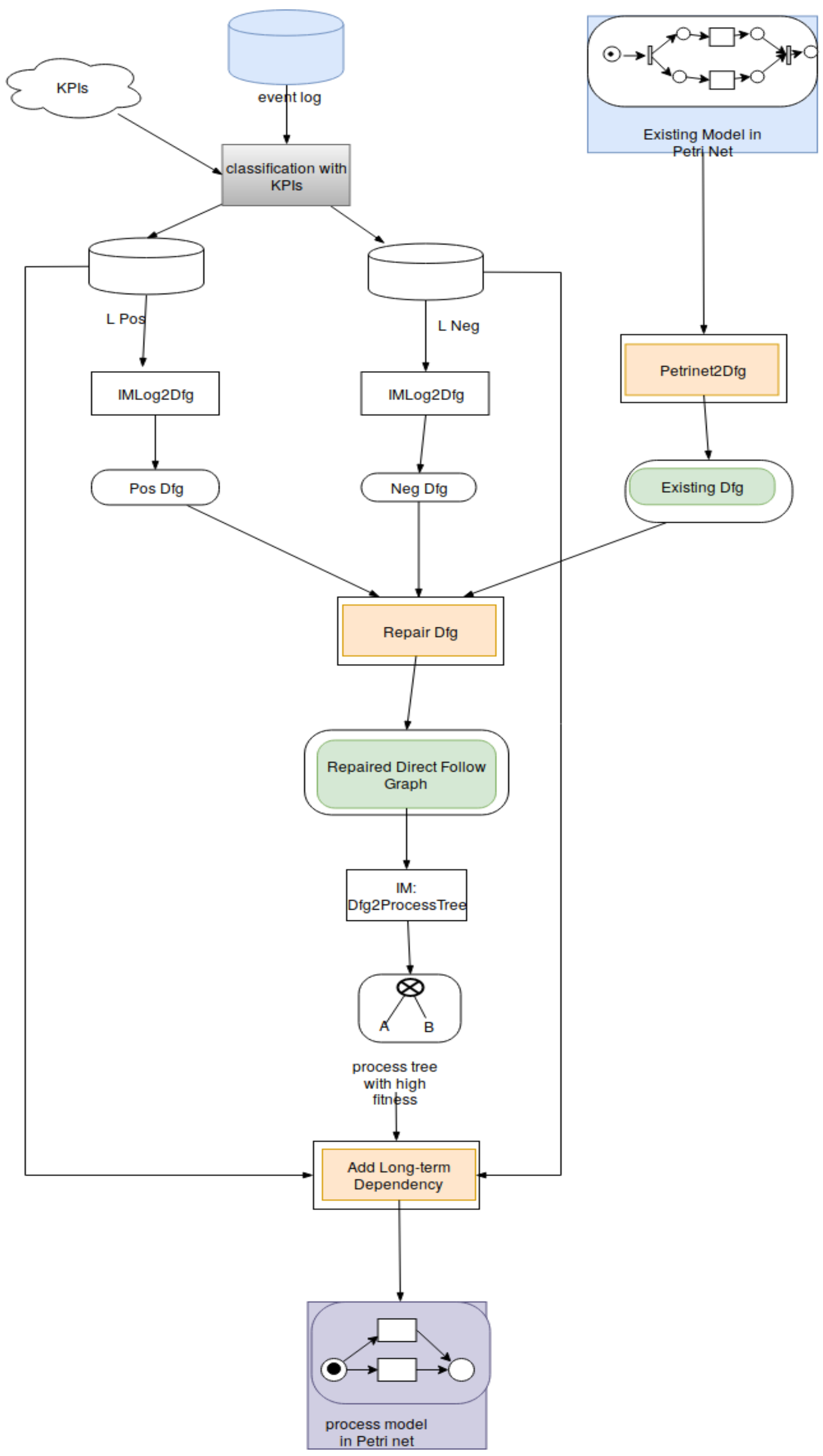

Figure 4: Model Repair Architecture – Rectangles represents processes and output data in eclipse shape, especially customized processes and data are in doubled lattice shape. Input event log and existing model are in blue, KPIs are in cloud. The output is a petri net in purple.

3 Algorithm Implementation

After combining this algorithm, the algorithm is completed to incorporate negative information to repair model as shown in Figure 4.

3.1 Detect the long-term dependency of xor branches

According to Definition 2.13, it gives constrains on xor branches. In term of process tree structure, constrains are:

- xor blocks where the xor branches belong to are also directly-followed; Only they are able to create xor block pair.
- the combination of potentially long-term dependency in xor blocks is not complete. If complete, any xor branch can go to next branches, the structure keep the same; So no necessary is required to add places and transitions to connect the xor branches.

With this knowledge, we develop Algorithm **??** to detect the long-term dependency.

Algorithm 1: General steps to add long-term dependency

```
   Result: Discover Long-term Dependency In Model
1  create a list including all xor pairs in process tree;
2  while pair in xor pair list do
3      generate all the combination of xor branches set ;
4      check the all set if they have potential long-term dependency if this
         pair has complete long-term dependency combination then
5          remove this pair from xor pair list;
6      end
7      keep the pair in list
8  end
9  transfer process tree into Petri net;
10 add places in Petri net for every branch pair with long-term
     dependency;
```

3.2 Add long-term dependency on Petri net

After detecting long-term dependency on xor branches from process tree, silent transitions and extra places are added to express those dependency on Petri net. Given, the basic idea is to add places after $XORB_X$ and places before $XORB_Y$, then connect the places by silent transition. Due to the structure of xor branches, situations differ, as listed in the Algorithm 2.

Algorithm 2: Add long-term dependency between pure xor branch

```
$XORB_Y$ is dependent on $XORB_X$;
if $XORB_X$ *is leaf node* then
    One place is added after this leaf node. ;
end
if $XORB_X$ *is Seq* then
    Add a place after the end node of this branch;;
    The node points to the new place;;
end
if $XORB_X$ *is And* then
    Create a place after the end node of every children branch in this
      And xor branch; ;
    Combine all the places by a silent transition after those places; ;
    Create a new place directly after silent transition to represent the
      And xor branch; ;
end
if $XORB_Y$ *is leaf node* then
    One place is added before this leaf node. ;
end
if $XORB_Y$ *is Seq* then
    Add a place before the end node of this branch;;
    The new place points to this end node;;
end
if $XORB_Y$ *is And* then
    Create a place before the end node of every children branch in this
      And xor branch; ;
    Combine all the places by a silent transition before those places; ;
    Create a new place directly before silent transition to represent the
      And xor branch; ;
end
Connect the places which represent the $XORB_X$ and $XORB_Y$ by
  creating a silent transition.
```

References

[1] S. J. Leemans, D. Fahland, and W. M. van der Aalst, "Discovering block-structured process models from event logs-a constructive approach," in *International conference on applications and theory of Petri nets and concurrency*, pp. 311–329, Springer, 2013.

[2] S. J. Leemans, D. Fahland, and W. M. van der Aalst, "Discovering block-structured process models from incomplete event logs," in *International Conference on Applications and Theory of Petri Nets and Concurrency*, pp. 91–110, Springer, 2014.

[3] W. van der Aalst, *Process Mining: Data Science in Action.* Springer Publishing Company, Incorporated, 2nd ed., 2016.