Model Repair by Incorporating Negative Instances In Process Enhancement

Master Thesis

Author : **Kefang Ding**

Supervisor : Dr. Sebastiaan J. van Zelst

Examiners : Prof. Wil M.P. van der Aalst
Prof. Thomas Rose

Registration date : 2018-11-15

Submission date : 2019-04-08

This work is submitted to the institute

PADS RWTH University

Acknowledgments

At first, I would like to express my deep gratitude to Prof. Wil M.P. van der Aalst for his valuable and constructive suggestions for planning and development of my thesis. Also, the support from Prof. Thomas Rose as my second supervisor on my thesis is greatly appreciated.

For the help given by Dr. Sebastiaan J. van Zelst, I am particularly grateful. His patience guidance and enthusiastic encouragement helped me keep my progress on schedule. Moreover, he kept pushing me into a higher level into scientific research through useful critiques. With those critiques, I realized the limits not only of my methods but also the working strategy, which benefits me a lot in the scientific field.

I also want to thank the whole PASD group for their valuable technical support; Especially, the advice from Alessandro Berti has saved me a lot of troubles and improved my work. Finally, I wish to thank my friends and parents for their support and encouragement throughout my study.

Abstract

Based on business execution history recorded in event logs, Process Mining provides visual insight on the business process and supports process analysis and enhancements. It bridges the gap between traditional business process management and advanced data analysis techniques such as data mining and gains more interests and application in recent years.

Process enhancement, as one of the main focuses in process mining, improves the existing processes according to actual business execution in the form of event logs. The records in an event log can be classified as positive and negative according to predefined Key Performance Indicators, e.g. the logistic time, and production cost in a manufacture. Most of the current enhancement techniques only consider positive instances from an event log to improve the model, while the value hidden in negative instances is simply neglected.

This thesis provides a novel strategy that considers not only the positive instances and the existing model but also incorporate negative information to enhance a business process. Those factors are balanced on directly-follows relations of activities and generate a process model. Subsequently, long-term dependencies of activities are detected and added to the model, in order to block negative instances and obtain a higher precision.

We validate the ability of our methods to incorporate negative information with synthetic data at first. Then, we conduct experiments in a scientific workflow platform KNIME to show the statistical performance of our methods. The results showed that our method is able to overcome the shortcomings of the current repair techniques in some situations and repair models with a higher precision.

Contents

List of Figures

List of Tables

Chapter 1

Algorithm

This chapter begins with a general framework to repair a reference process model by incorporating the negative instances. A concrete algorithm within the framework is proposed in the subsequent sections.

1.1 General Framework for Repairing Process Models

Figure 1.1 shows our proposed framework to repair a process model with negative information. The inputs are a reference process model and a labeled event log. The reference process model can be in multiple types, e.g. Petri net, process tree. Traces in the labeled event log are classified as positive or negative in respect to some KPIs of business processes. The output is a repaired process model with the same type as the reference model.

The basic idea behind the framework is to unify the impact from the reference model M, positive sublog L^+ and negative sublog L^- into data models. Those data models are of the same type and denoted as D^M, D^+ and D^- respectively. Then we consider all impact from models and generate a new data model D^n. D^n is later transformed into a process model M^r as the output. Several post-processes are optional to improve the repaired model M^r.

After defining a data model to unify the impact and implementing the process modules in the general framework, a solution which also considers the negative information on model repair is developed.

1.2 Algorithm

Given the inputs, an event log and a Petri net as the reference process model M, our task is to repair the referenced Petri net based on the actual event log with consideration of negative information.

Based on those scope, the main modules in the framework in Figure 1.1is designed and developed to repair the reference Petri net. First of all, we define a proper data model to represent the impact from Petri net M, and the event log L. Then, we list all the modules and describe our basic ideas to implement them.

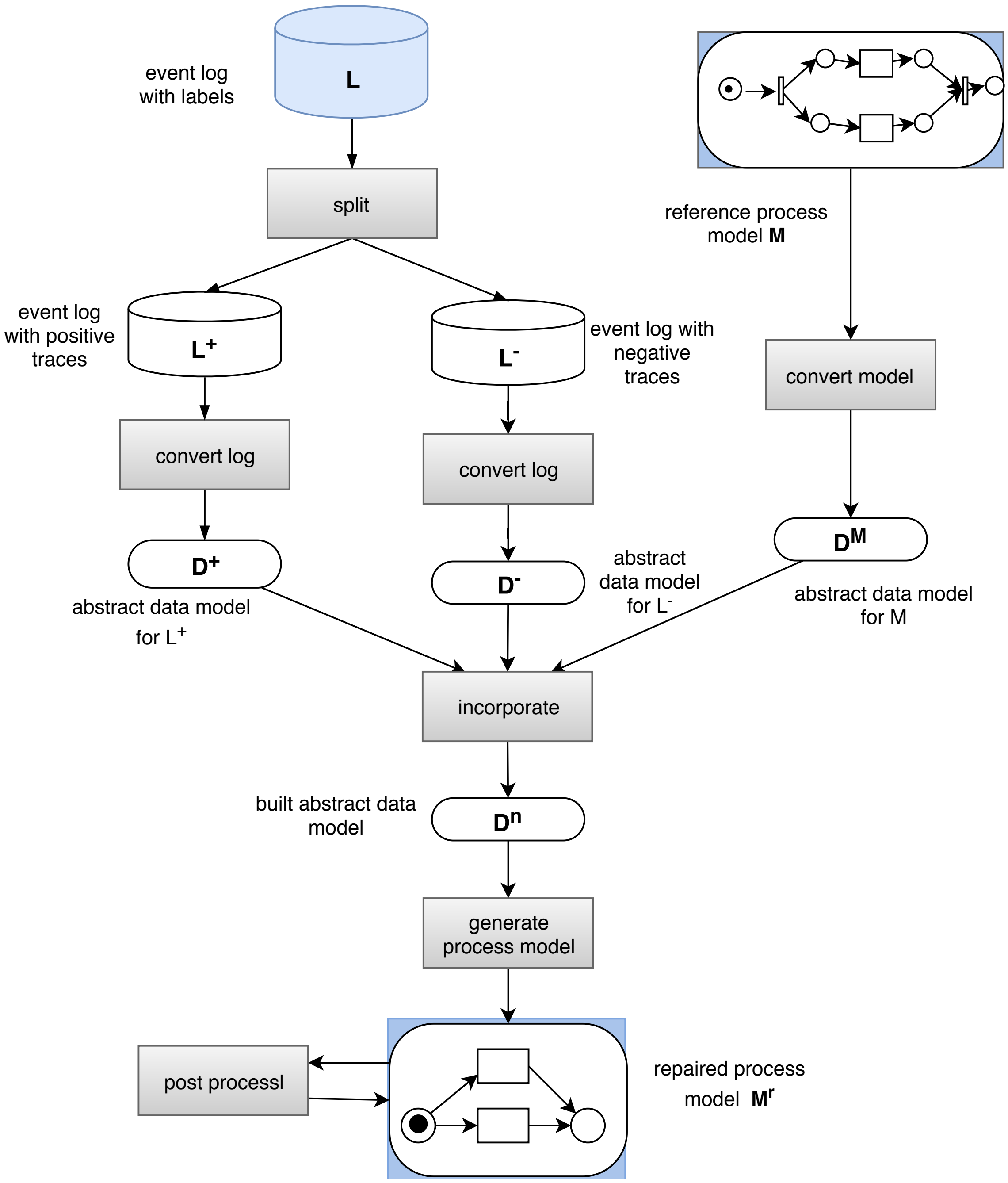


Figure 1.1: Model Repair Architecture – Rectangles represents processes and data models in eclipse shape. Input event log and the reference model are in blue, while the repaired model is in green.

1.2.1 Unified Data Model

We choose the directly-follows graph as the basis of our unified data model to represent the impact from the reference model and the event log. The reasons are, (1) there exist transformation algorithms to extract a directly-follows graph from an event log and to convert directly-follows graph into process tree or Petri net, which saves our effort;(2) the cardinality of directly-follows graphs can be used to express the impact strength.

Although we can derive three directly-follows graphs from the reference model, the positive and negative event logs respectively, their cardinalities are in different level and not able to incorporate with each other. So we introduce a concept called unified cardinality to bring all the impact into the same level, which is the percentage to the sum of total cardinalities with a range [0-1].

Definition 1.1 (Unified cardinality)**.** Given a directly-follows graphs $G(L) = (A, F, A_{start}, A_{end})$ for a model, the unification for this graph is a function $u : F \to N$ that has the following definition:
for each directly-follows relation $(a, b) \in F$,

$$u(a, b) = \frac{c(a, b)}{\sum_{(a',b') \in F} c(a', b')}$$

for start activities $a \in A_{start}$,

$$u(a) = \frac{c(a)}{\sum_{a' \in A_{start}} c(a')}$$

Similarly for end activities $a \in A_{end}$,

$$u(a) = \frac{c(a)}{\sum_{a' \in A_{end}} c(a')}$$

A directly-follows graph with unified cardinality is denoted as a unified directly-follow graph $D(L) = (A, F, A_{start}, A_{end}, u)$. After analyzing the positive , negative instances from event logs and the reference process model, $D(L_{pos})$, $D(L_{neg})$ and $D(L_{ext})$ are generated as the directly-follows graphs with unified cardinalities.

1.2.2 Modules List

After fixing the unified data models, in order to repair the reference model, the following list of modules are necessary.

- *Split event log into positive and negative sublogs* The event log L is split into an event log L^+ with only positive traces and an event log $L^-$with negative traces.
- *Convert event logs into unified directly-follows graphs, D^+, D^-* Two unified directly-follows graphs are generated respectively for the positive instance and negative instances from the event log.
- *Convert reference model into unified directly-follows graph D^M*
- *Incorporate unified directly-follows graphs* Three unified directly-follows graphs D^M, D^+, D^- are combined into one single directly-follows graph D^n after balancing their impact.

- *Generate process models from D^n* Process models are mined from the repaired data model D^n.
- *Post process the repaired model* Several post-processes are optionally applied on the generated process model, in order to improve the process model quality according to certain criteria.

In those modules,for the simplicity, we skip the details for the module *Split event log into positive and negative sublogs.* The details of the concrete algorithms to implement other modules are provided in the subsequent sections.

1.2.3 Convert Event Logs into Unified Directly-follows Graphs

Given an event log, to retrieve its unified directly-follows graph, we need to obtain its directly-follows graph at first. There is an existing procedure *IMLog2Dfg* from [8]. *IMLog2Dfg* traverses traces in the event log, extracts directly-follows relations of activities, and generates a directly-follows graph based on those relations.

By applying *IMLog2Dfg* separately on the event logs L^+ and L^-, we generate two directly-follows graphs $G(L_{pos})$ and $G(L_{neg})$. In the next step, the cardinalities from those graphs are unified according to Definition 1.1 and become a part of unified data model $D(L_{pos})$ and $D(L_{neg})$

1.2.4 Convert Reference Model into Unified Directly-follows Graph

The basic idea behind this convert is to transform a reference process model into a directly-follows graph and then unify this directly-follows graph into D^M.

To generate a directly-follows graph from a Petri net, we investigate the execution order of activities with help of a transition system. In a transition system, the set of activities directly before a state have to be executed to reach this state, while the activities after this state are enabled to execute only after reaching this state. This implies the set of the activities before the state is executed before the set after the state. By analyzing the transition system, directly-follows relations between activities are extracted and used later to a directly-follows graph for the reference model.

From the positive and negative event logs, we can get the cardinality for corresponding directly-follows graph 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 cardinality with 1 for each arc. Based on this cardinality assignment, we attain the unified data model D^M for the reference Petri net.

1.2.5 Incorporate Unified Directly-follows Graphs

After the unification, the impact from the existing model, the impact from positive and negative instances are in the same level and represented in D^M,D^+, and D^-. The strategy to incorporate those three models is that directly-follow relations are added into the repaired data model D^n if the total support from D^M and D^+ exceeds the rejection force from D^-. The other directly-follows relations are rejected. Namely, we balance all impact by subtracting the unified cardinality of D^- from the sum of unified cardinality in D^M andD^+.

Definition 1.2 (Incorporating method)**.** For any directly-follows relation from D^M,D^+, and D^-, we balance all forces on it in the following way.

- For one directly-follows relation,

$$u^n(a,b) = u^M(a,b) + u^+(a,b) - u^-(a,b)$$

- For a start activity $a \in A^M_{start} \cup A^+_{start} \cup A^-_{start}$,

$$u^n(a) = u^M(a) + u^+(a) - u^-(a)$$

- For an end activity $a \in A^M_{end} \cup A^+_{end} \cup A^-_{end}$

$$u^n(a) = u^M(a) + u^+(a) - u^-(a)$$

In the real life, there exists various needs to address the impact either from the existing model, the positive instances or the negative instances. To meet this requirement, three control parameters w^M,w^+, and $w^- \in [0,1]$ are assigned respectively to each unified cardinality from the existing model, and positive and negative instances. The weighted unification is modified in the way bellow.

Definition 1.3 (Weighted incorporating method)**.** Given the control weight w^M,w^+, and $w^- \in [0,1]$, the weighted incorporating method to balance forces from D^M,D^+, and D^- for D_n is defined below.

- For one directly-follows relation,

$$u^n_w(a,b) = w^M * u^M(a,b) + w^+ * u^+(a,b) - w^- * u^-(a,b)$$

- For a start activity $a \in A^M_{start} \cup A^+_{start} \cup A^-_{start}$,

$$u^n(a) = w^M * u^M(a) + w^+ * u^+(a) - w^- * u^-(a)$$

- For an end activity $a \in A^M_{end} \cup A^+_{end} \cup A^-_{end}$

$$u^n(a) = w^M * u^M(a) + w^+ * u^+(a) - w^- * u^-(a)$$

By adjusting the weights of w^M,w^+, and w^-, different focus can be reflected by the model. For example, by setting $w^M = 0, w^+ = 1, w^- = 1$, the existing model is ignored in the repair, while the original model is kept in situation $w^M = 1, w^+ = 0, w^- = 0$.

Next, we filter the directly-follows relation according to its weighted cardinality. If the cardinality over one certain threshold, it indicates a significant support to add this relation into the repaired data model D^n. By adding those directly-follows relation, we build a unified directly-follow graph D^n over this threshold t.

1.2.6 Generate process models from D^n

The result of the last step above is a unified directly-follows graph D^n with weighted cardinality. To generate a process model from it, we convert D^n firstly into a general directly-follows graph G^n. Analyzing the weighted cardinality, we transform the weighted cardinality by multiplying the number of traces in labeled event log.

Later, based on G^n, an existing procedure called *Dfg2ProcessTree* is applied to mine process models like process tree and Petri net[8]. It finds the most prominent split from the set of exclusive choice, sequence, parallelism, and loop splits on a directly-follows graph. Afterward, the corresponding operator to the split is used to build a block-structured process model called a process tree. Iteratively, the split sub graphs are passed as inputs for the same procedure until one single activity is reached and no split is available. A process tree is output as the mined process model and can be converted into another process model called Petri net.

1.2.7 Post Procedure on the Process Model

Due to the intrinsic characters of Inductive Miner, the dependency from activities which are not directly-followed can't be discovered. To improve precision of the generated model, we propose one post procedure to add the long-term dependencies to the model. Additionally, another post procedure to simplify the model is introduced by deleting redundant silent transitions and places.

1.2.7.1 Add Long-term Dependency

Obviously, long-term dependency relates to the structure of choices in process model, such as **exclusive choice**, **loop** and **or structure**. Due to the complexity of handling long-term dependency on **or** and **loop structure**, we skip them but focus only on the long-term dependency in **exclusive choice structure**.

The generated process tree mined by Inductive Miner is used as an intermediate process model to detect long-term dependency. A process tree is (1) easy to extract the exclusive choice structure, since it is block-structured. (2) easy to transform a process tree to a Petri net as the final model.

An exclusive choice structure is represented as ***xor block*** in a process tree. For the sake for convenience, we name a subtree of an xor block as one ***xor branch***, and denote the set of xor blocks B(Q) as and the set of xor branches as BB(Q) for a tree Q. For two arbitrary xor branches with long-term dependency, they have to satisfy the conditions: (1) they have a sequential order;(2) they always happen together, namely with significant correlation. The order of xor branch follows the same rule of node in process tree which is explained in the following.

Definition 1.4 (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 context 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.

To define the correlation of xor branches, several concepts listed below are necessary.

Definition 1.5 (The execution of a process tree Q)**.** With respect to a trace $\sigma \in L$, a process tree Q is executed in σ, denoted as $\sigma \models Q$, when

- for $Q = (a)$, if $\exists i, 1 \leq i \leq |\sigma| \sigma(i) = a$.
- for $Q = \rightarrow (Q_1, Q_2, ..Q_n)$, if $\forall Q_i, \sigma \models Q_i$ is executed in order.
- for $Q = \times(Q_1, Q_2, ..Q_n)$, if $\exists Q_i, \sigma \models Q_i$.
- for $Q = \wedge(Q_1, Q_2, ..Q_n)$, if $\forall Q_i, \sigma \models Q_i$.
- for $Q = \circlearrowleft (Q_1, Q_2, ..Q_n)$, if $\sigma \models Q_1$ and $\exists Q_i, \sigma \models Q_i \Rightarrow \sigma \models Q_1$ after Q_i.

With the definition above, we can formalize the frequency of xor branch in the event log as below.

Definition 1.6 (Xor branch frequency)**.** The frequency for an xor branch X in event log L is the count of traces, $f : X \rightarrow N$.

$$f_L(X) = \sum_{\sigma \in L} |\{\sigma | \sigma \models X\}|$$

For multiple xor branches, the frequency of their coexistence in event log L is defined as the count of traces with all the occurrence of xor branches Xi ,

$$f_L(X_1, X_2, ..., X_n) = \sum_{\sigma \in L} |\{\sigma | \forall X_i, \sigma \models X_i\}|$$

The frequency of the coexistence of multiple xor branches in positive and negative event logs reflects the correlation of those xor branches. The long-term dependency in the existing model also affects the long-term dependency in the repaired model. However, since the repaired model possibly differs from the existing model, the impact of long-term dependency from the existing model becomes difficult to detect. With limits of time, we only consider the impact from positive and negative instances on the long-term dependency.

Definition 1.7 (Correlation of xor branch)**.** The correlation to express dependency of two branches $X, Y \in BB(Q)$ is expressed into

$$d(X, Y) = d^{+}(X, Y) - d^{-}(X, Y)$$

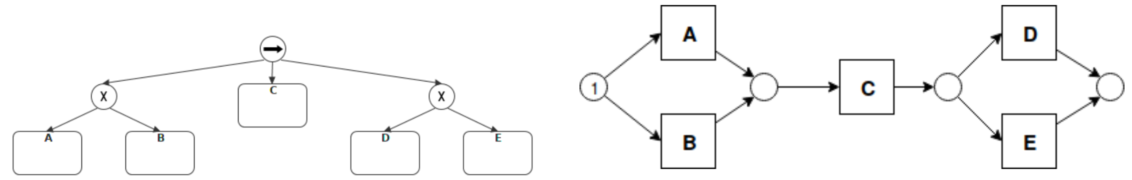
, where

$$d^{+}(X, Y) = \frac{f_{L^{+}}(X, Y)}{\sum_{Y' BB(Q), Y' \neq X} f_{L^{+}}(X, Y')}, \quad d^{-}(X, Y) = \frac{f_{L^{-}}(X, Y)}{\sum_{Y' BB(Q), Y' \neq X} f_{L^{-}}(X, Y')}$$

$f_{L^{+}}(X, Y)$ and $f_{L^{-}}(X, Y)$ are the frequency of the coexistence of X and Y, respectively in positive and negative instances. As an example, given the same model M4 and reviewed here in Figure 1.2a, and a labeled event log L,

$$L_1 := \{< A, C, D >^{10,pos}, < A, C, E >^{50,pos}, < B, C, E >^{10,pos}, < B, C, D >^{40,neg}\}$$

the frequencies of the coexistence of different xor branches are $f_{L^{+}}(A, D) = 10$, $f_{L^{+}}(A, E) = 50$, $f_{L^{+}}(B, E) = 10$, and $f_{L^{-}}(B, D) = 40$.



(a) The reviewed process tree PT

(b) The reviewed Petri net

1.2.7.2 Cases Analysis

There are various sorts of long-term dependencies that are able to happen for two xor blocks. To explain those situations better, we define concepts called sources and targets of long-term dependency and then give an example of one Petri net with long-term dependency.

Definition 1.8 (Source and target set of Long-term Dependency)**.** The source set of the long-term dependency in two xor blocks is the set of all xor branches, $LT_S := \{X | \exists Y, X \rightsquigarrow Y \in LT\}$, and target set is $LT_T := \{Y | \exists X, X \rightsquigarrow Y \in LT\}$.
For one xor branch $X \in S$, the target xor branch set relative to it with long-term dependency is defined as: $LT_T(X) = \{Y | X \rightsquigarrow Y \in LT\}$ Respectively, the source xor branch relative to one xor branch in target is $LT_S(Y) = \{X | X \rightsquigarrow Y \in LT\}$

At the same time, we use S and T to represent the set of xor branches for source and target xor block with long-term dependency. Given an Petri net in Figure **??**, two xor blocks are contained in the model which allows the following long-term situations.

1. $LT = \{A \rightsquigarrow D, A \rightsquigarrow E, B \rightsquigarrow D, B \rightsquigarrow D\}$.
   $LT_S = \{A, B\}, LT_T = \{D, E\}, |LT| = |S| * |T|$, which means long-term dependency has all combinations of source and target xor branches.

2. $LT = \{A \rightsquigarrow D, A \rightsquigarrow E, B \rightsquigarrow E\}$.
   $LT_S = \{A, B\}, LT_T = \{D, E\}$ $LT_S = S$ and $LT_T = T, |LT| < |S| * |T|$. it doesn't cover all combinations. But for one xor branch $X \in S, LT_T(X) = T$, it has all the full long-term dependency with T.

3. $LT = \{A \rightsquigarrow D, B \rightsquigarrow E\}$.
   $LT_S = \{A, B\}, LT_T = \{D, E\}$ $LT_S = S$ and $LT_T = T, |LT| < |S| * |T|$. For all xor branch $X \in S, LT_T(X) \subsetneq T$, none of xor branch X has long-term dependency with T.

4. $LT = \{A \rightsquigarrow D, B \rightsquigarrow D\}$.
   $LT_S = S, LT_T \subsetneq T$. There exists at least one xor branch $Y \in T$ which has no long-term dependency on it.

5. $LT = \{A \rightsquigarrow D, A \rightsquigarrow E\}$.
   $LT_S \subsetneq S, LT_T = T$. There exists at least one xor branch in source $X \in S$ which has no long-term dependency on it.

6. $LT = \{A \rightsquigarrow E\}$.
   $LT_S \subsetneq S, LT_T \subsetneq T$. There exists at least one xor branch in source $X \in S$ and one xor target xor branch which has no long-term dependency on it.

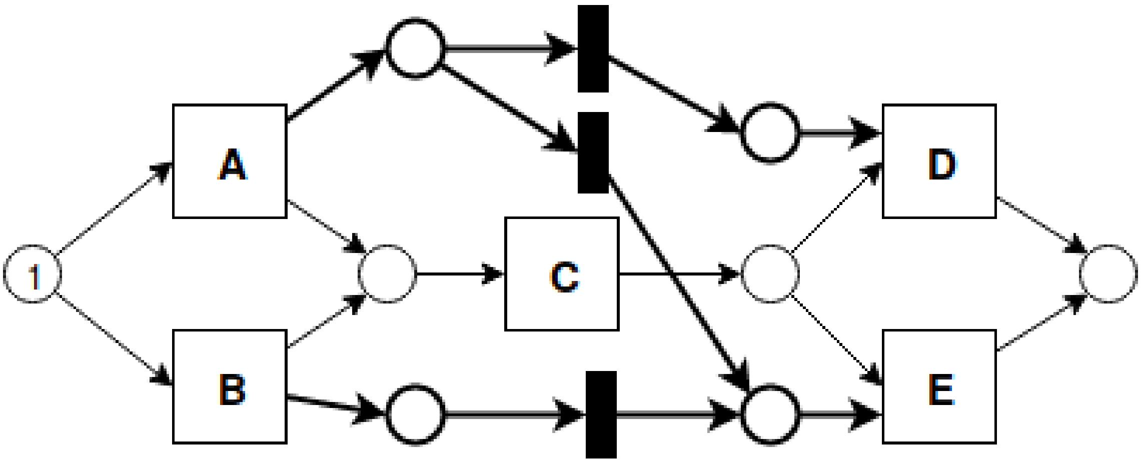


Figure 1.3: Model with long-term dependency $LT = \{A \rightsquigarrow D, A \rightsquigarrow E, B \rightsquigarrow E\}$.

7. $\emptyset$. There is no long-term dependency on this set.

In the following, we propose a method to express long-term dependency on Petri net.

1.2.7.3 Way to Express Long-term Dependency

Adding places to Petri net can limit the behavior[9], since its output transitions demand a token from it to fire themselves. When there is no token at this place, the transitions are enabled. By injecting extra places on Petri net, it can block negative behaviors which are not expected in the aspect of business performance.

Long-term dependency limits the available choices to fire transitions after the previous xor branch executes. So to express long-term dependency, our basic idea is to add places to the Petri net model. What's more, because one xor branch can be as a source to multiple long-term dependencies and one xor branch can be as a target to multiple long-term dependencies, silent transitions are also needed to address a long-term dependency explicitly.

Given arbitrary two xor blocks, $S = \{X_1, X2, ...X_m\}$ and $T = \{Y_1, Y_2, ...Yn\}$ with long-term dependency $LT = \{X_i \rightsquigarrow Y_j | 1 \leq i \leq m, 1 \leq j \leq n\}$, we add places after the source xor branches, $P_S = \{p_{X_i} | X_i \in LT_S\}$, and places before target xor branches,$P_T = \{p_{Y_j} | Y_i \in LT_T\}$. For each long-term dependency $X_i \rightsquigarrow Y_j$ in LT, there is silent transition t with $p_{X_i} \rightarrow t \rightarrow p_{Y_j}$. The steps to add silent transitions and places according to the long-term dependency are listed in algorithm 1.

One simple example is given in the Figure **??** to explain this algorithm. Given the long-term dependencies $LT = \{A \rightsquigarrow D, A \rightsquigarrow E, B \rightsquigarrow E\}$, two extra places are added respectively after A and B; Next, two places before D and E are created to express that the xor branches are involved with long-term dependency. At end, for each long-term dependency, a silent transition is generated to connect the extra places after the source xor branch to the place before target place.

Algorithm 1: Add long-term dependency between pure xor branch

```
$Y$ is dependent on $X$;
if $X$ is leaf node then
    One place is added after this leaf node ;
end
if $X$ is Seq then
    Add a place after the end node of this branch;
    The node points to the new place;
end
if $X$ 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 $Y$ is leaf node then
    One place is added before this leaf node ;
end
if $Y$ is Seq then
    Add a place before the end node of this branch;
    The new place points to this end node;
end
if $Y$ 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 $X$ and $Y$ by creating a silent transition.
```

1.2.7.4 Soundness Analysis

Following algorithm 1 by adding silent transitions and places to express long-term dependency, the model soundness can be violated. In the following section, we discuss the soundness in different situations.

Given a Petri net with long-term dependency $LT = \{X_i \rightsquigarrow Y_j | 1 \leq i \leq m, 1 \leq j \leq n\}$ on two xor blocks $S = \{X_1, X2, ...X_m\}$ and $T = \{Y_1, Y_2, ...Yn\}$, following the Algorithm 1, $P_S = \{p_{X_i} | X_i \in LT_S\}$, $P_T = \{p_{Y_j} | Y_i \in LT_T\}$, and silent transitions $E = \{\epsilon | p_{X_i} \to \epsilon \to p_{Y_j}\}$ are added.

The Petri net is sound if and only if (1)the soundness outside xor blocks with long-term dependency is not violated; and (2) soundness between xor blocks is kept. In the following, we check the model soundness with long-term dependency after applying Algorithm 1.

Soundness outside xor blocks.
Proof: he added silent transitions and places do not violate the execution outside of the xor blocks, because the extra tokens that are generated due to long-term dependency are constrained in the xor blocks, and it doesn't affect the token flows outside. As we know, the original model is sound. So the soundness outside xor blocks is not violated.

Soundness inside xor blocks.
For all xor branches in S, only one branch can be fired. Without loss of generality, X_i is assumed to be enabled. After firing X_i, the marking distribution on the extra places are

$$M(p_{X_i}) = 1; \quad \forall p_{X_i'} \in P_S, i' \neq i, M(p_{X_i'}) = 0$$

If $LT_S = S, LT_T = T$, adding the long-term dependency in this situation doesn't violate the model soundness, we prove it in the following part.

- safeness. Places cannot hold multiple tokens at the same time.
  For all extra places p_{X_i} and p_{Y_j},

  $$\forall p_{X_i} \in P_S, \sum M(p_{X_i}) = 1$$

  Because $LT_S = S, X_i \in S$, so $X_i \in LT_S$, there exists one Y_j with $X_i \to Y_j$ and one $\epsilon, p_{X_i} \to \epsilon \to p_{Y_j}$. After firing X_i, the transition ϵ becomes enable. After executing ϵ, the marking distribution turns to

  $$M(p_{Y_j}) = 1; \quad \forall p_{Y_j'} \in P_T, j' \neq j, M(p_{Y_i}) = 0$$

  So whenever the marking distribution in the extra places are

  $$\sum M(p_{X_i}) \leq 1, \sum M(p_{Y_j}) \leq 1$$

- proper completion. If the sink place is marked, all other places are empty.
  After firing Y_j, all the extra places hold no token. So it does not violate the proper completion.

- option to complete. It is always possible to reach the final marking just for the sink place.
  There is always one Y_j enabled after firing X_i to continue the subsequent execution.

- no dead part. For any transition there is a path from source to sink place through it.
  Because all $Y_j \in T$ are also in LT_T, there exists at least one $X_i \in S$ with long-term dependency with Y_j. After X_i is fired, one token is generated on the extra place p_{X_i} and can be consumed by silent transition ϵ in $p_{X_i} \to \epsilon \to p_{Y_j}$ to produce a token in p_{Y_j}, which enables xor branch Y_j and leaves no dead part.

Else, in other situation, the model becomes unsound.
If $LT_S \neq S, or \quad LT_T \neq \emptyset$, there exists one xor branch X_i with $X_i \notin LT_S$. When X_i is fired, it generates one token at place p_{X_i}, this token cannot be consumed by any Y_j. So it violates the proper completion.
If $LT_T \neq T$, there exists one $Y_j \notin LT_T, \nexists X_i, X_i \rightsquigarrow Y_j$, so with two input places but $Token(p_{Y_j}) = 0$, Y_j becomes the dead part, which violates the soundness again.

As a conclusion, to keep Petri net with long-term dependency sound, only situation $LT_S = S, LT_T = T$ is considered. However, when the long-term dependency is full connected where each combination of xor branches from source and target xor block has long-term dependency, namely xor branches can be chosen freely, we don't add any places and silent transitions on the model.

1.2.8 Reduce Silent Transitions

Our method to represent long-term dependency can introduce redundant silent transitions and places, which complicates the model. So, we post process the Petri net with long-term dependency to delete redundant silent transitions and places.

Proposition 1.9. Given a silent transition ϵ in Petri net with one input place P_{in} and one output place P_{out}, if $|Outedges(P_{in})| \geq 2 \quad and \quad |Inedges(P_{out})| \geq 2$, the silent transitions can not be deleted. Else, the silent transitions is able to delete, meanwhile the P_{in} and P_{out} can be merged into one place. This reduction does not violate the soundness and does not change the model behavior.

Soundness Proof. If a silent transition t is able to delete, then

$$|Outedges(P_{in})| \leq 1 \quad (1) \quad or,$$

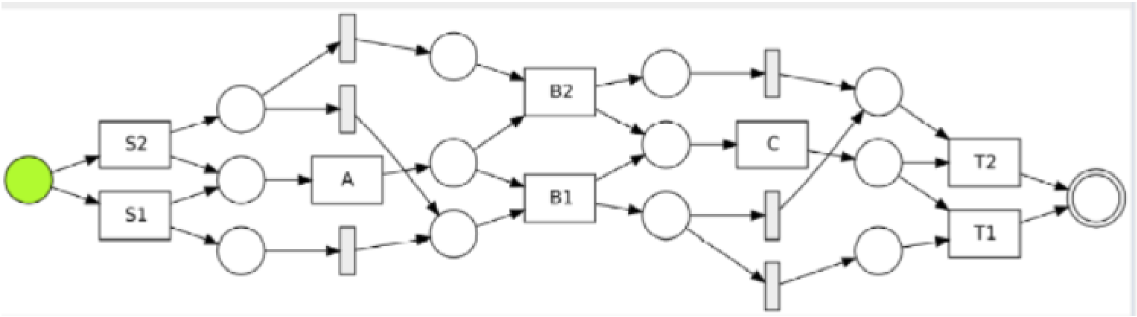
$$|Inedges(P_{out})| \leq 1 \quad (2).$$

When in case (1), P_{in} contains a token that is always passed to P_{out} by silent transitions t. After deleting the silent transition, the token is generated directly on P_{out}. Since t is silent transition, it won't affect the model behavior.
when in case (2), P_{out} contains a token that is always passed from P_{in} by silent transitions t. After deleting the silent transition, the token is remained on P_{in}, which enables the later execution after the original P_{out}. Since t is silent transition, it won't affect the model behavior.
In other cases, $Outedges(P_{in})| \geq 2 \quad and \quad |Inedges(P_{out})| \geq 2$, which means that the source xor branch with output place P_{in} has at lest two long-term dependencies; the target xor branch with input place P_{out} has at least two long-term dependencies. If we delete this silent transitions, the long-term dependencies are mixed together which allows more unexpected behaviors. Therefore, silent transition in this situation is necessary to hold long-term dependency. □

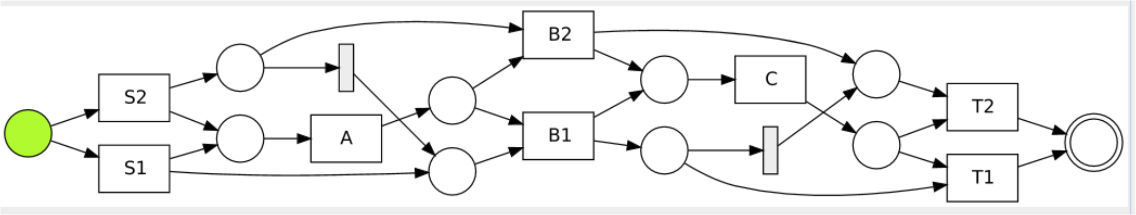
One example is given in the following graph. M_{lt} in Figure **??** has long-term dependency expressed in the silent transitions and places. The silent transition for $S2 \rightsquigarrow B1$ and silent transition for $B1 \rightsquigarrow T2$ belongs to the case (1). So they are kept in the model, while the other silent transitions are deleted. After reducing the redundant silent transitions, the model becomes M_r shown in Figure **??**. Those two models have the same behavior, yet the reduced model is simpler.

1.2.9 Concrete Architecture

At last, we assemble all the modules together and give an overview architecture of our repair techniques. We reuse existing modules in gray rectangles in Figure 1.5, e.g. IM-Log2Dfg to convert an event log into directly-follow graph, Petrinet2TransitionSystem to



(a) A Petri net M_{lt} with redundant silent transitions



(b) Petri net M_r with reduced silent transitions

transform a Petri net into a transition system. The other modules are programmed according to our specific needs and achieve the repair algorithm mentioned before. To achieve a preciser Petri net, the module to add long-term dependency becomes a necessary part. Yet, reduction on redundant places and silent transitions is optional.

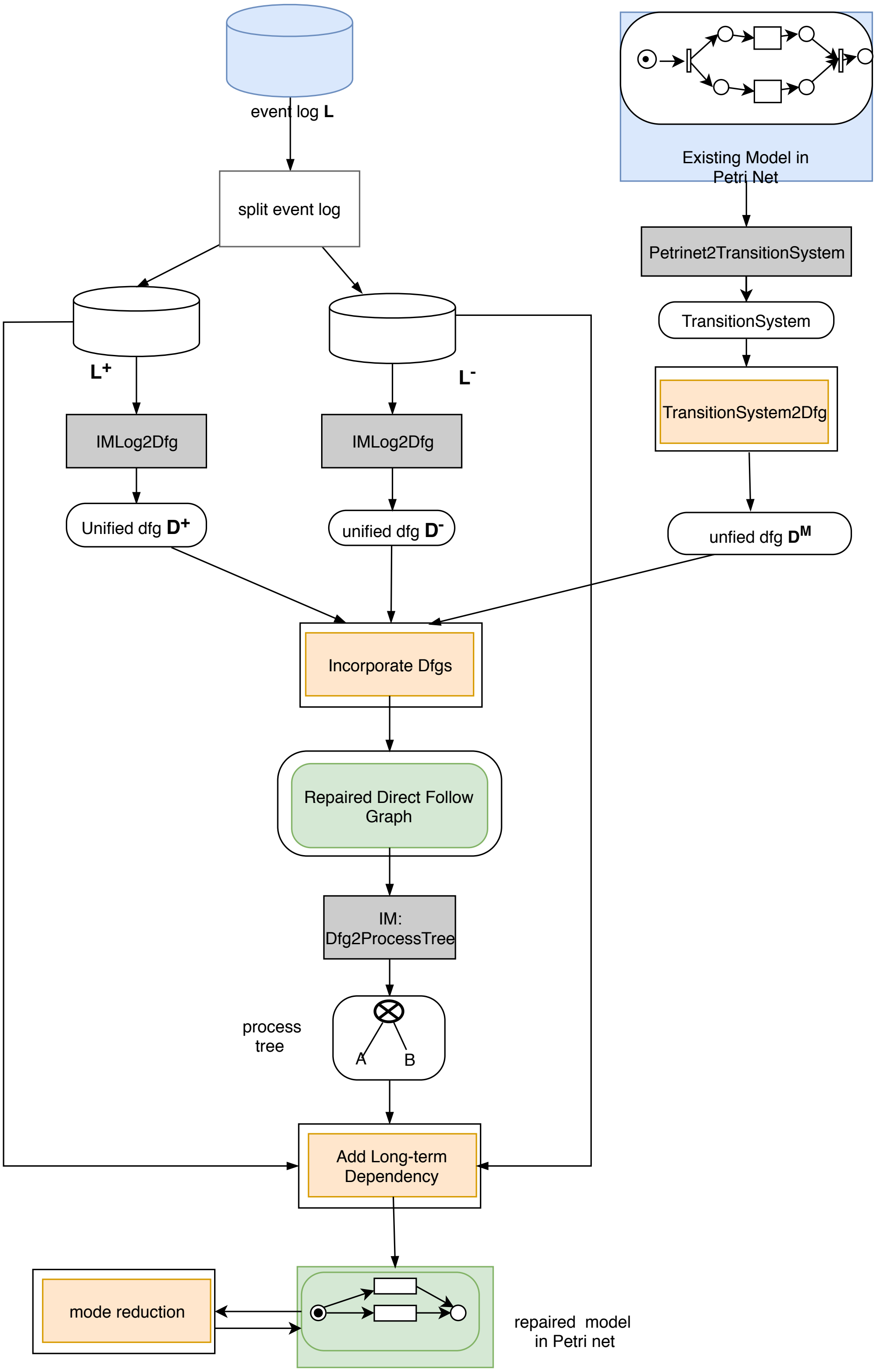


Figure 1.5: Model Repair Architecture – Rectangles represents processes and output data in eclipse shape, especially customized processes and data are in doubled lattice shape. Input event log and existing model are in blue, KPIs are in cloud. The output is a petri net in purple.

Bibliography

[1] Wil Van Der Aalst. *Process mining: discovery, conformance and enhancement of business processes*, volume 2. Springer, 2011.

[2] Dirk Fahland and Wil MP van der Aalst. Repairing process models to reflect reality. In *International Conference on Business Process Management*, pages 229–245. Springer, 2012.

[3] Mahdi Ghasemi and Daniel Amyot. From event logs to goals: a systematic literature review of goal-oriented process mining. *Requirements Engineering*, pages 1–27, 2019.

[4] Marcus Dees, Massimiliano de Leoni, and Felix Mannhardt. Enhancing process models to improve business performance: a methodology and case studies. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 232–251. Springer, 2017.

[5] Dirk Fahland and Wil MP van der Aalst. Model repair—aligning process models to reality. *Information Systems*, 47:220–243, 2015.

[6] Wil Van der Aalst. Data science in action. In *Process Mining*, pages 3–23. Springer, 2016.

[7] Wil Van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.

[8] Sander JJ Leemans, Dirk Fahland, and Wil MP 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*, pages 311–329. Springer, 2013.

[9] Robin Bergenthum, Jörg Desel, Robert Lorenz, and Sebastian Mauser. Process mining based on regions of languages. In *International Conference on Business Process Management*, pages 375–383. Springer, 2007.

[10] Jordi Cortadella, Michael Kishinevsky, Luciano Lavagno, and Alex Yakovlev. Synthesizing petri nets from state-based models. In *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pages 164–171. IEEE, 1995.

[11] Jan Martijn EM Van der Werf, Boudewijn F van Dongen, Cor AJ Hurkens, and Alexander Serebrenik. Process discovery using integer linear programming. In *International conference on applications and theory of petri nets*, pages 368–387. Springer, 2008.

[12] Boudewijn F Van Dongen, AK Alves De Medeiros, and Lijie Wen. Process mining: Overview and outlook of petri net discovery algorithms. In *Transactions on Petri Nets and Other Models of Concurrency II*, pages 225–242. Springer, 2009.

[13] Ana Karla A de Medeiros, Anton JMM Weijters, and Wil MP van der Aalst. Genetic process mining: an experimental evaluation. *Data Mining and Knowledge Discovery*, 14(2):245–304, 2007.

[14] Anton JMM Weijters and Wil MP Van der Aalst. Rediscovering workflow models from event-based data using little thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, 2003.

[15] Stijn Goedertier, David Martens, Jan Vanthienen, and Bart Baesens. Robust process discovery with artificial negative events. *Journal of Machine Learning Research*, 10 (Jun):1305–1340, 2009.

[16] Seppe KLM vanden Broucke, Jochen De Weerdt, Jan Vanthienen, and Bart Baesens. Determining process model precision and generalization with weighted artificial negative events. *IEEE Transactions on Knowledge and Data Engineering*, 26(8): 1877–1889, 2014.

[17] Hernan Ponce-de León, Josep Carmona, and Seppe KLM vanden Broucke. Incorporating negative information in process discovery. In *International Conference on Business Process Management*, pages 126–143. Springer, 2016.

[18] Josep Carmona and Jordi Cortadella. Process discovery algorithms using numerical abstract domains. *IEEE Transactions on Knowledge and Data Engineering*, 26(12): 3064–3076, 2014.

[19] BF Van Dongen, Jan Mendling, and WMP Van Der Aalst. Structural patterns for soundness of business process models. In *2006 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06)*, pages 116–128. IEEE, 2006.

[20] Wil van der Aalst. *Process Mining: Data Science in Action*. Springer Publishing Company, Incorporated, 2nd edition, 2016. ISBN 3662498502, 9783662498507.

[21] Joos C. A. M. Buijs, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. On the role of fitness, precision, generalization and simplicity in process discovery. In *OTM Conferences*, 2012.

[22] Eindhoven Technical University. © 2010. Process Mining Group. Prom introduction. URL `http://www.promtools.org/doku.php`.

[23] Kefang Ding. Incorporatenegativeinformation. URL `http://ais-hudson.win.tue.nl:8080/job/IncorporateNegativeInformation/`.

[24] Felix Mannhardt, Massimiliano De Leoni, and Hajo A Reijers. The multi-perspective process explorer. *BPM (Demos)*, 1418:130–134, 2015.