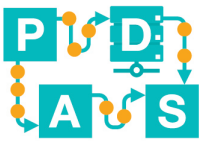


RWTH AACHEN UNIVERSITY

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

The acknowledgments and the people to thank go here, don't forget to include your project advice.

Abstract

Process mining is based on business execution history in the form of event log, aim to bring visual insights on the business process and to support process analysis and enhancements. It bridges the gap between traditional business process management and advanced data analysis techniques like 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 throughput time, and cost. 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.

Chapter 1

Algorithm

This chapter describes our repair algorithm to incorporate the negative instances on process enhancement. In the beginning, the architecture of this algorithm is provided to give an overview. Details of main steps are represented in the following order. Firstly, the impact of the existing model, positive and negative instances are balanced in the media of the directly-follow relations. Inductive Miner is then applied to mine process models from those directly-follows relations. Next, we detect and add long-term dependency on the generated process models. Furthermore, the model in Petri net with long-term dependency is post-processed for the sake of simplicity.

1.1 Architecture

Figure 1.1 shows the steps of our strategy to enhance a process model. The basic inputs are an event log, and a Petri net. The traces in event log have an attribute for the classification labels of positive or negative in respect to some KPIs of business processes. The Petri net is the referenced model for the business process. To repair model with negative instances, those steps are conducted.

- *Generate directly-follows graphs* Three directly-follows graphs are generated respectively for the existing model, positive instance and negative instances from the event log.
- *Repair directly-follows graph* The three directly-follows graphs are combined into one single directly-follows graph after balancing their impact.
- *Mine models from directly-follows graph* Process models are mined from the repaired directly-follows graph by Inductive Miner as intermediate results.
- *Add long-term dependency* Long-term dependency is detected on the intermediate models and finally added on the Petri net. To simplify the model, the reduction of silent transitions can be applied at the end.

More details can be provided in the following sections.

1.2 Generate directly-follows graph

Originally, the even log L is split into two sublogs, called L_{pos} and L_{neg}. L_{pos} contains the traces which are labeled as positive, while L_{neg} contains the negative instances in

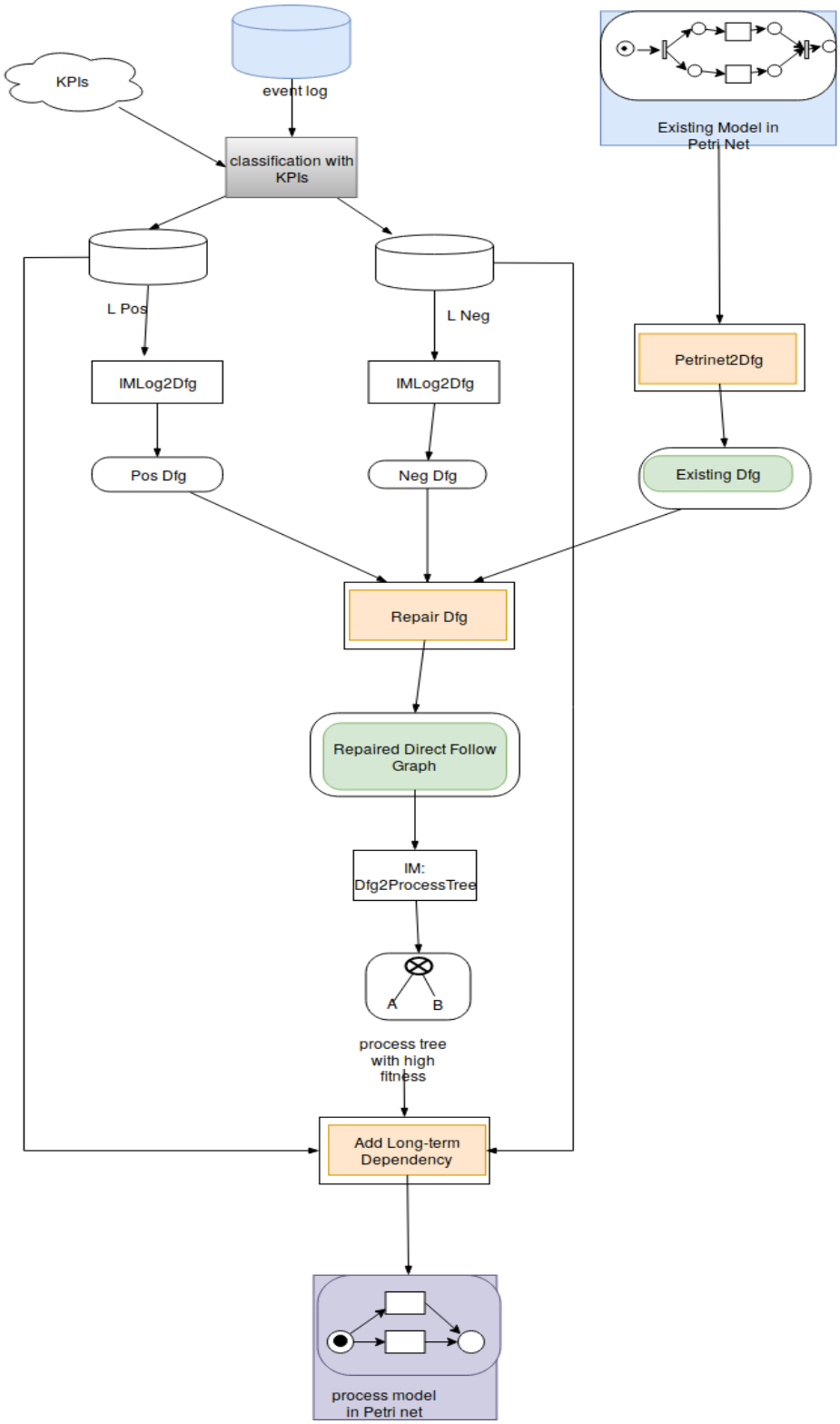


Figure 1.1: Model Repair Architecture – Rectangles represents processes and output data in eclipse shape, especially customized processes and data are in doubled lattice shape. Input event log and existing model are in blue, KPIs are in cloud. The final output is a Petri net in purple.

the event log. Then, those two sublogs are passed to an existing procedure *IMLog2Dfg* from [1]. *IMLog2Dfg* traverses traces in the event log, extracts directly-follows relations of activities, and generates a directly-follows graph based on those relations. By using *IMLog2Dfg*, $G(L_{pos})$ and $G(L_{neg})$ are generated respectively from positive and negative instances.

To generate a directly-follows graph from a Petri net, we gather the model behaviors by building a transition system. A transition system for a Petri net is composed of *states* and *transitions*. *States* are the possible markings of this Petri net. *Transitions* are activities from event log. Those transitions connect states as an arc from its enable marking to the marking after its execution. For each state, transitions pointed from this state are directly followed by transitions pointed to this state. By checking the transition system. directly-follows relations for the models are extracted. Based on those relations, we create a directly-follows graph for the existing 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.

1.3 Repair directly-follows graph

To combine all information of the directly-follows graphs from the positive , negative instances and the existing model, namely $G(L_{pos})$, $G(L_{neg})$ and $G(L_{ext})$, the cardinality in directly-follows graphs is unified as the percentage to the sum of total cardinalities with a range [0-1].

Definition 1.1 (Unified cardinality)**.** Given a directly-follows graphs $G(L)$ for a model, the unified cardinality of each directly-follows relation is defined as

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

$$Cardinality(E(*,*)) = \sum Cardinality(E(X,Y)|E(X,Y) \in G(L))$$

for start activities A,

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

Similarly for end activities B,

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

$E(*,*)$ means all edges in the directly-follows graph, $E(*,B)$ means all edges with target B, $Start(*)$ represents all start nodes, and $End(*)$ represents all end nodes.

After the unification, the impact from existing model, positive and negative instances is in the same level and is balanced by subtracting the negative unification from the sum of existing and positive unification. The output from this balance is as the generated unification for the repaired model.

Definition 1.2 (Unified cardinality for directly-follows graph G_{new})**.** The unified cardinality for the repaired directly-follows relations is defined in the following.

- For one directly-follows relation,

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

if $U(E_{G_{new}}(A,B)) > 1, U(E_{G_{new}}(A,B)) = 1$,
if $U(E_{G_{new}}(A,B)) < 0, U(E_{G_{new}}(A,B)) = 0$;

- For start acivities A, we have

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

if $U(Start_{G_{new}}(A)) > 1, U(Start_{G_{new}}(A)) = 1$,
if $U(Start_{G_{new}}(A)) < 0, U(Start_{G_{new}}(A)) = 0$;

- For end activities B, we have

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

if $U(End_{G_{new}}(A)) > 1, U(End_{G_{new}}(A)) = 1$,
if $U(End_{G_{new}}(A)) < 0, U(End_{G_{new}}(A)) = 0$.

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 in range [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 unification of directly-follows graph G_{new})**.** Given the control weight C_{ext},C_{pos}, and C_{neg} in range of [0-1], the weighted unification for G_{new} is defined below.

- For one directly-follows relation,

$$Wu(E_{G_{new}}(A,B)) = C_{ext} * U(E_{G_{ext}}(A,B)) + C_{pos} * U(E_{G_{pos}}(A,B)) - C_{neg} * U(E_{G_{neg}}(A,B))$$

if $Wu(E_{G_{new}}(A,B)) > 1, Wu(E_{G_{new}}(A,B)) = 1$,
if $Wu(E_{G_{new}}(A,B)) < 0, Wu(E_{G_{new}}(A,B)) = 0$;

- For start acivities A, we have

$$Wu(Start_{G_{new}}(A)) = C_{ext} * U(Start_{G_{ext}}(A)) + C_{pos} * U(Start_{G_{pos}}(A)) - C_{neg} * U(Start_{G_{neg}}(A))$$

if $Wu(Start_{G_{new}}(A)) > 1, Wu(Start_{G_{new}}(A)) = 1$,
if $Wu(Start_{G_{new}}(A)) < 0, Wu(Start_{G_{new}}(A)) = 0$;

- For end activities B, we have

$$Wu(End_{G_{new}}(A)) = C_{ext} * U(End_{G_{ext}}(A)) + C_{pos} * U(End_{G_{pos}}(A)) - C_{neg} * U(End_{G_{neg}}(A))$$

if $Wu(End_{G_{new}}(A)) > 1, Wu(End_{G_{new}}(A)) = 1$,
if $Wu(End_{G_{new}}(A)) < 0, Wu(End_{G_{new}}(A)) = 0$.

By adjusting the weight of C_{ext},C_{pos},C_{neg}, different focus can be reflected by the model. For example, by setting $C_{ext} = 0$,$C_{pos} = 1$,$C_{neg} = 1$, the existing model is ignored in the repair, while $C_{ext} = 1$,$C_{pos} = 0$,$C_{neg} = 0$, the original model is kept.

After getting the weighted unification, we can assign cardinality for the directly-follows graph G_{new} by multiplying weighted unification with the total number of cardinalities of $G(L_{pos})$, $G(L_{neg})$ and $G(L_{ext})$.

1.4 Mine models from a directly-follows graph

The result from the last step above is a generated directly-follows graph with weighted unified cardinality. We mine process models from this graph by procedure *Dfg2ProcessTree* in the Figure 1.1. This procedure was introduced with Inductive Miner[1]. 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.

1.5 Add long-term dependency

Due to the intrinsic characters of Inductive Miner, the dependency from activities which are not directly-followed can't be discovered. To make the generated model preciser, we detect the long-term dependency and add it on the process model.

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

To analyze the exclusive choice structure, we use process tree as an intermediate process model due to several reasons: (1) easy to extract the exclusive choice structure from process tree, since process tree is block-structured. (2) easy to transform a process tree to a Petri net.

An exclusive choice structure be represented as Xor in a process tree. A subtree of this structure is called xor branch in this thesis for convenience.

Definition 1.4 (Xor branch)**.** Given an xor block $Q = \times(Q_1, Q_2, ..Q_n)$, Q_i is one xor branch with respect to Q. It can be rewritten as $XORB_{Q_i}$ to represent one xor branch Q_i. 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 two arbitrary xor branches with long-term dependency, they have to satisfy the conditions: (1) they have a sequential 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 1.5 (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 listed below are necessary.

Definition 1.6 (Xor branch frequency)**.** Xor branch $XORB_X$ frequency in event log L, $F_L(XORB_X)$, is the count of traces with the execution of $XORB_X$.
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 $XORB_{Xi}$, written as

$$F_L(XORB_{X1}, XORB_{X2}, ..., XORB_{Xn}).$$

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 for two branches is expressed into

$$Wlt(XORB_X, XORB_Y) = Wlt_{pos}(XORB_X, XORB_Y) - Wlt_{neg}(XORB_X, XORB_Y)$$

, where

$$Wlt_{pos}(XORB_X, XORB_Y) = \frac{F_{pos}(XORB_X, XORB_Y)}{F_{pos}(XORB_X, *)},$$

$$Wlt_{neg}(XORB_X, XORB_Y) = \frac{F_{neg}(XORB_X, XORB_Y)}{F_{neg}(XORB_X, *)}$$

The $F_{pos}(XORB_X, XORB_Y)$ and $F_{neg}(XORB_X, XORB_Y)$ are the frequency of the coexistence of $XORB_X$ and $XORB_Y$, respectively in positive and negative event logs.

1.5.1 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 $X \in XORB_S$, 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 $XORB_S$ and $XORB_T$ 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.

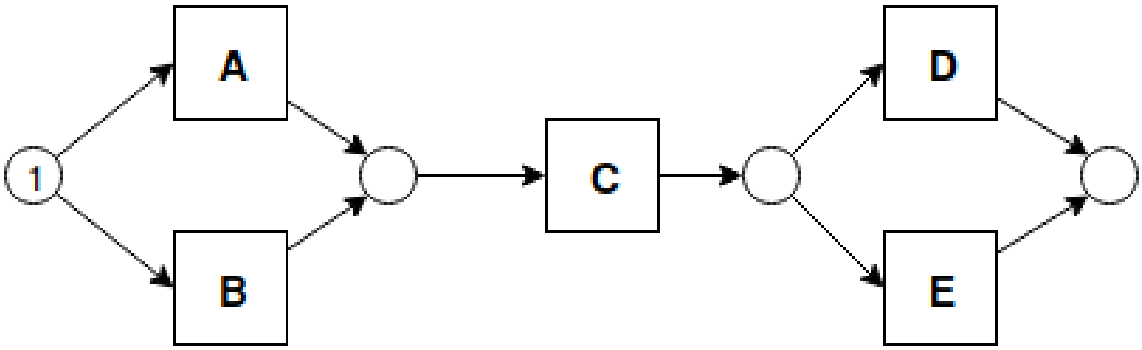


Figure 1.2: One Petri net with two two xor blocks

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

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

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

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

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

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

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.

1.5.2 Way to express long-term dependency

Adding places to Petri net can limit the behavior[2], 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.

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

```
1  XORB_Y is dependent on XORB_X;
2  if XORB_X is leaf node then
3  |  One place is added after this leaf node. ;
4  end
5  if XORB_X is Seq then
6  |  Add a place after the end node of this branch;;
7  |  The node points to the new place;;
8  end
9  if XORB_X is And then
10 |  Create a place after the end node of every children branch in this And xor
   |   branch; ;
11 |  Combine all the places by a silent transition after those places; ;
12 |  Create a new place directly after silent transition to represent the And xor
   |   branch; ;
13 end
14 if XORB_Y is leaf node then
15 |  One place is added before this leaf node. ;
16 end
17 if XORB_Y is Seq then
18 |  Add a place before the end node of this branch;;
19 |  The new place points to this end node;;
20 end
21 if XORB_Y is And then
22 |  Create a place before the end node of every children branch in this And xor
   |   branch; ;
23 |  Combine all the places by a silent transition before those places; ;
24 |  Create a new place directly before silent transition to represent the And xor
   |   branch; ;
25 end
26 Connect the places which represent the XORB_X and XORB_Y by creating a
     silent transition.
```

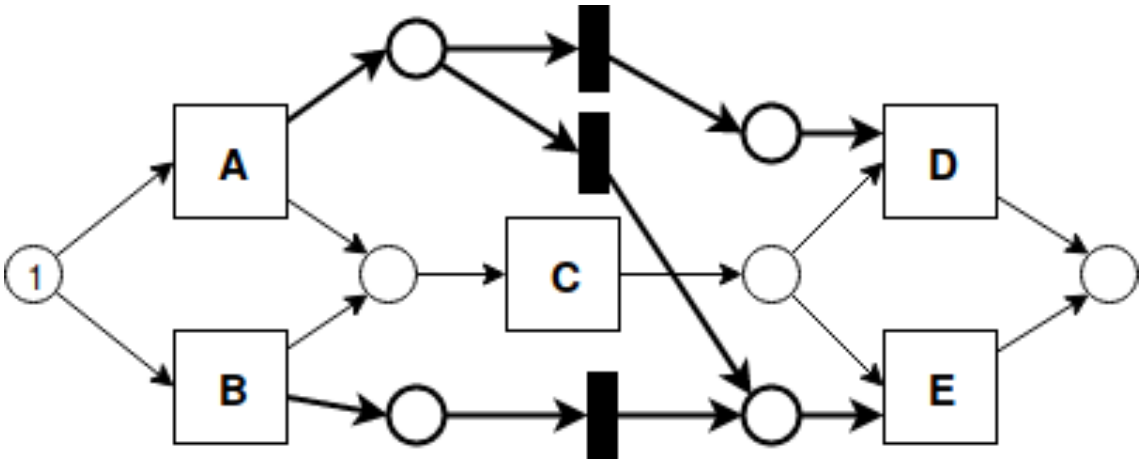


Figure 1.3: Model with long-term dependency $LT = \{A \rightsquigarrow D, A \rightsquigarrow E, B \rightsquigarrow E\}$.

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.

1.5.3 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 $E = \{\epsilon | p_{X_i} \rightarrow \epsilon \rightarrow p_{Y_j}\}$ 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.

(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.

(2) 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

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

If $LT_S = S, LT_T = T$, the following conditions are checked.

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

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

- 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 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 t in $p_{X_i} \to t \to p_{Y_j}$ to produce a token in p_{Y_j}, which enables xor branch Y_j and leaves no dead part.

Since it fulfills all the conditions above, adding the long-term dependency in this situation doesn't violate the model soundness.

Else, in other situation $LT_S \neq S = X_i, LT_T \neq \emptyset$, there exists one xor branch$X_i with X_i \notin LT_S, Y_j \in LT_T$, when X_i is selected, it generates one token at place p_{X_i}, this token can not be consumed by any Y_j. So it violates the proper completion. $If \quad LT_T \neq T = Y_j$, 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.
In source xor block S, only one xor branch X_i can be triggered.

As a conclusion, to keep Petri net with long-term dependency sound, only situation $LT_S = S, LT_T = T$ is considered. In this situations, when the long-term dependency is full connected where xor branches can be chosen freely, we don't add any places and silent transitions on the model.

1.6 Reduce Silent Transitions

Our method to represent long-term dependency can introduce redundant silent transitions and places. On one hand, it complicates the model; on the other hand, it causes the model unsound where the extra silent transitions suspend the execution and therefore violates the soundness condition of proper completion. So, in this step, we postprocess the Petri net to reduce silent transitions.

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

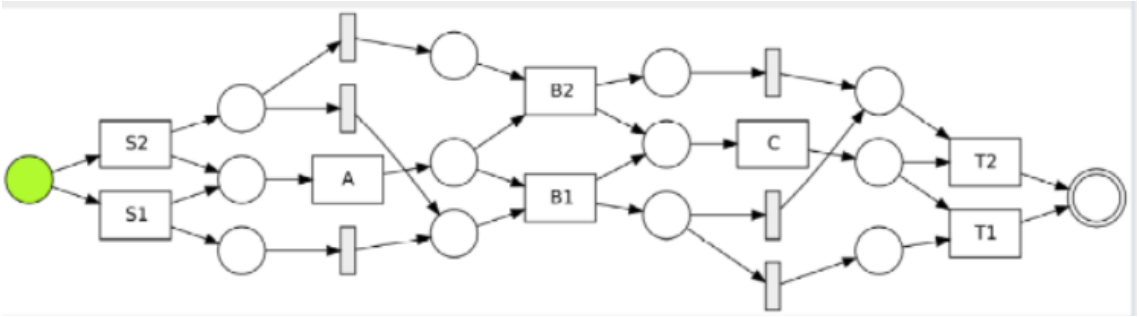
$$|Outedges(P_{in})| \leq 1$$

, or

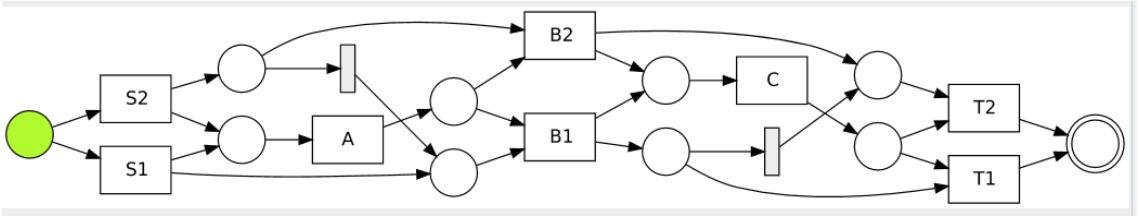
$$|Inedges(P_{out})| \leq 1$$

,
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. □

One example is given in the following graph.



(a) A Petri net with redundant silent transitions



(b) A Petri net with reduced silent transitions

Chapter 2

Implementation

ProM is an open-source extensible framework. It supports wide process mining techniques in the form of plug-ins[3]. The algorithm to incorporate negative information is implemented as one plug-in called ***Repair Model By Kefang*** in ProM and released online[4]. This chapter is divided into four parts to describe the whole implementation. Firstly, the inputs and outputs of this implementation is introduced. After accepting the inputs, a directly-follows graph is constructed by dfg-method and later converted into process tree or Petri net process model. Next, the implementation to add long-term dependency on the process model from last step is shown. At last, another feature is displayed to show the brief evaluation result based on confusion matrix.

2.1 Inputs And Outputs

The plug-in inputs are an event log L with labels to classify each trace and an existing model N possible in multiple forms.

- Acceptable event log L with labels. Labels are one trace attribute and used to identify the positive and negative instances.
- Acceptable Process Models
  - Petri net + Initial Marking.
  - Accepting Petri net.
    The initial marking is already included into the accepting Petri net.
  - Petri net.
    In this situation, the initial marking is guessed automatically in the background program.

The outputs are one process model and its corresponding initial marking. They are exported from the control panel in the result view and can be in various forms.

- Petri net with long-term dependency after Reducing Silent Transition
- Petri net with long-term dependency
- Petri net without long-term dependency
- Process tree

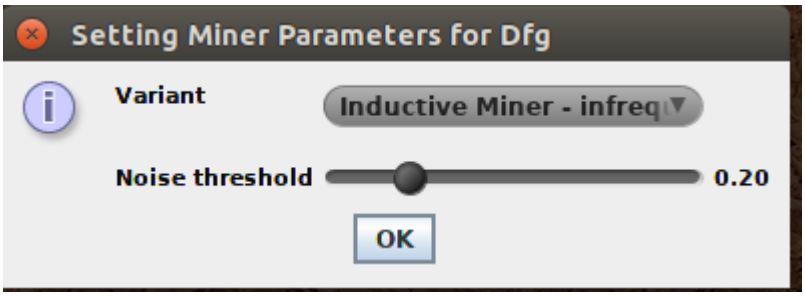


Figure 2.1: Inductive Miner Parameter Setting

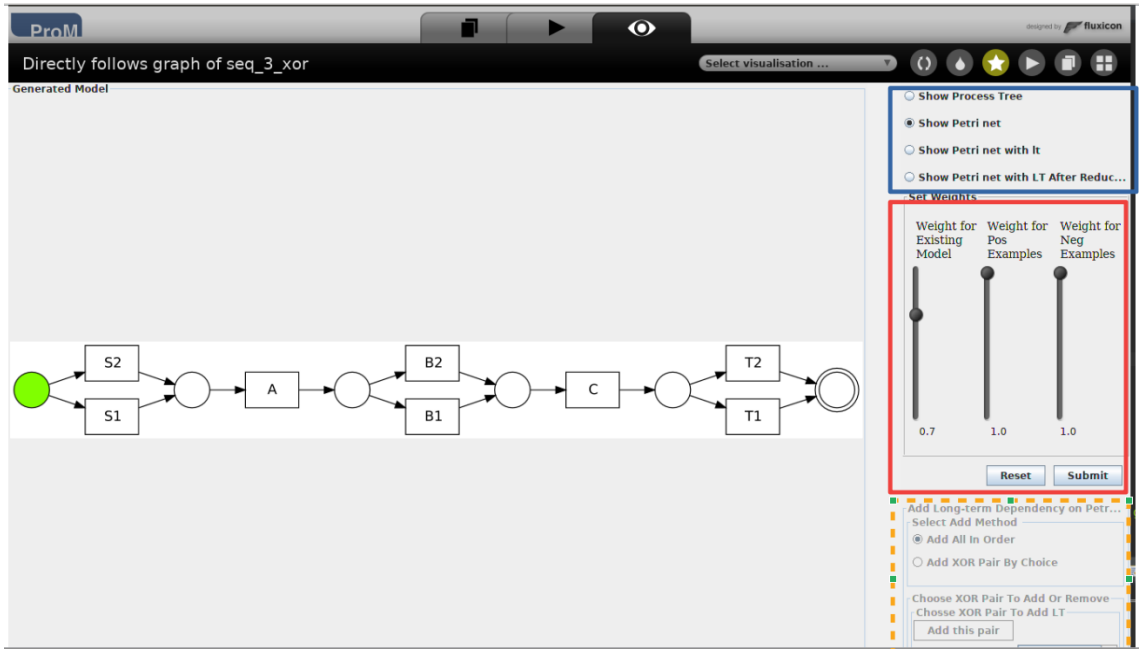


Figure 2.2: Generated Petri net without long-term dependency

2.2 Implementation of dfg-method

Firstly the dialog for options to generate directly-follows graphs from event log pops up. Event classifier are set by those dialogs. Subsequently, a dialog is shown to set the Inductive Miner parameters. The parameters include the Inductive Miner variant and the noise threshold to filter the data. The dialog is displayed in Figure 2.1.
After setting the parameters, process models of process tree and Petri net without long-term dependency can be generated by Inductive Miner and displayed in the result view in Figure 2.2. The left side is the model display area. To allow more flexibility, this plug-in are interactive by the control panel, which is the right side of result view. Originally, only the generated model type and the weight sliders are enabled, while the control panel for adding long-term dependency are invisible.

The model type are in the blue rectangle marked in Figure 2.2. It has 4 options to control the generated model type. Currently, the option "Show Petri net" is chosen, so the constructed model is Petri net without long-term dependency. The weights sliders are in red rectangle. It enables to adjust the weights on the existing model, positive

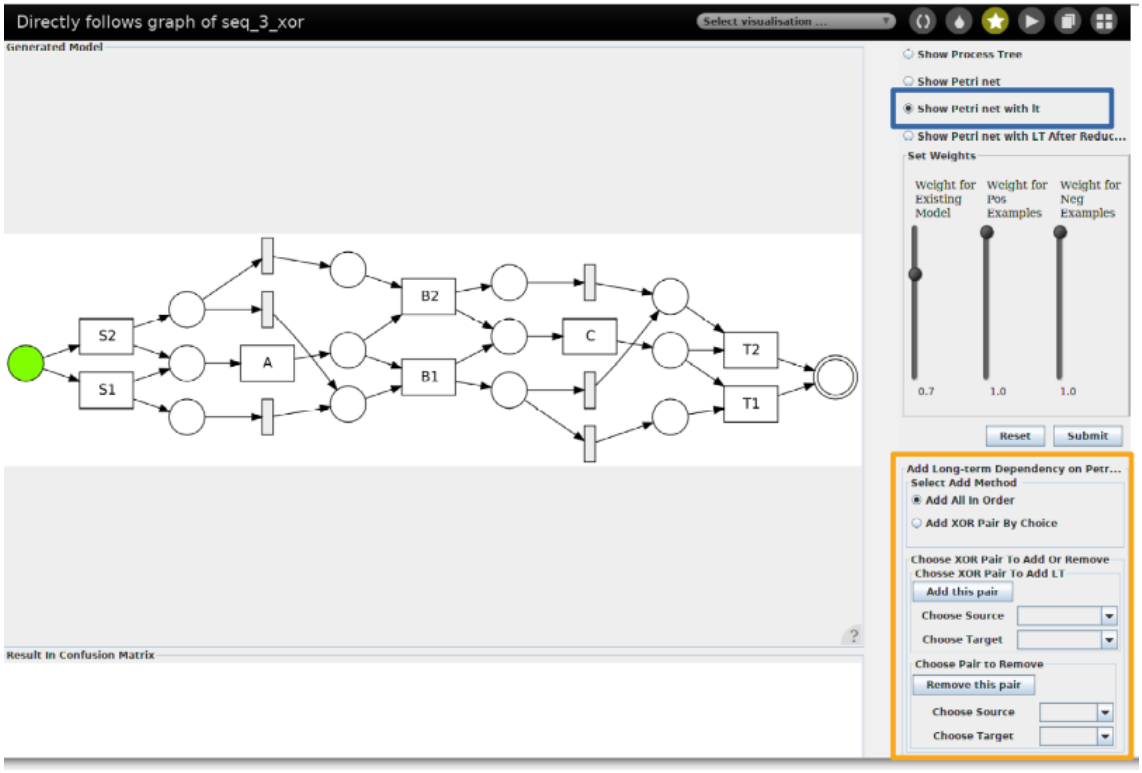


Figure 2.3: Petri Net with long-term dependency

and negative instances. Once submitted those options, different process models are mined under different weights. The rectangle in orange are the invisible part to control long-term dependency options. It is discussed in the next section.

2.3 Implementation of Adding Long-term Dependency

If the model ro generate is Petri net with long-term dependency, the program to add long-term dependency is triggered. This program in the background detects and puts places and silent transitions on Petri net directly mined from Inductive Miner to add long-term dependency. As comparison, the same weight setting is kept like the Figure 2.2, but the option to show a Petri net with long-term dependency is chosen. The resulted model is Figure 2.3.

Meanwhile, the control part of adding long-term dependency turns visible, which is in the orange rectangle in Figure 2.3. It has two main options, one is to consider all long-term dependency existing in the model, the other is to choose the part manually. It allows more flexibility for users. Below those two options, it is the manual selection panels, including control part to add and remove pair. As an example, the blocks Xor(S1,S2) and Xor(T1,T2) are chosen to add long-term dependency. It results in the model in Figure 2.4.

By choosing *Petri net with LT After Reducing* in model type option panel, silent transitions are reduced to simplify the model. Under the same setting in Figure 2.2, the simpler model in Figure 2.5 is constructed, after the post processing of reducing silent transitions.

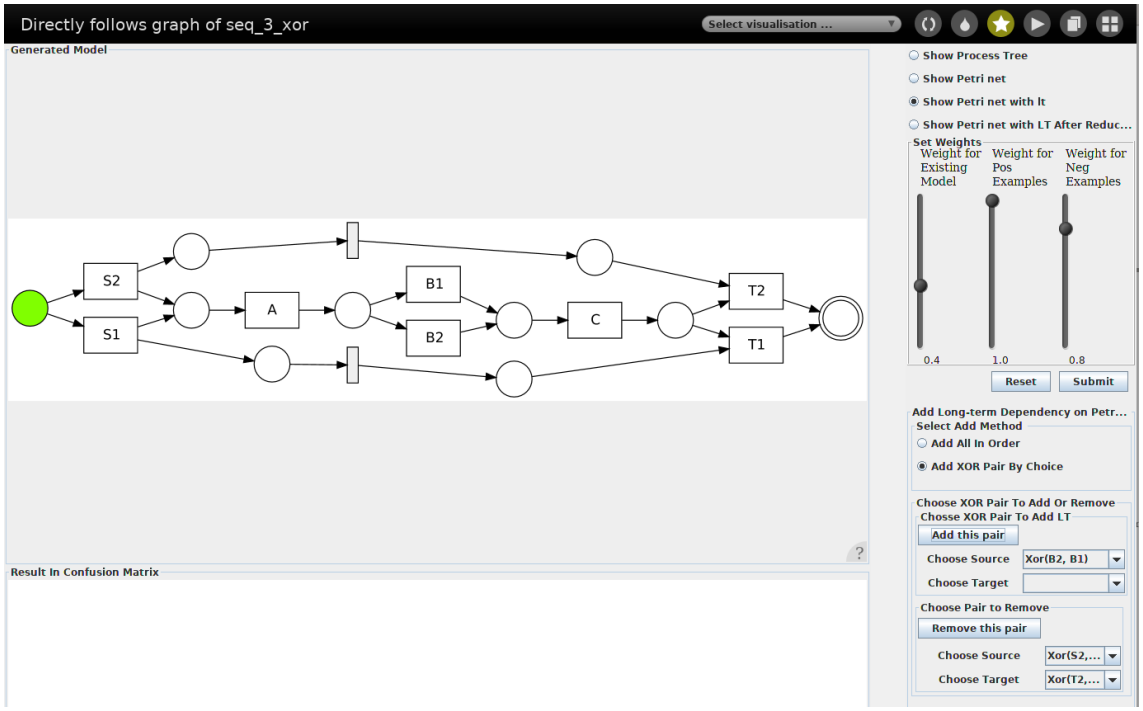


Figure 2.4: Petri net with selected long-term dependency

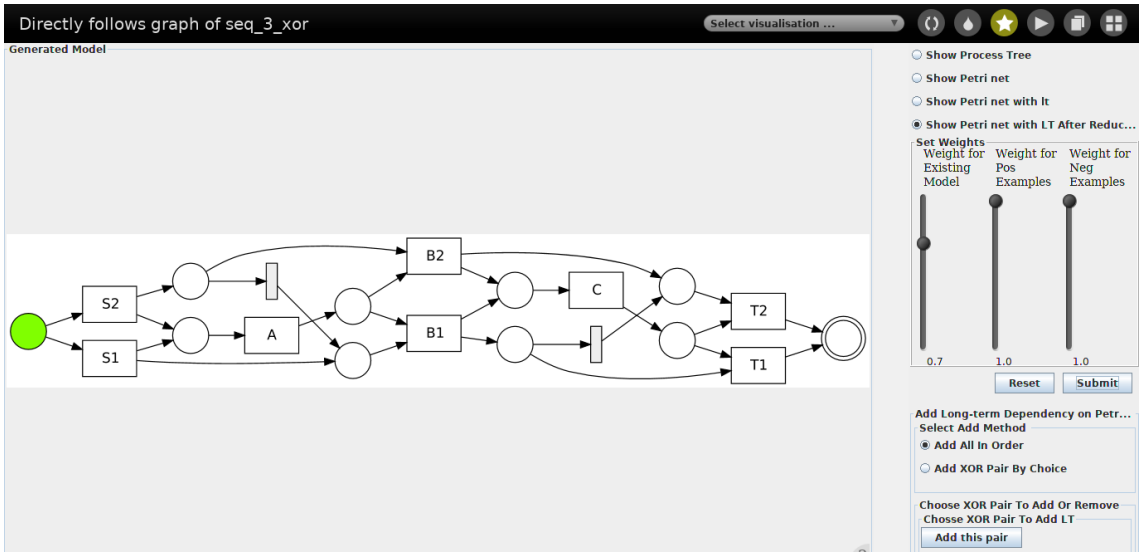


Figure 2.5: Petri net after reducing the silent transitions

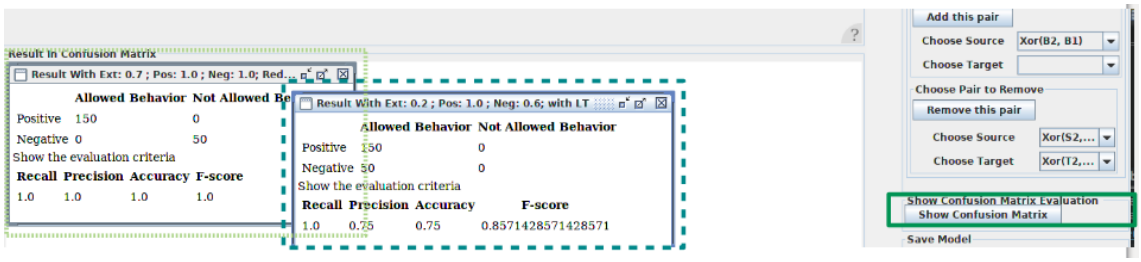


Figure 2.6: Generated Process Tree Model

2.4 Implementation of Showing Evaluation Result

Another feature in this plugin is to show the evaluation result based on confusion matrix. With the brief evaluation result, it helps set the parameter and select the final process model.

It works in this way. After creating the current model in the left view, the evaluation program in background uses the event log and the current Petri net in the view as inputs. It applies a naive fitness checking and generates a confusion matrix with relative measurements like recall, precision. This evaluation result is then shown in the bottom of the left view in Figure 2.6. If the button of green rectangle in the right view *Show Confusion Matrix* is pressed again, the program is triggered again and generates a new confusion matrix result in dark green dashed rectangle which will be listed above the previous result in light green dashes area.

Chapter 3

Conclusion

Bibliography

[1] 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.

[2] 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.

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

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