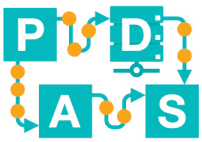


RWTH AACHEN UNIVERSITY

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

Big data projects have becomes a normal part of doing business, which raises the interest and application of process mining in organizations. Process mining combines data analysis with modeling, controlling and improving business processes, such that it bridges the gap of data mining on big data and business process management.

Process enhancement, as one of the main focuses in process mining, improves the existing processes according to actual execution event logs . It enables continuous improvement on business performance in organizations. However, most of the enhancement techniques only consider the positive instances which are execution sequences but lead to high business performance outcome. Therefore, the improved models tend to have a bias without the use of negative instances.

This thesis provides a novel strategy to incorporate negative information on process enhancement. Firstly, the directly-follows relations of business activities are extracted from the given existing reference process model, positive and negative instances of actual event log. Next, those relations are balanced and transformed into process model of Petri net by Inductive Miner. At end, long-term dependency on Petri net is further analyzed and added to block negative instances on the execution, in order to provide a preciser model.

Experiments for our implementation are conducted into scientific platform of KNIME. The results show the ability of our methods to provide better model with comparison to selected process enhancement techniques.

Chapter 1

Algorithm

This chapter describes the repair algorithm to incorporate the negative instances on process enhancement. At the beginning, the main architecture is listed to provide an overview of our strategy. Main modules of the algorithm are described in the next sections. 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. Again, we review the negative instances and express its impact by adding the long-term dependency. To add long-term dependency, extra places and silent transitions are created on the model, aiming to enforce the positive instances and block negative instances. Furthermore, the model in Petri net with long-term dependency can be post-processed by reducing the silent transitions 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, the main steps are conducted.

- *Generate directly-follows graph* Three directly-follows graph are generated respectively for the existing model, positive instance and negative instances from 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 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 end.

More details can be provided in the following sections.

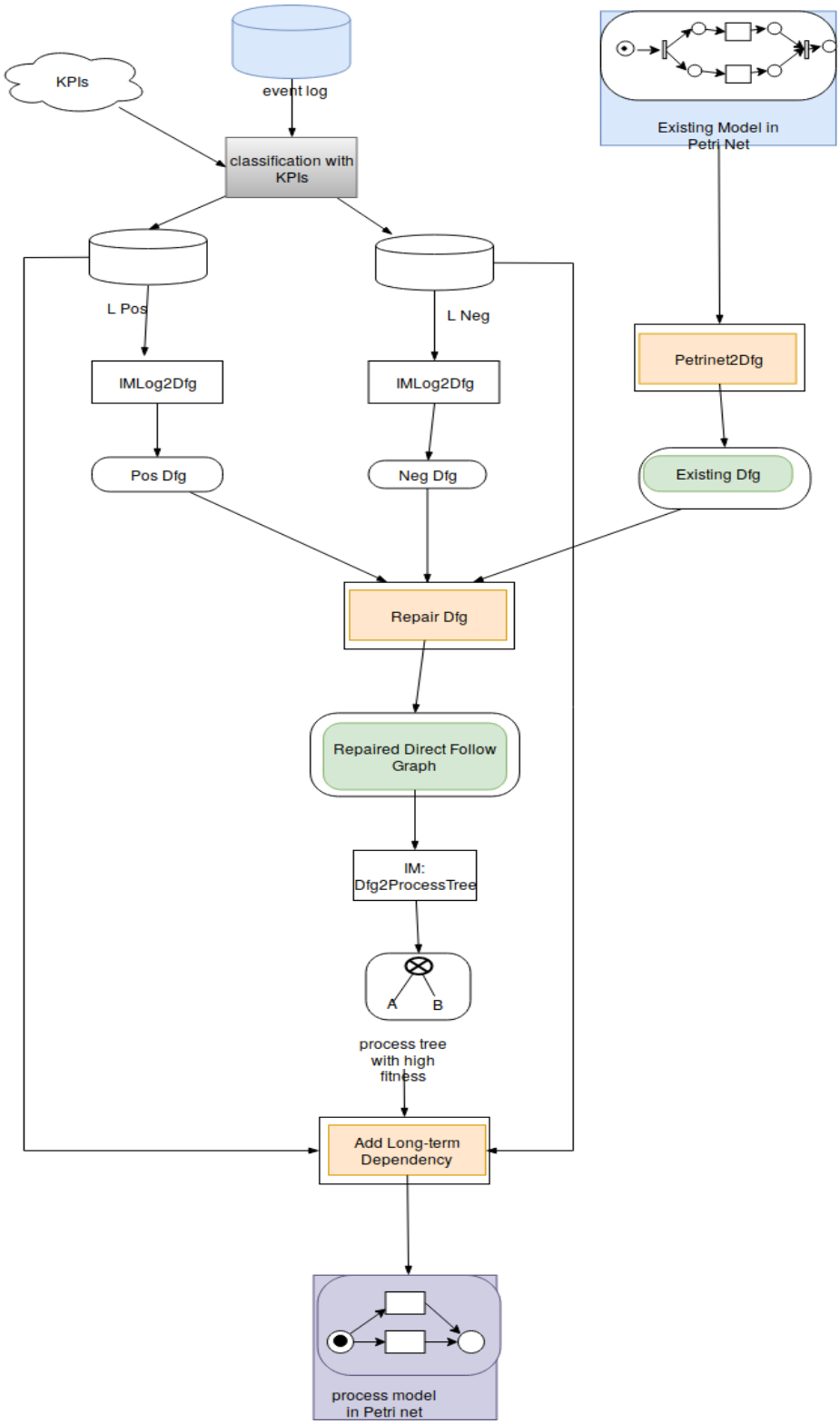


Figure 1.1: Model Repair Architecture – Rectangles represents processes and output data in eclipse shape, especially customized processes and data are in doubled lattice shape. Input event log and existing model are in blue, KPIs are in cloud. The output is a petri net in purple.

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 is labeled as positive, while L_{neg} contains the negative instances in the event log. Then, the two sublogs are passed to procedure *IMLog2Dfg* to generate directly-follows graphs, respectively $G(L_{pos})$ and $G(L_{neg})$. More details about the procedure is available in [1].

To generate a directly-follows relation from a Petri net, we gather the model behaviors by building a transitions system of its states. Then the directly-follows relations are extracted from state transitions. Based on those relations, we create a directly-follows graph for the existing model.

From the positive and negative event log, 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 edge.

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 into the same range [0-1]. Since $G(L_{ext})$ is derived differently, its unified cardinality is based only on the given directly-follows graph and defined in the following.

Definition 1.1 (Unified cardinality for the existing Model)**.** Given a directly-follows graphs $G(L_{ext})$ for a model, the unified cardinality of each directly-follows relation is defined as

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

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

for start activities A,

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

Similarly for end activities B,

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

$E(A,*)$ 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.

The unification of cardinality for positive and negative instances from an event log should consider the whole event log as its basic.

Definition 1.2 (Unified cardinality for $G(L_{pos})$, $G(L_{neg})$)**.** Given a directly-follows graphs $G(L_{pos})$ for a model, the unified cardinality of each directly-follows relation is defined as

$$U_{pos}(E(A,B)) = \frac{Cardinality_{pos}(E(A,B))}{Cardinality(E(A,*))}, with$$
$$Cardinality(E(A,*)) = \sum Cardinality_{pos}(E(A,X) + Cardinality_{neg}(E(A,Y)) where$$
$$E(A,X) \in G(L_{pos}) and E(A,Y) \in G(L_{neg}))$$

for start activities A,

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

for end activities B,

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

The unification for negative instances is defined in a similar way.

Considering all the unified cardinalities, we derive a concept called weight for directly-follows relation to combine the factors from the existing model, positive and negative instances. Later, a new directly-follows graph is built based on those weights.

Definition 1.3 (Weight of directly-follows relation G_{new})**.**

- For one directly-follows relation,

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

- For start acivities A, we have

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

- For end activities B, we have

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

In the real life, there exists various needs to address either on the existing model, the positive instances or the negative instances. To meet this requirement, three control parameters are assigned respectively to each unified cardinality from the existing model, and positive and negative instances. The weight for one directly-follow relation is modified in the way bellow.

Definition 1.4 (Weight of directly-follows relation G_{new})**.**

- For one directly-follows relation,

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

- For start acivities A, we have

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

- For end activities B, we have

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

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.

1.4 Mine models from directly-follows graph

The result from last step is a generated directly-follows graph with weighted unified cardinality. To apply the Inductive Miner on directly-follows graph, more procedures are in need. The directly-follows relations are firstly filtered when its weighted unified cardinality is less than 0. Secondly, its cardinality is transformed into the form which is acceptable by the Inductive Miner.

1.5 Add long-term dependency

Due to the intrinsic characters of Inductive Miner, the dependency from activities which are not directly-followed can not be discovered. To make the generated model more precise, we detect the long-term dependency and add it on the model in Petri net. Obviously, 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, the long-term dependency in exclusive choice is considered only in this thesis.

To analyze the exclusive choice of activities, we uses process tree as a intermediate process. We use process trees as one internal result in our approach in two factors: The reasons are: (1) easy to extract the exclusive choice structure from process tree. Process tree is block-structured and benefits the analysis of exclusive choices. (2) easy to get the model in process tree. Inductive Miner can generate process model in process tree. (3) easy to transform process tree to Petri net. The exclusive choice can be written as Xor in process tree. For sake of convenience, we define the concept called xor branch.

Definition 1.5. 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($XORB_{Q_i}$).

Two properties of xor block, purity and nestedness are demonstrated to express the different structures of xor block according to its branches.

Definition 1.6 (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.

For two arbitrary 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 1.7 (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 1.8 (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 1.9 (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.

We add long-term dependency on model by injecting silent transitions and extra places into Petri net. An example is given to describe this method.

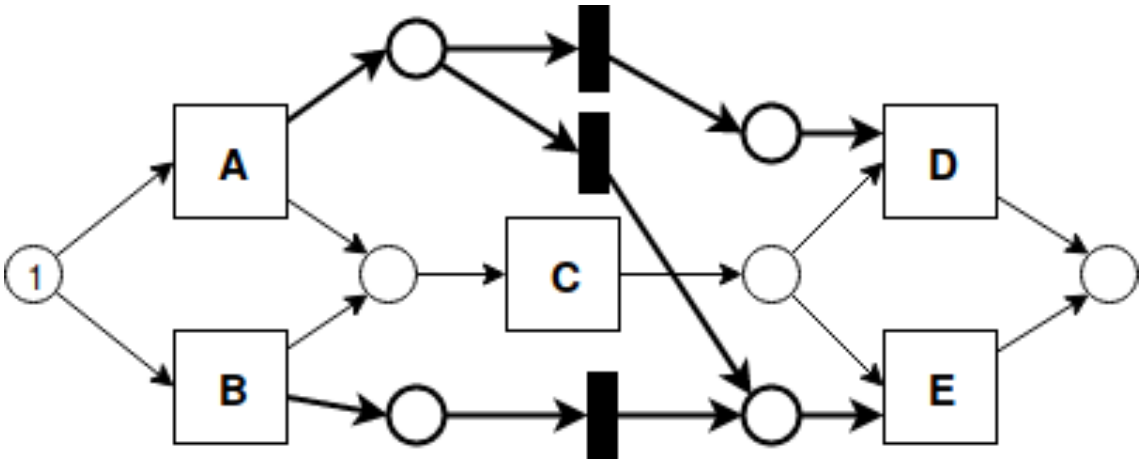
However, only with data from positive and negative information on the long-term dependency, it is possible to result in unsound model, because the existing model can keep some directly-follows relations about xor branches. When those relations do not show in the positive event log or shows only in negative event log, we get incomplete information. Consequently, when we detect this long-term dependency on those xor branches, there is no evidence of long-term dependency on those xor branches. It results in an unsound model, since those xor branches can't get fired to consume the tokens generated from the choices before.

There are 7 sorts of long-term dependency that is able to happen in this model as listed in the following. Before this, we need to define some concepts at the sake of convenience.

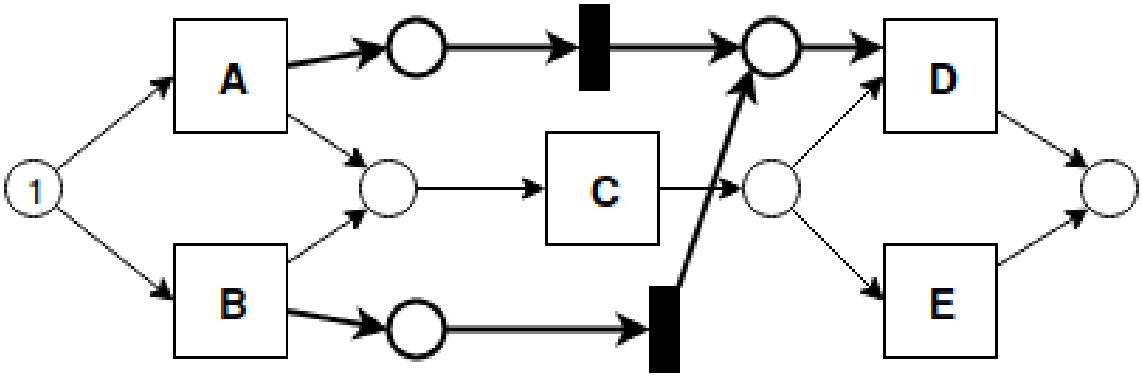
Definition 1.10 (Source and Target of Long-term Dependency)**.** We define the source set of long-term dependency is $LT_S := \{X | \exists X, X \rightsquigarrow Y \in LT\}$, and target set is $LT_T := \{Y | \exists Y, X \rightsquigarrow Y \in LT\}$.
For one xor branch $X \in XORB_S$, the target xor branch set relative to it with long-term dependency is defined as: $LT_T(X) = \{Y | \exists Y, X \rightsquigarrow Y \in LT\}$ Respectively, the source xor branch relative to one xor branch in target is $LT_S(Y) = \{X | \exists X, X \rightsquigarrow Y \in LT\}$

At the same time, we use $XORB_S$ and $XORB_T$ to represent the set of xor branches for source and target xor block.

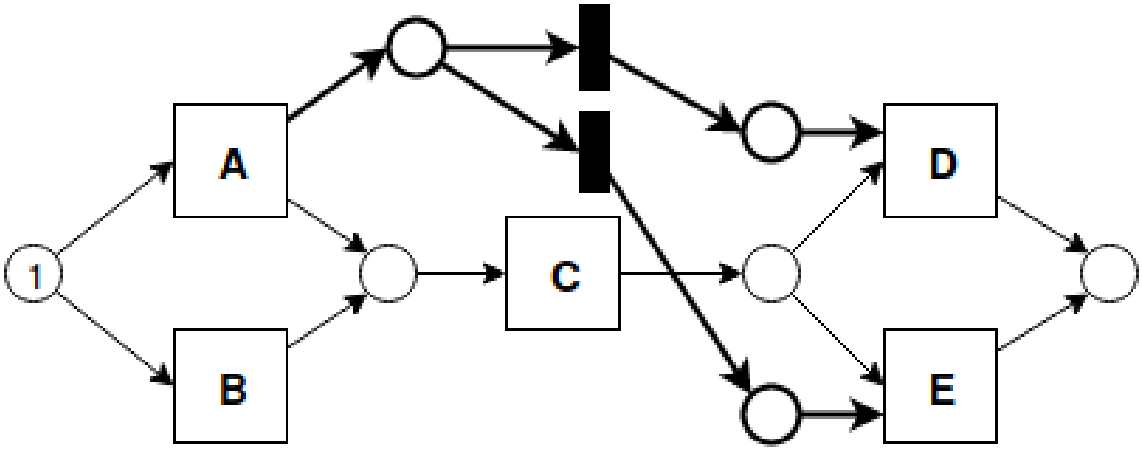
1. $LT = \{A \rightsquigarrow D, A \rightsquigarrow E, B \rightsquigarrow D, B \rightsquigarrow D\}$.
   $LT_S = \{A, B\}, LT_T = \{D, E\}, |LT| = |XORB_S| * |XORB_T|$, which means long-term dependency has all combinations of source and target xor branches.



(a) For situation 2



(b) For situation 4



(c) For situation 5

Figure 1.2: Silent Events for Long-term Dependency

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.

For situation 1, it's full connected and xor branches can be chosen freely. So there is no need to add explicit connection on model to represent long-term dependency, therefore the model keeps the same as original. For Situation 2 and 3, if $LT_S = XORB_S, LT_T = XORB_T$, then we can create an sound model by adding silent transitions. If not, then we need to use the duplicated transitions to create sound model. But before, we need to prove its soundness. For situation 4, 5 and 6, there is no way to prove the soundness even by adding duplicated events.

By adding constraints, we make sure only situations 1,2,3 happen when adding long-term dependency. It requires that after dfg method considers the negative, positive and existing model to delete the unused xor branches for long-term dependency. Before doing it, we recover some definitions for long-term dependency detection. The definition 1.9 is rephrased into weight.

Definition 1.11 (Rephrased Correlation of xor branch)**.** The correlation for two branches is expressed into

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

, where $Wlt_{ext}(XORB_X, XORB_Y) = \frac{1}{|XORB_{Y*}|}$, $|XORB_{Y*}|$ means the number of possible directly-follows xor branche set $XORB_{Y*} = \{XORB_{Y1}, XORB_{Y2}, ...XORB_{Yn}\}$ after $XORB_X$.

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

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 log.

1.6 Reduce Silent Transitions

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.