Chair of Process and Data Science

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 gains more interest and wider application in organizations. It bridges the gap between traditional business process management and advanced data analysis techniques like data mining. By providing comprehensive tools, process mining brings visual insights on business and supports bussiness improvement.

Process enhancement, as one of the main focuses in process mining, improves the existing processes according to actual execution event logs. The records in event log can be classified as positive and negative according to predefined Key Performance Indicators. Most of the enhancement techniques only consider positive instances to improve the model, while the value hidden in negative instances is simply neglected.

This thesis provides a novel strategy that incorporates negative information with positive instances and the existing model to enhance on business process. Those factors are balanced on directly-follows relations of activities and generate a process model. Subsequently, long-term dependency of activities is detected and added in the model. In this way, the repaired model blocks negative instances and obtains a higher precision.

Experiments for our implementation are conducted into scientific platform of KNIME. The results show how our method incorporates negative information and the ability to properly repair model with a higher precision compared to selected repair 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.

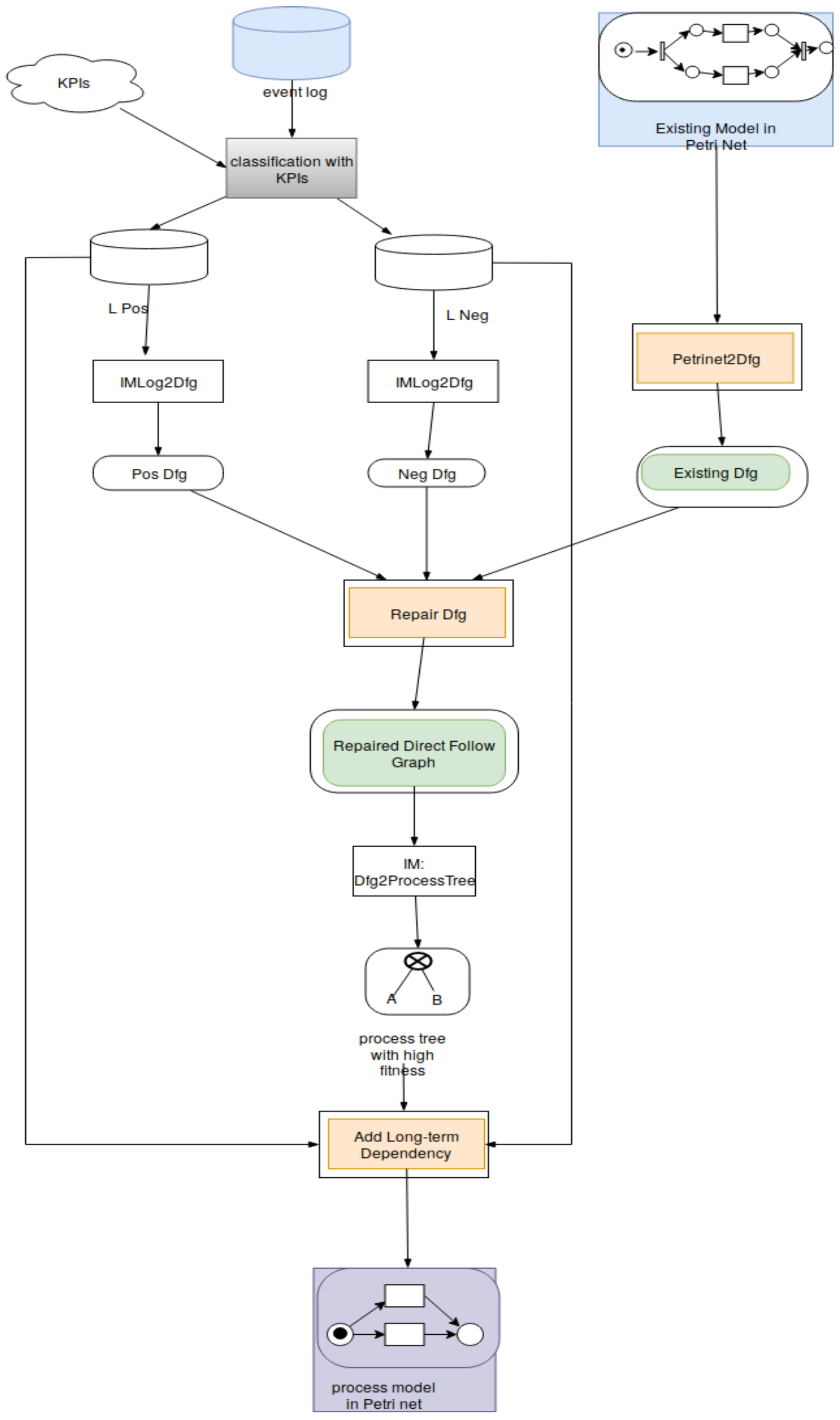


Figure 1.1: Model Repair Architecture – Rectangles represents processes and output data in eclipse shape, especially customized processes and data are in doubled lattice shape. Input event log and existing model are in blue, KPIs are in cloud. The output is a petri net in purple.

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 [6].

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

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

$$Cardinality(E(A,*)) = \sum Cardinality_{pos}(E(A,X) + Cardinality_{neg}(E(A,Y))where$$
$$E(A,X) \in G(L_{pos})andE(A,Y) \in G(L_{neg}))$$

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 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

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

.

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 to reflect 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.

The existing model can also affect the long-term dependency by supporting the existence of xor branches. In this assumption, the full long-tern dependency is approved. To incorporate the influence from the existing model, we rephrase the definition for xor branch correlation.

Definition 1.10 (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.

1.5.1 Cases Analysis

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 1.11 (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.

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

In dynamic aspect of the process model, long-term dependency can be seen as a way to block certain behaviors. By injecting silent transitions and extra places on Petri net, it limits the executions of certain behaviors. The steps to add silent transitions and places according to the long-term dependency are listed bellow.

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.
```

1.5.3 Soundness Analysis

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

Given arbitrary two xor block, $S = \{X_1, X2, ...X_m\}$ and $T = \{Y_1, Y_2, ...Yn\}$ with long-term dependency $LT = \{X_i \rightsquigarrow Y_j | 1 \le i \le m, 1 \le j \le n\}$, following the algorithm1, places after the source xor branches, $P_S = p_{X_i} | X_i \in LT_S$, and places before target xor branches,$P_T = p_{Y_j} | Y_i \in LT_T$ are firstly added.
For each long-term dependency $X_i \rightsquigarrow Y_j$ in LT, there is silent transition t with $p_{X_i} \to t \to p_{Y_j}$.

To prove the model soundness after adding long-term dependency, we need to prove

- the algorithm keep soundness between xor blocks.

Proof. For all xor branches in S, only one branch can be enabled. Without loss if generality, X_i is assumed to be enabled. After firing X_i, the marking distribution is those xor block are

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

The silent transition t with $p_{X_i} \rightarrow t \rightarrow p_{Y_j}$, can pass one token from p_{X_i} to p_{Y_j}. After firing t, the marking distribution changes to

$$M(p_{Y_j}) = 1, \forall p_{Y_{j\prime}} \in P_T, j\prime \neq j, M(p_{Y_i}) = 0$$

For each xor branch in T, it has two input places, one of them is the new added place $p_{Y_j}, Y_j \in T$. Now only Y_j is enabled. After executing the Y_j, it consumes the extra token at place p_{Y_j}. □

- In Situation 1,2 and 3, $LT_S = S, LT_T = T$, the original model is sound, now only consider the extra places and transitions, if they satisfy the conditions, too, then the whole model is sound.
  - safeness.
    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.
    After firing Y_j, all the extra places hold no token. So it does not violate the proper completion.
  - option to complete.
    There is always one Y_j enabled to continue the subsequent execution.
  - no dead part.
    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 $p_{X_i} \rightarrow t \rightarrow p_{Y_j}$ to produce a token in p_{Y_j}, which enables xor branch Y_j and leaves no dead part.
- In the rest situations, $LT_S \neq S = X_i, LT_T \neq \emptyset,$ 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.

In the target xor block T, only one xor block Y_j can be triggered.

- the algorithm does not violate the soundness outside of xor blocks.
  the added silent transitions and places do not damage the execution outside of the xor blocks. Because the tokens are only stays in those xor blocks, so it does not affect the execution of other parts. The tokens flow through the network like it does before.

An example is given in Figure **??**. Given the long-term dependency in situation 2 with $LT = \{A \rightsquigarrow D, A \rightsquigarrow E, B \rightsquigarrow E\}$, firstly, 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 with xor branches, a silent transition is generated to connect the extra places after the source xor branch to the place before target place.

1.5.4 Feasible cases due to soundness

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

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 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.

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 1.12. 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

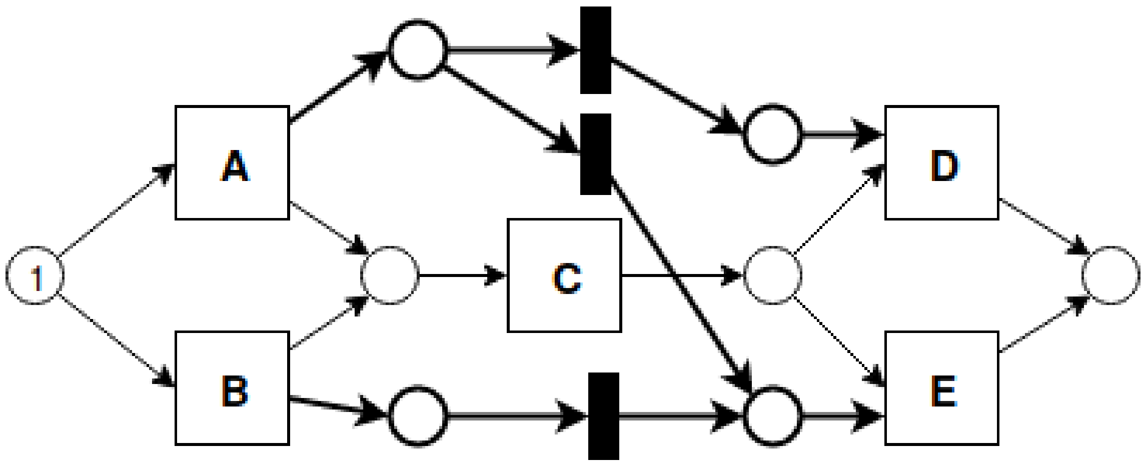
$$|Outedges(P_{in})| \leq 1$$

, or

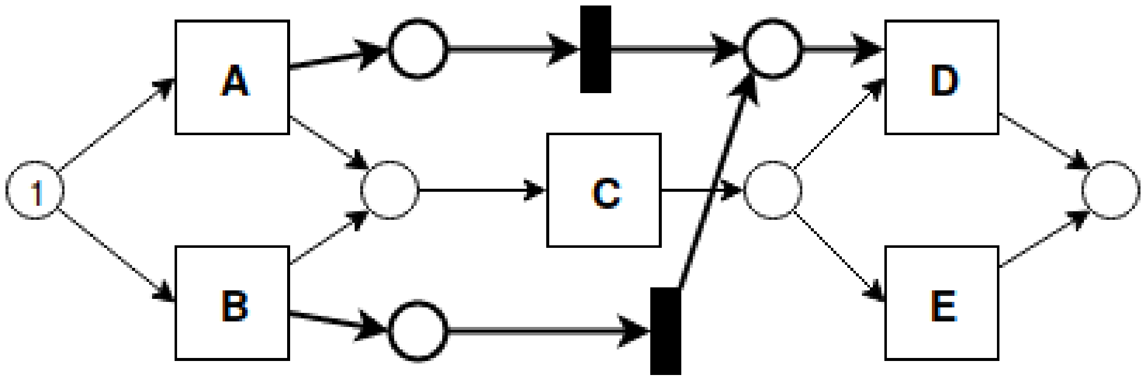
$$|Inedges(P_{out})| \leq 1$$

,

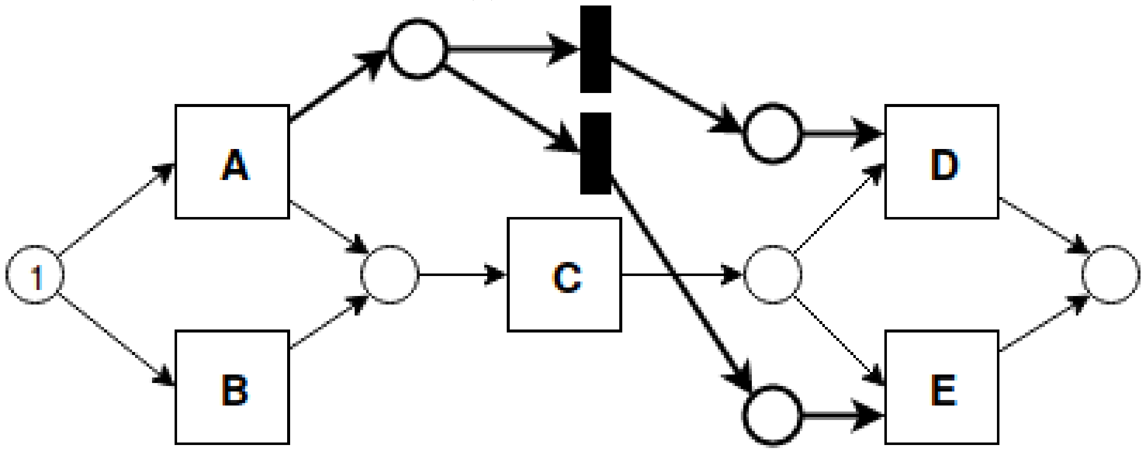
when in case (1), P_{in} contains a token that is always passed to P_{out} by silent transitions t. After deleting the silent transition, the token is generated directly on P_{out}. Since t is



(a) For situation 2



(b) For situation 4



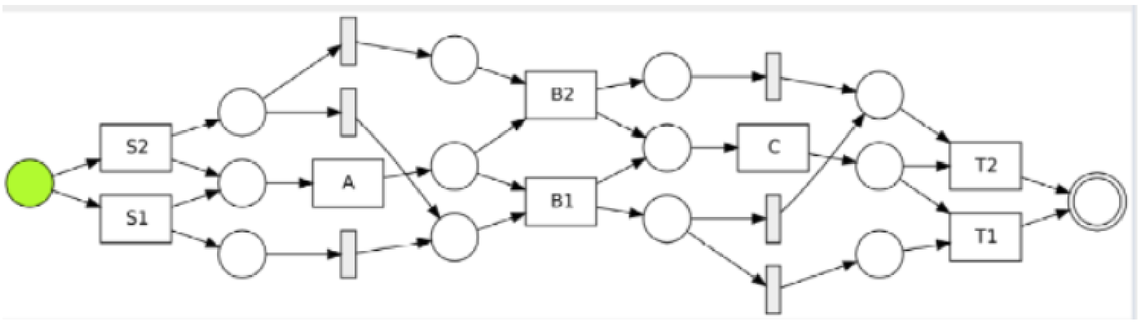
(c) For situation 5

Figure 1.2: Silent Events for Long-term Dependency

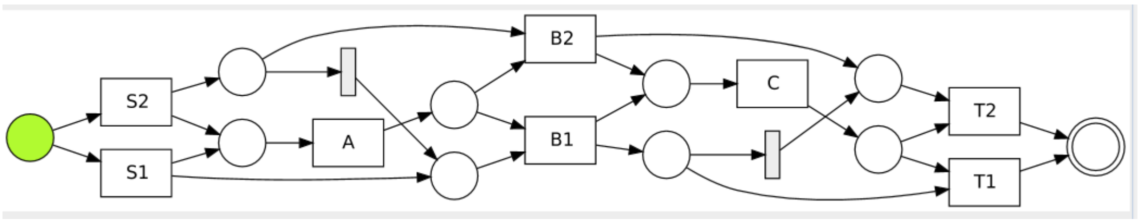
silent transition, it won't affect the model behavior.
when in case (2), P_{out} contains a token that is always passed from P_{in} by silent transitions

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. $\square$

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

Conclusion

Bibliography

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

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.