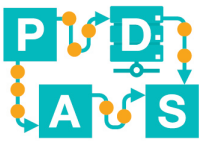


RWTH AACHEN UNIVERSITY

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

Introduction

Process mining is a relatively new discipline that has emerged from the need to bridge the gap of data mining and business process management. The objective of process mining is to support the analysis of business process, provide valuable insights on processes and further improve the business execution. According to [15], techniques of process mining are divided into three categories: process discovery, conformance checking and process enhancement. Process discovery techniques focus on deriving process models from event logs of the information system, allowing the vision into the real business process. Conformance checking analyzes the deviations between an referenced process model and observed behaviors driven from its execution. Enhancement adapts and improves existing process models by extending the model with additional data perspectives or repairing the existing model to accurately reflect observed behaviors.

Due to the increasing availability of detailed event logs of information systems, process mining techniques have recently enabled wider applications of process mining in organizations around the world[15]. After applying process discovery in organizations, a process model is fixed in information system to guide the execution of business. However,in real life, business processes often encounter exceptional situations where it is necessary to execute process differing from the predefined model. To reflect the reality, the organizations need to adapt the existing process model. Basically, one can apply process discovery techniques again to obtain a new model from event log. However, due to the facts, (1) the cost of rediscovery, and (2) the discovered model tend to have less similarity with the original model[8]. As shown in [8], there is a need to change an existing model similar to the original model while replaying the current process execution. Here comes the model repair.

Model repair belongs to process enhancement and stands between process discovery and conformance checking. It analyzes the workflow deviations between event log and process model, and fix the deviations mainly by adding sub processes on the model. As known, business in organizations is goal-oriented and aims to have high performance according to a set of Key Performance Indicators(KPIs), for example, average conversion time for the sales, payment error rate for the finance. However, there are few researches on applying the process mining with consideration of performance[10]. [10] points out the rare contributions like [6] to combine performance into process mining. Deviations are firstly analyzed to determine if they have a positive impact on the process performance. Model repair techniques in [9] are applied into traces with positive deviations.

However, the current repair methods have some limits. Model repair fixes the model

by adding subprocesses, silent transitions or loops, it guarantees the model fitness but overgeneralizes the model, such that it allows more behaviors than expected. On the other hand, it increases the model complexity. Even the performance is considered in [6], but only deviations in positive is used to add subprocesses, the negative information is ignored, which disables the possibility to block negative behaviors from model. A motivation example is listed to describe those limits.

1.1 Motivation Example

The example is extracted from the registration procedure of thesis project one German university. The main activities are topic selection, make proposal, a meeting with supervisor to discuss the topic, and check course requirements. After finishing all of those activities, the formal registration is enabled.
Figure 1.1a shows the original process in Petri net. The activities are modeled by the corresponding *transitions* which is represented by a square. Transitions are connected through a circle called *place*. The network structure is static but governed by the firing rule.

As an example, there is a model presented in Figure 1.1a, where A is followed directly by B. During its execution in real life, an event log is generated:

$$< A, B >^{55}, < B, A >^{105}$$

When considering KPIs performance, the log is divided into a positive and a negative set:

$$Positive\ examples : < A, B >^{5}, < B, A >^{100}$$

$$Negative\ examples : < A, B >^{50}, < B, A >^{5}$$

After applying current model repair techniques in [9], the process model is repaired using all examples. A can be skipped and duplicated later in a self-loop. In [6] methods, only the positive examples are taken for the model repair. Yet, since both $< A, B > and < B, A >$ contributes to good performance, the repaired model keeps the same like in Figure 1.1(b).

However, it's obvious that $< A, B >$ often leads to bad performance and therefore should be excluded. The Figure 1.1(c) shows the expected model with incorporating the negative information. This model reinforces positive examples and avoids negative instances, which provides us a more accurate view of the business process.

Clearly, the use of negative information can bring significant benefits, e.g, enable a controlled generalization of a process model: the patterns to generalize should never include negative instances. The demand to improve current repair model techniques with incorporating negative instances appears. In the next section, the demand is analyzed and defined in a formal way.

1.2 Research Problem And Questions

We analyze the current model repair methods, and give the formal definitions.

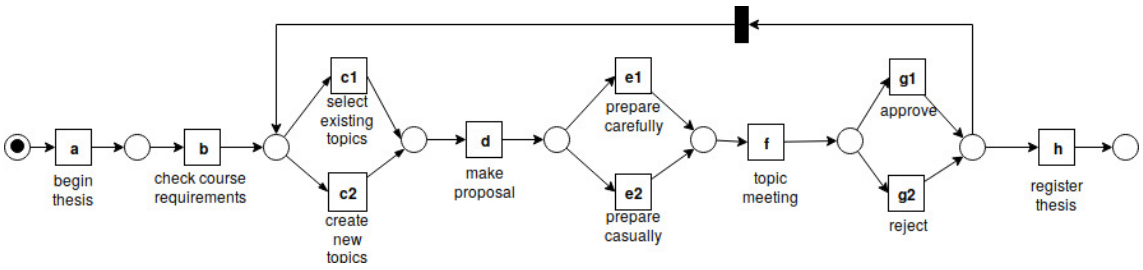
Definition 1.1. Given an input of one existing process model M, an event log L and a set of KPIs, how to improve current process enhancement techniques by incorporating

negative information, and generate a process model to enforce the positive instances while blocking the negative instance, with condition that the generated model should be as similar to the original model as possible Therefore, the repaired model provides a better way to understand and execute the real business process compared to the original model.

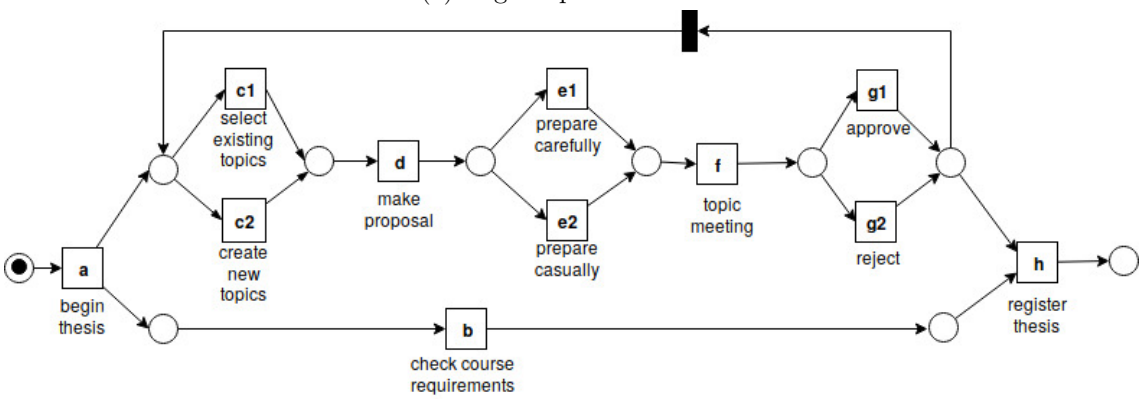
This paper tries to provide a solution for it. Our idea is to analyze the positive and negative impact on process performance of each trace. It balances the existing model, positive traces and negative traces on directly-follows relation, in order to incorporate all the factors on model generation. Later, the directly-follows relation is used to create process model by Inductive Miner. What's more, the impact of the existing model, positive and negative instances are parameterized by weights, to allow more flexibility of the generated model.

1.3 Outline

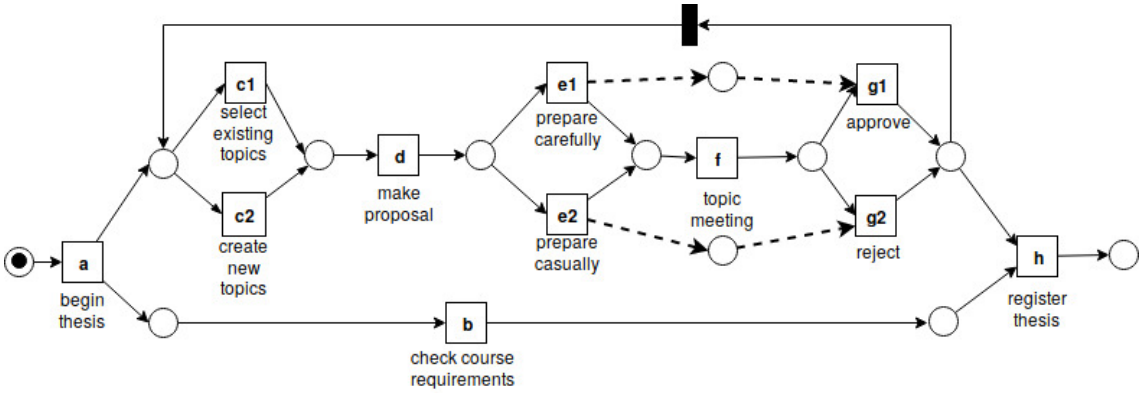
The reminder is organized in the following order. Section 2 recalls the basic notions on process mining and list the preliminary to solve the problem. The next section lists our methods are introduced and formal definitions are given. In Implementation Section, the details of algorithms are given. Later, we evaluate our methods with simulated data and real data respectively and list the results. Subsequently, the discussion on this paper is presented. At last section, a conclusion is drawn on the paper.



(a) original process model



(b) process model
with high fitness



(c) process model
with good KPI

Figure 1.1: example for model change under model repair

Chapter 2

Related Work

To update an existing process model in organizations, there are two strategies, rediscovery and process enhancement. Process rediscovery applies the discovery techniques on the actual event log to mine a new model. Process enhancement improves the model based on not only the actual event log but also the existing model.

Process discovery has been intensively researched in the past two decades and many algorithms have been proposed[16]. Directly-follows[18, 12] methods investigates the activities order in the traces and extracts higher relations which are used to build process models. State-based methods like [1, 4] builds a transition system to describe the event log, and then group the state regions into corresponding Petri net node. Language-based algorithms uses integer linear system to represent the place constraint where the token at one place can never go negative. By solving the system, a petri net is created. Its representative techniques are Integer Linear Programming(ILP) Miner[19]. Other methods due to [20] include search-based algorithm like Genetic Algorithm Miner[5], heuristic-based algorithm Heuristics Miner[22].

Among those discovery methods, Inductive Miner is widely applied[12]. It investigates the activity order in the traces and represents the order in a directly-follows graph. Based on the graph, it finds the most prominent split from the set of exclusive choice, sequence, parallelism and loop splits on the event log. Afterwards, the corresponding operator to the split is used to build a block-structured process model called process tree. Iteratively, the split sublogs are passed as inputs for the same procedure until single activity is reached and no split is available. A process tree is output as the mined process model.

When the actual event log differs a lot from the referred process model, it is suitable to use the rediscovery method to improve the business execution. However, in some cases, the process enhancement focuses to extend or improve an existing process model by using an actual event log[15]. Besides extending the model with more data perspectives, repair is another type of enhancement. It modifies the model to reflect observed behavior while keeping the model as similar as possible to the original model.

In [8], model repair is firstly introduced into process enhancement. By using conformance checking, the deviations of the event log and process model are detected. The consecutive deviations in log only are collected in the form of subtraces at specific location Q in the model. Later, the subtraces are grouped into sublog that share the same location Q for subprocess discovery. In the earlier version in [8], the sublogs are obtained in a greedy way, while in [9], sublogs are gathered by using ILP Miner to guarantee the fitness. Additional subprocesses and loops are introduced into the existing model to ensure the

fitness, which also brings variants of execution paths into the model.

Later, compared to [8, 9], where all deviations are incorporated in model repair, [6] considers the impact of negative information. In [6], the deviations of the model and event log are firstly analyzed, in order to find out which deviations enforces the positive performance. Given a trace and a selected KPI, an observation instance is built to correlate the number of each log move with KPI output. Based on the observation instance, a set of rules are derived in the form of a decision tree. According to the rules, the original event log is divided into sublogs with traces matching the rules. The sublogs are then repaired to contain only trace deviations which have a positive KPI output. Following repair, the sublogs are merged as the input for model repair in [9]. According to the study case in [6], it provides better result than [9] on the aspect of performance.

As described above, the state-of-the-art repair techniques are based on positive instances, meanwhile the negative information are neglected. Without negative information, it is difficult to balance the fitness and precision of those model. Likewise, few researches give a try to incorporate negative information in multiple forms on process discovery.

In [11], the negative information is artificially generated by analyzing the available events set before and after one position and represented in the form of the complement of positive event sets. Based on the positive and negative event sets, Inductive Logic Programming is applied to detect the preconditions for each activity. Those preconditions are then converted to Petri net after applying a pruning and post-process step. Similar work on model discovery based on artificial negative events are published later. In [21], the author improves the method in [11] by assigning weights on artificial events with respect to unmatching window, in order to offer generalization on model.

The work in [13] uses traces in the event log with negative outcomes as negative information. It extends the techniques of numerical abstract domains and Satisfiability Modulo Theories(SMT) proposed in [3] to incorporate negative information for model discovery. Each trace as positive or negative is transformed as one point in n-dimensional space, n is the number of distinct activities. The execution of a trace reflects the token transmission and marking limits on places in the model. Those limits are represented into the a set of marking inequalities and in a form of convex polyhedron in n-dimensional space. Given half-space hypotheses, SMT solves the inequalities and gives the limits on the process model. Before SMT, negative information is incorporated to shift and rotate the polyhedron, which limits the generalization of the solution space. Because half-space is used, this method can not deal with negative instances overlapped into positive instances.

However, the field of model repair which considers the negative information is new. Furthermore, the idea to incorporate negative instances on trace level into model repair is innovative.

Chapter 3

Preliminary

This chapter introduces the most important ground concepts and notations that are used in the remainder of this article. Firstly, the data and process models in process mining are described; Later, one related discovery technique is introduced.

3.1 Event Log

Business process in organizations is reflected by its execution of a set of activities. The historical execution data is usually in a form of event logs in information systems and can be used by Process Mining to analyze the business execution. To specify the event log, we begin with formalizing the various notations[16] .

Definition 3.1 (Event)**.** An event corresponds to an activity in business execution and can be marked by e. An event is characterized by attributes, like a timestamp, name, and associated costs, etc. The finite set of events in the process is written as $\mathcal{E}$.

Definition 3.2 (Trace)**.** A trace is a finite sequence of events $\sigma \in \mathcal{E}^*$ with conditions that *(i)* each event can not appear twice in a trace, $\forall i, j, 1 \leq i, j \leq |\sigma|, if\ i \neq j, then\ \sigma(i) \neq \sigma(j)$. *(ii)* one event can only appear in one trace, $\forall e \in \sigma, if\ e \in \sigma\prime, then\ \sigma = \sigma\prime$. A trace also has a set of attributes, which describes the trace characters, like the identifier, the trace cost.

Definition 3.3 (Event Log)**.** An event log L is a set of traces. If an event log contains timestamps, then the ordering in a trace should respect these timestamps.

3.2 Process Models

After gathering the event log from information systems, process mining can discover a process model based on the event log, aims to improve understanding and insights on the business process. Multiple process modeling languages are applied in process mining in the last years, such as Petri net, BPMN models, etc. Petri nets are bipartite graph to describe concurrent systems. BPMN models, fully named Business Process Model and Notation models, include many different elements and are flexible but complex tool to visualize business process. Process tree is based on a tree structure to organize the event relation. In this article, due to simplicity, we focus on Petri net and process tree.

3.2.1 Petri Net

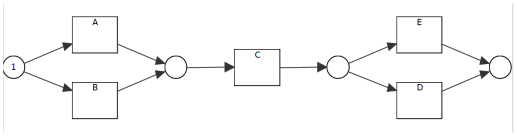
Among multiple models which are proposed to describe business process, Petri net has been best studied thoroughly to allow for the modeling of concurrency of activities and is one of the main process model in process mining. Activities from event log correspond to observable transitions in Petri net. Places in Petri net connect the transitions to express execution rules of activities. Except the observable transitions, there is silent transitions which is not visible.

Definition 3.4 (Silent transition)**.** A silent transition is an internal transition that is non-observable from the outside, but changes the state of the Petri net. It is usually denoted by ϵ.

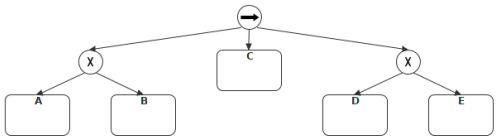
The observable transitions with silent transition ground a basic element set for the static structure of Petri net. To describe the dynamic behaviors of Petri net, the concept called *token* is introduced. It is a mark in the place to enable execution of the following transition. If all the input places for the transition hold a token, the transition is enable, namely the corresponding activity can be triggered. After this execution, the token in the input places are consumed and new tokens are generated in the output places. Initially, only the start place contains a token.

Definition 3.5 (Petri net)**.** A Petri net N is composed of a finite set of places P, transitions T, and a set of directed arcs $F \subseteq (P \times T) \cup (T \times P)$, which can be written as $N = (P, T, F)$. A marked Petri net is (P,T,F,M) where M the marking of the net. A marking of a net N is a multi-set over P, $M \in \mathbb{P}$ and used to express the dynamic state of the Petri net.

An example is shown in Figure 3.1a. It has transitions $T = \{A, B, C, D, E\}$ and four places with the initial marking in the place before $T = \{A, B\}$.



(a) A Simple Petri net



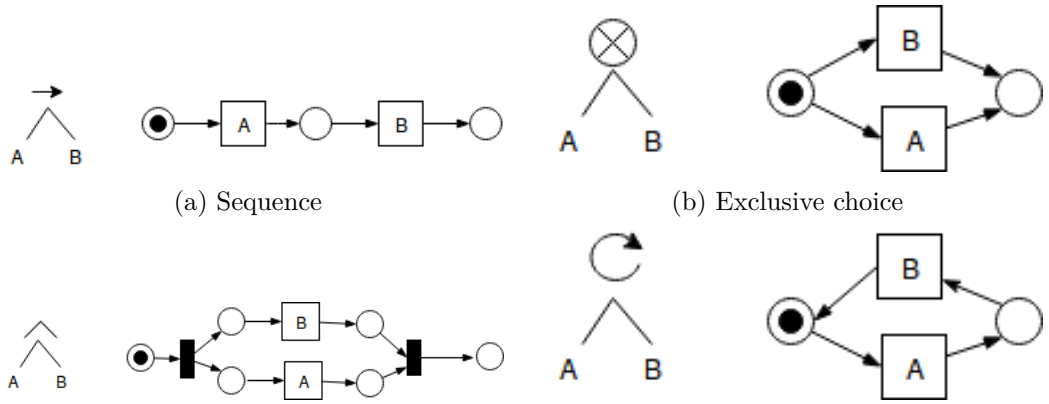
(b) A process tree corresponding to Fig 3.1a

3.2.2 Process Tree

Process tree is block-structured and sound by construction, while Petri nets, BPMN models possibly suffer from deadlocks, other anomalies[16]. Here we give the definition of process tree.

Definition 3.6 (Process Tree)**.** Let $A \subseteq \mathbb{A}$ be a finite set of activities with silent transition $\tau \in \mathbb{A}$, $\bigoplus \subseteq \{\rightarrow, \times, \wedge, \circlearrowleft\}$ be the set of process tree operators.

- $Q = a$ is a process tree with $a \in A$, and
- $Q = \oplus(Q_1, Q_2, ..Q_n)$ is a process tree with $\oplus \in \bigoplus$, and Q_i is a process tree, $i \in 1, 2, .., n, n \in \mathbb{N}$.



(a) Sequence (b) Exclusive choice

(c) Parallelism (d) Loop

Figure 3.2: Semantics of process tree operators w.r.t. Petri net

Process tree operators represents different block relation of each subtree. Their semantics are standardized from [17, 2] and explained with use of Petri net in Figure 3.2[2].

Definition 3.7 (Operator Semantics)**.** The semantics of operators $\bigoplus \subseteq \{\rightarrow, \times, \wedge, \circlearrowleft\}$ are,

- if $Q = \rightarrow (Q_1, Q_2, ..Q_n)$, the subtrees have sequential relation and are executed in order of $Q_1, Q_2, ..Q_n$
- if $Q = \times(Q_1, Q_2, ..Q_n)$, the subtrees have exclusive choice relation and only one subtree of $Q_1, Q_2, ..Q_n$ can be executed.
- if $Q = \wedge(Q_1, Q_2, ..Q_n)$, the subtrees have parallel relation and $Q_1, Q_2, ..Q_n$ they can be executed in parallel.
- if $Q = \circlearrowleft (Q_1, Q_2, ..Q_n)$, the subtrees have loop relation and $Q_1, Q_2, ..Q_n$ *with* $n \geq 2$, Q_1 is the do-part and is executed at least once, $Q_2, ..Q_n$ are redo part and have exclusive relation.

According to the corresponding semantic relations, a process tree can be easily transformed into Petri net. In Figure 3.1b, it is the process model in process tree which describes the same process as in Figure 3.1a.

3.3 Inductive Miner

To discover a process model from an event log, we choose one of the leading process discovery approaches – Inductive Miner, because it guarantees the construction of sound model, and is flexible and scalable to event log data. Its steps are listed bellow.

3.3.1 Construct a directly-follows graph

A the start, the event log L is scanned to extract the directly follows relation of events. The directly-follows relation is like the one in α-algorithm [18, 12], but the frequency information is stored for each relation. Later, those relations are combined together to build a directly-follows graph with frequency. According to [16, 12], a directly-follows graph is defined bellow.

Definition 3.8 (Directly-follows Graph)**.** The directly-follows relation $a > b$ is satisfied iff there is a trace σ *where*, $\sigma(i) = a$ *and* $\sigma(i+1) = b$. A directly-follows graph of an event log L is $G(L) = (A, F, A_{start}, A_{end})$ where A is the set of activities in L, $F = (a, b) \in A \times A | a >_L b$ is the directly-follows relation, A_{start}, A_{end} are the set of start and end activities respectively.

The frequency information of the directly-follows relation is called cardinality and defined below.

Definition 3.9 (Cardinality in directly-follows graph)**.** Given a directly-follows graph G(L) derived from an event log L, the cardinality of each directly-follows relation in G(L) is :

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

3.3.2 Split Log Into Sublogs

Based on the directly-follows graph, it finds the most prominent cut which is applied afterwards to split the event log into smaller sublogs. Cuts compose of *exclusive-choice cut, sequence cut, parallel cut and redo-loop cut* which correspond to the process tree operators $\{\rightarrow, \times, \wedge, \circlearrowleft\}$. They are selected in the following order. A maximal exclusive-choice cut is firstly tried to split the directly-follows graph; if it is not available, then a maximal sequence cut, a maximal parallel cut and a redo-loop cut are applied in sequence. Sublogs are created due to this available operator. Meanwhile, this operator is used to build the process tree.

The same procedure is applied again on the sublogs until single activities. What's more, this process tree can be converted into Petri net for further analysis.

Chapter 4

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.

4.1 Architecture

Figure 4.1 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.

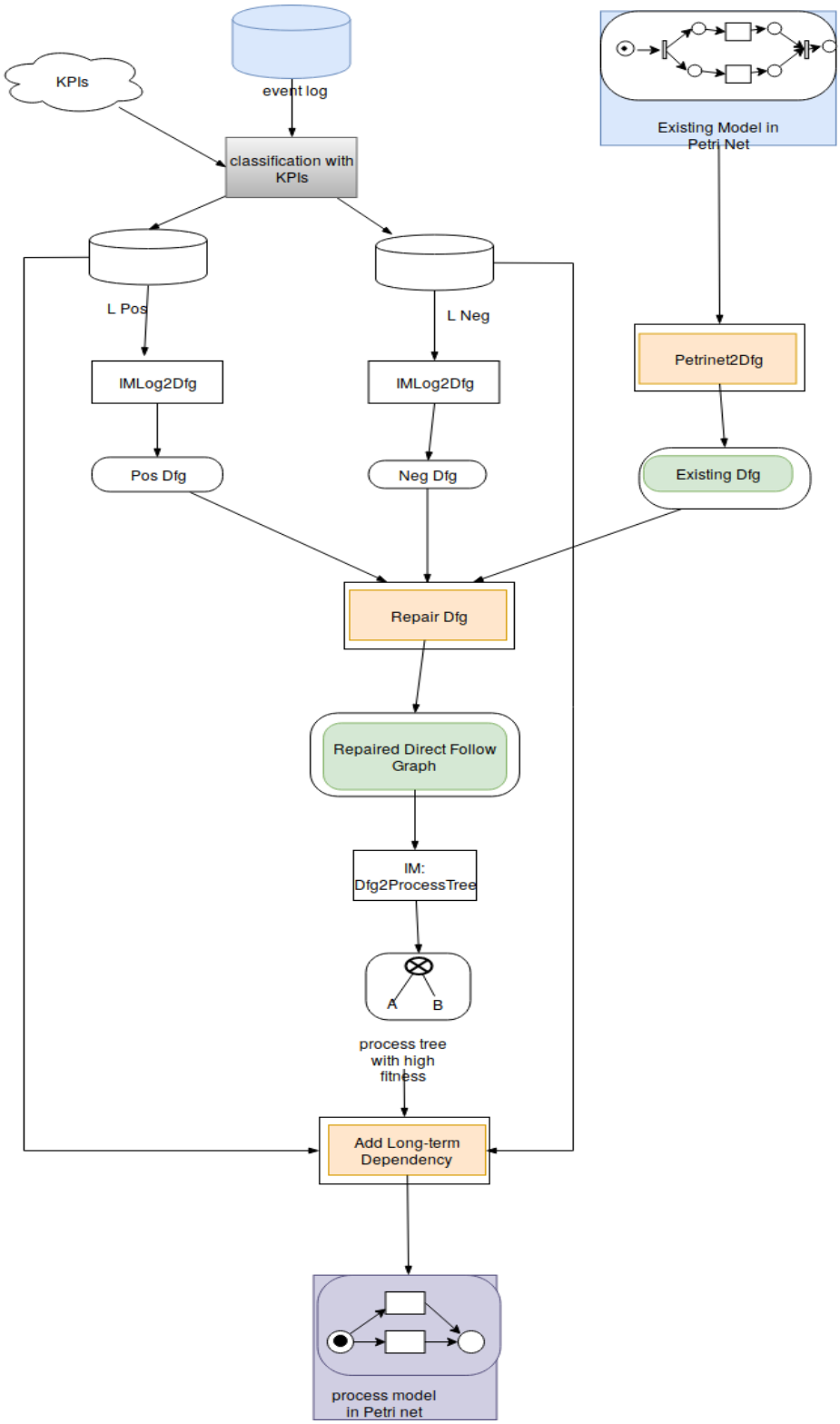


Figure 4.1: Model Repair Architecture – Rectangles represents processes and output data in eclipse shape, especially customized processes and data are in doubled lattice shape. Input event log and existing model are in blue, KPIs are in cloud. The output is a petri net in purple.

4.2 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 [12].

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.

4.3 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 4.1 (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 4.2 (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 4.3 (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 4.4 (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.

4.4 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.

4.5 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 4.5. 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 4.6 (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 4.7 (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 4.8 (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 4.9 (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 4.10 (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.

4.5.1 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 4.11 (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.

4.5.2 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.

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

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

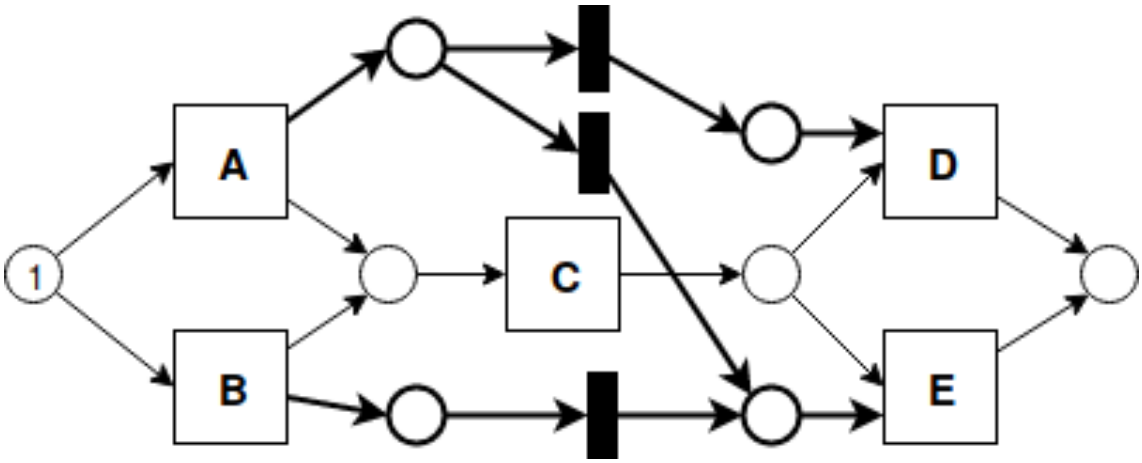
$XORB_Y$ is dependent on $XORB_X$;
if $XORB_X$ *is leaf node* **then**
    One place is added after this leaf node. ;
end
if $XORB_X$ *is Seq* **then**
    Add a place after the end node of this branch;;
    The node points to the new place;;
end
if $XORB_X$ *is And* **then**
    Create a place after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

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.

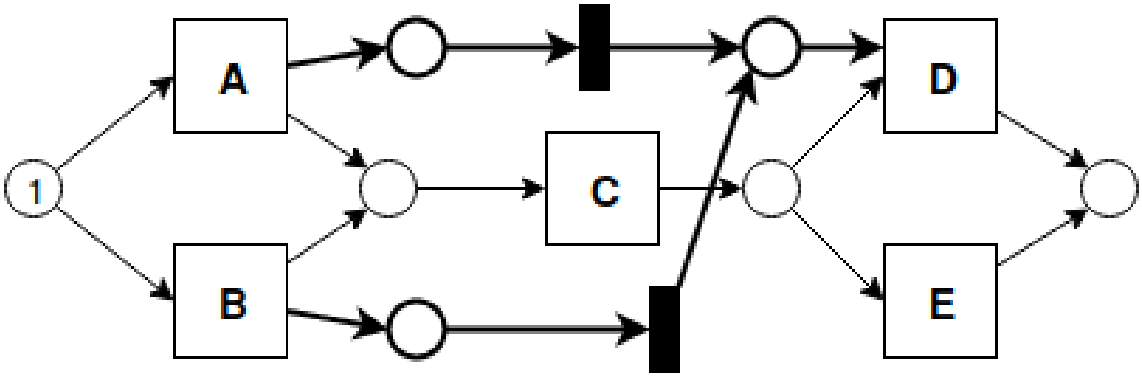
4.5.3 Feasible cases due to soundness

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.

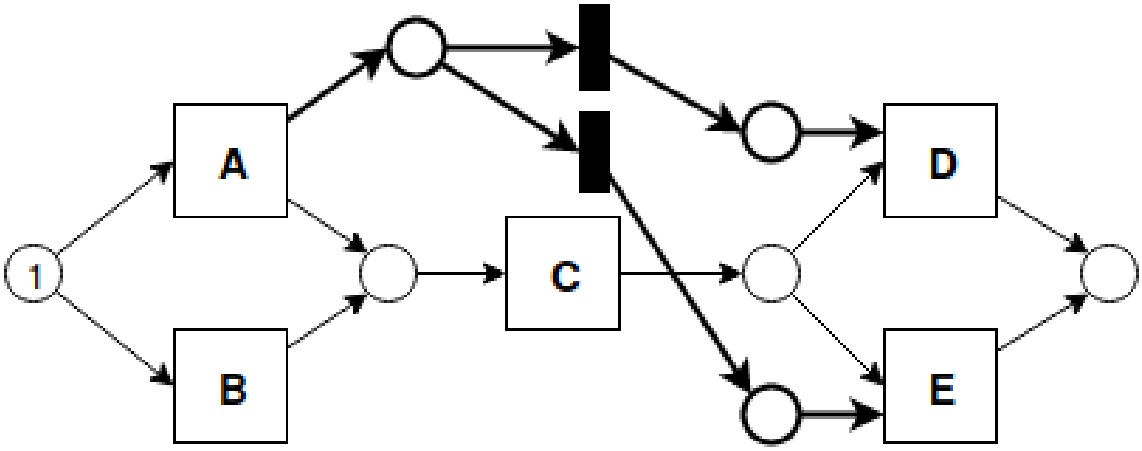
For situation 1, it's full connected and xor branches can be chosen freely. So there is no



(a) For situation 2



(b) For situation 4



(c) For situation 5

Figure 4.2: Silent Events for Long-term Dependency

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.

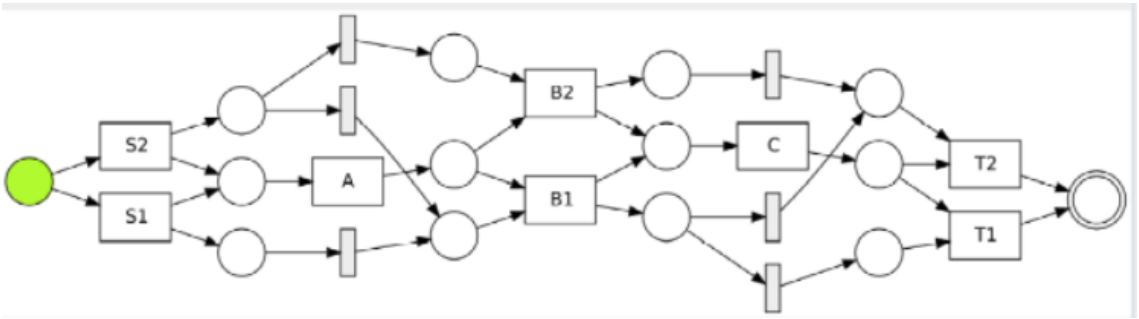
By adding constraints, we make sure only situations 1,2,3 happen when adding long-term dependency.

4.6 Reduce Silent Transitions

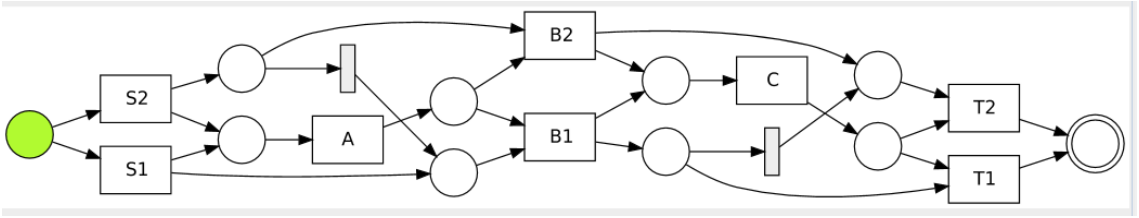
Our method to represent long-term dependency can introduce redundant silent transitions and places. On one hand, it complicates the model; on the other hand, it causes the model unsound where the extra silent transitions suspend the execution and therefore violates the soundness condition of proper completion.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

Proposition 4.12. Given a silent transition ϵ in Petri net with input place set P_{in} and output place set P_{out}, if $|P_{in}| \geq 2$ *and* $|P_{in}| \geq 2$, the silent transitions can not be deleted. Else, the deletion is ok, and it does not damage the process model.

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 5

Implementation

ProM is an open-source extensible framework. It supports wide process mining techniques in the form of plug-ins[14]. The algorithm to incorporate negative information is implemented as one plug-in called ***Repair Model By Kefang*** in ProM and released online[7]. 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.

5.1 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

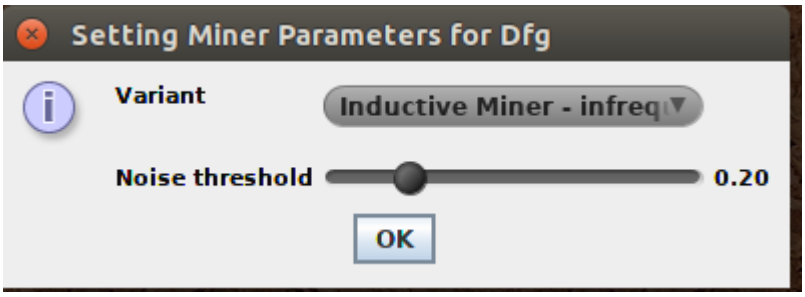


Figure 5.1: Inductive Miner Parameter Setting

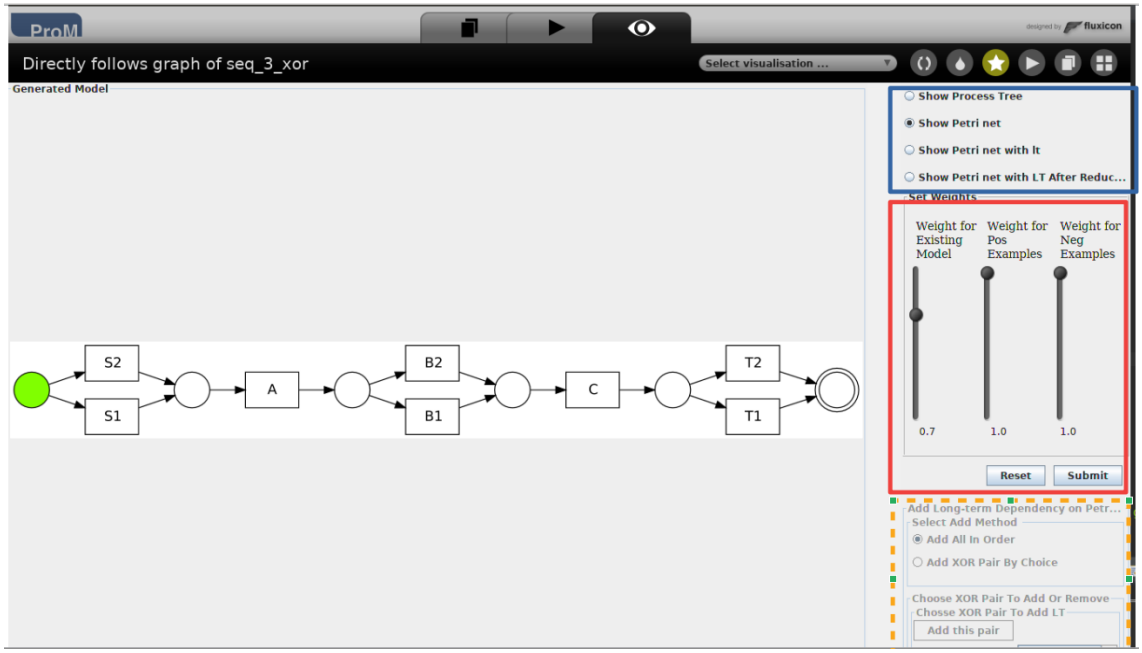


Figure 5.2: Generated Petri net without long-term dependency

5.2 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 5.1.
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 5.2. 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 5.2. 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

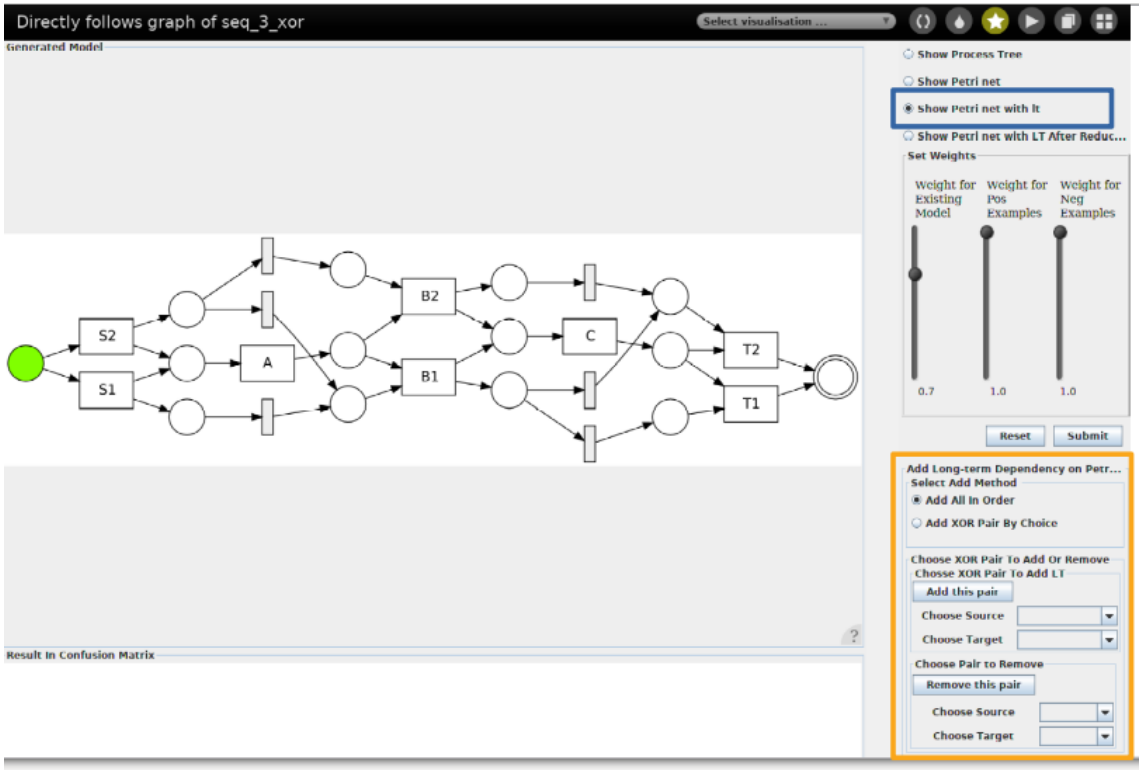


Figure 5.3: Petri Net with long-term dependency

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.

5.3 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 5.2, but the option to show a Petri net with long-term dependency is chosen. The resulted model is Figure 5.3.

Meanwhile, the control part of adding long-term dependency turns visible, which is in the orange rectangle in Figure 5.3. 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 5.4.

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 5.2, the simpler model in Figure 5.5 is constructed, after the post processing of reducing silent transitions.

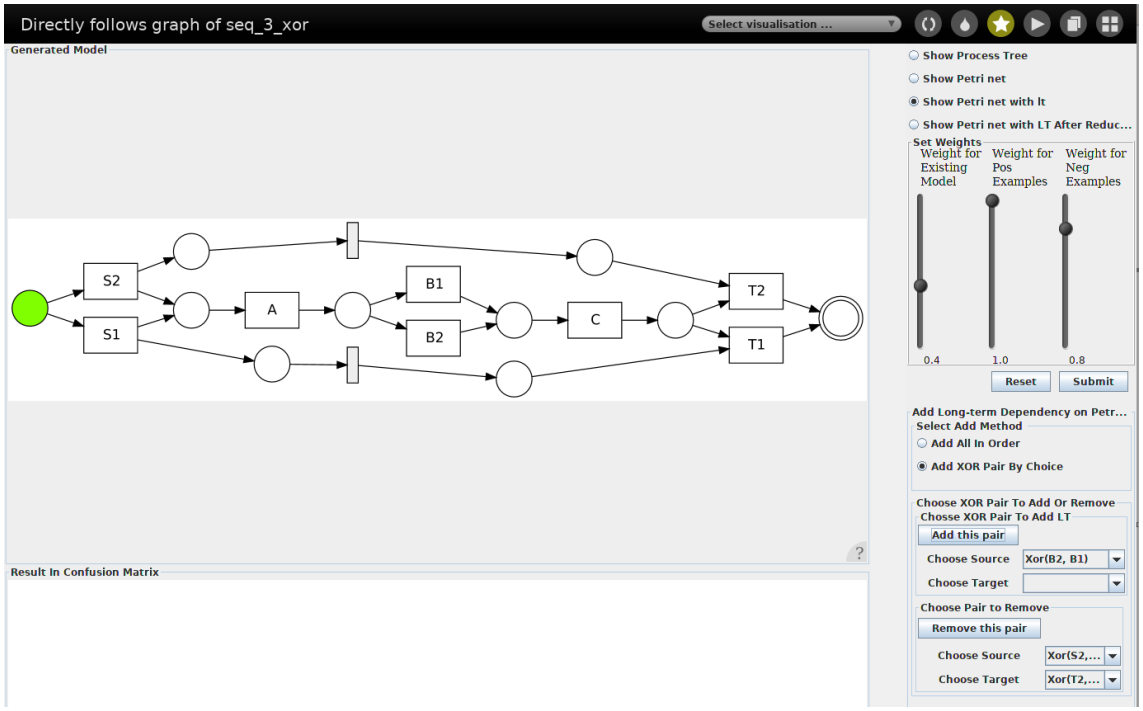


Figure 5.4: Petri net with selected long-term dependency

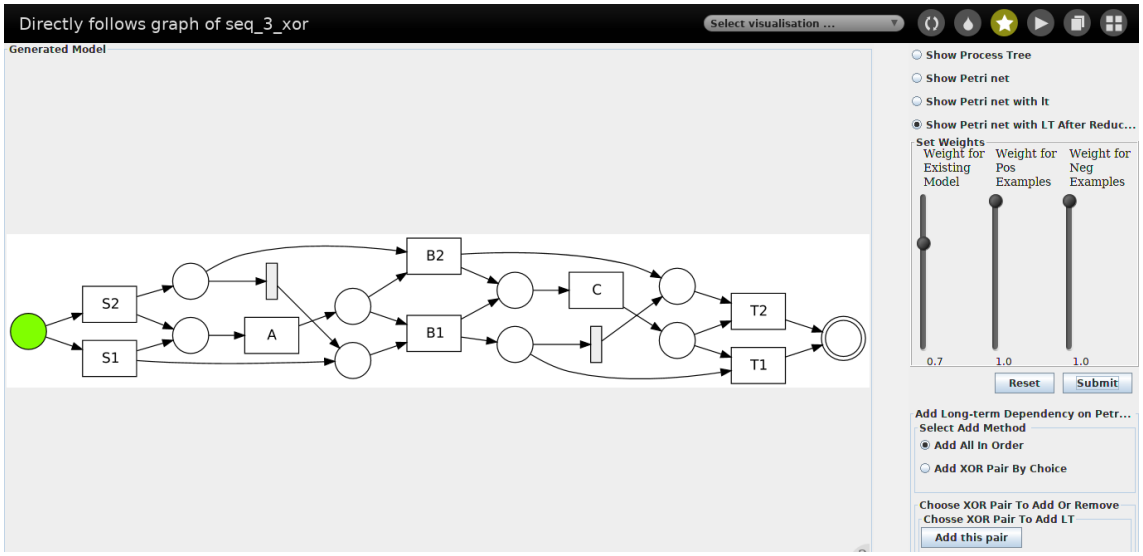


Figure 5.5: Petri net after reducing the silent transitions

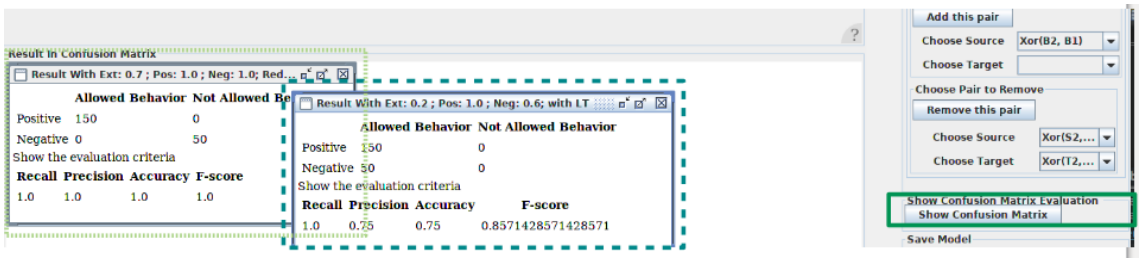


Figure 5.6: Generated Process Tree Model

5.4 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 5.6. 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 6

Evaluation

Chapter 7

Conclusion

Bibliography

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] Kefang Ding. Incorporatenegativeinformation.

[8] 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.

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

[10] Mahdi Ghasemi and Daniel Amyot. Process mining in healthcare: a systematised literature review. 2016.

[11] 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.

[12] 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.

[13] 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.

[14] Eindhoven Technical University. © 2010. Process Mining Group. Prom introduction.

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

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.