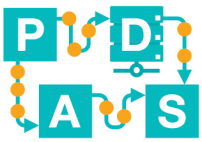


Chair of Process and Data Science

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

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

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

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

Abstract

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

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

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

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Process mining is a relatively new discipline that bridges the gap of data mining and business process management. The objective of process mining is to support the analysis of business processes, provide valuable insights in processes and further improve the business execution based on the business execution data which ~~is~~ recorded in event logs. According to [1], process mining techniques are divided into three categories: *process discovery*, *conformance checking*, and *process enhancement*. *Process discovery* techniques derive visual models from event logs of ~~the information system~~, aiming at a better understanding of real business processes. *Conformance checking* analyzes the deviations between a ~~referenced~~ process model and observed behaviors driven from its execution. *Process enhancement* adapts and improves existing process models by extending the models with additional data perspectives or repairing the reference models to accurately reflect observed behaviors.

Most of the organizations have predefined process execution rules of activities which are captured in a process model. One execution of related activities in this model is called a trace. A trace which has no deviation to the model is a fitting trace. However, in ~~real life~~, business processes often encounter exceptional situations where it is necessary to execute process differing from the reference model. With accumulation of deviations, the reference process model needs amending to reflect reality.

Basically, one can apply process discovery techniques again on the event log to obtain a new model only based. This method is referred as process rediscovery. However, there is a need that the improved model should be as similar as possible to the original model while replaying the current process execution [2]. In this situation, the rediscovery method tends to fail due to the ignorance of the impact from the existing model. To meet this need, **model repair** techniques are proposed in [2].

Model repair belongs to process enhancement [2]. It analyzes the workflow deviations between an event log and a process model, and fixes the deviations mainly by adding subprocesses on the model. As known, organizations are goal-oriented and aim to have high performance according to a set of Key Performance Indicator(KPI)s, ~~e.g.~~ the production time for a car industry, the logistic cost for importer companies. However, little research in process mining is conducted on the basis of business performance [3]. The authors of [3] point out several contributions e.g. [4] to consider business performance into process

mining. The work in [4] divides deviations of model and the event log into positive and negative according to certain KPIs. Then it applies repair techniques in [2] only with positive deviations which are deviations leading to positive KPI outcomes. This reason behind this approach is to avoid introducing negative instances into the repaired model.

However, the current repair methods have some limits. Model repair techniques fix the model by adding subprocesses. They guarantee that the repaired model replays the whole event log but overgeneralizes the model, such that more behaviors are allowed than expected. Furthermore, the model complexity ~~is increased by~~ the repair techniques in [2]. Even the performance is considered in [4], but only positive deviations are used to add subprocesses, the negative information is ignored, which disables the possibility to block negative behaviors from model.

In the following sections, motivating examples are given to describe those limits of the current repair techniques in several situations. Then we propose research questions to overcome those limits and define our research scope. At the end, we give the outline for the whole thesis.

1.1 Motivating Examples

For better understanding, examples are extracted from the common master thesis procedure to illustrate those situations. The process is described by the Petri net M_0 in Figure 1.1. Activities are denoted as rectangle and connected by circles called places. The directed arcs in the Petri net indicate the execution order of activities. And the black dots in places of the model represents current execution states. In addition, those dots can be substituted by other symbols due to visualization tools, like numbers in places.

According to the model, a student can **select existing topics** or **create new topics**. Those two activities are exclusive. The exclusive choice relation also exist in **learn existing tools** or **explore new tools**. Activities **development** and hold a **write thesis** are executed currently. Activity **presentation** is necessary to complete a master thesis project. For convenience, alphabets are used to represent the corresponding activities and annotated in the model later.

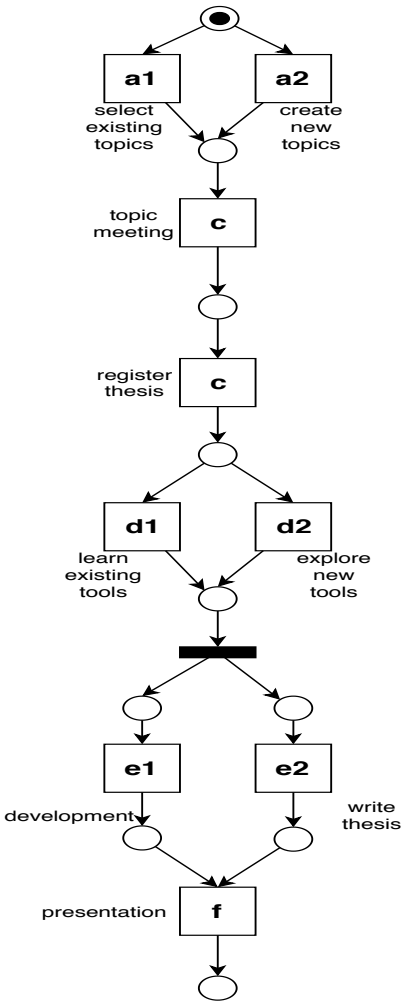


Figure 1.1: original master study process M_0

1.1.1 Situation 1: Add Subprocesses as Loops

One obvious shortcoming of repair techniques in [5] is that subprocesses for deviations are created and added as loops, when the deviations start and end at the same place. If there is only one such subprocess, it is fixed as a sequence in the model with post procedure. Yet, the algorithm does not discover orders between different subprocesses at overlapping locations. Therefore, the subprocesses are kept in a loop form, which causes a lower precision of the repaired model. An example is given with respect to the master study.

In some universities, before registering a master thesis, the activities **write proposal** and **check course requirement** might be necessary in the master study procedure. The real process are recorded in the event log L_1. Traces with either of those activities are considered as positive. **x1, x2** represent the activities **write proposal** and **check course requirement**.

$$\begin{aligned} L_1 := \{ &< a1, b, \mathbf{x1}, c, d1, e1, e2, f >^{50,pos}, \\ &< a1, b, \mathbf{x2}, c, d2, e1, e2, f >^{50,pos}, \} \end{aligned}$$

Because the existing repair techniques [5] don't consider the performance of traces in event log, all instances with negative labels are ignored to compute the deviations. L_1 has deviations of **x1, x2** at the same place without order. The corresponding subprocesses for them are added in the loop as shown in Figure **??**, which brings more behavior than expected into model.

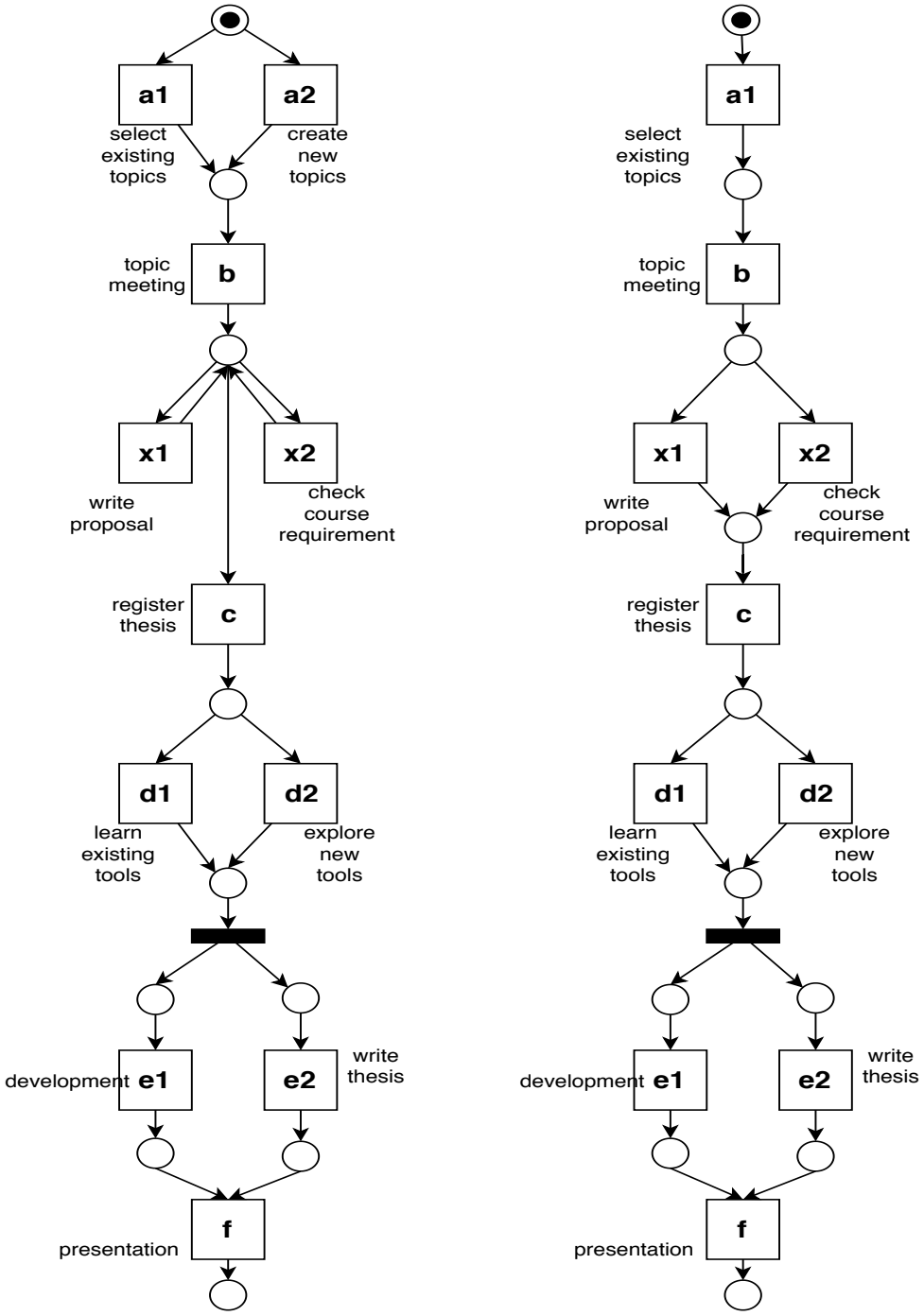
The repair algorithm in [4] builds upon [5] and considers the performance of the event log. However, the repaired model is the same as the one in Figure 1.2a. The reasons are: (1) there is no deviation from negative factors. (2) positive deviations are used to add subprocesses in the same way as [5].

When using rediscovery strategy with setting, the Inductive Miner for Infrequent and noise threshold with 20, a new model $M_{1.2}$ based on the positive instances of L_1 is generated in Figure 1.2b. **x1, x2** position with exclusive choices as implied by event log L_1. However, **a1** is kept in the new model and **a2** has been deleted, because no traces in L_1 includes **a2**.

Compared to the model $M_{1.1}$ and $M_{1.2}$ in Figure 1.2 where the two extra activities are shown in loop, or the main original structure has been changed, we expect the subprocesses with **x1, x2** can be added in sequence without affecting the main structure. In this way, the model has a higher precision of the repaired model, while keeping similar to original one.

1.1.2 Situation 2: Unable to Adapt Model with Fit Traces

This situation describes the existing problem in the current methods that fitting traces with negative performance outcomes cannot be used to repair a model. Given an actual event log L_2, The execution order of **development, write thesis** affects the performance outcomes. When **development** is executed before **write thesis**, more positive instances are given. In contrast, when **write thesis** is executed before **development**, more negative



(a) repaired model $M_{1.1}$ with techniques in [5] (b) rediscovered model $M_{1.2}$ by IM

Figure 1.2: example for situation 1 where $M_{1.1}$ is repaired by adding subprocess in the form of loops, which results in lower precision compared with the expected model $M_{1.2}$.

instances are caused.

$$\begin{aligned} L_2 := \{ &< a1, b, c, d2, \mathbf{e1}, \mathbf{e2}, f >^{30,pos}, \\ &< a2, b, c, d1, \mathbf{e1}, \mathbf{e2}, f >^{20,pos}; \\ &< a2, b, c, d2, \mathbf{e2}, \mathbf{e1}, f >^{10,pos}; \qquad < a1, b, c, d2, \mathbf{e2}, \mathbf{e1}, f >^{20,neg}, \\ &< a1, b, c, d1, \mathbf{e2}, \mathbf{e1}, f >^{20,pos}; \qquad < a2, b, c, d1, \mathbf{e1}, \mathbf{e2}, f >^{5,neg} \} \end{aligned}$$

Compared to M_0, the event log L_2 contains no deviation. When we apply the techniques in [5] and [4] to repair the model, the model remains unchanged. Also, IM mines the same model as the original one. Apparently, the fact that those two methods can't incorporate the negative information in fitting traces causes this shortcoming. A model as M_2 is expected because it enforces the positive instances and avoids the negative instance. Unfortunately, the current methods don't allow us to obtain such results.

1.1.3 Situation 3: Disable to Detect Long-term Dependency

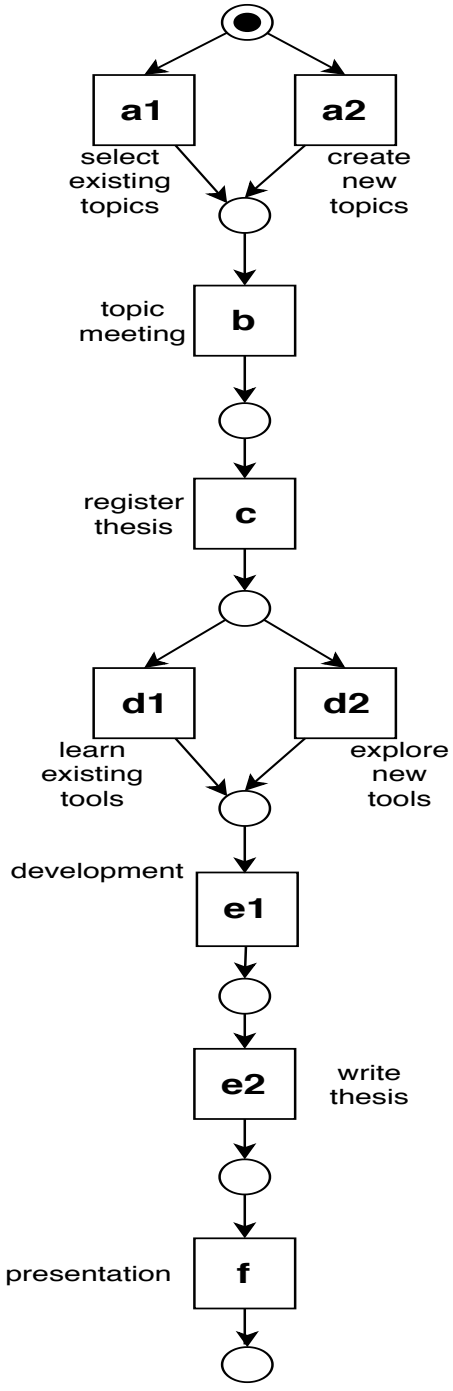
Another problem is that current methods are unable to detect the long-term dependency in the Petri net, which causes a lower precision. The long-term dependency describes the phenomenon that the execution of an activity decides the execution of activities that do not follow directly. Due to the long distance of this dependency, current methods cannot detect it and improve the precision by adding long-term dependency on the model. One example on M_0 is used to demonstrate this problem.

Given the time consumption on the whole study as the KPI, if the total sum goes over one threshold, the trace is negative, else as positive. Since the activity **create new topics** usually demands new knowledge rather than **checking the existing tools**. If students choose to learn existing tools, it's possibly not useful and time-wasting. In the other case, if we select existing topics with existing background, it saves time when we directly learn the existing tools. According to this performance standard, we classified those event traces into positive and negative shown above. An event log L_3 is given in the following.

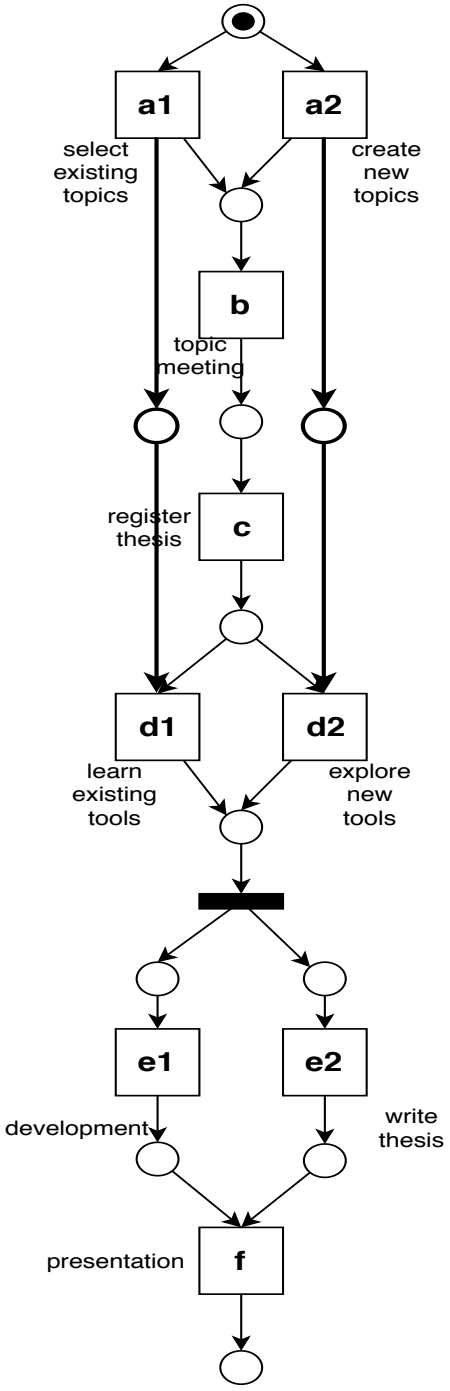
$$\begin{aligned} L_3 := \{ &< a, b, \mathbf{c1}, d, e, \mathbf{f1}, g1, g2, h, i >^{50,pos}, \\ &< a, b, \mathbf{c2}, d, e, \mathbf{f2}, g2, g1, h, i >^{50,pos}; \\ &< a, b, \mathbf{c1}, d, e, \mathbf{f2}, g2, g1, h, i >^{50,neg}, \\ &< a, b, \mathbf{c2}, d, e, \mathbf{f1}, g1, g2, h, i >^{50,neg} \} \end{aligned}$$

As observed, ***c1*** decides ***f1*** while ***c2*** decides ***f2***. They have long-term dependencies. However, there are no deviations of the model and event log L_3 according to the algorithms in [5] and [4]. Also, the Inductive Miner can't detect long-term dependency. Therefore, the original model stays the same and negative instances can't be blocked.

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. This leads to the demand of improving current repair model techniques with incorporating negative instances. In the next section, the demand is analyzed and defined in a formal way.



(a) model M_2 with order change



(b) model M_3 with long-term dependency

Figure 1.3: example for situation 2 and 3

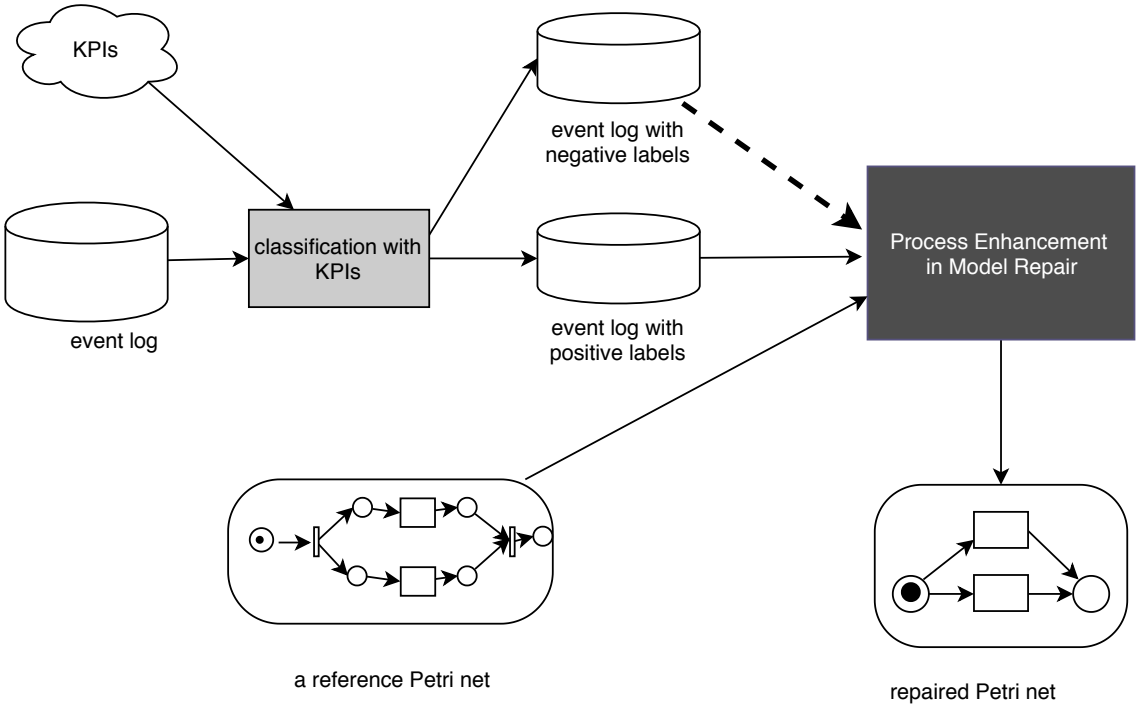


Figure 1.4: The resaerch problem scope

1.2 Research Scope And Questions

After analyzing the current model repair methods, we limit our research scope as shown in Figure 1.4. The inputs for our research are one existing process model M, an event log L . According to predefined KPIs, each trace in event log is classified into positive or negative. After applying repair techniques in the black box, the model should be improved to enforce the positive instances while disallowing negative instance, with condition that the generated model should be as similar to the original model as possible.

In this scope, we come up with several research questions listed in the following.

RQ1: How to overcome the shortcomings of current repair techniques in situations 1-3 above?

RQ2: How to balance the impact of the existing model, negative and positive instances together to repair model?

RQ3: How to block negative instances from the model while enforcing the positive ones?

In this thesis, we propose a solution for the black box. It analyzes process performance on trace level and balances the existing model, positive traces and negative traces on directly-follows relation, to incorporate all the factors on model generation.

1.3 Outline

This thesis aims to answer the questions presented in section 1.2 in the remaining chapters and provides a solution for the black box. Chapter 2 introduces the related works. Chapter 3 recalls the basic notions on process mining and list the preliminaries to solve the problem. Chapter 4 firstly presents an original framework to incorporate negative information into model repair. Based on this framework, a concrete algorithm are proposed. In Chapter 5, screenshots of the implementation tools are shown to demonstrate the usage. Chapter 6 answers the last question RQ3, by conducting a bundle of experiments. Later, results are analyzed and discussed. At last, we summarize our work in Chapter 7.

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 [6]. Directly-follows methods [7, 8] investigate the order of activities in the traces and extract higher relations which are used subsequently to build process models. State-based methods like [9, 10] build 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 [11]. Other methods due to [12] include search-based algorithms, like Genetic Algorithm Miner [13], and heuristic-based algorithms, like Heuristics Miner [14].

Among those discovery methods, Inductive Miner is widely applied [8]. 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. Afterward, the corresponding operator to the split is used to build a block-structured process model called a process tree. Iteratively, the split sublogs are passed as inputs for the same procedure until one single activity is reached and no split is available. The mined process model is a process tree that can be converted into the classical process model Petri net.

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 [1]. Besides extending the model with more data perspectives, model 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 one.

In [2], 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 are only collected in the form of subtraces at a 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 [2], the sublogs are obtained in a greedy way, while in [5], sublogs are gathered by using ILP Miner to guarantee the fitness. Additional subprocesses are introduced into the existing model to ensure that all traces fit the model, but it introduces more behavior into the model and lowers the precision.

Later, compared to [2, 5], where all deviations are incorporated in model repair, [4] considers the impact of negative information. In [4], 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 activities in event logs which are deviating from models. 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 with trace deviations which lead to positive KPI outputs. Following repair, the sublogs are merged as the input for model repair in [5]. According to the study case in [4], [4] provides a better result than [5] on the aspect of performance.

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

In [15], the negative information is artificially generated by analyzing the available event 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 [16], the author improves the method in [15] by assigning weights on artificial events with respect to an unmatching window, in order to offer generalization on the model.

The work in [17] 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 [18] 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. By adding places between transitions, it limits the transition execution by demanding tokens on those added places. This limit can be reflected by the sequence of traces. In linear system, those limits are represented into a set of 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 on 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

Preliminaries

This chapter introduces the most important concepts and notations that are used in this thesis. Firstly, the event data and process models typically used in process mining are described. Later, details of Inductive Miner techniques are listed.

3.1 Event Log

Business processes in organizations can be reflected by the execution of related activities. The historical execution data is usually stored as event logs in information systems and can be used by process mining techniques to analyze, understand, and improve the business execution. To specify the event log, we begin with formalizing various notations [6] .
Definition 3.1 (Event)**.** $\mathcal{A}$ is the universe of activities. An event e for an activity $a \in \mathcal{A}$ corresponds to one execution of this activity. The event set for a process is a finite set $\mathcal{E}$.

An event is characterized by attributes, like a timestamp, activity name, associated costs, etc. The executions of activities in a process are recorded as events. The set of events for a complete process case is a trace. Furthermore, the multiset of traces combines as an event log for the process.
Definition 3.2 (Trace And Event Log)**.** A trace σ is a finite sequence of event, $\sigma \in \mathcal{E}^*$. An event log L is a multiset of traces, $L \in \mathcal{B}(\mathcal{E}^*)$.

A trace also has a set of attributes, like its unique identifier, the cost. In this thesis, we extend this definition to handle traces with performance output according to certain KPIs. We call a trace labeled when this trace has a positive or negative label according to certain KPIs. Correspondingly, an event log is labeled if all of its traces have labels according to certain KPIs.

3.2 Process Models

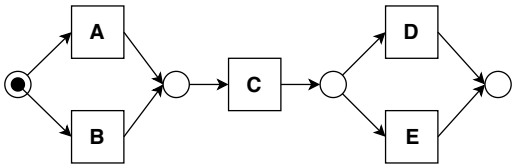
After gathering an event log from the information system, process mining can discover a process model based on the event log. With this process model, the understanding of

the business process can be improved. To describe the process, multiple process modeling languages are proposed in the past years, e.g, Petri net, BPMN models, etc.

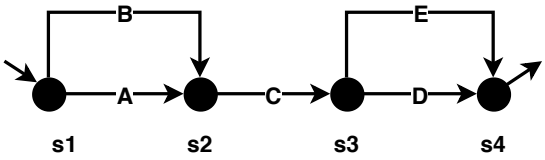
Among those model languages, Petri nets have been best studied thoroughly. Petri nets are capable to capture concurrent systems in a compact manner. Process trees are based on a tree structure to organize the event relation and simple to understand in comparison with other models, like BPMN models. In this thesis, Petri nets and process trees are used to represent our process.

3.2.1 Petri Nets

Petri nets are bipartite graphs which consists of ***transitions*** represented by a square and ***place*** by a circle. ***Tokens*** in black dots are put in places to express the dynamic states of a Petri net. An Petri net example is shown in Figure 3.1a. This Petri net is a workflow net with a token at the source place and satisfy the soundness conditions.



(a) A simple Petri net



(b) A transition system corresponding to Fig 3.1a

In the following, we define Petri nets in a formal way.
Definition 3.3 (Petri net)**.** A Petri net is a tuple $N = (P, T, F)$ where $P \cap T = \emptyset$. F is the set of arcs to connect places and transitions, $F \subseteq (P \times T) \cup (T \times P)$.

Further, we can assign activity labels to transitions in Petri nets to describe the corresponding activities in business process. Usually, transitions with activity labels are seen *observable actions*. To represent particular transitions that are not observable, we reserve label τ for them and name such transitions *silent or invisible* transitions. A Petri net with labels are called labeled Petri nets and can be formalized in Definition 3.4.
Definition 3.4 (Labeled Petri nets)**.** Labeled Petri nets are a tuple $N = (P, T, F, \Sigma, \lambda)$, where Σ is a set of activity labels, λ is a function defined as $\lambda : T \to \Sigma \cup \tau$. A transition t with $\lambda(t) = \tau$ is a silent or an invisible transition.

To express the dynamic states of Petri nets, we introduce a concept called ***marking*** ~~and~~ extend the form $N = (P, T, F)$ and define a marked Petri net.
Definition 3.5 (Marked Petri nets)**.** Marked Petri nets are 4-tuple nets, $N = (P, T, F, M)$ where $M \in \mathbb{B}(\mathbb{P})$ is a place multiset called marking, which denotes the number of tokens in places.

The firing sequence of transitions from Petri net corresponds to business execution in reality. A transition can be fired if it is in an enabled state where all the input places for this transition hold at least one token. After firing the transition, the token in the input places are consumed and new tokens are generated in the output places for this transition. Before we formalize the firing rule, we need several concepts below.
Definition 3.6 (Input and output nodes)**.** For a node x in Petri net $N = (P, T, F)$, the set of its input nodes is denoted as $\bullet x = \{y | (y, x) \in F\}$. The set of its output nodes is

denoted as $x\bullet = \{y|(x,y) \in F\}$.
Definition 3.7 (Firing rule)**.** In a marked Petri net $N = (P, T, F, M)$, a transition $t \in T$ is enabled, written as $N[t\rangle$, if and only if $\bullet t \subseteq M$. After firing t, the marking changes in this way, $M \Rightarrow (M \setminus \bullet t) \cup t\bullet$.

In real life, a subclass of Petri nets known as Workflow nets is often used to model business process.
Definition 3.8 (Workflow nets)**.** A workflow net is a labeled Petri net $N = P, T, F, \lambda$ with constraints:

- P contains a source place denoted as i, where $\bullet i = \emptyset$.
- P contains one sink place denoted as o, where $o\bullet = \emptyset$.
- When connecting the sink place to source place by an arc, this net seen as a graph becomes strongly connected.

However, not every Workflow net represents a correct process. To perform the business on an enterprise level, we define a minimum correctness criterion, known as soudness [19].
Definition 3.9 (Soundness)**.** A Petri net is sound if and only if it satisfies the following conditions.

- Safeness. Places cannot hold multiple tokens at the same time.
- Proper completion. If the sink place is marked, all other places are empty.
- Option to complete. It is always possible to reach the final marking from any reachable marking.
- No dead parts. For any transition, there exists a path from source to sink place through it.

3.2.2 Transition System

A transition system is a basic process model. As displayed in Figure 3.1b, it is composed of *states* and *transitions*. *States* are represented by black circles and include an initial and a final state. The initial state **s1** in Figure 3.1b is denoted with an input arc without any label and the final state **s4** is with an output arc without label. *Transitions*, also called actions, correspond to activities in the processes. They connect states by an arc.
Definition 3.10 (Transition System)**.** A transition system is a triple $TS = (S, A, T)$ where S is the set of states, A is the set of activities, $T \subseteq S \times A \times S$ is the set of transitions, $(s_i, a, s_j) \in T$ is represented as $s_i \xrightarrow{a} s_j$.

Transition systems are eligible to describe the dynamic behavior and can easily be translated to Petri nets. The example models in Figure 3.1b are mapped to each other. There are 5 transitions and four states in the transition system, which correspond to the transitions and places in Petri net.

3.2.3 Process Tree

Process tree is a tree and is suitable to represent a sound WF-net by construction according to [6]. Moreover, as block-structured in a tree, process tree is easy to understand and use.

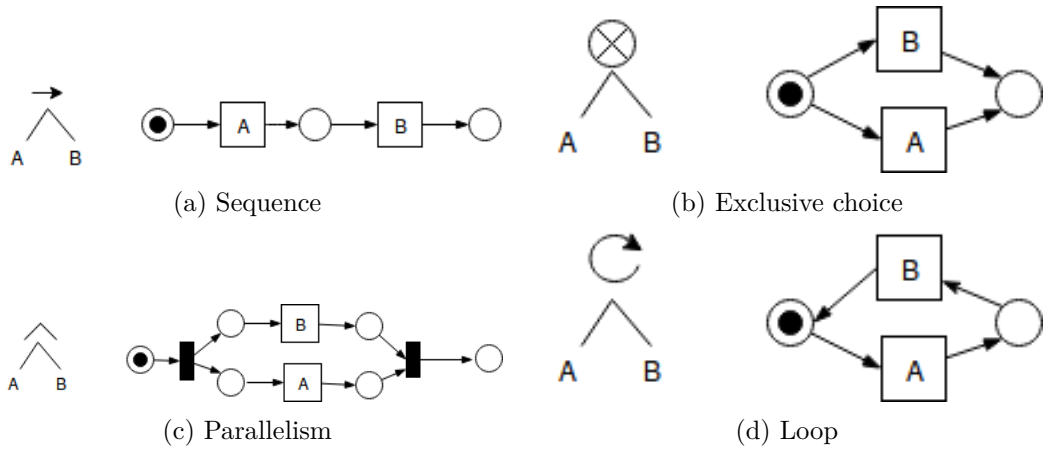
Therefore, we introduce it in our thesis.

Definition 3.11 (Process Tree)**.** Let $A \subseteq \mathbb{A}$ be a finite set of activities with silent transition $\tau \notin \mathbb{A}$, $\bigoplus \subseteq \{\rightarrow, \times, \wedge, \circlearrowleft\}$ 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 also a process tree, $1 \leq i \leq n$.

For convenience, given a process tree $Q = \oplus(Q_1, Q_2, ..Q_n)$, we denote Q the parent for $Q_1, Q_2, ..Q_n$, Q_i as one child of Q. For a node in a tree without children, we call it a ***leave node***; Else, we call them ***blocks***.

Process tree operators represent different block relations of each subtree. Their semantics are standardized in [20, 21] and compared with Petri net in Figure 3.2 [21]. We provide an informal description in the following.



(a) Sequence (b) Exclusive choice

(c) Parallelism (d) Loop

Figure 3.2: Translation of process tree operators to Petri net

Definition 3.12 (Operator Semantics)**.** The semantics of operators $\bigoplus \subseteq \{\rightarrow, \times, \wedge, \circlearrowleft\}$ are defined firstly in the context of its structure. The semantics with respect to a trace σ are also given.

- if $Q = (a)$, the tree only has one leaf node.
- if $Q = \rightarrow (Q_1, Q_2, ..Q_n)$, the subtrees have a 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 an 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 a 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 a loop relation for $\{Q_1, Q_2, ..Q_n\}$. $\{Q_2, ..Q_n\}$ have an exclusive relation. Q_1 executes at first and is triggered again once one subtree of $\{Q_2, ..Q_n\}$ executes.

According to the corresponding semantic relations, a process tree can be easily transformed into Petri net. In Figure 3.3b, it is the process model in process tree which describes the

same process as in Figure 3.1a.

To describe the execution of a process tree Q, we need concepts called ***start nodes*** and ***end nodes***. The start nodes represent the firstly executed leaves nodes in Q; Similarly, the end nodes represent the last executed leaves node set in Q.
Definition 3.13 (Start node and end node set)**.** Given a process tree Q, its start node $S(Q)$ and end node set $E(Q)$ is defined as following.

- $Q = (a) \quad \Rightarrow S(Q) = \{(a)\} = E(Q)$;
- $Q = \rightarrow (Q_1, Q_2, ..Q_n) \Rightarrow S(Q) = S(Q_1); E(Q) = E(Q_n)$. The start node set is the start nodes of its first child, and the end node set is the end nodes from its last child.
- $Q = \times(Q_1, Q_2, ..Q_n) \Rightarrow S(Q) = \cup_{i\in\{1,..n\}} S(Q_i), E(Q) = \cup_{i\in\{1,..n\}} E(Q_i)$.
  For an exclusive choice block, its start and end node sets are the union of start and end nodes from each child; Any set from the union is an option start node set to represent the execution of Q.
- $Q = \wedge(Q_1, Q_2, ..Q_n) \Rightarrow S(Q) = \prod_i S(Q_i), E(Q) = \prod_i E(i)$.
  For parallel relation, the start nodes are all the start nodes from each child and end nodes are all the end nodes from each child.
- $Q = \circlearrowleft (Q_1, Q_2, ..Q_n) \Rightarrow S(Q) = S(Q_1), E(Q) = E(Q_1)$
  The start node sets and end node sets depend on its first child.

3.3 Inductive Miner

Among multiple discovery techniques to mine a process model from an event log, Inductive Miner suits well our needs, because it guarantees the construction of a sound models, and is flexible and scalable w.r.t. event log. In this section, we explain its algorithms in details.

3.3.1 Construct a Directly-Follows Graph

At the start, an event log L is scanned to extract the *directly-follows relation* of events which describes the execution order of pair activities. We denote directly-follows relation in an event log by $>_L$.

For example, given an event log

$$L_{IM} = \{< a, c, d >^{20}, < b, c, e >^{10}, < a, c, e >^{20}, < b, c, d >^{10}\},$$

a is directly followed by b and c, denoted as $a >_L b, a >_L c$; d directly follows b and c, $b >_L d, c >_L d$.

later, those directly-follows relations are combined together to build a directly-follows graph with frequency. According to [6, 8], formal definitions for the concepts above are given.

Definition 3.14 (Directly-follows Graph)**.** The directly-follows relation $a >_L b$ is satisfied iff

$$\exists \sigma \in L, 1 \leq i < |\sigma|, \sigma(i) = a \; and \; \sigma(i+1) = b.$$

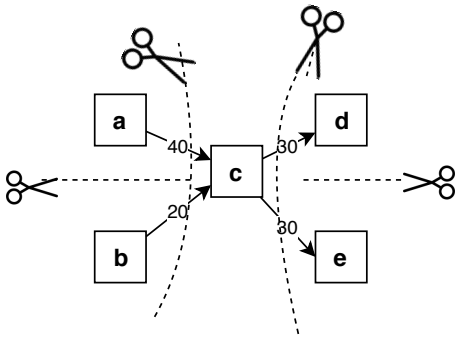
A directly-follows graph of an event log L is $G(L) = (A, F, A_{start}, A_{end})$ where A is the set of activities in L, $F = \{(a, b) \in A \times A | a >_L b\}$ is the directly-follows relation set, A_{start}, A_{end} are the set of start and end activities respectively, $A_{start} = \{a | \exists \sigma \in L, a = \sigma(1)\}$, $A_{end} = \{a | \exists \sigma \in L, a = \sigma(|\sigma|)\}$.

What's more, the frequency information of the directly-follows relation is stored to express the relation strength and denoted as cardinality.

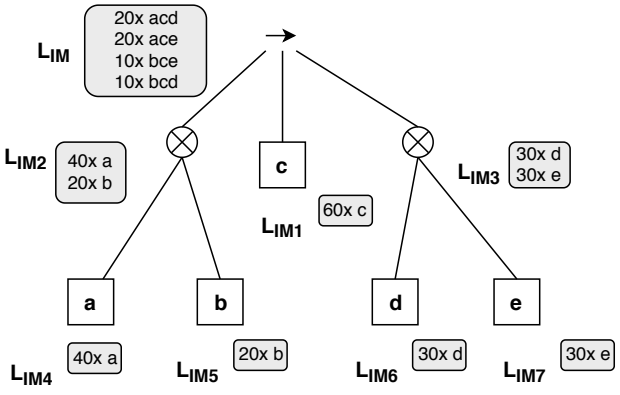
Definition 3.15 (Cardinality in a directly-follows graph)**.** Given a directly-follows graph $G(L) = (A, F, A_{start}, A_{end})$ derived from an event log L, the cardinality in G(L) is defined respectively on the F, A_{start}, and A_{end}.

- $\forall (a, b) \in F, c(a, b) = \sum_{\sigma \in L} |\{i \in \{1, 2...|\sigma|\} | \sigma(i) = \sigma(i+1)\}|$ is the frequency for directly-follows relation $a >_L b$.
- $\forall a \in A_{start}, c(a) = \sum_{\sigma \in L} |\{\sigma | \sigma(1) = a\}|$ is the frequency for a start activity.
- $\forall a \in A_{end}, c(a) = \sum_{\sigma \in L} |\{\sigma | \sigma(|\sigma|) = a\}|$ is the frequency for an end activity.

According to definitions above, we obtain a directly-follows graph from event log L_{IM} as shown in the Figure 3.3a. Cardinality information is listed on the connections of activities.



(a) The directly-follows graph from L_{IM}. The cut on directly-follows graph is shown in dashed line.



(b) IM to split L_{IM} for building a process tree

Figure 3.3: Inductive Miner to discover a process tree from event log L_{IM}

3.3.2 Split Log Into Sublogs

Based on the directly-follows graph, the Inductive Miner finds the most prominent cut which is applied afterwards to split the event log into smaller sublogs. Corresponding to process tree operators $\{\rightarrow, \times, \wedge, \circlearrowleft\}$, cuts consist of *exclusive-choice cut, sequence cut, parallel cut and redo-loop cut*. 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. As a result, sublogs are generated based on those cuts. Meanwhile, the

operators which corresponds to those cuts are used to build the process tree. The same procedure is applied again on the sublogs until single activity set.

As an example, we apply the splitting on event log L_{IM} and $G(L_{IM})$ illustrated in Figure 3.3b. Firstly, the $\rightarrow$ cut is applied and divides the whole event log into three sublogs $L_{IM1}, L_{IM2}, L_{IM3}$. Since L_{IM1} includes only one single activity, so we stop splitting and build a corresponding node in the process tree. Next, L_{IM2}, L_{IM3} are split by exclusive choice cuts separately until meeting single set in $L_{IM4}, L_{IM5}, L_{IM6}, L_{IM7}$.

Furthermore, a process tree is able to be converted into Petri net.

Chapter 4

Algorithm

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

4.1 General Framework for Repairing Process Models

Figure 4.1 shows our proposed framework to repair a process model with negative information. The inputs are a reference process model and a labeled event log. The reference process model can be in multiple types, e.g. Petri net, process tree. Traces in the labeled event log are classified as positive or negative in respect to some KPIs of business processes. The output is a repaired process model with the same type as the reference model.

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

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

4.2 Algorithm

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

Based on those scope, the main modules in the framework in Figure 4.1is designed and developed to repair the reference Petri net. First of all, we define a proper data model to

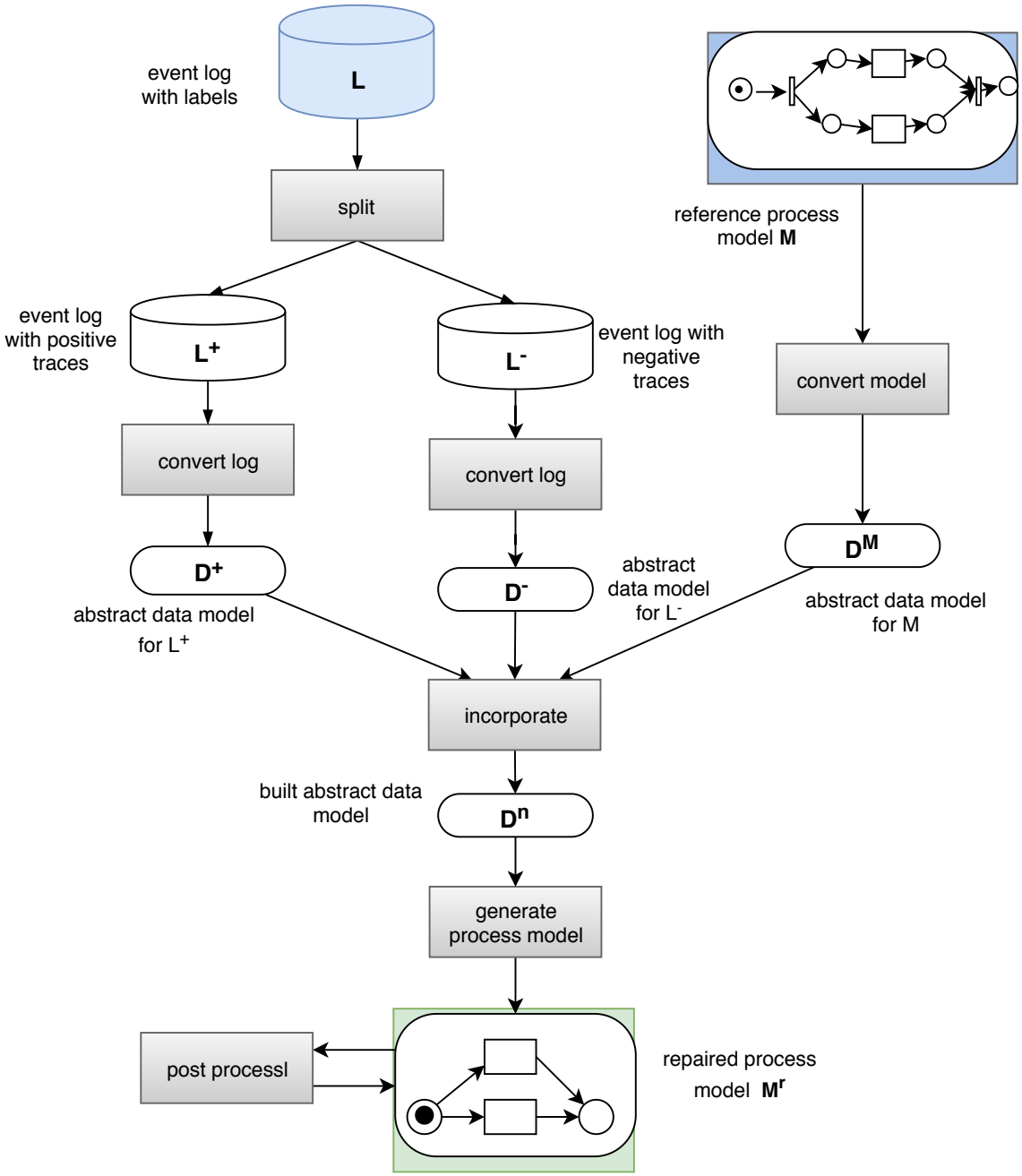


Figure 4.1: Model Repair Architecture – Rectangles represents processes and data models in eclipse shape. Input event log and the reference model are in blue, while the repaired model is in green.

represent the impact from Petri net M, and the event log L. Then, we list all the modules and describe our basic ideas to implement them.

4.2.1 Unified Data Model

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

Although we can derive three directly-follows graphs from the reference model, the positive and negative event logs respectively, their cardinalities are in different level and not able to incorporate with each other. So we introduce a concept called unified cardinality to bring all the impact into the same level, which is the percentage to the sum of total cardinalities with a range [0-1].
Definition 4.1 (Unified cardinality)**.** Given a directly-follows graphs $G(L) = (A, F, A_{start}, A_{end})$ for a model, the unification for this graph is a function $u : F \to N$ that has the following definition:
for each directly-follows relation $(a, b) \in F$,

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

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

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

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

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

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

4.2.2 Modules List

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

- *Split event log into positive and negative sublogs* The event log L is split into an event log L^+ with only positive traces and an event log $L^-$with negative traces.
- *Convert event logs into unified directly-follows graphs, D^+, D^-* Two unified directly-follows graphs are generated respectively for the positive instance and negative instances from the event log.

- *Convert reference model into unified directly-follows graph* D^M
- *Incorporate unified directly-follows graphs* Three unified directly-follows graphs D^M, D^+, D^- are combined into one single directly-follows graph D^n after balancing their impact.
- *Generate process models from* D^n Process models are mined from the repaired data model D^n.
- *Post process the repaired model* Several post-processes are optionally applied on the generated process model, in order to improve the process model quality according to certain criteria.

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

4.2.3 Convert Event Logs into Unified Directly-follows Graphs

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

By applying *IMLog2Dfg* separately on the event logs L^+ and L^-, we generate two directly-follows graphs $G(L_{pos})$ and $G(L_{neg})$. In the next step, the cardinalities from those graphs are unified according to Definition 4.1 and become a part of unified data model $D(L_{pos})$ and $D(L_{neg})$

4.2.4 Convert Reference Model into Unified Directly-follows Graph

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

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

From the positive and negative event logs, we can get the cardinality for corresponding directly-follows graph to represent the strength of this directly-follows relation. However, when the existing model is transformed into directly-follows graph $G(L_{ext})$, there is no point to assign cardinality on each edge. So we just set cardinality with 1 for each arc. Based on this cardinality assignment, we attain the unified data model D^M for the reference Petri net.

4.2.5 Incorporate Unified Directly-follows Graphs

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

Definition 4.2 (Incorporating method)**.** For any directly-follows relation from D^M,D^+, and D^-, we balance all forces on it in the following way.

- For one directly-follows relation,
$$u^n(a,b) = u^M(a,b) + u^+(a,b) - u^-(a,b)$$
- For a start activity $a \in A^M_{start} \cup A^+_{start} \cup A^-_{start}$,
$$u^n(a) = u^M(a) + u^+(a) - u^-(a)$$
- For an end activity $a \in A^M_{end} \cup A^+_{end} \cup A^-_{end}$
$$u^n(a) = u^M(a) + u^+(a) - u^-(a)$$

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

Definition 4.3 (Weighted incorporating method)**.** Given the control weight w^M,w^+, and $w^- \in [0,1]$, the weighted incorporating method to balance forces from D^M,D^+, and D^- for D_n is defined below.

- For one directly-follows relation,
$$u^n_w(a,b) = w^M * u^M(a,b) + w^+ * u^+(a,b) - w^- * u^-(a,b)$$
- For a start activity $a \in A^M_{start} \cup A^+_{start} \cup A^-_{start}$,
$$u^n(a) = w^M * u^M(a) + w^+ * u^+(a) - w^- * u^-(a)$$
- For an end activity $a \in A^M_{end} \cup A^+_{end} \cup A^-_{end}$
$$u^n(a) = w^M * u^M(a) + w^+ * u^+(a) - w^- * u^-(a)$$

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

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

4.2.6 Generate Process Models from D^n

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

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

4.2.7 Post Procedure on the Process Model

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

4.2.7.1 Add Long-term Dependency

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

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

An exclusive choice structure is represented as ***xor block*** in a process tree. For the sake for convenience, we name a subtree of an xor block as one ***xor branch***. For two arbitrary xor branches X,Y with long-term dependency, denoted as $X \rightsquigarrow Y$, they have to satisfy the conditions:

- they have a sequential order;
- they have significant correlation. In this thesis, we assume, the more frequent they are executed in the same traces, the more significant correlation they have.

The order of xor branch follows the same rule of node in process tree which is explained in the following.

Definition 4.4 (Order of nodes in process tree)**.** Node X is before node Y, written in $X \prec Y$, if X is always executed before Y. In the context of process tree structure, $X \prec Y$, if the least common ancestor of X and Y is a sequential node, and X positions before Y.

To define the correlation of xor branches, several concepts listed below are necessary.
Definition 4.5 (The execution of a process tree Q)**.** With respect to a trace $\sigma \in L$, a process tree Q is executed in σ, denoted as $\sigma \models Q$, when

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

With the definition above, we can formalize the frequency of xor branch in the event log as below.
Definition 4.6 (Xor branch frequency)**.** The frequency for an xor branch X in event log L is the count of traces, $f : X \rightarrow N$.

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

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

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

As an example, given the same model M4 and reviewed here in Figure 4.2a, and a labeled event log L,

$$L := \{< A, C, D >^{10,pos}, < A, C, E >^{30,pos}, < B, C, E >^{10,pos}, < B, C, D >^{40,neg}\}$$

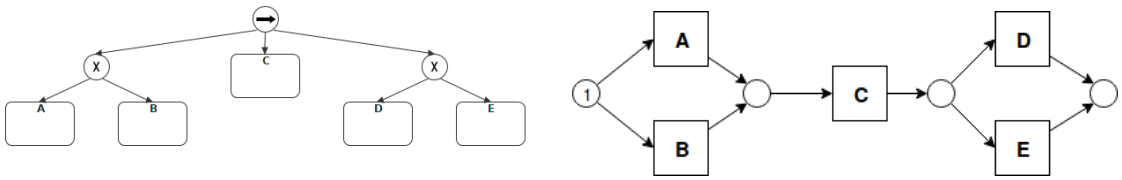
the frequencies of the coexistence of different xor branches are $f_{L^+}(A, D) = 10$, $f_{L^+}(A, E) = 30$, $f_{L^+}(B, E) = 10$, and $f_{L^-}(B, D) = 40$.

The frequency of the coexistence of multiple xor branches in positive and negative event logs reflects the correlation of those xor branches. The long-term dependency in the existing model also affects the long-term dependency in the repaired model. However, since the repaired model possibly differs from the existing model, the impact of long-term dependency from the existing model becomes difficult to detect. With limits of time, we only consider the impact from positive and negative instances on the long-term dependency.
Definition 4.7 (Correlation of xor branch)**.** The correlation to express dependency of two branches $X, Y \in BB(Q)$ is expressed into

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

, where

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



(a) The reviewed process tree PT (b) The reviewed Petri net

In this thesis, we call the correlation strong with respect to one threshold t, if

$$d(X, Y) > t.$$

Following the example above, the correlations of their xor branches are $d(A, D) = 0.2, d(A, E) = 0.6, d(B, E) = 0.2, d(B, D) = -1$. Since the xor branches of those pairs are already in order, if the threshold for long-term dependency is set as $t = 0$, we get a long-term dependency set, $LT = \{A \rightsquigarrow D, A \rightsquigarrow E, B \rightsquigarrow E\}$, for the process tree.

4.2.7.2 Cases Analysis

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

Definition 4.8 (Source and target set of long-term dependency)**.** The source set of the long-term dependency in two xor blocks is the set of all xor branches, $LT_S := \{X | \exists Y, X \rightsquigarrow Y \in LT\}$, and target set is $LT_T := \{Y | \exists X, X \rightsquigarrow Y \in LT\}$.
For one xor branch $X \in S$, the target xor branch set relative to it with long-term dependency is $LT_T(X) = \{Y | X \rightsquigarrow Y \in LT\}$ Similarly, the source set related to one target xor branch is $LT_S(Y) = \{X | X \rightsquigarrow Y \in LT\}$

At the same time, we use S and T to represent the set of xor branches for source and target xor block with long-term dependency. Given a process tree in Figure 4.2a, two xor blocks are contained in the model. They are able to have the following long-term dependencies.

1. $LT = \{A \rightsquigarrow D, A \rightsquigarrow E, B \rightsquigarrow D, B \rightsquigarrow E\}$.
   $LT_S = \{A, B\}, LT_T = \{D, E\}, |LT| = |S| * |T|$, which is a full long-term dependency where all combinations of source and target xor branches have significant correlation.

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

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

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

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

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

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

In the listed situations, situation 1 is a full long-term dependency, which allows all combinations like the original models, and no changes is necessary on the model. Situation 7 has no long-term dependency, where the correlation of them is either not so strong from the positive instances or rejected from negative instances. In this thesis, we don't take those two situations into consideration.

4.2.7.3 Way to Express Long-term Dependency

After detecting long-term dependencies with xor branches in process tree, it is not possible to express those dependencies on process tree by using the current process tree operators. Moreover, the final output of our task demands a Petri net. Therefore, we convert the generated process tree into a Petri net and express long-term dependency on the Petri net with corresponding structures.

According to [9], adding places to transitions in Petri net can limit the model behavior. Any transition demands a token from its input places to fire. When there is no token at this place, the transitions are not enabled. By adding extra places to the Petri net, it can block negative behaviors which are not expected in the aspect of business performance.

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

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

Reviewing the example of process tree of Figure 4.2a, it has long-term dependency set $LT = \{A \rightsquigarrow D, A \rightsquigarrow E, B \rightsquigarrow E\}$. To express long-term dependency, the process tree is firstly transformed into the Petri net as Figure 4.2b. Then two extra places are added

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

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

respectively after A and B; Next, two places before D and E are created to express that the xor branches are involved with long-term dependency. At end, for each long-term dependency, a silent transition is generated to connect the extra places after the source xor branch to the place before target place. The generated model with long-term dependency is displayed in Figure 4.3.

4.2.7.4 Soundness Analysis

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

Given a Petri net with long-term dependency $LT = \{X_i \rightsquigarrow Y_j | 1 \leq i \leq m, 1 \leq j \leq n\}$ on two xor blocks $S = \{X_1, X2, ...X_m\}$ and $T = \{Y_1, Y_2, ...Yn\}$. After applying the Algorithm

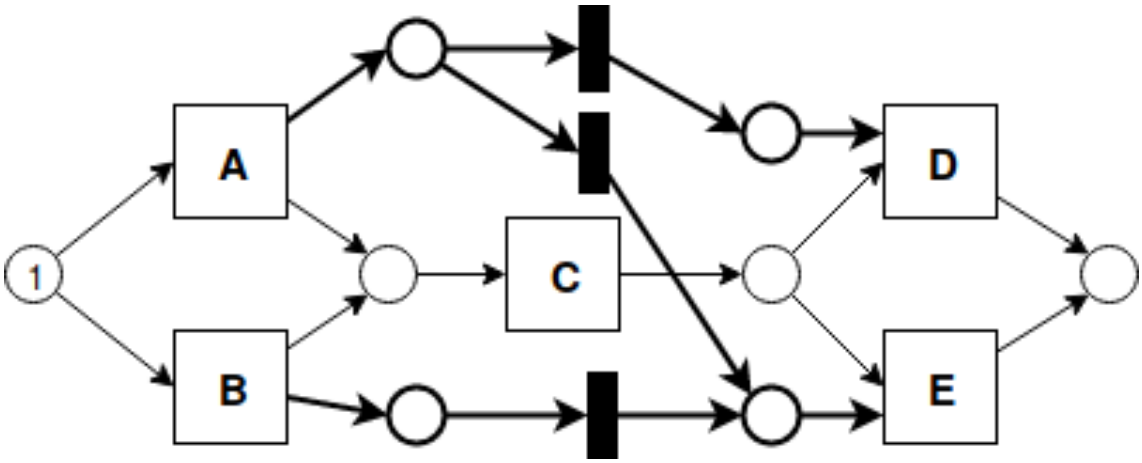


Figure 4.3: Model with long-term dependency $LT = \{A \rightsquigarrow D, A \rightsquigarrow E, B \rightsquigarrow E\}$.

1, $P_S = \{p_{X_i} | X_i \in LT_S\}$, $P_T = \{p_{Y_j} | Y_i \in LT_T\}$, and silent transitions $E = \{\epsilon | p_{X_i} \rightarrow \epsilon \rightarrow p_{Y_j}\}$ are added.

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

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

Soundness inside xor blocks.
For the xor blocks $S = \{X_1, X2, ...X_m\}$ and $T = \{Y_1, Y_2, ...Yn\}$ with long-term dependency $LT = \{X_i \rightsquigarrow Y_j | 1 \leq i \leq m, 1 \leq j \leq n\}$, only one xor branch can be fired in S. Without loss of generality, X_i is assumed to be enabled. After firing X_i, the marking distribution on the extra places are

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

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

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

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

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

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

So whenever the marking distribution in the extra places are

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

- Proper completion. If the sink place is marked, all other places are empty.
  After firing Y_j, all the extra places hold no token. So it does not violate the proper completion.
- Option to complete. It is always possible to reach the final marking just for the sink place.
  There is always one Y_j enabled after firing X_i to continue the subsequent execution.
- No dead part. For any transition there is a path from source to sink place.
  Because all $Y_j \in T$ are also in LT_T, there exists at least one $X_i \in S$ with long-term dependency with Y_j. After X_i is fired, one token is generated on the extra place p_{X_i} and can be consumed by silent transition ϵ in $p_{X_i} \rightarrow \epsilon \rightarrow p_{Y_j}$ to produce a token in p_{Y_j}, which enables xor branch Y_j and leaves no dead part.

In other situation, the model becomes unsound.
If $LT_S \neq S, or \quad LT_T \neq \emptyset$, there exists one xor branch X_i with $X_i \notin LT_S$. When X_i is fired, it generates one token at place p_{X_i}, this token cannot be consumed by any Y_j. So it violates the proper completion.
If $LT_T \neq T$, there exists one $Y_j \notin LT_T, \nexists X_i, X_i \rightsquigarrow Y_j$, so with two input places but $Token(p_{Y_j}) = 0$, Y_j becomes the dead part, which violates the soundness again. As a conclusion, to keep Petri net with long-term dependency sound, only situations with $LT_S = S, LT_T = T$ are considered.

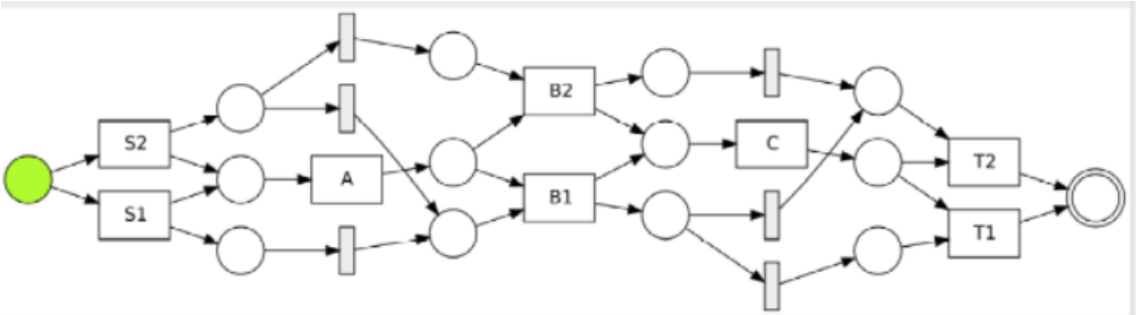
4.2.8 Reduce Silent Transitions

Our method to represent long-term dependency can introduce redundant silent transitions and places, which complicates the model. So, we post process the Petri net with long-term dependency and delete redundant silent transitions and places in it. Silent transitions and places of a Petri net are redundant when the Petri net preserves the behavior without those transitions and places [22, 23]. There is an existing algorithm developed in work of [23]. We reuse it as one post procedure in our thesis.

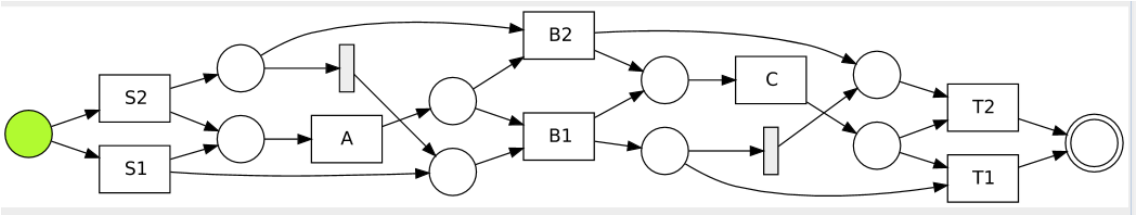
One example is given in the following graph. M_{lt} in Figure 4.4a has long-term dependency expressed in the silent transitions and places. The silent transition for $S2 \rightsquigarrow B1$ and silent transition for $B1 \rightsquigarrow T2$ belongs to the case (1). So they are kept in the model, while the other silent transitions are deleted. After reducing the redundant silent transitions, the model becomes M_r shown in Figure 4.4b. Those two models have the same behavior, yet the reduced model is simpler.

4.2.9 Concrete Architecture

At last, we assemble all the modules together and give an overview architecture of our repair techniques. We reuse existing modules in gray rectangles in Figure 4.5, e.g. IMLog2Dfg to convert an event log into directly-follow graph, Petrinet2TransitionSystem to



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



(b) Petri net M_r with reduced silent transitions

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

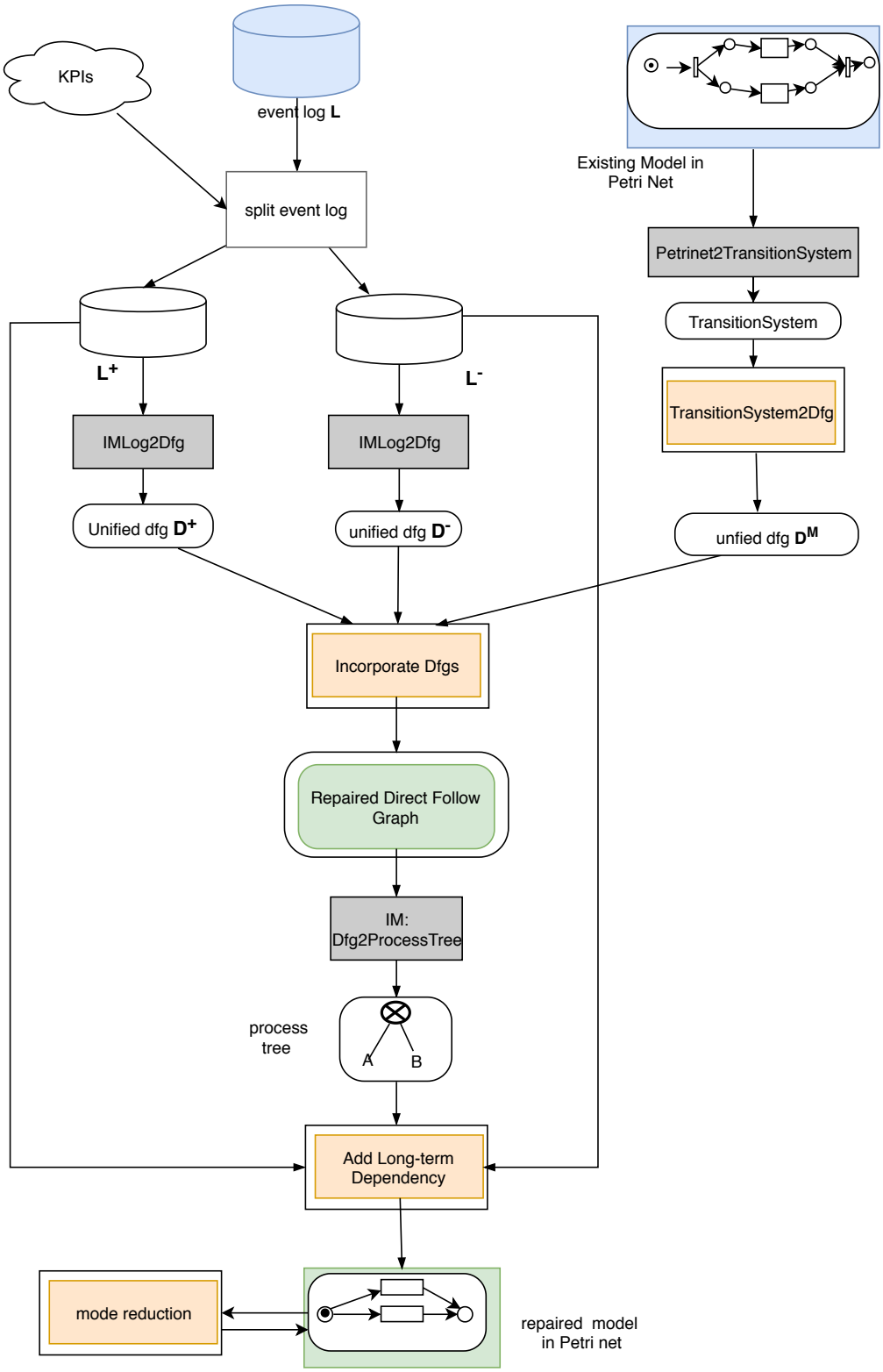


Figure 4.5: Model Repair Architecture – Rectangles represents processes and output data in eclipse shape, especially customized processes and data are in doubled lattice shape. Input event log and existing model are in blue, KPIs are in cloud. The output is a petri net in purple.

Chapter 5

Implementation

In this chapter, we begin with the introduction of implementation platforms for our methods and then show the use of those applications step by step.

5.1 Implementation Platforms

5.1.1 Process Mining Platform – ProM

ProM is an open-source process mining tool in Java that is extensible by adding a set of plug-ins [24]. ProM supports a wide variety of process mining techniques and is usually used for academic research. We implement the algorithm on ProM 6.8, which is the latest stable version. The corresponding plugin is ***Repair Model By Kefang*** and released online [25].

5.1.2 KNIME

KNIME Analytics Platform is an open-source software to help researchers analyze data. Multiple modules are integrated into this platform for loading, transforming and processing data. Researchers can achieve their goals by creating visual workflows composed of expected modules implemented as nodes with an intuitive, drag and drop style graphical interface, rather than focusing on any particular application area.

The reasons to integrate our techniques into KNIME are (1)KNIME is widely used in scientific research and benefits the application of our techniques;(2)KNIME supports automation of test workflow, which helps conduct more efficient experiments. However, the integration requires additional development effort.

5.2 Generate a Petri net

Firstly two dialogs are poped up to set the arguments, such as the event classifier to generate directly-follows graphs from event logs. Subsequently, a dialog is shown to set

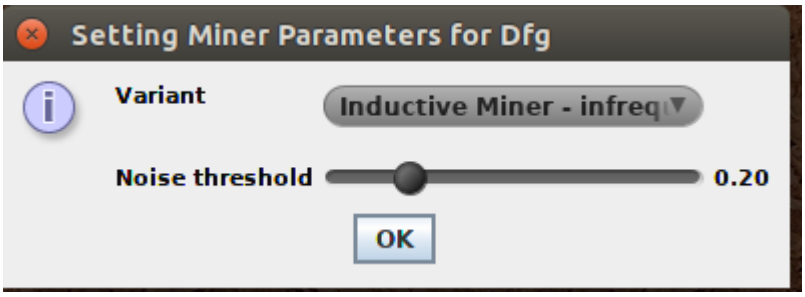


Figure 5.1: Inductive Miner Parameter Setting

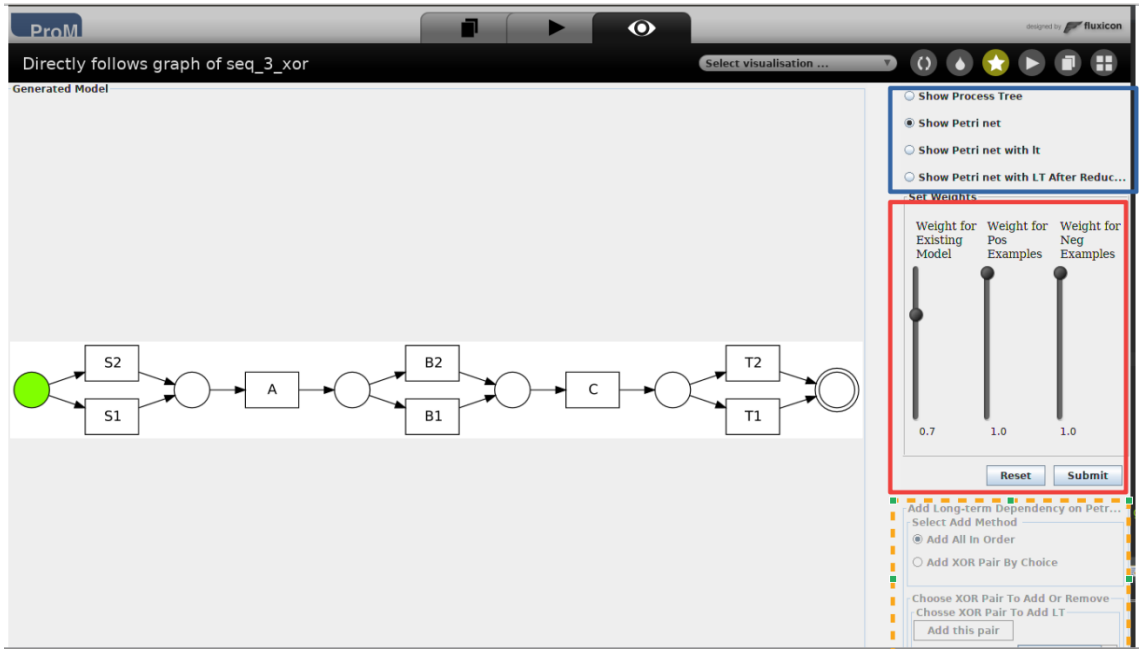


Figure 5.2: Generated Petri net without long-term dependency

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, where the right panel is to set the control parameters for the existing model, positive or negative instances. In interactive way, more flexibility is allowed by this plug-in to repair model. By default, the generated model type and the weight sliders are enabled at first. The control panel for adding long-term dependency are only triggered after choosing the option to repair model with long-term dependency.

The model type is selected 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. They adjust the weights for the existing model, positive and negative instances. Once those options are submitted, different process models are mined under different weights. The rectangle in orange are the invisible part to control long-term

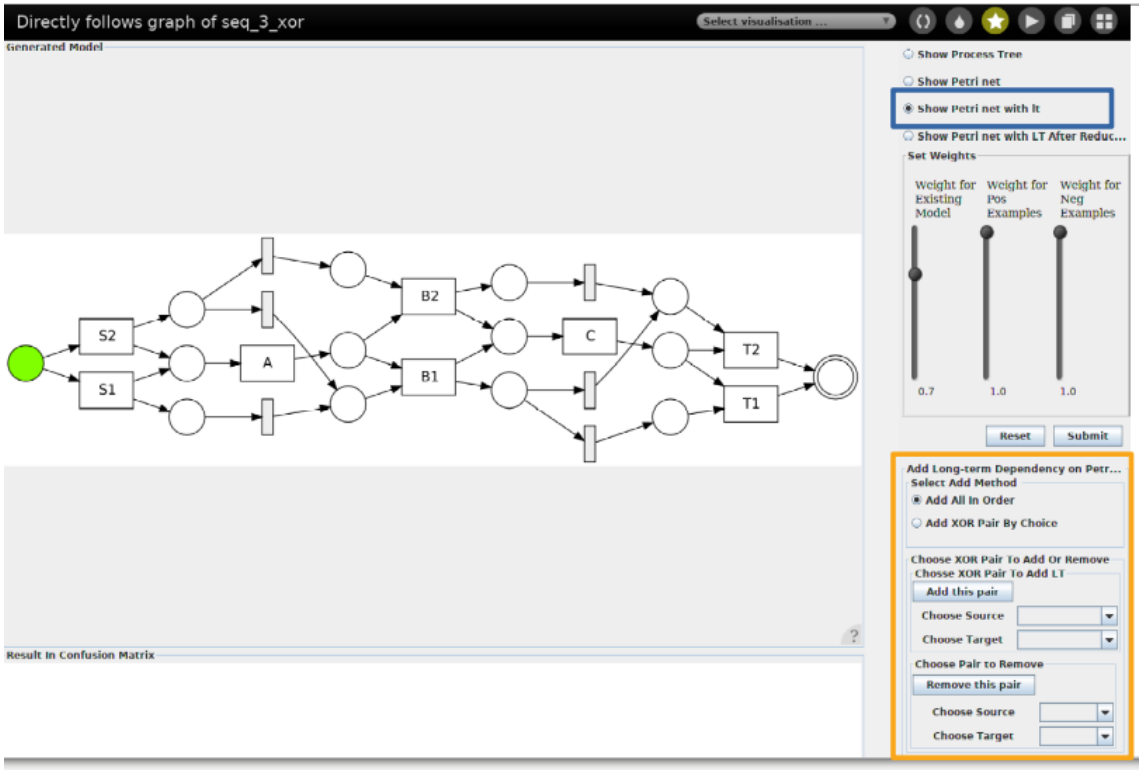


Figure 5.3: Petri Net with long-term dependency

dependency options. It will be discussed in the next section.

5.3 Post Process to Add Long-term Dependency

If we want to repair the Petri net with long-term dependency, one post procedure is triggered to add long-term dependency . 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 displayed in Figure 5.3.

Meanwhile, the control part of adding long-term dependency turns visible in the orange rectangle like in Figure 5.3. It has two main options, one is to consider all long-term dependencies existing in the model, the other is to choose the part manually. It allows more flexibility for users. Below those two options are the manual selection panels, including a 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.

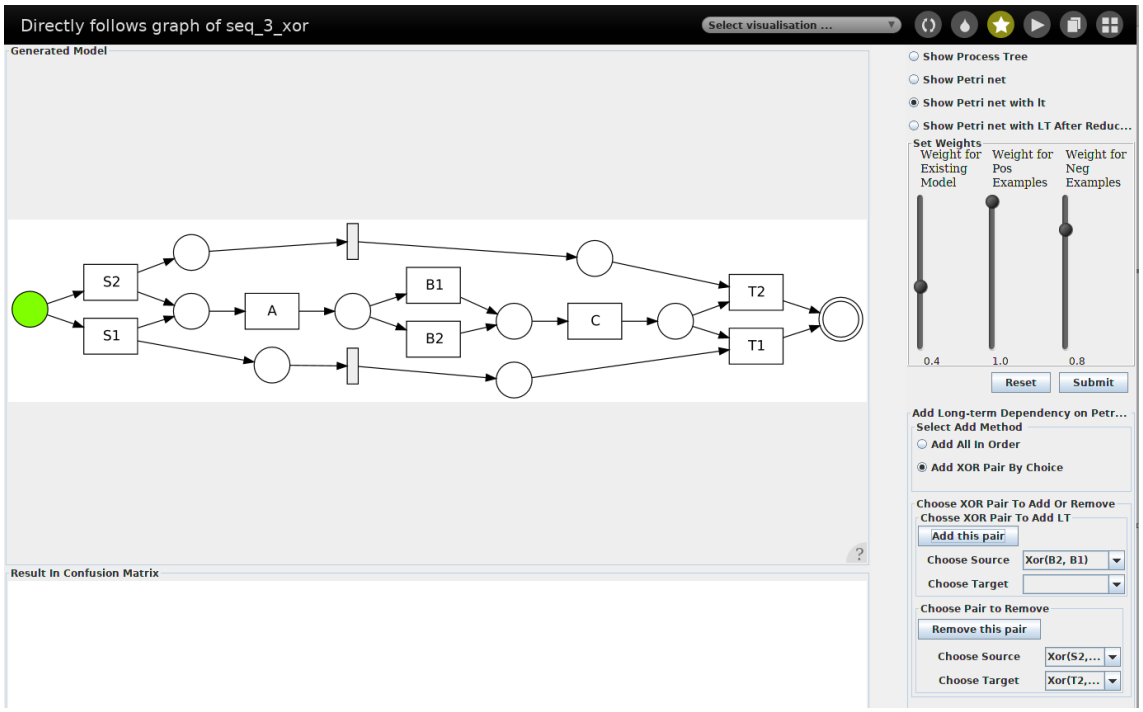


Figure 5.4: Petri net with selected long-term dependency

5.4 Post Process to Reduce Redundant Silent Transitions and Places

By choosing the option of *Petri net with LT After Reducing* in model type, silent transitions and places 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.

5.5 Additional Feature to Show 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 parameters for the optimal Petri net.

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. A naive fitness checking is applied on the repaired model with the event log. This procedure is based on the existing plugin in ProM – **PNetReplayer**. This plugin checks if the trace fits the model and give out the one possible deviation with minimal cost. In our implementation, either the deviation on model or in trace is set with the same cost. Based on the deviation result and the label information on each trace, a confusion matrix is generated. Moreover, relative measurements like recall, precision are calculated and 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

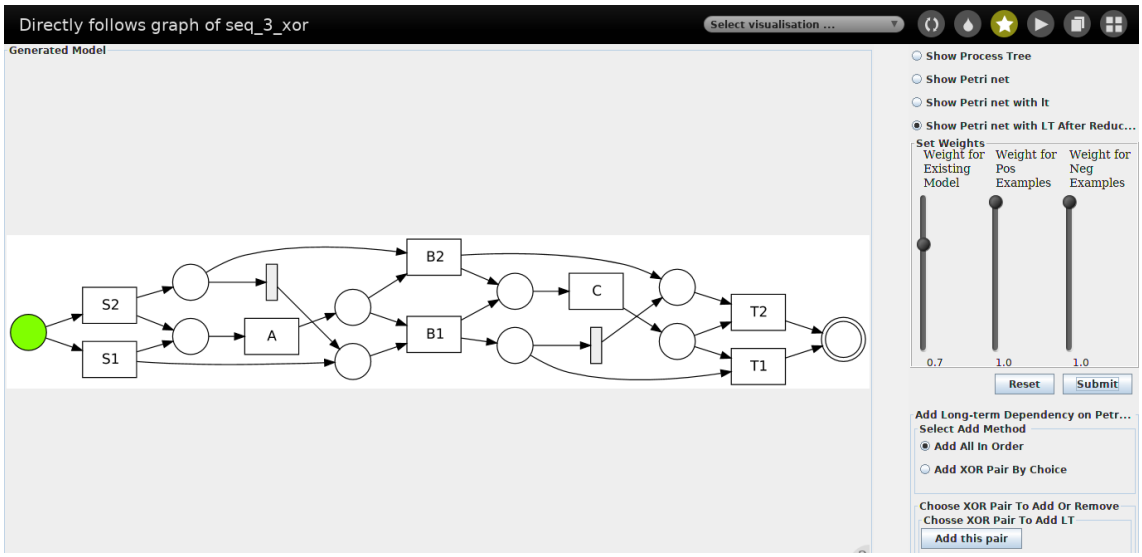


Figure 5.5: Petri net after reducing the silent transitions

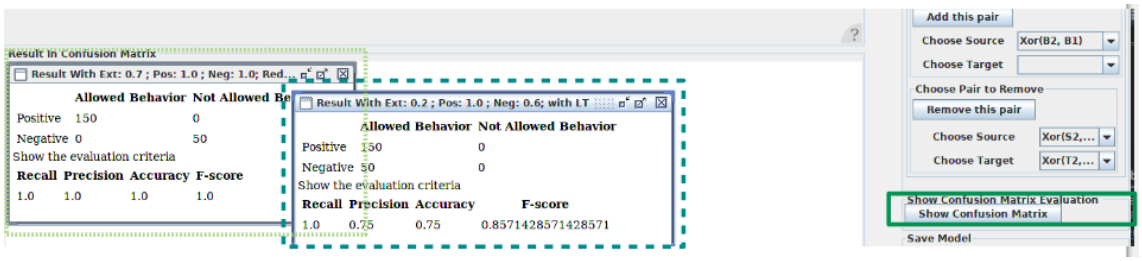


Figure 5.6: Generated Process Tree Model

matrix result in the dark green dashed rectangle which will be listed above the previous result in light green dashes area.

5.6 Integration into KNIME

Nodes in the workflow represents different modules corresponding the plugins in ProM. Each node has certain input ports on the left side to represent the required parameters and ports on the right to output result. By connecting the ports between nodes, data are passed and processed by one node after another. To integrate our algorithm into KNIME, other related modules on process mining are necessary, which can be divided into the following categories:

1. Data importer and exporter. The importers and exporters for event logs, process trees and Petri nets are implemented to load and save basic data for Process Mining.
2. Event logs manipulation. Nodes for splitting, sampling and assigning labels to event logs are implemented to benefit our experiments.
3. Classic discovery algorithms. Inductive Miner and Alpha Miner are integrated into KNIME to provide baselines for our algorithm.

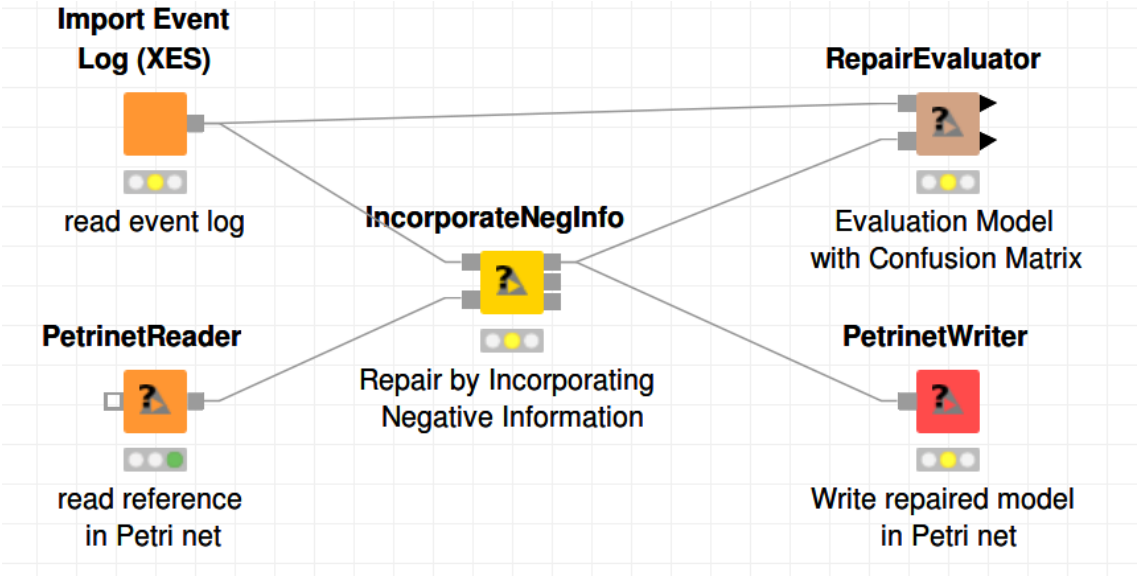


Figure 5.7: Integration of our repair techniques into KNIME

4. Model enhancement. Our proposed method is integrated in KNIME to repair model in Petri net.

To integrate our repair algorithm from ProM into KNIME, we need to create the workflow in the Figure 5.7. After reading a Petri net by *PetrinetReader* and an event log by *Import Event Log(XES)*, Node *IncorporateNegInfo* applies the algorithm in this thesis to repair a model in Petri net with incorporating negative information. The outputs have different kinds of Petri nets to match the ones generated in ProM, eg. reduced Petri net with long-term dependency, Petri net without long-term dependency. In addition, we can evaluate our repaired model by using the node *RepairEvaluator*. At last, we can save the repaired Petri net by *PetrinetWriter*.

Chapter 6

Evaluation

In this chapter, we evaluate our repair techniques based on the quality of repaired model. At first, we define the evaluation criteria. Next, we briefly introduce the test platforms KNIME and relevant ProM plugins tools. Then, we conduct two kinds of tests. One is based one the demo example proposed in the introduction part, one is on the real life data.

6.1 Evaluation Measurements

We evaluate repair techniques based on the quality of repaired models with respect to the given event logs. In process mining, there are four quality dimensions generally used to compare the process models with event logs.

- *fitness.* It quantifies the extent of a model to reproduce the traces recorded in an event log which is used to build the model. Alignment-based fitness computation aligns as many events from trace with the model execution as possible.
- *precision.* It assesses the extent how the discovered model limits the completely unrelated behavior that doesn't show in the event log.
- *generalization.* It addresses the over-fitting problem when a model strictly matches to only seen behavior but is unable to generalize the example behavior seen in the event log.
- *simplicity.* This dimension captures the model complexity. According to Occam's razor principle, the model should be as simple as possible.

The four traditional quality criteria are proposed in semi-positive environment where only positive instances are available. Therefore, when it comes to the model performance, where negative instances are also possible, the measurement metrics should be adjusted. With labeled traces in the event log, the repaired model can be seen as a binary prediction model where the positive instances are supported while the negative ones are rejected. Consequently, the model evaluation becomes a classifier evaluation.

Confusion matrix has a long history to evaluate the performance of a classification model. A confusion matrix is a table with columns to describe the prediction model and rows

Table 6.1: Confusion Matrix

		repaired model	
		allowed behavior	not allowed behavior
actual data	positive instance	TP	FN
	negative instance	FP	TN

for actual classification on data. The repaired model can be seen a binary classifier and produces four outcomes- true positive, true negative, false positive and false negative shown in the Table 6.1.

- True Positive(TP): The execution allowed by the process model has an positive performance outcome.
- True Negative(TN): The negative instance is also blocked by the process model.
- False Positive(FP): The execution allowed by the process model has an negative performance outcome.
- False Negative(FN):The negative instance is enabled by the process model.

Various measurements can be derived from confusion matrix. According to our model, we choose the following ones as the potential measurements.

- recall. It represents the true positive rate and is calculated as the number of correct positive predictions divided by the total number of positives.

$$Recall = \frac{TP}{TP + FN}$$

- precision. It describes the ability of the repaired model to produce positive instances.

$$Precision = \frac{TP}{TP + FP}$$

- accuracy. It is the proportion of true result among the total number. It measures in our case how well a model correctly allows the positive instances or disallows the negative instances.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

- F-score is is the harmonic mean of precision and recall.

$$F_1 = \frac{2 * Recall * Precision}{Precision + Recall}$$

Generally, there is a trade-off between the quality criteria. So the measurements are only used to compare specific aspects of our techniques.

6.2 Experiment Platforms

KNIME, as a scientific workflow analytic platform, supports automation of test workflow, which helps us repeat experiments efficiently. Yet, traditional evaluation plugins in ProM are not integrated into KNIME, so partial experiments are conducted in ProM.

6.2.1 KNIME

KNIME supports automation of test workflow mainly through the following mechanisms.

- Loop Control Structure. KNIME provides a bunch of control nodes which support re-executing workflow parts. Nodes representing *Loop Start* appear in pairs with nodes for *Loop Nodes*, the workflow between pairs is executed recursively in a fixed number, or until certain conditions are met. In our test, we repeat our repair techniques for different parameter settings by applying loop structure into KNIME workflow.
- Flow Variables. Flow Variables are used inside a KNIME workflow to parameter node settings dynamically. When it combines with loop control structure, tests with different settings is able to conduct automatically.

Furthermore, there are nodes provided by KNIME to optimize the value of some parameters with respect to a cost function. As long as a cost function is provides, KNIME is able to automatically optimize any kind of parameters.

6.2.2 ProM Evaluation Plugins

Although KNIME offers a powerful approach to conduct experiments, the integration of traditional process mining evaluation plugins into KNIME is out of our capability due to the time limits. To complete the experiments, the following plugins in [1] are in need.

- Model Repair. This plugin is developed on the work in [2, 5]. It repairs a Petri net according to an event log.
- PNetReplayer. It checks conformance of the process model in Petri net with an event log.

6.3 Experiment Results

We conduct our experiments into two main parts, one is to verify if our method overcomes the limits of current repair algorithms and provides a repaired model like expected. This experiment is based on the synthetic data and models from Introduction chapter. Next, real life data is used to test our method.

[1] ProM`http://www.promtools.org`

6.3.1 Test on Demo Example

In this part, experiments are performed on the motivating examples which are listed in Introduction. Thereby, we are able to answer the question: Will our repair method overcome shortcomings of current techniques which are shown in the introduction chapter?

6.3.1.1 Answer to Situation 1

Situation 1 in Introduction shows the drawbacks of current repair methods [4, 5]. Additional activities **x1,x2** lead to good performance as shown in the actual event log L_1. Given the original Petri net M_0 and event log L_1, subprocesses for **x1,x2** are added as loops in the model, which brings more unexpected behaviors in the model. Rediscover strategy doesn't take the original model into account and generates a new model $M_{1.2}$ in Figure 1.2b that deviates from the original model M_0.

When we apply our repair techniques on all situations listed on Introduction, and get repaired models listed in Figure 6.1. The parameters for our method are set in the following : weight for the existing model is 0.45, weight for positive examples is 1.0, the Inductive Miner for Infrequent is chosen and has a noise with 20, the same setting as the rediscovery method by Inductive Miner in Introduction. As seen in Figure 6.1a, the subprocesses for **x1,x2** are added in a sequence with others and can be skipped by the silent transition. In this way, $M_{1.3}$ is able to reflect the deviations in positive instances while keeping similar to the reference model M_0. Compared to techniques in [5], it increases the precision without loops.

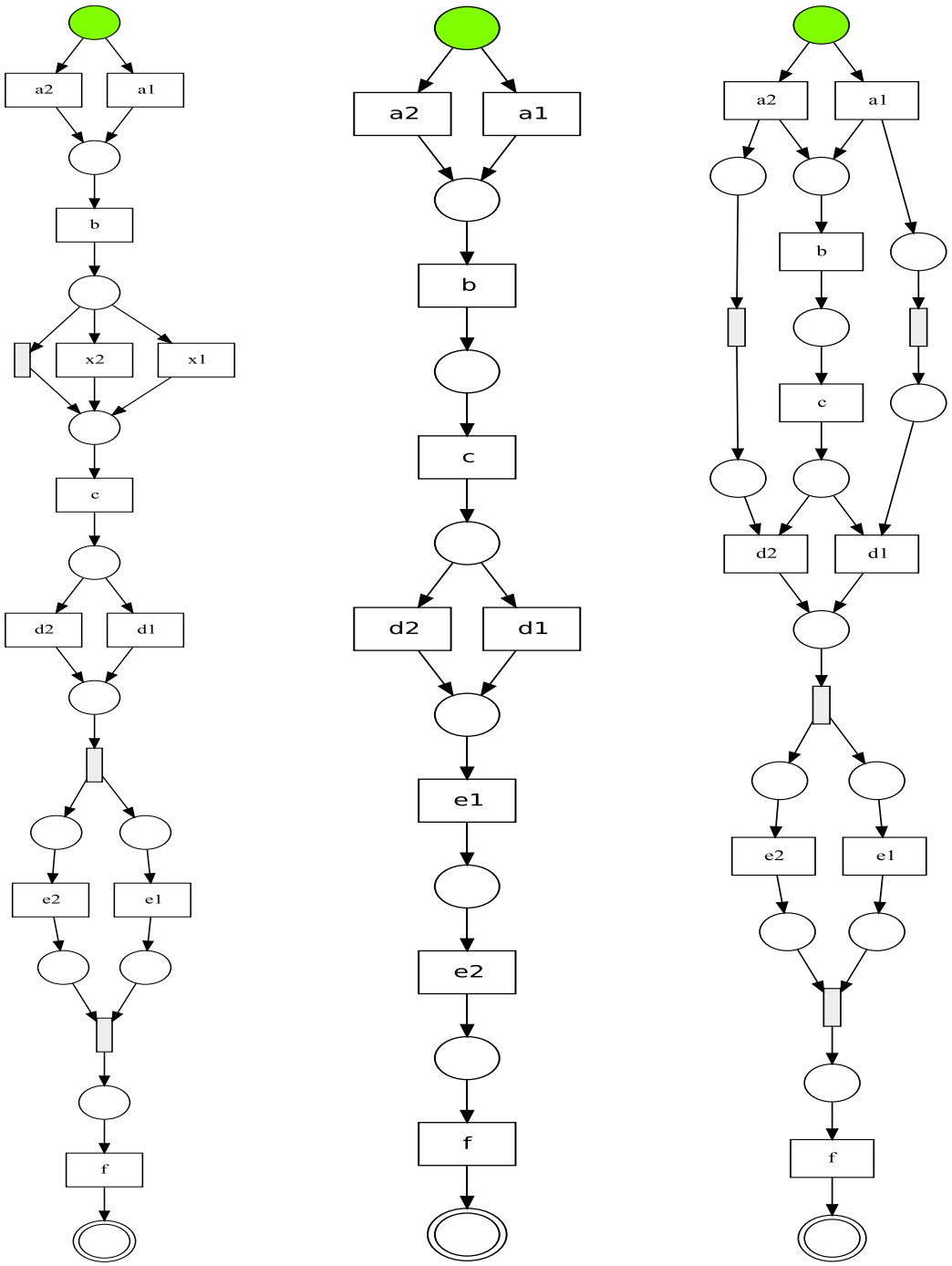
6.3.1.2 Answer to Situation 2

Situation 2 describes the inability of current repair methods that fitting traces with negative performance outcomes cannot be used to repair a model. The execution order of **e1, e2** affects the performance outcomes and **e1** is expected to position before **e2**. Without negative information, the repaired models have the same structure as the reference ones, because the execution of **e2** before **e1** brings also the positive outcomes.

If we apply our repair methods on the model M_0 and event log L_2, with setting 1.0 for all control weights, and the same Inductive Miner-Infrequent with noise 20, the repaired model $M_{2.3}$ is obtained. In $M_{2.3}$, **e1** is executed before **e2**. The reason behind our method is that it incorporates the negative information and balances the forces from the existing model, positive and negative instances.

6.3.1.3 Answer to Situation 3

Situation 3 concerns the long-term dependency in Petri nets. As observed in event log, there exists the long-term dependency set, $LT = \{a1 \rightsquigarrow d1, a2 \rightsquigarrow d2\}$. With adding long-term dependency as expected in Figure 1.3b, precision and accuracy increase, since the model limits activity selection and blocks the negative behavior due to free execution of xor branches. Yet none of the current repair and rediscovery techniques are able to detect and add long-term dependencies in the Petri net.



(a) $M_{1.3}$ for situation 1 (b) $M_{2.3}$ for situation 2 (c) $M_{3.3}$ for situation 3

Figure 6.1: repaired models with our techniques for situation 1,2 and 3 in Introduction part. The green place is the initial marking of the Petri net and the doubled place is the final marking.

In our repair techniques, the long-term dependency is taken into account with negative information. With inputs of the Petri net M_0 and event log L_3, our methods produces the repaired model $M_{3.3}$ with long-term dependency. Two silent transitions that are used to explicitly represent the long-term dependencies can be deleted with post procedure to reduce the redundant silent transitions and places. After reduction, our repaired model is simplified as the model M_3.

As conclusion, our proposed method is able to overcome shortcomings of current techniques which are shown in the Introduction chapter. It avoids the loops in model by repairing the model with additional activities, incorporates the negative information in the data to adjust the model, also detect and add the long-term dependency into model. In this way, the repaired model has better recall, accuracy.

6.3.2 Test On Real life Data

We choose a publicly available event log from BPI challenge 2015 as our user cases and compare current repair techniques on it.

6.3.2.1 Data Description

The data set for BPI Challenge 2015 contain 5 event logs which are provided by five Dutch municipalities respectively. Those event logs describe the building permit application around four years. We choose it as our user cases due to the following reasons.

- The event logs hold attributes as potential KPIs to classify traces. Attribute **SUMleges** which records the cost of the application is a candidate to label traces as positive or negative if its value is over one threshold. What's more, we can take the throughput time of the application as another potential KPI.
  In a word, this data set provides us information to reasonably label traces.
- The five event logs describe an identical process, but includes deviations caused by the different procedures, regulations in those municipalities. Also, the underlying processes have changes over four years.
  So, this data set gives us a basic process but also allows deviations of the actual event logs and predefined process, which builds the environment for repair techniques.

Firstly, we conduct our experiments on event log called **BPIC15_1.xes.xml**. This event log includes 1199 cases and 52217 events. But the event classes for those events are with the sum of 398. So we preprocess the event log and get a proper subset of data as our user case.

We filter the raw event log by ***Filter Log By Simple Heuristic*** in ProM with the following setting. 40 for the start, end activities and the events between them, at end. We get the event log $D1$. After this, we calculate the throughput time for each trace and add it as a trace attribute **throughput time**. Then we classify traces according to **SUMleges** and **throughput time** separately. When our performance goal is to reduce the cost of application, if **SUMleges** of one trace is over 0.7 of the whole traces, this trace is treated as negative, else as positive. The similar strategy is applied on the attribute **throughput time**. A trace with **throughput time** higher than 0.7 of all traces is considered as a

Table 6.2: Test event log from real life data BPI15-1

Data ID	Data Description	Traces Num	Events Num	Event Classes
D1	Heuristic filter with 40	495	9565	20
D2	Apply heuristic filter on D1 with 60	378	4566	12
D3.1	classify on SumLedges; values below 0.7 as positive	349	6744	20
D3.2	classify on SumLedges; values above 0.7 as negative	146	2811	20
D3.3	union of D3.1 and D3.2	495	9596	20
D4.1	classify on throughput time; values below 0.7 as positive	349	6744	20
D4.2	classify on throughput time; values above 0.7 as negative	146	2811	20
D4.3	union of D4.1 and D4.2	495	9596	20

negative instance. Following this preprocess, we have event logs in Table 6.2 available for our tests.

Table 6.3: Generated reference models for test

Model ID	**Used Data**	**Setting**	**Event Class**	**CM Evaluation**								
				Data	TP	FP	TN	FN	recall	precision	accuracy	F1
M1	D1	IM-infrequent: Noise Setting: 20	20	D3.3	112	40	106	237	0.321	0.737	0.440	0.447
				D4.3	131	21	128	215	0.379	0.862	0.523	0.526
M2	D1	IM-infrequent: Noise Setting: 50	20	D3.3	106	39	107	243	0.304	0.731	0.430	0.429
				D4.3	125	20	129	221	0.361	0.862	0.513	0.509
M3	D2	IM-infrequent: Noise Setting: 20	12	D3.3	0	0	146	349	0	NaN	0.295	0
				D4.3	0	0	149	346	0	NaN	0.301	0
M4	D2	IM-infrequent: Noise Setting: 50	12	D3.3	0	0	146	349	0	NaN	0.295	0
				D4.3	0	0	149	346	0	NaN	0.301	0

Based on the filtered data, we derive corresponding Petri nets as reference process models. The Table 6.3 lists the models with different setting. **IM-infrequent** is one variant of Inductive Miner working on event logs with infrequent traces. **Noise** is set as the threshold to filter out infrequent traces. After mining a reference model, we compare them with corresponding event logs to get the basis lines for later evaluation.

As seen in table above, the reference models don't apply well to the corresponding event logs. So changes on the models are in demand, to reflect better the reality and also to enforce the positive instances and avoid negative instances.

6.3.2.2 Test Result

We conduct experiments in the following types.

- **Type 1** Inductive Miner only on the positive event log to discover a model. The default setting with infrequent variant and noise threshold as 20 is chosen. Later, the mined model is checked on the labeled event with positive and negative instances. This method is abbreviated as IM.
- **Type 2** Repair Model from [5] is applied on the positive event log to discover a model. The default setting is chosen. Later, the mined model is checked on the labeled event with positive and negative instances. This method is abbreviated as Fahland, named after the name of main author.
- **Type 3** The method proposed from our thesis, is applied on the labeled event log with positive and negative instances. Default setting for the control parameters is 1.0 while the parameters to generate Petri nets from directly-follows graph are set as the same as experiment Type 1. Later, the repaired model is evaluated on the labeled data. This method is abbreviated as Dfg.

Those types are applied on pairs of event log groups of D3.1,D3.2,D3.3 and D4.1,D4.2,D4.3 in Table 6.2 and models from Table 6.3. The experiment result is shown in the Table 6.4.

Table 6.4: Test Result on BPI15-M1 data

event log	**reference model**	**method**	**confusion matrix metrics**							
			TP	**FP**	**TN**	**FN**	**recall**	**precision**	**accuracy**	**F1**
D3.1	M1,M2,M3,M4	IM	137	48	118	289	0.32	0.74	0.43	0.45
D3.1	M1	Fahland	343	136	10	6	0.983	0.716	0.713	0.829
D3.3	M1	dfg	124	52	94	225	0.355	0.705	0.44	0.472
D3.1	M2	Fahland	317	133	13	32	0.908	0.704	0.667	0.793
D3.3	M2	dfg	124	52	94	225	0.355	0.705	0.44	0.472
D3.1	M3	Fahland	349	145	1	0	1.0	0.706	0.707	0.828
D3.3	M3	dfg	0	0	349	146	0	NaN	0.295	0
D3.1	M4	Fahland	271	107	77	39	0.779	0.718	0.628	0.747
D3.3	M4	dfg	0	0	349	146	0	NaN	0.295	0
D4.1	M1	IM	131	21	128	215	0.379	0.862	0.523	0.526
D4.1	M1	Fahland	325	133	16	21	0.939	0.710	0.689	0.808
D4.3	M1	dfg	139	36	113	207	0.402	0.794	0.509	0.534
D4.1	M2	Fahland	325	130	19	21	0.939	0.714	0.695	0.811
D4.3	M2	dfg	139	36	113	207	0.402	0.794	0.509	0.534
D4.1	M3	Fahland	87	29	120	259	0.251	0.75	0.418	0.377
D4.3	M3	dfg	0	0	346	149	0	NaN	0.303	0
D4.1	M4	Fahland	63	20	129	283	0.182	0.759	0.388	0.294
D4.3	M4	dfg	0	0	346	149	0	NaN	0.303	0

With the similar measurements, it is observed that the repaired models from **Type 2** are more complex that Dfg method. One example is given for the tests of M1 on event log D4.1 for **Type 2** and M1 on event log D4.3.

Due to the different settings in our method, forces from the reference model, positive, and negative event logs are balanced differently during repair, which results in different process models. To investigate the effect of those setting on the repaired model, we repeat our experiments on the following setting.

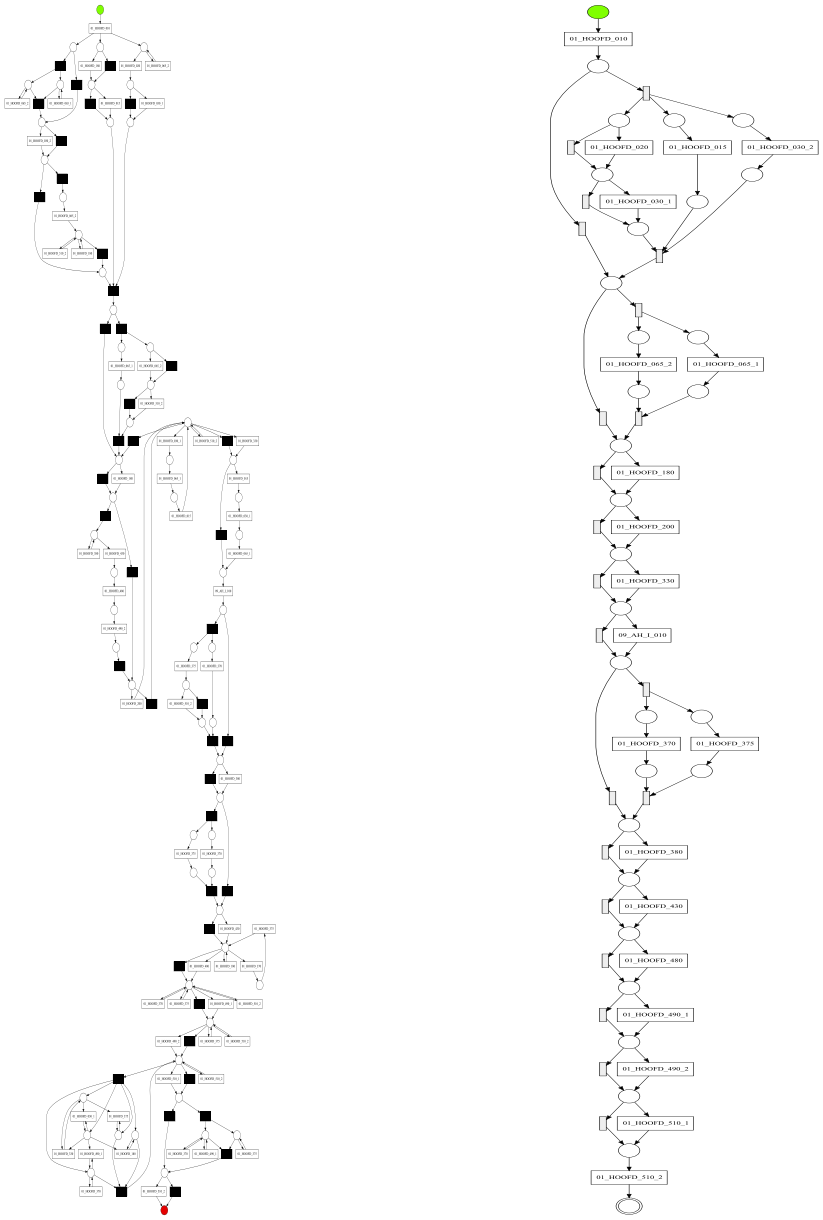
Each of three control parameters for the existing model, positive and negative instances changes value from 0.0 to 1.0 with step 0.1. With this setting, directly-follows relation is generated. Afterward, the default setting of Inductive Miner Infrequent with noise threshold 20 is used to mine Petri nets from the generated directly-follows graph.

So in total, we conduct over thousand experiments to investigate our methods. Based on those results, we draw plots to show the tendency of evaluation results on the parameters for the existing model, positive and negative event logs.

From the Figure 6.3, with the parameter for the existing model going up, recall goes up while accuracy and precision goes down. The reason behind it is possibly ????.

Figure 6.4 shows that if the parameter for negative event log increases, precision and accuracy go up. By addressing negative force, our techniques tend to block behavior which leads to low performance output. However, if the negative force is over the force from positive event log and the existing model, certain behavior which contributes to positive performance will also be deleted from the models. In contrast, this creates a model with less recall.

Figure 6.5 displays the tendency with the parameter for positive event log. When the positive parameter rises, the recall increases. Precision and accuracy also increases but with ???.



(a) repaired model with techniques in [5]

(b) repaired model with

Figure 6.2: example for situation 1 where $M_{1.1}$ is repaired by adding subprocess in the form of loops, which results in lower precision compared with the expected model $M_{1.2}$.

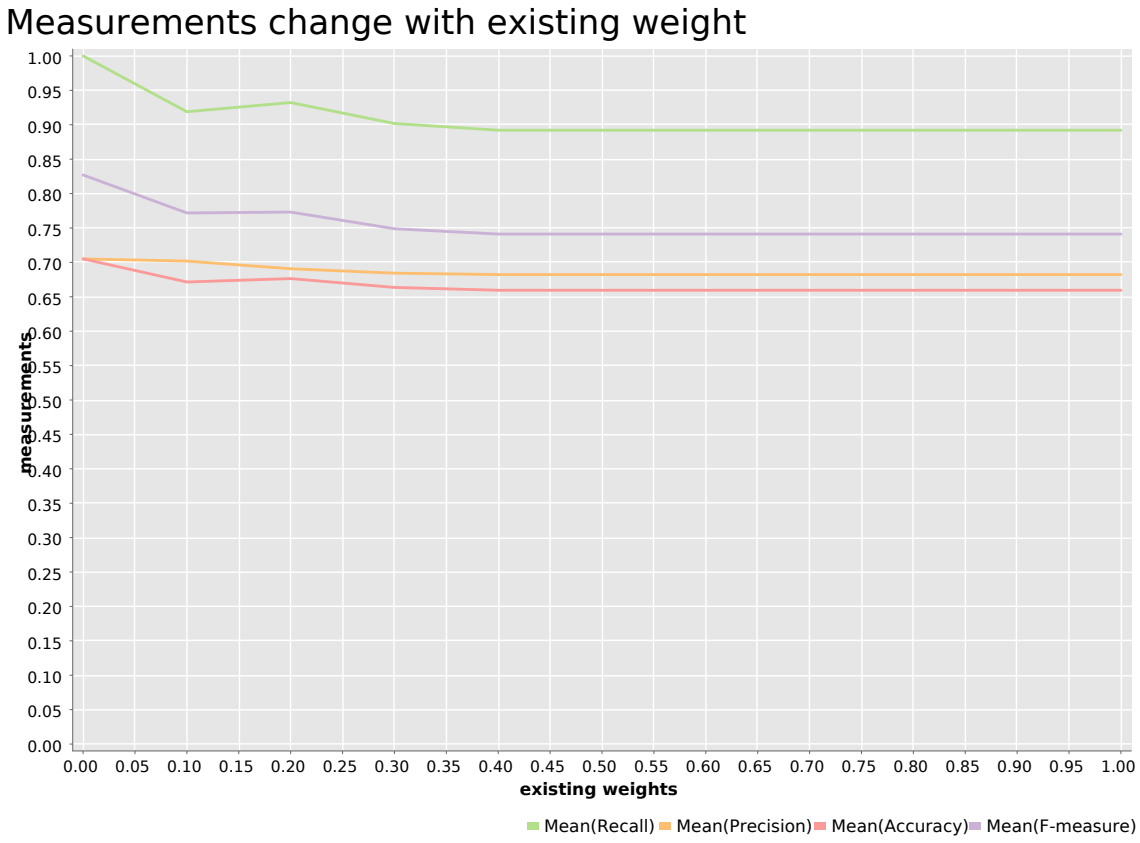


Figure 6.3: result with control parameter for existing model on event log D3.3 and model M3

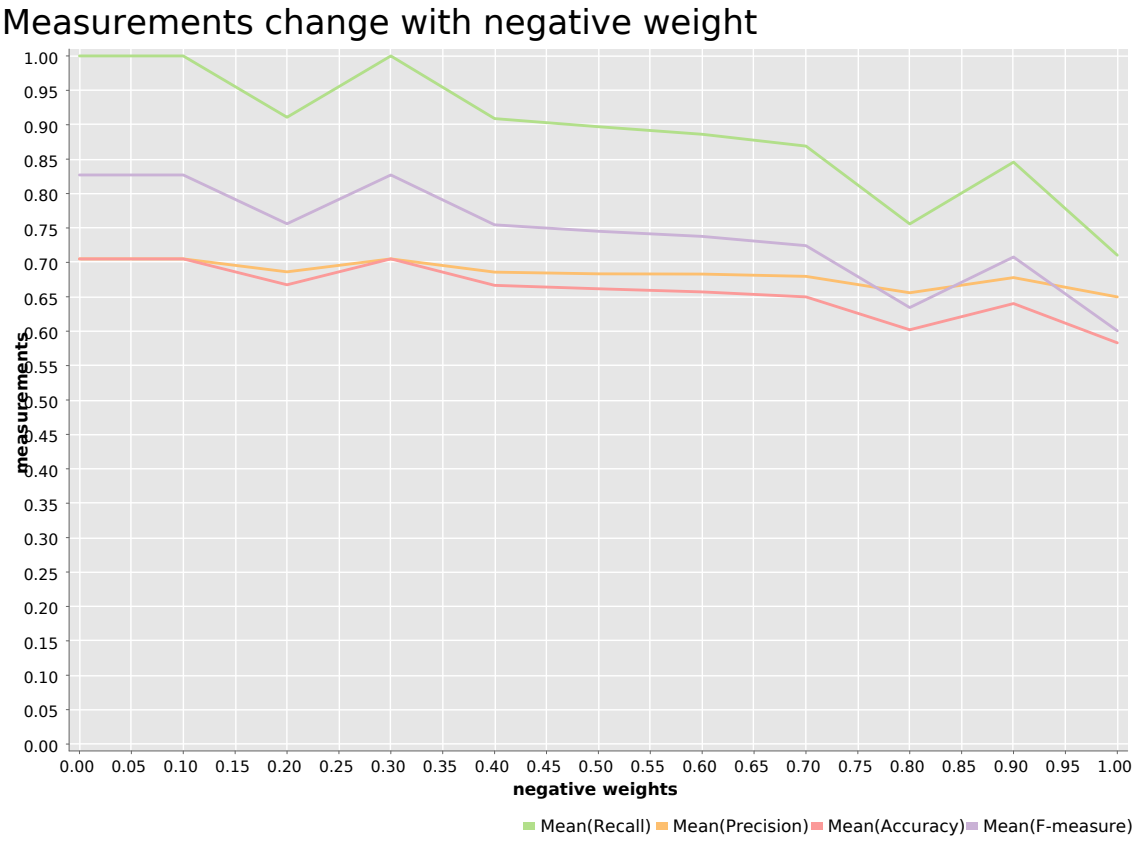


Figure 6.4: result with control parameter for negative instance on event log D3.3 and model M3

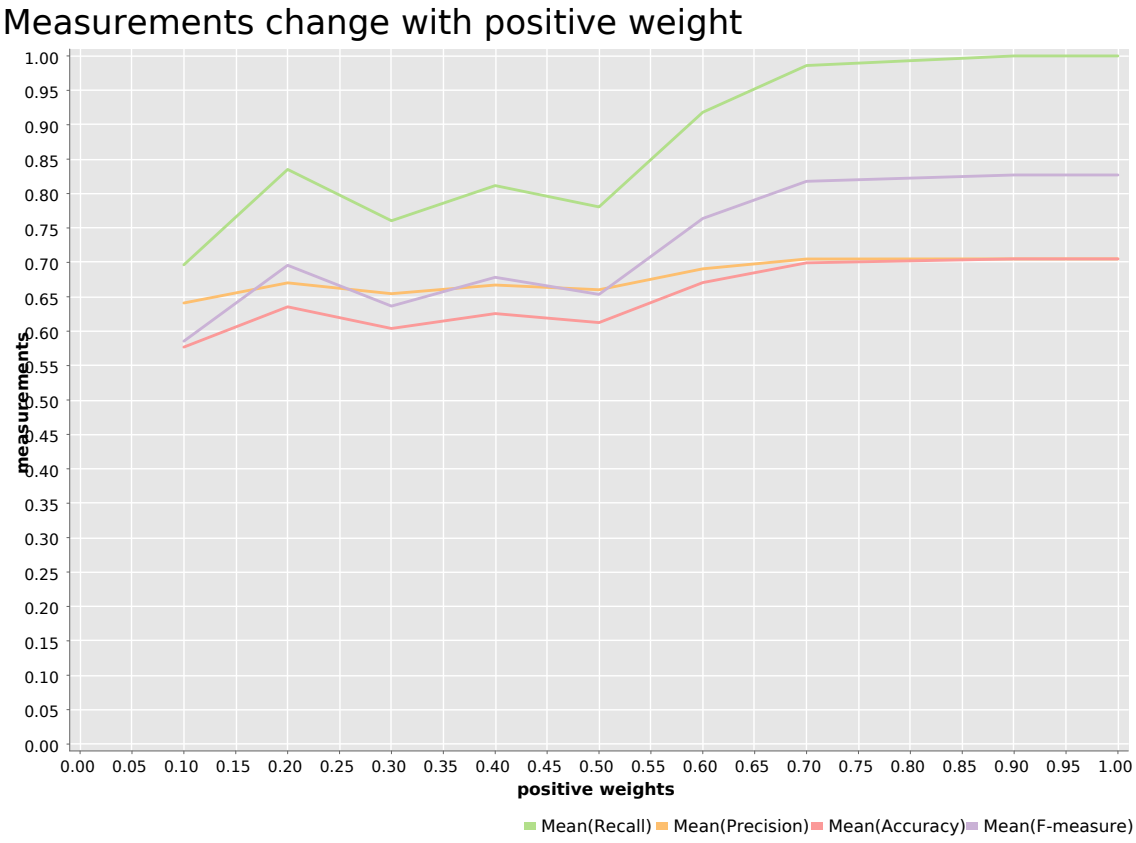


Figure 6.5: result with control parameter for positive instance on event log D3.3 and model M3

Chapter 7

Conclusion

In this thesis, we explore ways to use negative information in model repair and propose our innovative method. Firstly, we analyzed the current techniques on model repairs based on performance and detect their shortcomings. Then we proposed a general framework to incorporate the forces from the existing model, positive and negative event logs. Three abstraction data models in the same type are built to represent those forces. Later, forces are balanced based on data models and expressed in a new data model. From this new data model, process models are discovered and converted into repaired models with the required type. Optional post processes include long-term dependency detection and silent transition reduction, which further improves the repaired model.

Moreover, we demonstrate the usage of our method by conducting experiments with real life data. Our method is able to provide better repaired models than other repair methods in some situations. Also, with respect to other methods, it runs faster and generates simpler models.

Future work might be to improve the rules of balancing different forces, which choose the directly-follows relation on the simple subtraction and sum of those forces. Advanced data mining techniques such as association rules discovery, and Inductive Logistic Programming can be used on those forces to derive rules for building a process model. The same improvement can be applied on the long-term dependency discovery. Moreover, in this implementation, the long-term dependency discovery is restricted on the activities with exclusive choices relation. Later, we should extend the long-term discovery on other possible relations, like parallel relation. Also, we can drop the process tree as our intermediate result and adopt it directly on the Petri net.

Bibliography

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

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

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

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

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

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

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

[8] Sander JJ Leemans, Dirk Fahland, and Wil MP van der Aalst. Discovering block-structured process models from event logs-a constructive approach. In *International conference on applications and theory of Petri nets and concurrency*, pages 311–329. Springer, 2013.

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

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

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

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

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

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

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

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

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

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

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

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

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

[22] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[23] HMW Verbeek. Decomposed replay using hiding and reduction as abstraction. In *Transactions on Petri Nets and Other Models of Concurrency XII*, pages 166–186. Springer, 2017.

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

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