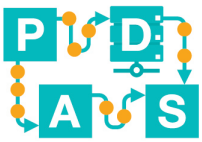


RWTH AACHEN UNIVERSITY

Model Repair by Incorporating Negative Instances In Process Enhancement

Master Thesis

Author : **Kefang Ding**

Supervisor : Dr. Sebastiaan J. van Zelst

Examiners : Prof. Wil M.P. van der Aalst
Prof. Thomas Rose

Registration date : 2018-11-15

Submission date : 2019-04-08

This work is submitted to the institute

PADS RWTH University

Acknowledgments

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

Abstract

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

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

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

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

Contents

Aknowledgement iii

Abstract v

1 Introduction 1
- 1.1 Motivating Examples . . . 2
  - 1.1.1 Situation 1: Repairing Model with Unfitting Traces . . . 3
  - 1.1.2 Situation 2: Repairing A Model with Fitting Traces . . . 3
  - 1.1.3 Situation 3: detect long-term dependency . . . 6
- 1.2 Research Scope And Questions . . . 6
- 1.3 Outline . . . 7

2 Related Work 9

3 Preliminaries 1
- 3.1 Event Log . . . 1
- 3.2 Process Models . . . 2
  - 3.2.1 Petri Nets . . . 2
  - 3.2.2 Transition System . . . 3
  - 3.2.3 Process Tree . . . 4
- 3.3 Inductive Miner . . . 5
  - 3.3.1 Construct a Directly-Follows Graph . . . 5
  - 3.3.2 Split Log Into Sublogs . . . 6

4 Algorithm 9
- 4.1 General Framework . . . 9
- 4.2 Algorithm . . . 9
  - 4.2.1 Unified Data Model . . . 11
  - 4.2.2 Modules List . . . 11
  - 4.2.3 Convert event logs into unified directly-follows graphs, D^+, D^- . . . 12
  - 4.2.4 Convert process model into unified directly-follows graph D^M . . . 12
  - 4.2.5 Incorporate unified directly-follows graphs . . . 12
  - 4.2.6 Generate process models from D^n . . . 14
  - 4.2.7 Post process on the process model . . . 15
  - 4.2.8 Reduce Silent Transitions . . . 20
  - 4.2.9 Concrete Architecture . . . 21

5 Implementation **23**
5.1 Implementation Platforms . . . 23
5.1.1 Process Mining Platform – ProM . . . 23
5.1.2 KNIME . . . 23
5.2 Generate A Process Model . . . 23
5.3 Post Process to Add Long-term Dependency . . . 25
5.4 Post Process to Reduce Redundant Silent Transitions and Places . . . 26
5.5 Additional Feature to Show Evaluation Result . . . 26
5.6 Integration into KNIME . . . 28

6 Evaluation **29**
6.1 Evaluation Measurements . . . 29
6.2 Experiment Platform . . . 31
6.2.1 PLG . . . 31
6.2.2 KNIME . . . 31
6.2.3 ProM Evaluation Plugins . . . 31
6.3 Experiment Result . . . 31
6.3.1 Test On Property . . . 31
6.3.2 Comparison To Other Techniques . . . 33
6.3.3 Test On Synthetic Data . . . 33
6.3.4 Test On Real life Data . . . 33

7 Conclusion **35**

Bibliography **38**

List of Figures

1.1 original master study process M_0 2
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 $M_{1.2}$. 4
1.3 example for situation 2 and 3 5
1.4 The problem description 7

3.2 Translation of process tree operators to Petri net 4
3.3 Inductive Miner to discover a process tree from event log L_{IM} 6

4.1 Model Repair Architecture 10
4.2 One Petri net with two two xor blocks 16
4.3 Model with long-term dependency $LT = \{A \rightsquigarrow D, A \rightsquigarrow E, B \rightsquigarrow E\}$. 19
4.5 Model Repair Architecture 22

5.1 Inductive Miner Parameter Setting 24
5.2 Generated Petri net without long-term dependency 24
5.3 Petri Net with long-term dependency 25
5.4 Petri net with selected long-term dependency 26
5.5 Petri net after reducing the silent transitions 27
5.6 Generated Process Tree Model 27
5.7 Integration of our repair techniques into KNIME 28

6.1 Model M1 with sequence relation 32
6.2 example for model change under model repair 34

List of Tables

6.1 Confusion Matrix 30

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 model with additional data perspectives or repairing the existing model to accurately reflect observed behaviors.

Most of the organizations have predefined process execution rules which are captured in a process model. However, in real life, business processes often encounter exceptional situations where it is necessary to execute process differing from the reference model. To reflect reality, the organizations need to adjust the existing process model. Basically, one can apply process discovery techniques again to obtain a new model only based on the event log. 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 fix the deviations mainly by adding subprocesses on the model. As known, organizations are goal-oriented and aims to have high performance according to a set of Key Performance Indicator(KPI)s,e.g. average conversion time for the sales, payment error rate for the finance. However, little research in process mining is conducted on the basis of business performance[3]. The authors of [3] point out several contributions like [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 [5] only with positive deviations, 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 well the event log but overgeneralizes the model, such that it allows more behaviors than expected. Furthermore, it increases the model complexity. Even the performance is considered in

[4], but only deviations in positive is used to add subprocesses, the negative information is ignored, which disables the possibility to block negative behaviors from model.

In the following part, 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

This section describes some situations where current repair techniques tend to fail. For the sake of understanding, examples are extracted from the common master study procedure to illustrate those situations.

The main activities for the master study include *register master*, *finish courses* and *write a master thesis*. Here, we simplify the *finish courses* and only extend the activity *write a master thesis* into a set of sub activities. Those activities are shown in the Petri net model M_0 of Figure 1.1. The activities are modeled by the corresponding ***transitions*** which is represented by a square. Transitions are connected through a circle called ***place***. Transitions and places build the static structure of Petri net and describe the transition relations. ***Tokens*** in the black dot are put in the initial places and represent the dynamic state of the model.

M_0 is currently in an initial state where only one token is at the start place to enable the transition *register master*. After firing *register master*, the token at the initial place is consumed while two new token are generated in the output places of *register master*. In this way, activity *finish courses* can be executed concurrently the other branch except for the *get degree*. When multiple activities have the same input place, all of them are enabled but only one of them can be fired and executed, namely, they are exclusive to each other. As shown in the figure, *select existing topics* and *create new topics* are exclusive, and only one of them can be triggered. When a transition has multiple input places, it can be triggered with condition that all input places hold at least a token. *Get degree* is enabled only after *finish courses* and *representation* done.

Along the tokens flowing through the model, activities get fired and generate a sequence according to their execution order. One execution sequence is called a trace. A set of traces depicts the model behavior and is recorded into a data file called event log. In real life, activities might be executed with deviation to the process model. A trace which has no deviation to the model is fitting. Otherwise, it's a unfitting trace. With accumulation of deviations, process model needs

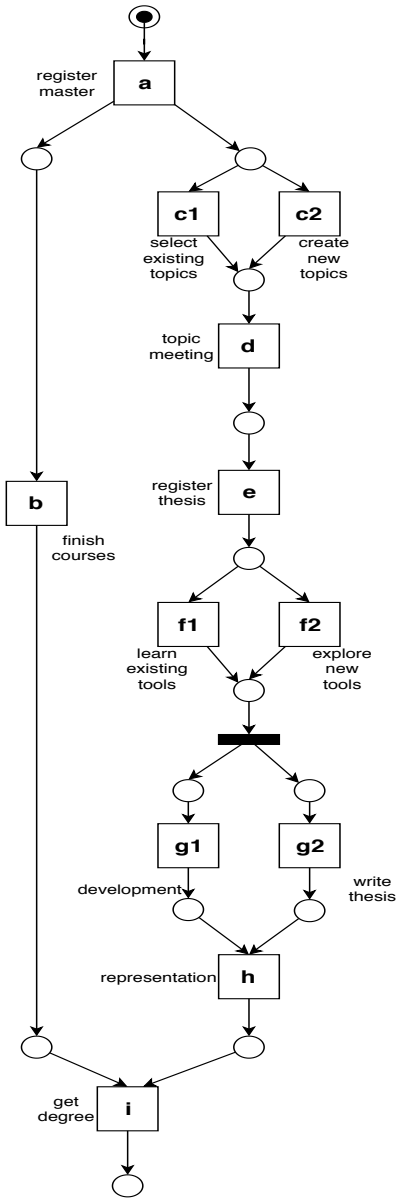


Figure 1.1: original master study process M_0

amending to reflect reality. In the following part, different
several situations are introduced to demonstrate the shortcomings of current techniques to repair a model.

1.1.1 Situation 1: Repairing Model with Unfitting Traces

In some universities, before registering a master thesis, the activities *write proposal* and *check course requirement* with exclusive choice relation 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. For convenience, alphabet characters are used to represent the corresponding activities and annotated in the model. **x1, x2** represent the activities *write proposal* and *check course requirement*. *Event Log* L_1 –

$$\begin{aligned} Positive : \{ &< a, b, c1, d, \mathbf{x1}, f, g1, g2, h, i >^{50}, \\ &< a, b, c2, d, \mathbf{x2}, f, g2, g1, h, i >^{50} \} \end{aligned}$$

Because the existing repair techniques [5] don't consider the performance of traces in event log, all instances with positive labels are used to repair the model. Firstly, the deviations of the existing model M0 and the event log L_1 are computed. After computation of deviations, each deviation has the same start and end place and two deviations appear at the same position in the model. When repairing this model, each subprocess has one place as its start and end place, which forms a loop in the model. If there is only one such subprocess, the subprocess is added in a sequence in the model, which leads to a higher precision. Yet the algorithm does not discover orderings between different subprocesses at overlapping locations. So the subprocesses are kept in a loop form.

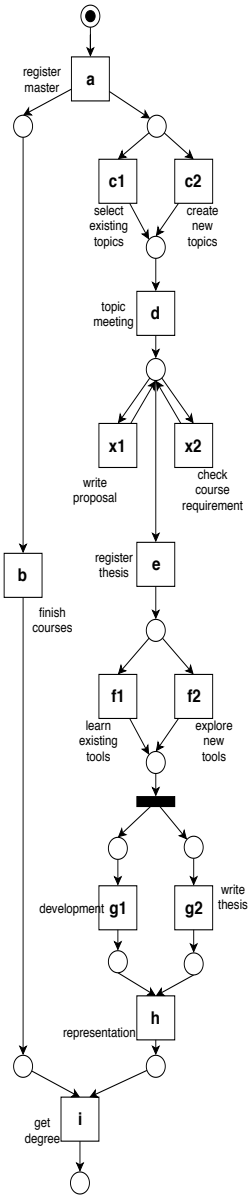
The repaired model is shown in Figure 1.2a, where the two additional activities are added in the form of loop. The repair algorithm in [4] builds upon [5] and considers the performance of the event log. However, the repaired model is the same as the one in Figure 1.2a. The reasons are: (1) there is no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

Compared to the model in Figure 1.2a where the two extra activities are shown in loop, the model in Figure 1.2b are more expected, since it includes the two activities in sequence and has a higher precision.

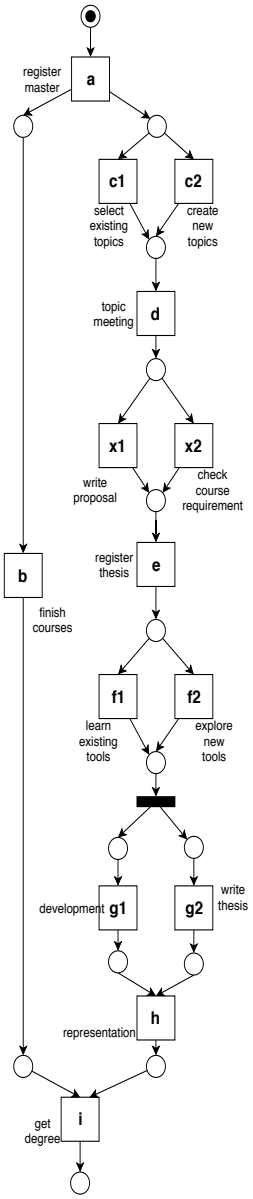
1.1.2 Situation 2: Repairing A Model with Fitting Traces

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, when activity *finish courses* is fired after *begin thesis* and before writing master thesis, it reduces the pressure for the master thesis phase and traces in such an order are treated as positive. Else, the negative outcomes are given. *Event Log* L_2 –

$$\begin{aligned} Positive : \{ &< a, \mathbf{b}, c1, d, f, g2, g1, h, i >^{50}, \\ &< a, \mathbf{b}, c2, d, f, g1, g2, h >^{50} \} \\ Negative : \{ &< a, c1, d, f, g2, g1, \mathbf{b}, h, i >^{50}, \\ &< a, c1, \mathbf{b}, d, f, g1, g2, h, i >^{50}, \} \end{aligned}$$



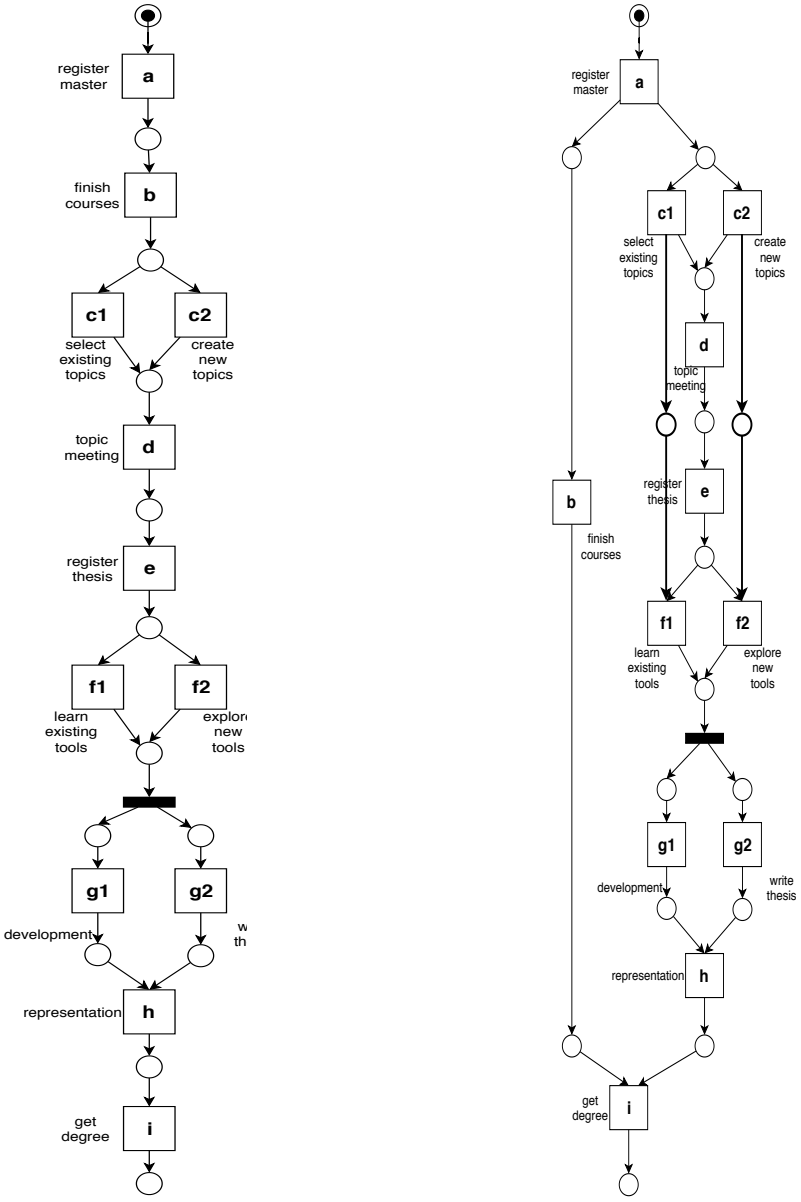
(a) repaired model $M_{1.1}$ with additional activitiesties



(b) expected model $M_{1.2}$ with additional activi-

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 $M_{1.2}$.

Compared to the model, the event log L_2 contains no deviations. When we apply the techniques in [5] and [4] to repair the model, the model keeps untouched due to no deviation. Apparently, the reason that those two methods can't incorporate the negative information in fit traces causes this shortcoming. When we expect a model which enforce the positive instances and avoid the negative instance as the model M_2, the current methods don't allow us to obtain such results.



(a) expected model M_2 with order change (b) model M_3 with long-term dependency

Figure 1.3: example for situation 2 and 3

1.1.3 Situation 3: detect long-term dependency

This part introduces a problem which causes a lower precision in process mining. It is the inability in current methods to detect the long-term dependency in the Petri net. The long-term dependency describes the phenomenon that one execution choice 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. An event log L_3 is given in the following. By using time consumption as one KPI, if the total sum goes over one threshold, we mark this trace as negative, else as positive. Since the activity *create new topics* usually demands new knowledge rather than checking the existing tools. So if students choose to learn existing tools, it's possibly not useful and time is wasted. 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. *Event Log* L_3 –

$$\begin{aligned} Positive : \{ &< a, b, \mathbf{c1}, d, e, \mathbf{f1}, g1, g2, h, i >^{50}, \\ &< a, b, \mathbf{c2}, d, e, \mathbf{f2}, g2, g1, h, i >^{50} \} \\ Negative : \{ &< a, b, \mathbf{c1}, d, e, \mathbf{f2}, g2, g1, h, i >^{50}, \\ &< a, b, \mathbf{c2}, d, e, \mathbf{f1}, g1, g2, h, i >^{50} \} \end{aligned}$$

There are no deviations of the model and event log L_3 according to the algorithms in [5] and [4]. Therefore, the original model stays the same and allows for the execution of negative instances. After checking the model and log, those long-term dependencies have significant evidence. Transition ***c1*** decides ***f1*** while ***c2*** decides ***f2***. After addressing long-term dependency like the model M_3 in Figure 1.3b by connecting transitions to extra places, negative instances are blocked and the model has higher precision.

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

1.2 Research 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?

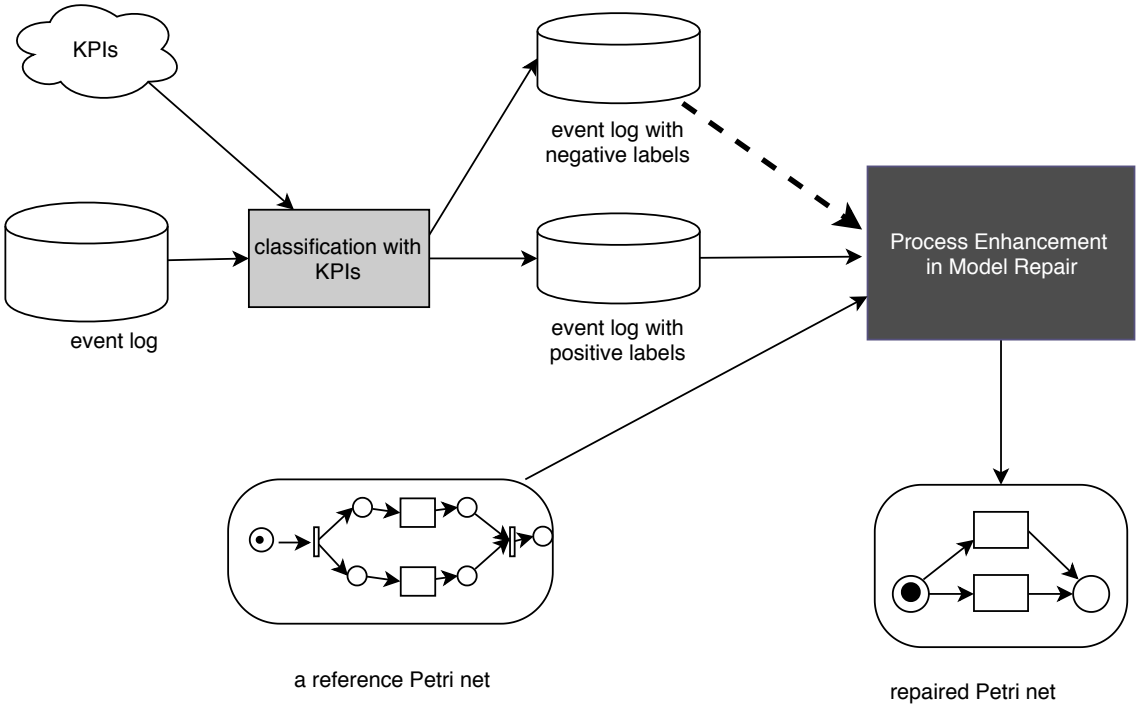


Figure 1.4: The problem description

In the remainder, 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, in order to incorporate all the factors on model generation.

1.3 Outline

This thesis tries to answer the questions presented in section 1.2 in the remainder chapters and provide a solution for the black box. Chapter 2 and 3 introduces the related work and recalls the basic notions on process mining and list the preliminary to solve the problem. Chapter 4 describes the algorithm to solve the problem. In Implementation part, screenshots are given from the finished tools of our algorithm to demonstrate the use. Experiment chapter answers the last question RQ3, by conducting a bundle of experiments. Later, results are analyzed and discussed. The last chapter is the summary of our work.

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[7, 8] methods investigate the activities order in the traces and extracts higher relations which are used to build process models. State-based methods like [9, 10] builds a transition system to describe the event log, and then group the state regions into corresponding Petri net node. Language-based algorithms uses integer linear system to represent the place constraint where the token at one place can never go negative. By solving the system, a petri net is created. Its representative techniques are Integer Linear Programming(ILP) Miner[11]. Other methods due to [12] include search-based algorithm like Genetic Algorithm Miner[13], heuristic-based algorithm 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. A process tree is output as the mined process model and can be converted into another process model called 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 model.

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 only are 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 the number of each log move with KPI output. Based on the observation instance, a set of rules are derived in the form of a decision tree. According to the rules, the original event log is divided into sublogs with traces matching the rules. The sublogs are then repaired to contain only trace deviations which have a positive KPI output. Following repair, the sublogs are merged as the input for model repair in [5]. According to the study case in [4], it 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 events set before and after one position and represented in the form of the complement of positive event sets. Based on the positive and negative event sets, Inductive Logic Programming is applied to detect the preconditions for each activity. Those preconditions are then converted to Petri net after applying a pruning and post-process step. Similar work on model discovery based on artificial negative events are published later. In [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. The execution of a trace reflects the token transmission and marking limits on places in the model. Those limits are represented into a set of marking inequalities and in a form of convex polyhedron in n-dimensional space. Given half-space hypotheses, SMT solves the inequalities and gives the limits on the process model. Before SMT, negative information is incorporated to shift and rotate the polyhedron, which limits the generalization of the solution space. Because half-space is used, this method can not deal with negative instances overlapped into positive instances.

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

Chapter 3

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 which relate to our work are listed.

3.1 Event Log

Business processes in organizations can be reflected by their activities execution. 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 the various notations[6] .

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

Definition 3.2 (Trace)**.** A trace is a finite sequence of events $\sigma \in \mathcal{E}^*$ with conditions that (i) each event appears only once in a trace.

$$\forall 1 \leq i, j \leq |\sigma|, i \neq j \Rightarrow \sigma(i) \neq \sigma(j)$$

(ii) one event can only appear in one trace.

$$\forall e \in \sigma, e \in \sigma\prime \Rightarrow \sigma = \sigma\prime$$

A trace also has a set of attributes, like its unique identifier, the cost. We extend this definition to handle traces with performance output according to certain KPIs.

Definition 3.3 (Labeled Trace)**.** A trace is labeled with respect to certain KPIs, if it has an attribute for the performance output. We call a trace positive, if the value of its performance attribute is positive, else the trace is negative.

Definition 3.4 (Event log and labeled Event log)**.** An event log L is a set of traces, $L \in \mathcal{B}(\mathcal{E}^*)$. A labeled event log is an event log if all of its traces have an performance attribute 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, aims to improve the understanding of the business process. To describe the process, multiple process modeling languages are proposed in the last years, e.g, Petri net, BPMN models, etc.

Among those model languages, Petri net has been best studied thoroughly. It captures 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, we use Petri net and process tree to represent our process.

3.2.1 Petri Nets

Petri nets are bipartite graphs which are built by ***transitions*** represented by a square and ***place*** in a circle. ***Tokens*** in black dots are put in places to express the states of a Petri net. In the following, we define Petri nets in a formal way.

Definition 3.5 (Petri net)**.** A Petri net is a tuple $N = (P, T, F)$ where
P and T are disjoint finite set of places and transitions, respectively, $P \cap T = \emptyset$.
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 that particular transitions 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 formally represented in the following way.

Definition 3.6 (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 3.7 (Marked Petri nets)**.** A marked Petri net is a 4-tuple $N = (P, T, F, M)$ where $M \in mathbbB(P)$ 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 3.8 (Input and output nodes)**.** For a node x in Petri net $N = (P, T, F)$, its input nodes set is $\bullet x = \{y | (y, x) \in F\}$. Its output nodes set is $x\bullet = \{y | (x, y) \in F\}$.

Definition 3.9 (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 $\bullet t \leq M$. 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 3.10 (Workflow nets)**.** A workflow net is a labeled Petri net $N = P, T, F, \lambda$ with constraints:

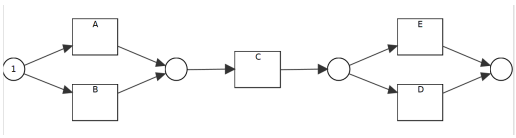
- P contains a source place $i, \bullet i = \emptyset$.
- P contains one sink place $o, o\bullet = \emptyset$.
- When connecting the sink place to source place by an arc, this net becomes strongly connected.

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].

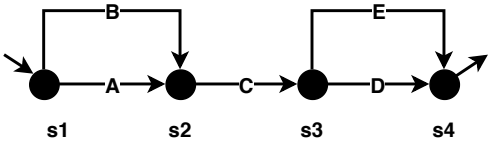
Definition 3.11 (Soundness)**.** A Petri net is sound if and only if it satisfies the following conditions.

- 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 just for the sink place.
- no dead part. For any transition there is a path from source to sink place through it.

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

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 an activity in the process and connect states as an arc.

Definition 3.12 (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 **??** are mapped to each other. There are 5 transitions and four states in the transition system, which corresponds to the transitions and places in Petri net. Transition systems are simple but have difficulty to express concurrency in business process. So in our thesis, we use more expressive models like Petri nets to represent our process mining result.

3.2.3 Process Tree

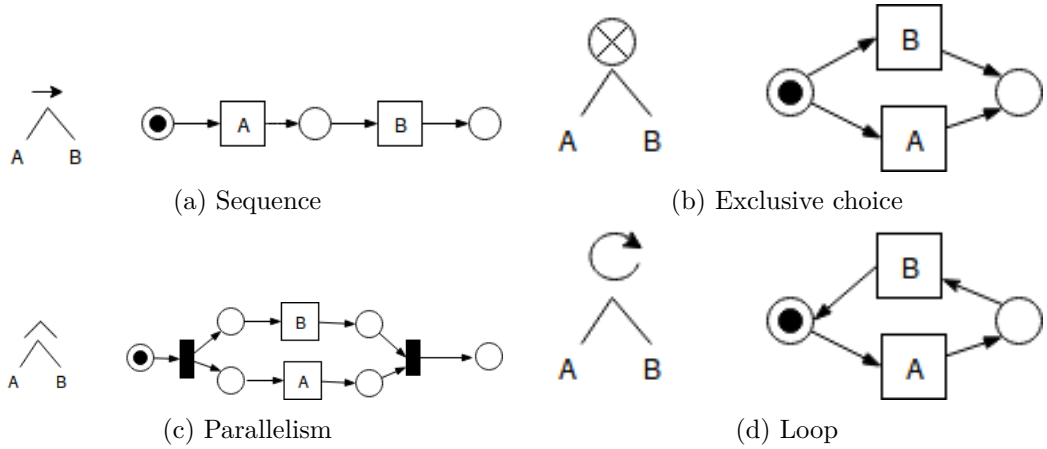
Process tree is a block-structured tree and sound by construction[6]. Also, due to the simplicity to understand, we use it in our thesis and give the formal definition for it.

Definition 3.13 (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, $i \in \mathbb{N}$.

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

- if $Q = (a)$, the tree only has one leaf node. With respect to a trace $\sigma \in L$, Q is executed in σ, denoted as $\sigma \models Q$, if $\exists i, 1 \leq i \leq |\sigma| \sigma(i) = a$.
- if $Q = \rightarrow (Q_1, Q_2, ..Q_n)$, the subtrees have a sequential relation and are executed in order of $Q_1, Q_2, ..Q_n$. For a trace σ,$\sigma \models Q$, iff $\forall Q_i, \sigma \models Q_i$ in order.
- 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. For a trace σ,$\sigma \models Q$, if $\exists Q_i, \sigma \models Q_i$.
- 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. For a trace σ,$\sigma \models Q$, if $\forall Q_i, \sigma \models Q_i$.

- 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. For a trace σ,$\sigma \models Q$, if $\sigma \models Q_1$ and $\exists Q_i, \sigma \models Q_i \Rightarrow \sigma \models Q_1$ after Q_i.

According to the corresponding semantic relations, a process tree can be easily transformed into Petri net. In Figure **??**, 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 3.15 (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 3.16 (Directly-follows Graph)**.** The directly-follows relation $a >_L b$ is satisfied iff

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

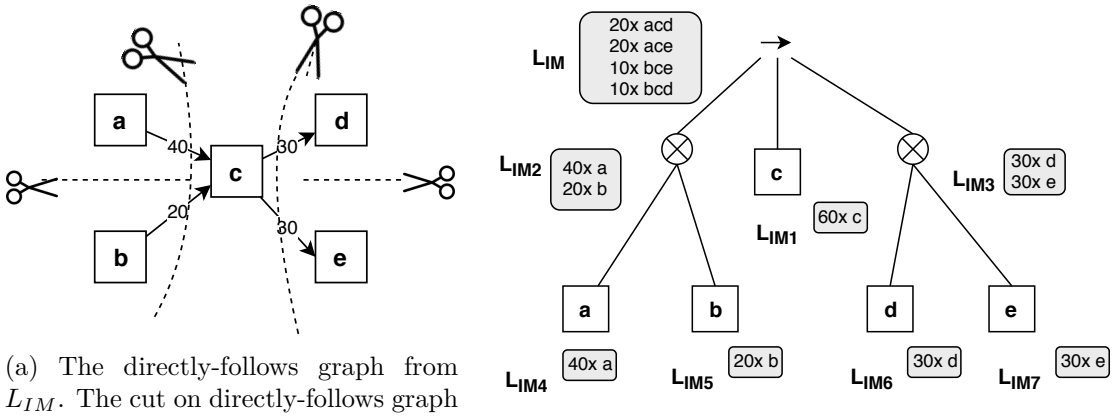
A directly-follows graph of an event log L is $G(L) = (A, F, A_{start}, A_{end})$ where A is the set of activities in L, $F = \{(a,b) \in A \times A | a >_L b\}$ is the directly-follows relation set, A_{start}, A_{end} are the set of start and end activities respectively, $A_{start} = \{a | \exists \sigma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

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

Definition 3.17 (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 as a function $c : F \to N$:

- $c(a,b) = \sum_{\sigma \in L} \{i \in \{1, 2...|\sigma|\} | \sigma(i) = \sigma(i+1)\}$ 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. Cuts compose of *exclusive-choice cut, sequence cut, parallel cut and redo-loop cut* which correspond to the process tree operators $\{\rightarrow, \times, \wedge, \circlearrowleft\}$. They are selected in the following order. A maximal exclusive-choice cut is firstly tried to split the directly-follows graph; if it is not available, then a maximal sequence cut, a maximal parallel cut and a redo-loop cut are applied in sequence. Sublogs are created due to this available operator. Meanwhile, this operator is

used to build the process tree. The same procedure is applied again on the sublogs until single activities.

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 activities 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 model by incorporating the negative instances. In subsequent sections, we propose a concrete solution to implement the modules in the framework.

4.1 General Framework

Figure 4.5 shows our proposed framework to repair a process model with negative information. The inputs are a reference process model and an labeled event log. The reference process model can be in multiple types, like 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 to repair model which also considers the negative information is able to find.

4.2 Algorithm

Given the inputs as shown in the problem scope Figure 1.4, an event log and a Petri net as the reference process model M, we repair the Petri net with actual data in the event log and output the repaired Petri net.

Based on those input and the condition with output, we give our algorithm to implement the modules to repair the reference Petri net. First of all, we define a proper data model to 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.

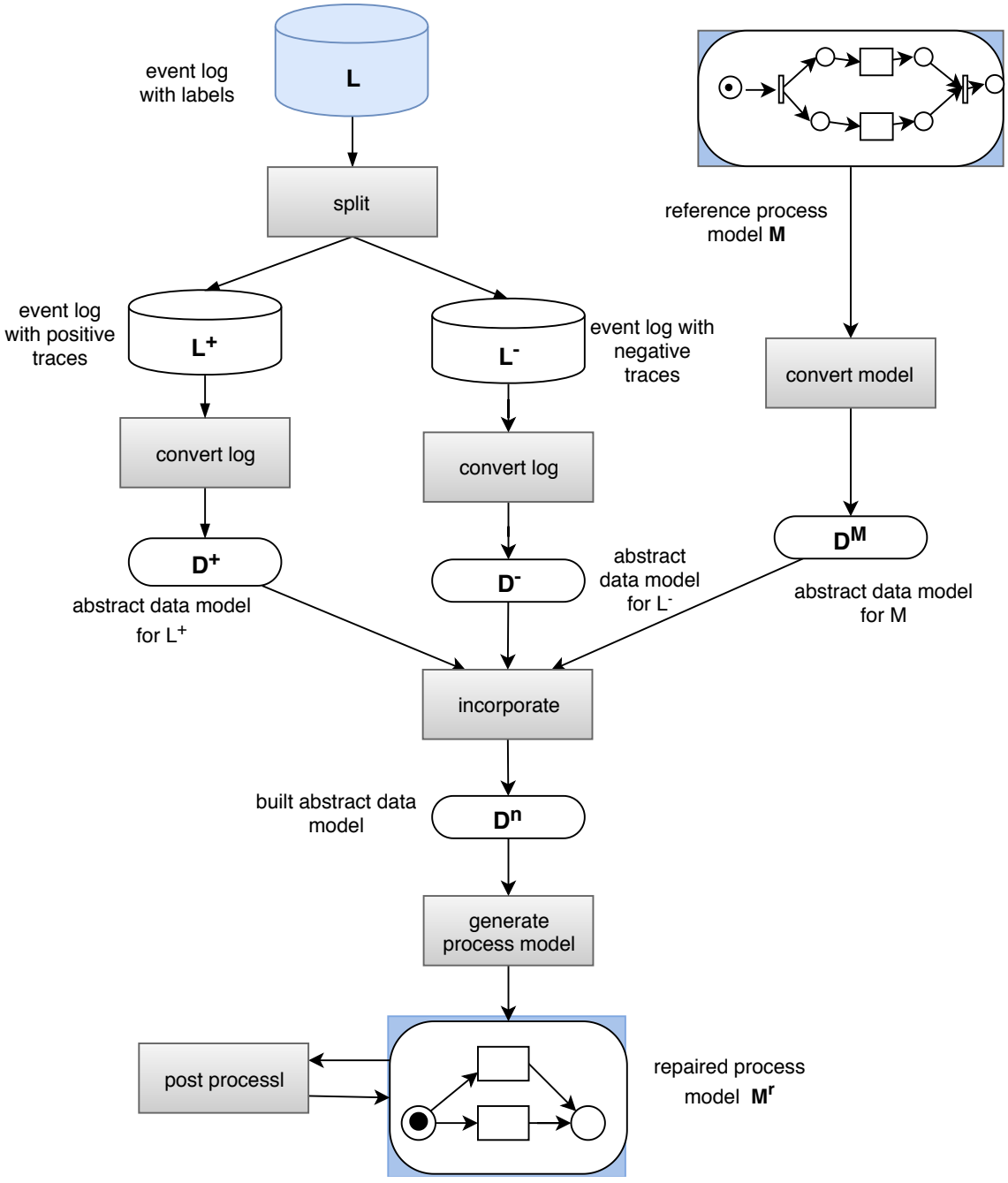


Figure 4.1: Model Repair Architecture – Rectangles represents processes and data models in eclipse shape. Input event log and the reference model are in blue. Output in blue is a repaired model with the same type as the reference model.

4.2.1 Unified Data Model

We choose 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) directly-follows relation for building a directly-follows graph can be obtained from an event log and a reference model;(2) the cardinality of directly-follows graphs can be used to express the impact strength; (3) 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.

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 disable 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 process 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 process model* Several post-processes on the generated process model are optional to conduct, in order to improve the process model quality according to certain criteria.

In those modules, due to the simplicity, we skip the detail 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, D^+, D^-

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 process model into unified directly-follows graph D^M

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 gather the model behaviors which are expressed in a transition system. By checking 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, impact from existing model, positive and negative instances is in the same level and separately represented in D^M,D^+, and D^-. To repair a reference model w.r.t. an actual event log with labels based on directly-follows relation, the strategy is that we add directly-follow relations into the repaired data model D^n if the total support from D^M and D^+ exceeds the rejection force from D^-; else, we reject directly-follows relations into the repaired model D^n. In other words, 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 weight 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. The algorithm is listed in Algorithm ??.

Algorithm 1: Build data model D^n after incorporating D^M,D^+, and D^-

Result: The repaired data model D^n

$D^n = (A = \emptyset, F = \emptyset, A_{start} = \emptyset, A_{end} = \emptyset, u = \emptyset)$;
Given a threshold $t \in [0, 1]$, the weighted unified cardinality set from D^M,D^+, and D^-, $U^n : F^M \cup F^+ \cup F^- \to \mathbb{R}$;
for *Each weighted unified cardinality* $u^n(a, b)$ **do**
    if $u^n(a, b) > t$ **then**
        if $u^n > 1.0$ **then**
            $u^n = 1.0$
        end
        $A = A \cup \{a\} \cup \{b\}$;
        $F = F \cup \{(a, b)\}$;
        $u = u \cup \{(a, b) \to u^n\}$;
    end
end
for *Each weighted unified cardinality for the start activity* $u^n(a)$ **do**
    if $u^n(a) > t$ **then**
        if $u^n > 1.0$ **then**
            $u^n = 1.0$
        end
        $A = A \cup \{a\}$;
        $A_{start} = A_{start} \cup \{a\}$;
        $u = u \cup \{(a) \to u^n\}$;
    end
end
for *Each weighted unified cardinality for the end activity* $u^n(a)$ **do**
    if $u^n(a) > t$ **then**
        if $u^n > 1.0$ **then**
            $u^n = 1.0$
        end
        $A = A \cup \{a\}$;
        $A_{end} = A_{end} \cup \{a\}$;
        $u = u \cup \{(a) \to u^n\}$;
    end
end

4.2.6 Generate process models from D^n

The result from 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 know all of them are in range [0,1]. We transform the weighted cardinality by multiplying the sum of cardinalities of $G(L^+)$, $G(L^-)$ and $G(L^M)$.

Later, based on G^n, an existing procedure called *Dfg2ProcessTree* is applied to mine process models like process tree and Petri net. This procedure was introduced with Inductive Miner[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 op-

erator 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 process 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 make the generated model preciser, as one post process, we detect the long-term dependency and add it on the process model. What's more, to simplify the model, we can delete redundant silent transitions and places as another post process.

4.2.7.1 Add long-term dependency

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

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

An exclusive choice structure be represented as ***xor block*** in a process tree. For the sake for convenience, we name a subtree of an xor block as one ***xor branch***, and denote the set of xor blocks B(Q) as and the set of xor branches as BB(Q) for a tree Q. For two arbitrary xor branches with long-term dependency, they have to satisfy the conditions: (1) they have a sequential order;(2) they have significant correlation. The order of xor branch follows the same rule of node in process tree which is explained in the following.

Definition 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 aspect of process tree structure, $X \prec Y$, if the least common ancestor of X and Y is a sequential node, and X positions before Y.

The correlation of xor branches is significant if they always happen together. To define it, several concepts listed below are necessary.

Definition 4.5 (Xor branch frequency)**.** The frequency for an xor branch X in event log L is the count of traces, $f : X \to N$.

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

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 ,

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

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,

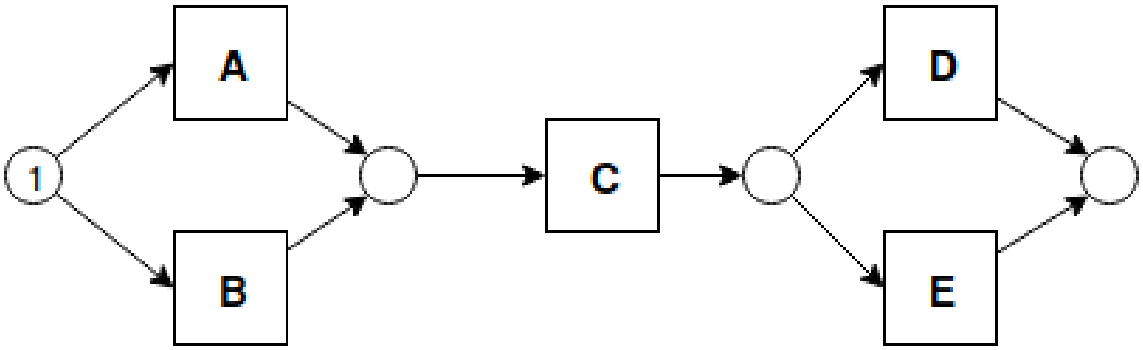


Figure 4.2: One Petri net with two two xor blocks

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 4.6 (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

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

$f_{L^{+}}(X,*)$ and $f_{L^{-}}(X,Y)$ are the frequency of the coexistence of X and Y, respectively in positive and negative event logs.

4.2.7.2 Cases Analysis

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

Definition 4.7 (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 defined as: $LT_T(X) = \{Y | X \rightsquigarrow Y \in LT\}$ Respectively, the source xor branch relative to one xor branch in target is $LT_S(Y) = \{X | X \rightsquigarrow Y \in LT\}$

At the same time, we use S and T to represent the set of xor branches for source and target xor block with long-term dependency. Given an Petri net in Figure **??**, two xor blocks are contained in the model which allows the following long-term situations.

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

2. $LT = \{A \rightsquigarrow D, A \rightsquigarrow E, B \rightsquigarrow E\}$.
   $LT_S = \{A, B\}, LT_T = \{D, E\}$ $LT_S = S$ 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 following, we propose a method to express long-term dependency on Petri net.

4.2.7.3 Way to express long-term dependency

Adding places to Petri net can limit the behavior[9], since its output transitions demand a token from it to fire themselves. When there is no token at this place, the transitions are enabled. By injecting extra places on Petri net, it can block negative behaviors which are not expected in the aspect of business performance.

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

Given arbitrary two xor blocks, $S = \{X_1, X2, ...X_m\}$ and $T = \{Y_1, Y_2, ...Yn\}$ with long-term dependency $LT = \{X_i \rightsquigarrow Y_j | 1 \leq i \leq m, 1 \leq j \leq n\}$, we add places after the source xor branches, $P_S = \{p_{X_i} | X_i \in LT_S\}$, and places before target xor branches,$P_T = \{p_{Y_j} | Y_i \in LT_T\}$. For each long-term dependency $X_i \rightsquigarrow Y_j$ in LT, there is silent transition t with $p_{X_i} \rightarrow t \rightarrow p_{Y_j}$. The steps to add silent transitions and places according to the long-term dependency are listed in algorithm 2.

One simple example is given in the Figure **??** to explain this algorithm. Given the long-term dependencies $LT = \{A \rightsquigarrow D, A \rightsquigarrow E, B \rightsquigarrow E\}$, two extra places are added respectively after A and B; Next, two places before D and E are created to express that the xor branches are involved with long-term dependency. At end, for each long-term dependency, a silent transition is generated to connect the extra places after the source xor branch to the place before target place.

Algorithm 2: 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.
```

4.2.7.4 Soundness Analysis

Following algorithm 2 by adding silent transitions and places to express long-term dependency, the model soundness can be violated. In the following section, we discuss the soundness in different situations.

Given a Petri net with long-term dependency $LT = \{X_i \rightsquigarrow Y_j | 1 \leq i \leq m, 1 \leq j \leq n\}$ on two xor blocks $S = \{X_1, X2, ...X_m\}$ and $T = \{Y_1, Y_2, ...Yn\}$, following the Algorithm 2, $P_S = \{p_{X_i} | X_i \in LT_S\}$, $P_T = \{p_{Y_j} | Y_i \in LT_T\}$, 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 2.

Soundness outside xor blocks.

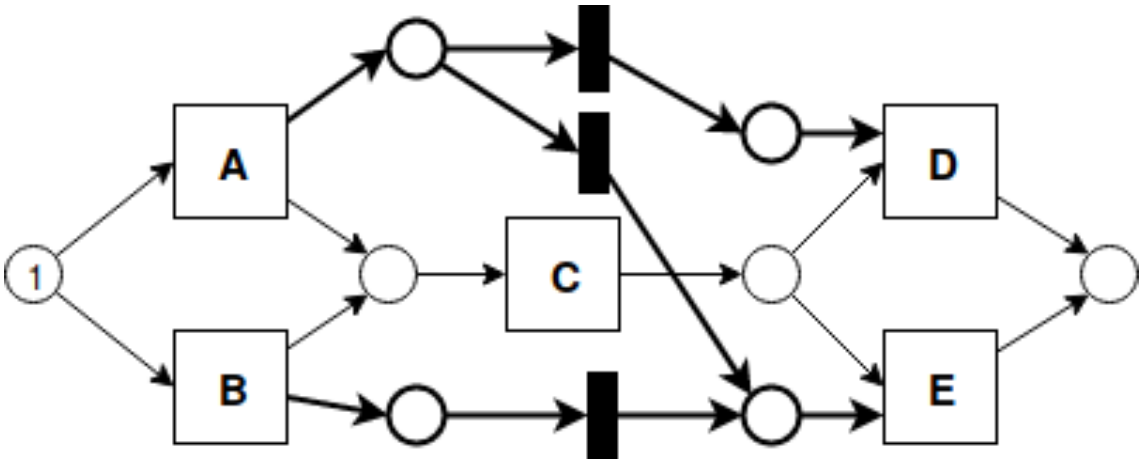


Figure 4.3: Model with long-term dependency $LT = \{A \rightsquigarrow D, A \rightsquigarrow E, B \rightsquigarrow E\}$.

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 all xor branches in S, only one branch can be fired. Without loss of generality, X_i is assumed to be enabled. After firing X_i, the marking distribution on the extra places are

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

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 through it.
  Because all $Y_j \in T$ are also in LT_T, there exists at least one $X_i \in S$ with long-term dependency with Y_j. After X_i is fired, one token is generated on the extra place p_{X_i} and can be consumed by silent transition ϵ in $p_{X_i} \to \epsilon \to p_{Y_j}$ to produce a token in p_{Y_j}, which enables xor branch Y_j and leaves no dead part.

Else, 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 situation $LT_S = S, LT_T = T$ is considered. However, when the long-term dependency is full connected where each combination of xor branches from source and target xor block has long-term dependency, namely xor branches can be chosen freely, we don't add any places and silent transitions on the model.

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 to delete redundant silent transitions and places.

Proposition 4.8. Given a silent transition ϵ in Petri net with one input place P_{in} and one output place P_{out}, if $|Outedges(P_{in})| \geq 2 \quad and \quad |Inedges(P_{out})| \geq 2$, the silent transitions can not be deleted. Else, the silent transitions is able to delete, meanwhile the P_{in} and P_{out} can be merged into one place. This reduction does not violate the soundness and does not change the model behavior.

Soundness Proof. If a silent transition t is able to delete, then

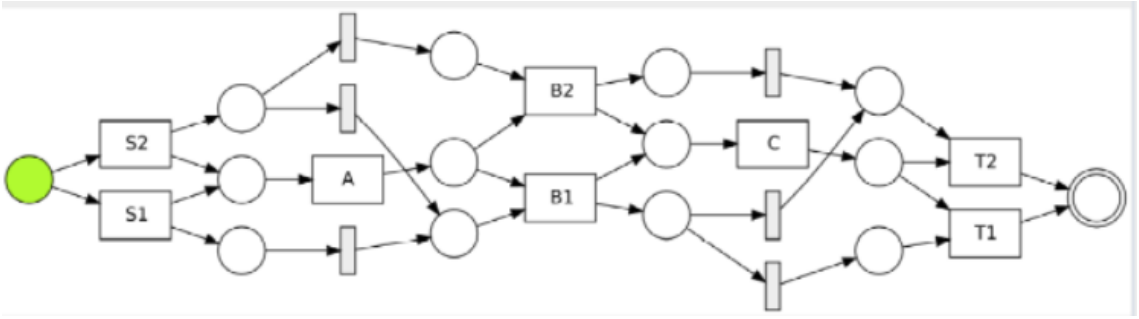
$$|Outedges(P_{in})| \leq 1 \quad (1) \quad or,$$

$$|Inedges(P_{out})| \leq 1 \quad (2).$$

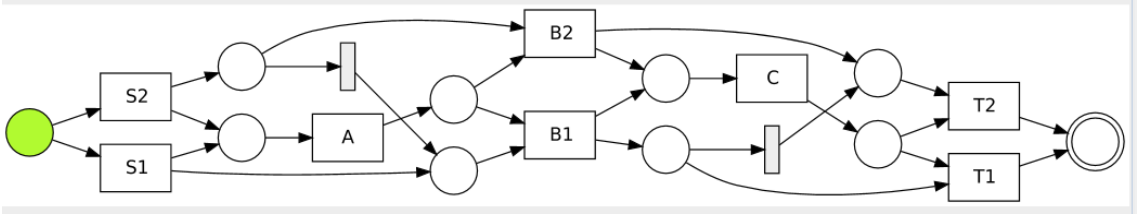
When in case (1), P_{in} contains a token that is always passed to P_{out} by silent transitions t. After deleting the silent transition, the token is generated directly on P_{out}. Since t is silent transition, it won't affect the model behavior.
when in case (2), P_{out} contains a token that is always passed from P_{in} by silent transitions t. After deleting the silent transition, the token is remained on P_{in}, which enables the later execution after the original P_{out}. Since t is silent transition, it won't affect the model behavior.
In other cases, $Outedges(P_{in})| \geq 2 \quad and \quad |Inedges(P_{out})| \geq 2$, which means that the source xor branch with output place P_{in} has at lest two long-term dependencies; the

target xor branch with input place P_{out} has at least two long-term dependencies. If we delete this silent transitions, the long-term dependencies are mixed together which allows more unexpected behaviors. Therefore, silent transition in this situation is necessary to hold long-term dependency. □

One example is given in the following graph. M_{lt} in Figure **??** 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 **??**. Those two models have the same behavior, yet the reduced model is simpler.



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



(b) Petri net M_r with reduced silent transitions

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 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.

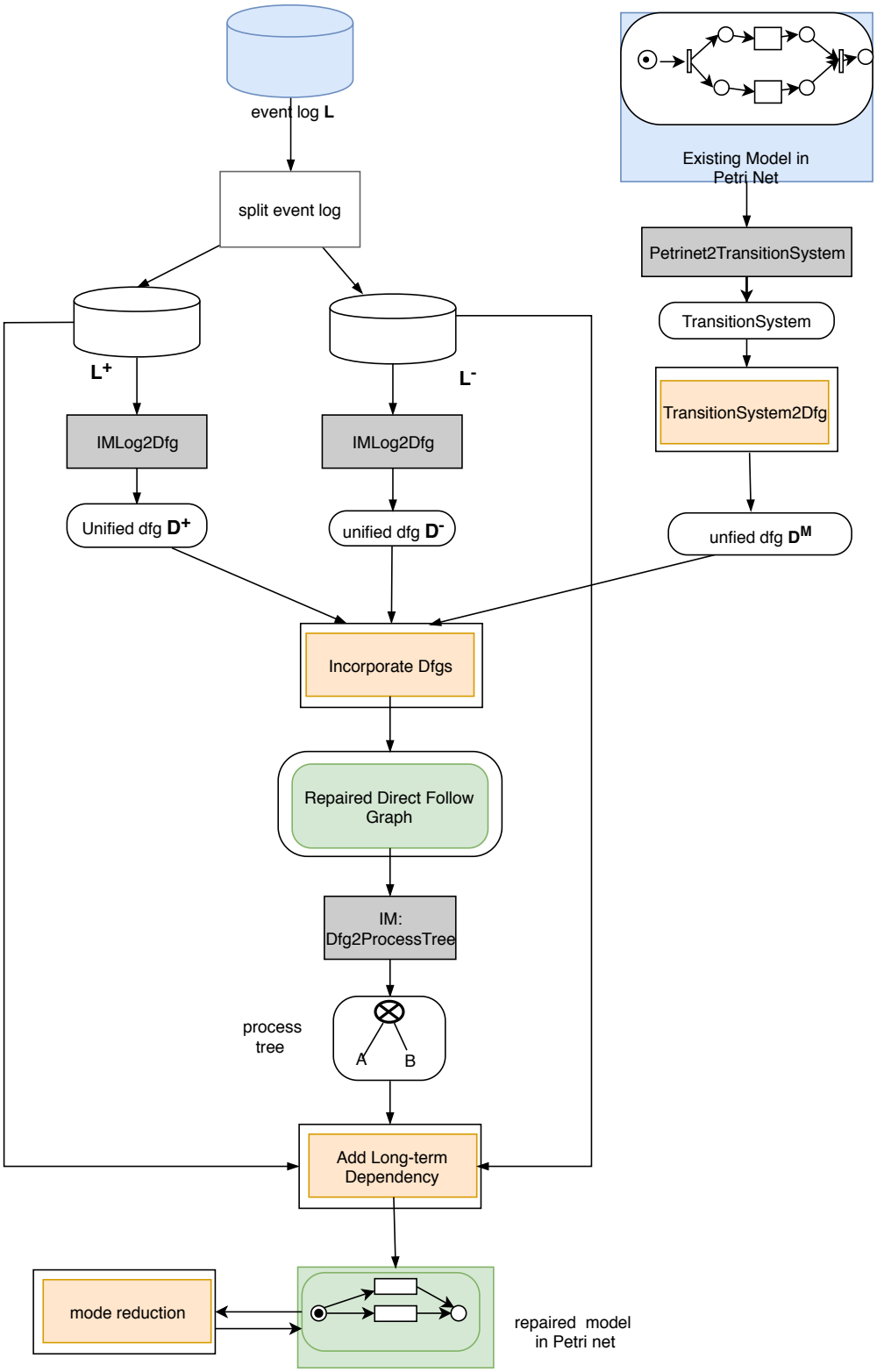


Figure 4.5: Model Repair Architecture – Rectangles represents processes and output data in eclipse shape, especially customized processes and data are in doubled lattice shape. Input event log and existing model are in blue, KPIs are in cloud. The output is a petri net in purple.

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[22]. 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[23].

5.1.2 KNIME

KNIME Analytics Platform is an open-source software to help researcher analyze data by integration of multiple modules for loading, process and transformation and machine learning algorithms. Researcher can achieve their goals by creating visual workflows composed of 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 Process Model

Firstly the dialog for options to generate directly-follows graphs from event log pops up. Event classifier are set by those dialogs. Subsequently, a dialog is shown to set the Inductive Miner parameters. The parameters include the Inductive Miner variant and the noise threshold to filter the data. The dialog is displayed in Figure 5.1.
After setting the parameters, process models of process tree and Petri net without long-term dependency can be generated by Inductive Miner and displayed in the result view in Figure 5.2. The left side is the model display area. To allow more flexibility, this plug-in are interactive by the control panel, which is the right side of result view. Originally, only

Figure 5.1: Inductive Miner Parameter Setting

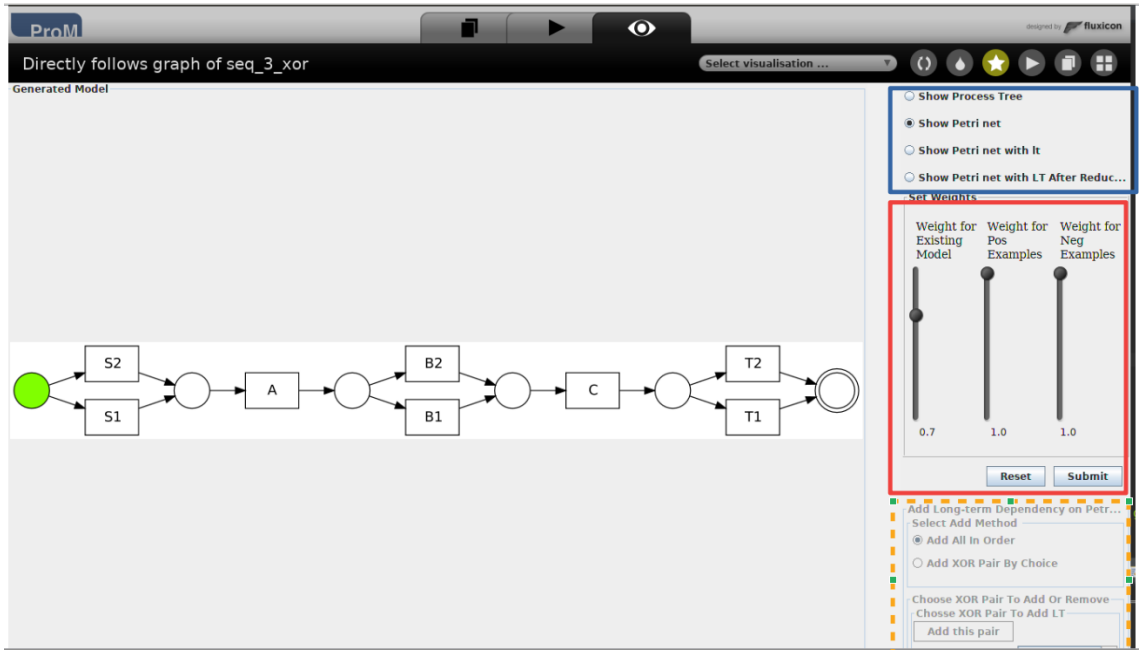


Figure 5.2: Generated Petri net without long-term dependency

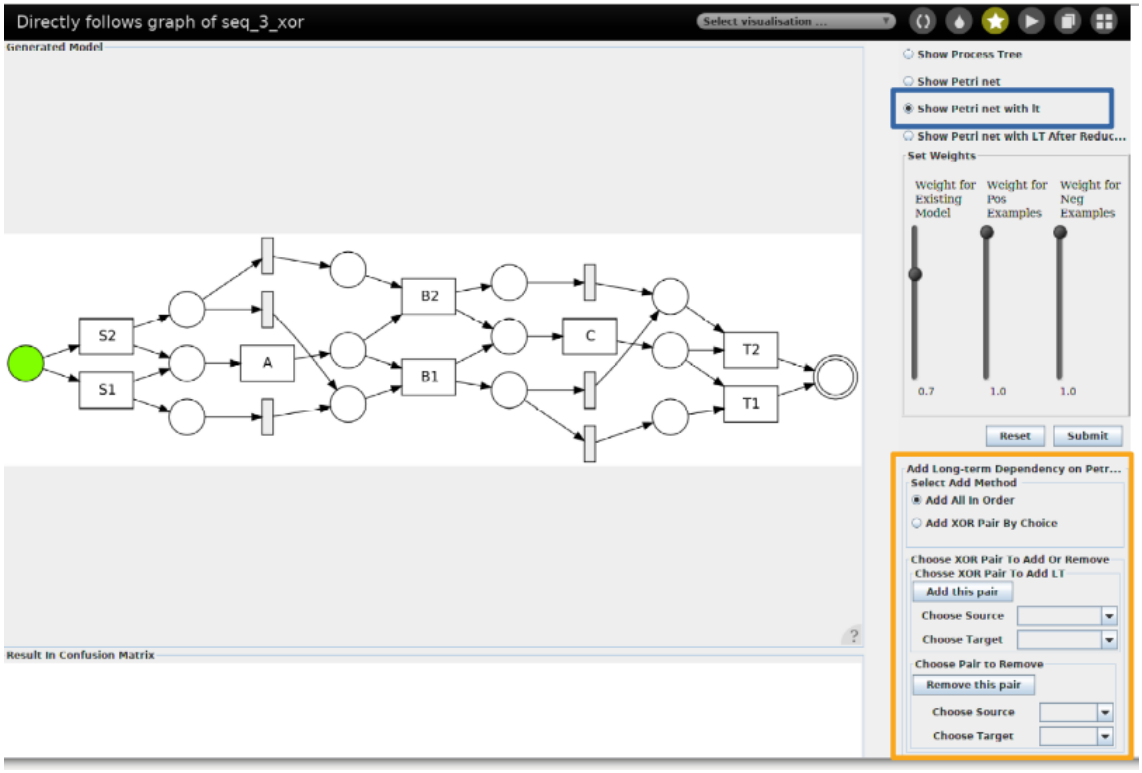


Figure 5.3: Petri Net with long-term dependency

the generated model type and the weight sliders are enabled, while the control panel for adding long-term dependency are invisible.

The model type are in the blue rectangle marked in Figure 5.2. It has 4 options to control the generated model type. Currently, the option "Show Petri net" is chosen, so the constructed model is Petri net without long-term dependency. The weights sliders are in red rectangle. It enables to adjust the weights on the existing model, positive and negative instances. Once submitted those options, different process models are mined under different weights. The rectangle in orange are the invisible part to control long-term dependency options. It is discussed in the next section.

5.3 Post Process to Add Long-term Dependency

If the model ro generate is Petri net with long-term dependency, the program to add long-term dependency is triggered. This program in the background detects and puts places and silent transitions on Petri net directly mined from Inductive Miner to add long-term dependency. As comparison, the same weight setting is kept like the Figure 5.2, but the option to show a Petri net with long-term dependency is chosen. The resulted model is Figure 5.3.

Meanwhile, the control part of adding long-term dependency turns visible, which is in the orange rectangle in Figure 5.3. It has two main options, one is to consider all long-term dependency existing in the model, the other is to choose the part manually. It allows more flexibility for users. Below those two options, it is the manual selection panels, including control part to add and remove pair. As an example, the blocks Xor(S1,S2) and

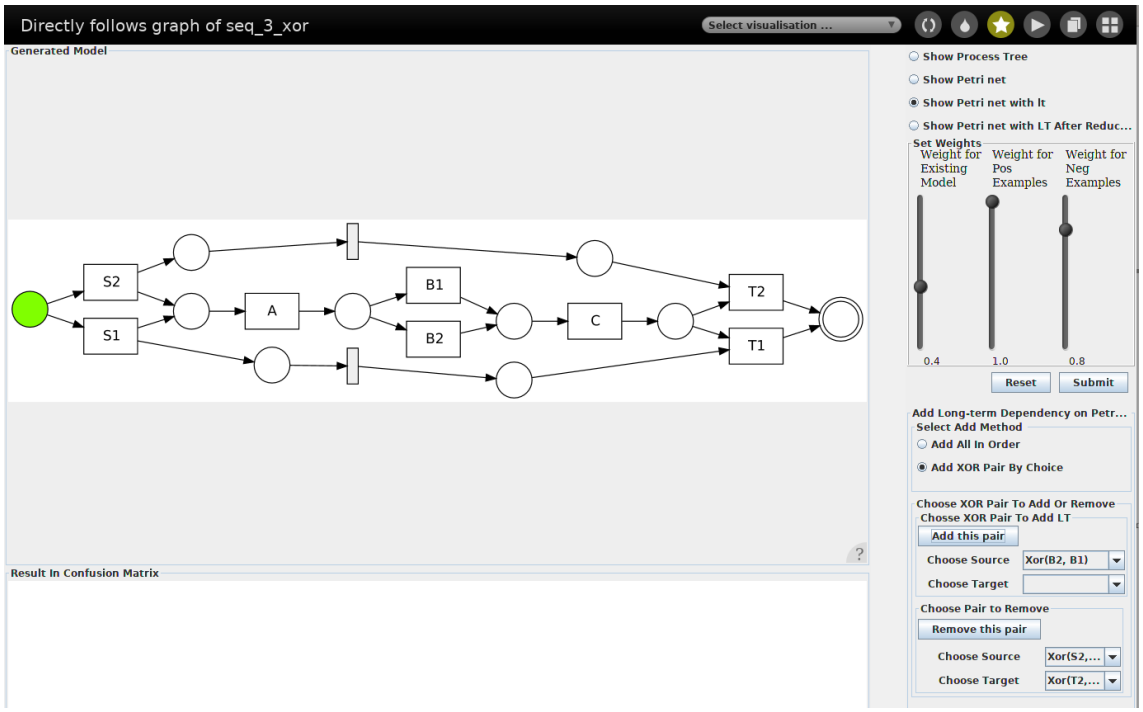


Figure 5.4: Petri net with selected long-term dependency

Xor(T1,T2) are chosen to add long-term dependency. It results in the model in Figure 5.4.

5.4 Post Process to Reduce Redundant Silent Transitions and Places

By choosing *Petri net with LT After Reducing* in model type option panel, silent transitions are reduced to simplify the model. Under the same setting in Figure 5.2, the simpler model in Figure 5.5 is constructed, after the post processing of reducing silent transitions.

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 parameter and select the final process model.

It works in this way. After creating the current model in the left view, the evaluation program in background uses the event log and the current Petri net in the view as inputs. It applies a naive fitness checking and generates a confusion matrix with relative measurements like recall, precision. This evaluation result is then shown in the bottom of the left view in Figure 5.6. If the button of green rectangle in the right view *Show Confusion Matrix* is pressed again, the program is triggered again and generates a new confusion matrix result in dark green dashed rectangle which will be listed above the previous result in light green dashes area.

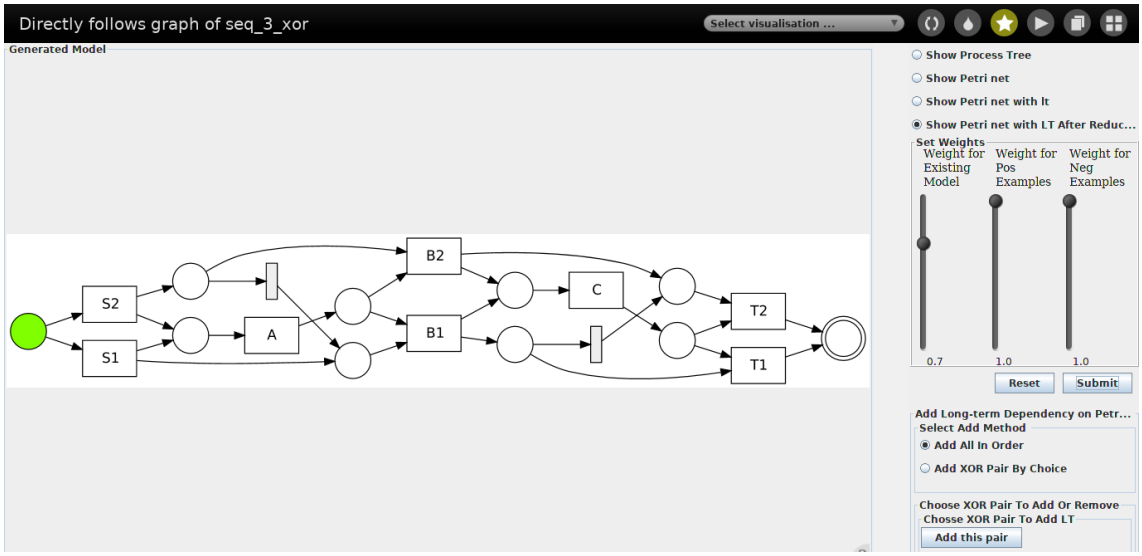


Figure 5.5: Petri net after reducing the silent transitions

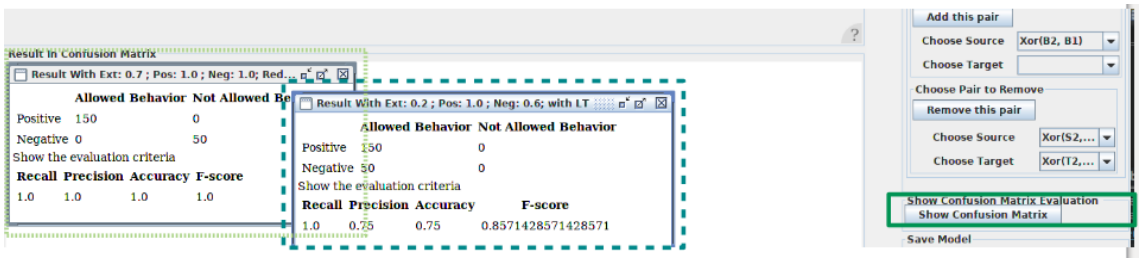


Figure 5.6: Generated Process Tree Model

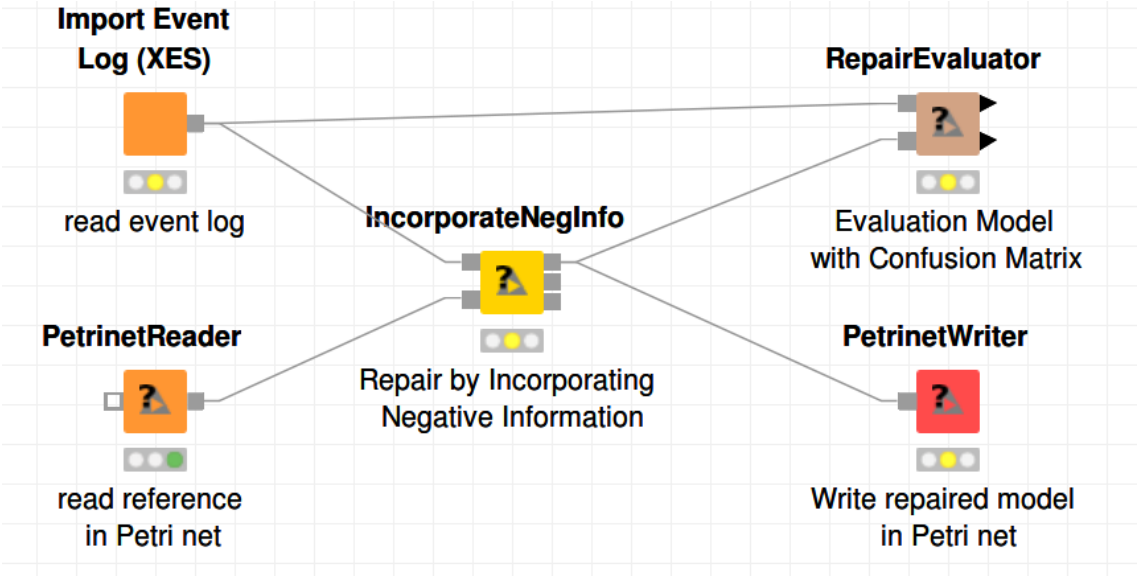


Figure 5.7: Integration of our repair techniques into KNIME

5.6 Integration into KNIME

To integrate our algorithm into KNIME, other related modules on process mining are necessary, which can be divided into the following categories: (1) event log and process models importer and exporter; (2) event logs manipulation;(3)classic discovery algorithms;(4) model enhancement with the algorithm in this thesis.

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 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. At last, to obtaining the feature of showing evaluation result, the node *RepairEvaluator* is called. What's more, we can save our Petri net by using node *PetrinetWriter*.

Chapter 6

Evaluation

This chapter presents an experimental evaluation of our techniques to repair model. At first, the evaluation measurements are defined. Next, we briefly introduce the test platform KNIME and ProM plugins tools for evaluation. In the following main part, the test on properties of our techniques is presented at the beginning. Then synthetic data is generated randomly to show the whole performance of our methods. At last, we conduct our experiments on real life data and also compare our techniques with other methods. The results show the ability of our techniques to repair model with high ranking according to defined measurements.

6.1 Evaluation Measurements

We evaluate our 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. The repair techniques in this thesis are based on the labeled data and the repaired model can be seen as a binary prediction model where the positive instances are supported while the negative ones are rejected. The model evaluation becomes a classification evaluation. Confusion matrix has a long history to evaluate the performance of a classification model.

Table 6.1: Confusion Matrix

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

A confusion matrix is a table with columns to describe the prediction model and rows 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.

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

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

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

- specificity. In opposite with recall, it measures the true negative rate.

$$Specificity = \frac{TN}{TN + FP}$$

- 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.

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

After experiments we can see that accuracy gaine more weights than precision.

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 Platform

Data simulation part Try to automatize and standardize experiments, in order to make this experiments more effective, and easier to repeat. So KNIME is selected as one of the test platform to perform tasks. Also, some existing evaluation plugins in ProM are used to evaluate on the other aspects.

6.2.1 PLG

Process Log Generator(PLG) is designed to generate random business processes models and event logs according by setting the complexity parameters. Complexity parameters for model generation in PLG include the maximum number of branches for activities relation parallel, or exclusive choices, maximum depth of nested pattern, the weights on relations, and so on. The parameters to generate an event log are, for example, number of traces, and noise on the workflow or missing rate of trace heads.

In this experiment, process models in Petri net are randomly created by inputting simple, normal and complex parameters settings. Event logs are also simulated according to different complexity level. The setting details is in Table??.

6.2.2 KNIME

KNIME Analytics Platform is open source software in Java to help researcher analyze data by integration of multiple modules for loading, process and transformation and machine learning algorithms. Researcher can achieve their goals by creating visual workflows composed of modules with an intuitive, drag and drop style graphical interface, rather than focusing on any particular application area. The reason to choose KNIME as our experiment platform is that KNIME supports automation of test workflow which is more efficient. However, to conduct experiments, the related modules of our algorithm have to be integrated into KNIME, which requires additional development effort. At end, process mining modules like event logs, basic process models reader and writer, basic logs manipulators, classic discovery algorithms, and model repair are imported into KNIME.

6.2.3 ProM Evaluation Plugins

Here we discuss the plugins to test other aspects of our methods.

6.3 Experiment Result

6.3.1 Test On Property

In this experiment, we aim to answer the questions: 1) How do our techniques incorporate the existing model, positive and negative information to repair model? 2) How do the techniques overcome the shortcomings of current state-of-the-art repair techniques? To answer the first question, we applied the repaired techniques on event logs with different relations of activities, such as sequence, parallel and loop, exclusive choice and investigated the effect from the existing model, positive and negative instances on the repaired model, by adjusting the corresponding weights. For the second question, we describe the situations where our techniques overcome the defects of other repair techniques and explain the reasons behind it.

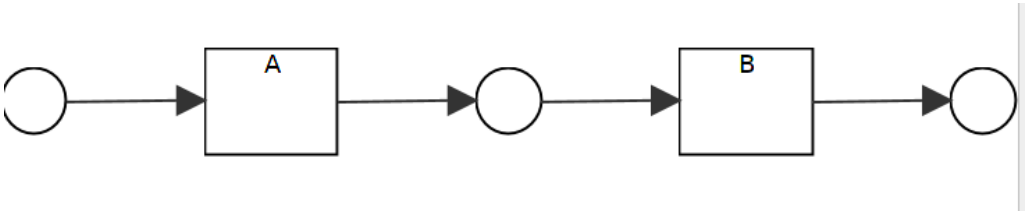


Figure 6.1: Model M1 with sequence relation

6.3.1.1 Test On Sequence

This part is used to show the effect of our techniques on the sequence relation of activities. Given a fixed model in Petri net with sequence relation, a set of event logs with different deviations are used to test if our repair techniques work properly. By changing the control weights of the existing model, positive and negative instances, different models are derived from the repair method. Next, we evaluate the model based on confusion matrix and plot the several measurements with the weight changes. By this way, we show how our method incorporates different effect of control weights.

In this experiment, we fix one weight as control variable while keeping the other weights unchanged. To make the experiments more reliable, for each fixed control variable value, we conduct the experiment in 5 different combinations of other weights that are randomly generated. At last, the average of those combination is used to represent the final measurement.

- *Experiment 1* – delete activity from sequence relation.
  Event Log –

$$Positive :< a >^{50}$$
$$Negative :< a, b >^{50}$$

  Result Analysis – Figure 6.2 shows the measurements results separately due to the weight change. In Figure 6.2a, with the weight for the existing model increases, precision, accuracy and specificity decrease while the recall keeps 1.0. After checking confusion matrix, we see TP and FN keep the same, FP goes up while TN goes down. The reason is that the higher weight for the existing model causes the higher possibility to keep the existing model during the repair. Due to the deviation of the existing model and actual event log, precision and accuracy goes down. With the weight for positive goes up, all measurements are stable in value 1.0. The reason for this is the TF and TN keeps the same, other values are all 0.0.

-

6.3.1.2 Test On Parallel

This part shows how the parallel relation of activities is affected by the weights for the existing model, positive and negative instances.

6.3.1.3 Test On Loop

This part investigated our repair method on activities with loop relation.

6.3.1.4 Test On Exclusive Choice

This part displays the changes of exclusive choices relation in the model under the different control weights.

6.3.2 Comparison To Other Techniques

This section represents some situations where current repair techniques can't handle properly, while our algorithm gives out an improved repaired model.

Situation 1, unfit part!! added subprocess are too much!! Where the addition of subprocesses and loops are allowed, while the structure changes are impossible, Fahrland's method applies the extension strategy to repair model by adding subprocesses and loops in the procedure. It introduces unseen behavior into the model. However, if the behaviors which are already in the model is unlikely to be removed from the model. One simple example is shown in the following part.

Dee's method is based on Fahrland's method. Deviations are calculated at first and used to build subprocesses for model repair. However, before building subprocesses, it classifies the deviations into positive and negative ones with consideration of trace performance. Only positive deviations are applied to repair model. Different to Fahrland's method, it improves the repaired model performance by limiting the introduced subprocesses. Still, it can't get rid of the defect mentioned before.

Situation 2, For fitted data in the model, can not recognize them!! where overlapped data noise can not be recognized, trace variant with more negative effect is treated as positive and kept in the model, which we should delete them.

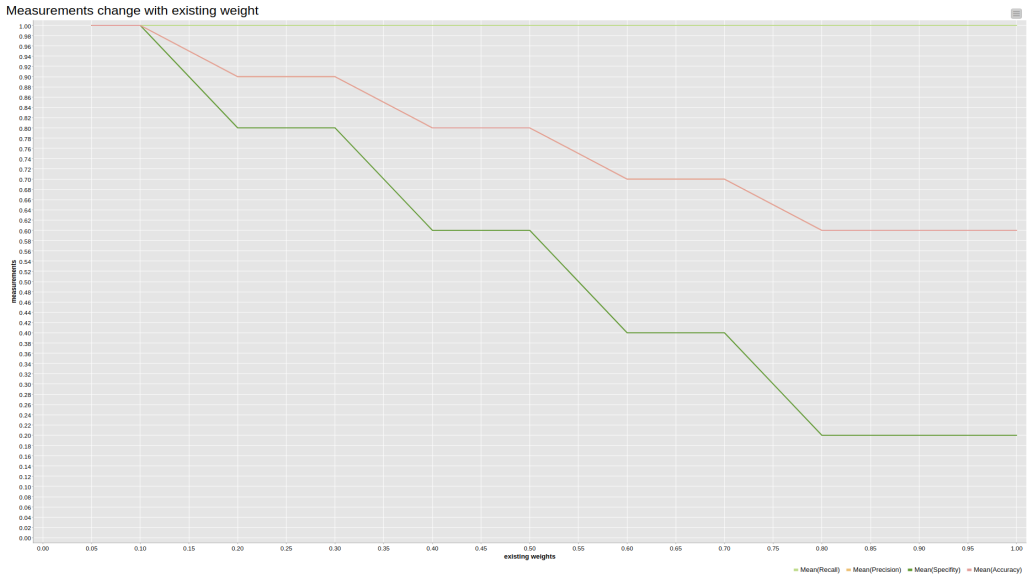
Situation 3, with long-term dependency!! fitted part or new added part!! none of the current techniques can handle this problem yet. Simple examples listed, but will this repeat the last section??

For one exclusive choices, but with long-term dependency detected and added in the model, precision and accuracy increase, since model with long-term dependency blocks the negative information by adding transitions and places to limit activity selection.

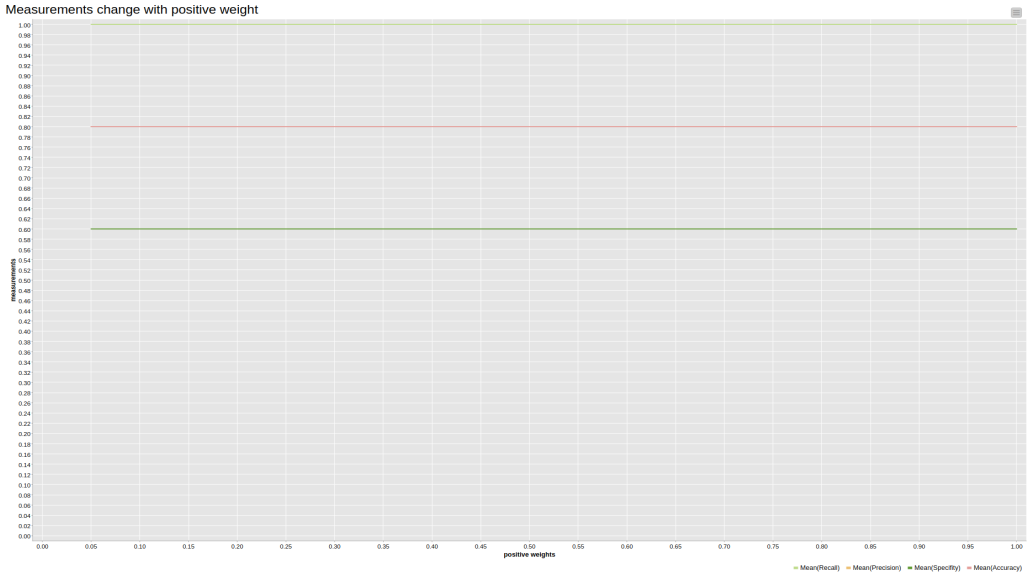
6.3.3 Test On Synthetic Data

Those synthetic data is generated randomly by using the simulation tool plg

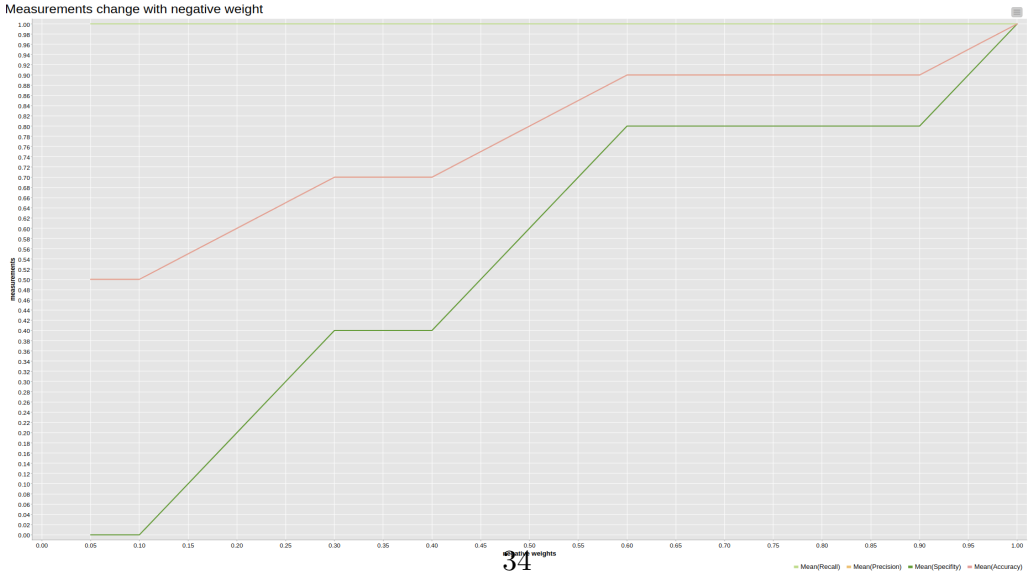
6.3.4 Test On Real life Data



(a) Measurements with the weight for the existing model



(b) Measurements with the weight for positive instances



(c) Measurements with the weight for negative instances

Figure 6.2: example for model change under model repair

Chapter 7

Conclusion

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

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