UNIVERSITY OF TARTU
FACULTY OF SCIENCE AND TECHNOLOGY
Institute of Computer Science
Software Engineering Curriculum

Juraj Jarábek

Exploring business process Deviance with Declare

Master's Thesis (30 ECTS)

Supervisor(s):
Fabrizio Maria Maggi, Fredrik Milani

Tartu 2016

Acknowledgment

I would like to thank big time to my supervisor Fabrizzio Maria Maggi for all the help and guidance, without him I would not be able to pull this off. Another big thanks to Irene Teinemaa, Ilya Verenich, Beatriz Mateos and to all my friends and family for endless support.

Exploring business process Deviance with Declare

Abstract: This thesis introduces business process deviance mining, which belongs to the group of process mining, and gives an overview on multiple deviance mining approaches. After that we focus on deviance mining using discriminative patterns, which belongs to the group of sequential patterns mining techniques. In this work we propose new discriminative pattern mining algorithm based on the Declare language. We implemented the approach as a plug-in of the Process Mining tool ProM. We describe the whole proposed approach from the labelling of the event logs until building the classifier and classifying the test logs. In the end of the thesis we evaluate the effectiveness of our proposed algorithm on variety of experiments on event logs.

Keywords: process mining, deviance mining, Declare, LTL grammar, classification, ProM, ProM plugins, Process modelling

CERCS: P170

Äriplaani protsessi hälbe kindlakstegemine Declarega

Abstrakt: See töö tutvustab äriprotsessi hälbe kaevandust, mis kuulub protsessi kaevandamise gruppi ja annab ülevaate mitme hälbivaga kaevandamise lähenemisviisidest. Keskendutakse ka hälbiva kaevandamise kasutamise diskrimineerivatele mustritele, mis kuulub vaadeldakse uute diskrimineerivate mustrite kaevandamise algoritmi, mis põhineb Declare keelel. Rakendati plug-in lähenemisviisi protsessi kaevandamise näitajaga ProMile. Kirjeldatakse kogu protsessi kuni klassifitseerimiseni. Töö lõpus hinnati algoritmi tõhusust ja eksperimente erinevates variatsioonides.

Märksõnad: protsessi kaevandamine, hälbe kaevandamine, tuvastamine, LTL grammatika, liigitus, klassifikatsioon, ProM, ProM pluginad

CERCS: P170

Contents

1 Introduction

Business process mining includes process management techniques that allow us to extract knowledge about business processes based on event logs [1]. Process mining consists of several branches like automated discovery of processes from event logs, conformance checking between event logs and process models, but also techniques for predicting and analyzing performance of different processes based on event logs. Process mining takes existing data from the existing systems as an input and produce relevant knowledge which can be used later on for discovering, analyzing and enhancing business processes.

We look at a business process as a collection of events or activities, which in their end has some accomplishment of business goal. Some executions of the process are normal or expected (customer receives product) but some, like minority are not desired (customer doesn't receive product). There might be errors in business processes. Some unexpected process executions are not so easy to recognize, or at least they are missing this visible effect and some are visible e.g. by their business goal accomplishment or not accomplishment in the end. These not desired/deviant executions can influence business in negative ways (high utilizations of resources, loss of profit, etc.), therefore special concerns should be made and ensuring the reliability of business processes should be very important task.

This thesis discusses *business process deviance mining*, which is a family of techniques used to analyze the event logs of business processes in order to find out the reasons why the process deviates from normal or expected execution. These deviant business process executions can be positive or negative. Positive deviation deviates in positive direction than expected or normal execution. It means that it represents the execution with high performance, like for example short execution time of the process, low resource usage, low costs or any other positive parameter. Negative deviations deviates in negative direction than expected execution. It represents the execution with low performance. For example, too long execution time, high resources usage but as well as negative outcomes like "invoice not sent" or "customer rejected invoice" etc. or any other violations.

Process deviance mining takes as input labeled log, which is a set of labeled traces. We called this labeled log also training log or training set. Each trace in the log corresponds to one execution of the business process. Trace consists of list of elements, which represents the execution of activities. Label of the trace represents whether the trace is deviant or normal. This is usually represent by "true/false" label or "0/1" numerical label. The main focus of deviance mining is explain why some traces deviates, in other words find and explain the root causes of deviance. The deviance mining produces function, called the "classifier", which can classify the trace and label it accordingly - normal or deviant. This produced classifier then can take on the input testing log, where the traces are already labeled, and produce the accurate label for these traces so it should guess the correct class (so we can see how accurate the classifier is by comparing the previous labels with our new labels calculated by our classifier) or take as input not labeled log, with "unseen" traces and produce labels accordingly. In other words, logs from the past can be used to train this classifier and then the classifier can be used for future logs to determine which cases are deviant and which are not.

The output of deviance mining should be explanatory to analyst, therefore some visualization is needed, in most of the cases in form of nicely represented constraints/frequent patterns/discriminative patterns, etc. (depending on the discovery techniques). Classifier is constructed of the set of these patterns/constraints, which were minded in test log, then these patters/constrains are shown as a results, since they are interpretable by analyst. Classifier like decision tree can be interpretable to the analyst as well, but the main purpose of classifiers are the evaluation of labelling accuracy.

There are multiple existing methods for deviance mining. Many of these traditional deviance mining techniques have been used in the context of software processes. Software behavior is the way how the software executes. Just like in execution of business process mentioned above, in software systems some execution are desired, normal executions, and some executions are not desired and we call them failures. Some failures are easily recognized (Blue screen of death in Windows, exception thrown etc.) while some are not so easy to recognize, since they are missing this “visible” effect. These failures can cause security risks and there should be special concerns made since they are not so easily identified.

Aim of this thesis is to apply deviance mining in the BPM context. In particular we use the declarative business process modelling language Declare [2] [3] to explain deviance. In this way we use process models to explain deviances that give more abstract representations of the properties than sequences (form different logs and process models). Our proposed framework works in three steps process: In the first step we mine Declare constraints/patterns from the event log. In the second step the constraint filtering and selection is applied to the constraints. The selection is based on the discriminativeness of the constraint which is represented by Fisher’s score. In the last third step we build the classifier from these selected constraints.

We implemented our algorithm as the *Declare Deviance Miner* plugin[1] for ProM open-source tool.

In [4] authors have done comparison of difference deviance mining tools (based on different deviance mining techniques), therefore our main research questions in this thesis are:

1. *Is it possible to provide business analyst with understandable feedback about business process deviances?*
2. *Is the accuracy obtained by explaining deviances with Declare constraints comparable with the one achieved with the state of the art deviance mining techniques?*

In summary, our main contribution to answer these research questions is the following:

- We proposes new approach for classifying event logs based on discriminative Declare patterns, which distinguish the deviant from normal traces in the event log. To identify highly discriminative Declare constraints, we proposes Declare constraints selection algorithms based on Fisher score.
- We design a new algorithm for mining Declare constraints, using Declare templates which are then satisfied or not satisfied within the traces.
- Throughout a set of experiments we evaluate accuracy of our proposed Declare mining algorithm and give comparable results with already existing deviance mining approaches.

This thesis is structured as follows. Section 2 discusses already existing methods for deviance mining, Section 3 discuss prior information need to understand this thesis. Section 4 discusses our entire proposed approach, from the event log to the result, Section 5 discusses implementation of our approach as ProM plugin, section 6 discusses evaluation of our labelling accuracy and comparison with existing tools and the last section 7 represents conclusion of our work.

[1] https://bitbucket.org/Duri_Boss/declaredevianceminerplugin

2 State of Art

There are many papers describing process deviance mining and different process deviance mining techniques. One of the first techniques to mention is *delta-analysis*.

In [5] delta analysis has been used in order to explain deviance in healthcare processes. In another paper [6] authors report on case about insurance company trying to describe why some claims took longer than normal. Analysts understand how different are "simple slow claims" from "simple quick claims" which took x more days to be handled. Delta-analysis used in this report consists of using automated process discovery, to specifically extract two process models, one for "deviant" and one for "normal" executions. Firstly the two logs were created from the main log, where one log consists only of "simple slow claims" and second one of "simple quick claims". Then the *Disco process discovery tool*[2] was used to extract two process models from these two logs. Then the manual comparison of these two models was done. After comparison authors found some visible differences between these models. Some sequences or paths of events were more frequent only in the first model and vice versa. Another heuristics, which authors came up with, which helped to discriminate between slow and fast claims, was comparing chosen event X occurrences numbers in the traces and comparing the percentage of traces where chosen event appears at least one time. In other words event X occurred in different amount of traces in these two models as well as the occurrences for particular traces were different. Delta analysis is a manual technique of deviance mining, in this thesis we focus just on automated techniques. Our contribution is similar to Delta analysis because we produce models but we do it automatically.

In [7] the authors applied a *discriminative pattern* mining approach on a log of over 2600 defect reports of four big software projects to discriminate between normal correct resolutions and deviant ones leading to complaints. Using discriminative patter mining approach they identified discriminative patterns which are frequent in deviant cases and not frequent in normal cases and vice-versa. First number of features were just "number of occurrences of particular activity X in the trace" and another features were "number of occurrences of activity X after activity Y". Since there is simple too many combinations of (X,Y), in order to avoid too big amount of discriminative features authors selected only some using mining techniques. Afterwards classifier – decision tree was built based on these selected features.

In [8] authors tried to discriminate between traces in the log leading to malfunctioning versus normal behaving of X-ray machines. In this paper authors were using *frequent pattern mining*, which belongs to the group of *sequence mining techniques* in order to find "maximal repeats", "tandem repeats" and "alphabet repeats" [9]. Maximal repeats represent repeated sequence of events which are not included in longer repeated sequence. In the process maximal repeats represent sub-process. Tandem repeats are sequences of events which represent loops in business processes. Alphabet repeats are intersections of sets of events which are found in different maximal or tandem repeats. In control-flow terms this represents parallelism. In this paper *frequent patterns* are extracted from all the traces together (normal and deviant cases) and afterwards those with highest support are selected and decision tree is built using these frequent patterns.

In [10] the authors tried to use sequential pattern mining as well to discriminate between cases, which lead to positive or negative clinical outcome in the process for congestive heart failure treatment. Authors used sequence mining techniques combined with Delta analysis mentioned in [6]. Authors first used sequence mining techniques to extract frequent patterns of the form

[2] http://www.fluxicon.com/

"activity B occurs after activity A after some time" from both negative and positive outcomes. After the extraction authors could determine which of these patterns are frequent in positive and which are frequent for negative cases. Authors then made additional comparison of process models which were extracted from positive and negative cases. Observations from this delta analysis were combined together with observations from frequent patterns mining and in the end authors got pathways characteristics of either negative or positive cases.

Authors in [11] tried to understand the causes for deviations in procurement processes from normative pathways. This study was done in large European financial institution were the dataset consists of close to 30 000 cases. After using process discovery tool analysts found that approximately 29% of the cases were deviant. Authors applied association rule mining technique in order to extract frequent patterns for deviant and for normal cases separately. It was found that total of 10 patterns could characterise almost all deviant cases.

In the above studies all discriminative mining techniques were using the process flow not data flow for deviance mining. In other words the inputs were some sequences of occurrences of activities without payload. Sometimes traces or events carry multiple information. For example traces hold some information or events consists of attributes like "customer age" or "type", which could be used for deviance mining as well. There were some studies, which tried to attempt to use this data for deviance mining [12] [13], in this paper we will focus on process flow, ignoring the data attributes so we make minimal assumptions on the log.

Based on the papers mentioned in this section we came up with Figure 1 which represents the Taxonomy of Process Deviance Mining approaches excluding manual techniques like Delta analysis. This figure represents only automated techniques. These papers helped us to identify three categories of deviance mining techniques: (1) based on individual activities (2) based on set-based patterns (3) based on sequence patterns which can be divided into sequential patterns based and discriminative patterns based.

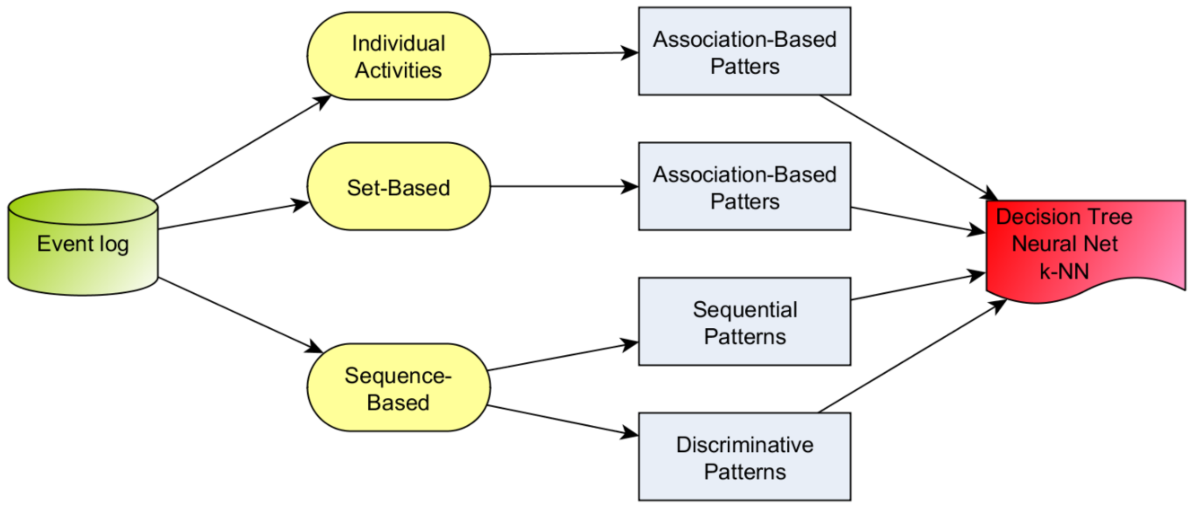


Figure 1. Taxonomy of automated Deviance mining approaches

As can been seen in the Figure 1, each technique has the event log on the input and build the classifier in the end, which in all papers was either Decision Tree or combination of these three classifiers: Decision Tree, Neural networks or k-NN.

In this thesis we will use only discriminative patterns as the main approach for our deviance mining.

3 Background

In this chapter, prior information needed to understand the thesis are described.

3.1 Declare

Pesic and van der Aalst introduced Declare in [2] [3] as declarative process modelling language. Even though Declare is declarative system, it can offer more than just declarative model development, but also automated model execution, model verification, changing model during run-time, decomposition of big processes and other features then traditional WFMSs have.

Declare language uses constraint-based system as opposed to classical imperative approaches and using declarative language grounded in temporal logic. Declare constraints build declarative model. Declare constraints are based on Declare templates, which we will describe more in following subsections.

In classical imperative model, there is need to specify control-flow by entering all the options how the process executes. Declare model defines control-flow unconditionally by specified list of constraints, which represents the rules. Declare language provides more flexibility than usual procedural notations like BPMN, UML, Petri Nets, ADs and BPEL. This language, proposed for process modelling, is fulfilling two main criteria: It is understandable for end-users and it has formal semantics.

Declare templates are notional objects which determine parameterised categories of properties, and Declare constraints are their concrete instantiations. These templates have graphical representation, which is easily understandable and readable for the end-user and using LTL formulas the semantics of these templates are specified. Table 1 below, which shows LTL operators semantics. Template defines semantics and graphical representation of each constraint which is generated from this template. Because of these features Declare is easily understandable and it has formal semantics.

Table 1. LTL operators semantics

operator	***semantics***
$\bigcirc\varphi$	*φ has to hold in the next position of a path.*
$\Box\varphi$	*φ has to hold always in the subsequent positions of a path.*
$\Diamond\varphi$	*φ has to hold eventually (somewhere) in the subsequent positions of a path.*
$\varphi\, U \psi$	*φ has to hold in a path at least until ψ holds. ψ must hold in the current or in a future position.*

So as mentioned above Declare language is using the set of constraints, which must be satisfied during process executions rather than directly specifying the process flow. So with comparison with classical procedural approaches we can say that Declare defines flexibility. Classical procedural models produce “closed” process models. By “closed” model we mean, that if something in this model is not directly specified, then it’s forbidden. On the other hand Declare give us more options for executions, more flexibility.

Let us focus on Declare templates now. The main goal of the templates is to highlight different attributes, which is worth to describe in process model as well as mainly characterise language. Declare templates are divided into four main groups: *relation, negative relation, existence and choice*. We give more details about specific group of templates and templates themselves in following subsections.

3.1.1 Existence Templates

Figure 2 below represents so-called existence templates. There is only one event (unary relationship) involved in existence templates. Existence templates defines the position or cardinality of the event in the trace.

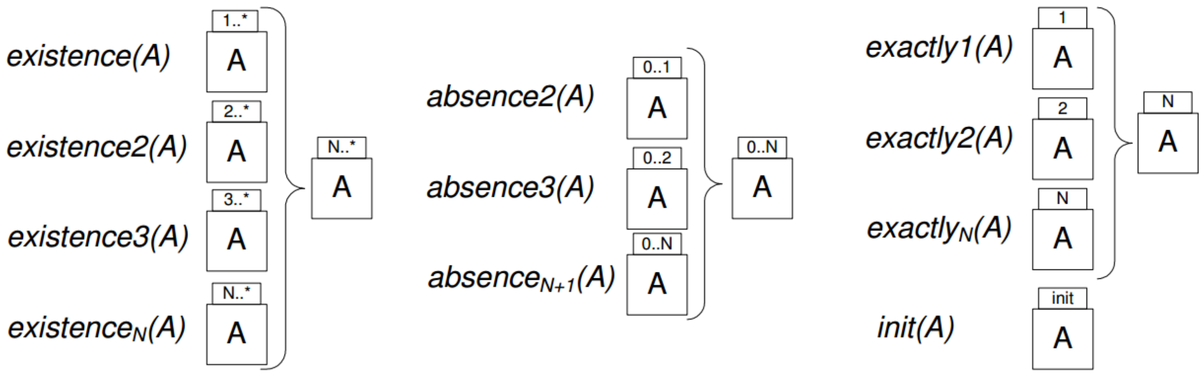


Figure 2. Notation for existence templates

The first templates *existence(A),* which has annotation "1..*" above the event A (see Figure 2), represents that A is executed at least once in the trace (event A is present at least one time and more in the trace). So others like *existence2, 3 and N*, the number behind the templates name presents that A should occur at least that amount of times in the trace. Absence templates, which has annotations "0..N", on the other hand specifies that event A should be present in the trace N times at most. E.g. *Absence3(A)* represents that event A should occur 2 times in the trace at most. Templates *ExactlyN(A)* specify that event A should occur exactly N times in the trace and template *Init(A)* specify that that event A should be the very first event in the trace.

Table 2. Existence Templates

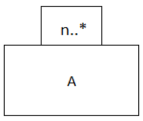
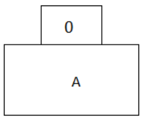
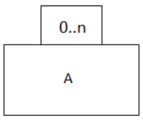
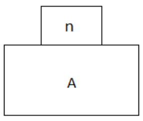
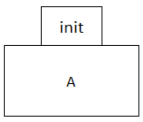
name of template	*LTL semantics*	*graphical representation*
$existence(1, A)$ $existence(2, A)$ ... $existence(n, A)$	$\Diamond A$ $\Diamond(A \wedge \bigcirc(existence(1, A)))$ ... $\Diamond(A \wedge \bigcirc(existence(n-1, A)))$	
$absence(A)$	$\neg existence(1, A)$	
$absence(2, A)$ $absence(3, A)$ ... $absence(n+1, A)$	$\neg existence(2, A)$ $\neg existence(3, A)$ ... $\neg existence(n+1, A)$	
$exactly(1, A)$ $exactly(2, A)$ ... $exactly(n, A)$	$existence(1, A) \wedge absence(2, A)$ $existence(2, A) \wedge absence(3, A)$ ... $existence(n, A) \wedge absence(n+1, A)$	
$init(A)$	A	

Table 2 shows LTL semantics or (LTL formula) and graphical notations of existence templates.

3.1.2 Relation Templates

Relation templates describe dependency between two events A and B in the contrary to existence templates, which specified cardinality of one event only. These templates there for represent binary relationships. Table 3 represents LTL semantics and graphical notations of relationships templates. All relation templates have two events as parameters. As can be seen from the Table 3, the column on the right represents graphical notations, where the line between events A and B is unique for each template.

The template *responded existence (A, B)* specifies that if event A is present in the trace, then also event B has to be present either after or before event A, so the event B can be executed at any time in the process.

The template *co-existence (A, B)* specify if one of the event A or B is present in the trace, then also the second event has to be present (if A is present in the trace then also B has to be present, if B is present in the trace then also A has to be present).

Templates *responded existence* and *co-existence* don't consider the ordering of the event A and B, they can be present anywhere in the trace. On the other hand next templates we will describe, *precedence*, *response* and *succession*, they consider ordering of the events, therefore what are the locations of these events in the traces.

The template *response (A, B)* specify that if the event A is present in the trace, then the event B has to be present after A. Event B doesn't have to present right after event A, but it can be present in any time in the trace after, as well as there can be another event A present between first event A and event B.

The template *precedence (A, B)* specify that if the event A is present, then the event B has to be present before event A. Just like in previous response template, we don't care about location, therefor event B can be anywhere before A, as well as, there can be another A after B and first event A.

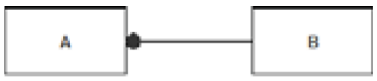
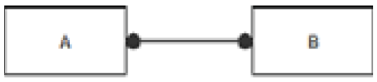
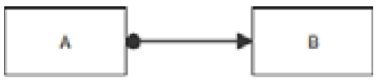
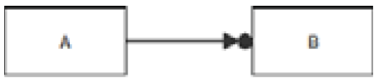
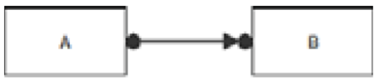
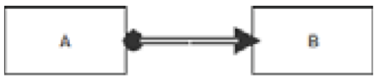
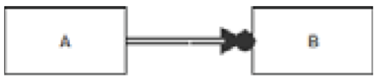
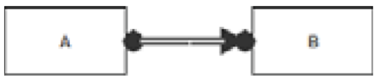
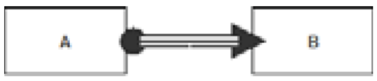
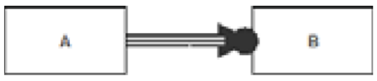
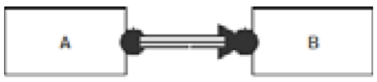
The template *succession (A, B)* is combination or response and precedence, so we can say it specify bi-directional execution of two events. So this template specifies if the event A is present then the event B have to be present both before and after event A.

Templates *alternate precedence*, *alternate response* and *alternate succession* strengthen three templates mentioned above in the way that events A and B have to alternate. In other words if we have for example *alternate response (A,B)* means that if event A occurs then the event B have to occur after (it doesn't have to be right after), then another event A can occur, but only after activity B not before (between first event A and event B).

Even more strict approach is when using *chain precedence*, *chain response* and *chain succession*. These templates specify that the two events A and B have to occur next to each other, e.g. *chain response(A,B)* represent that event A is followed immediately by event B.

See Table 3 for graphical notations notation and LTL semantics.

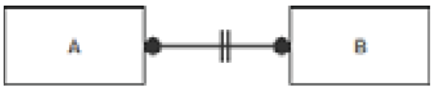
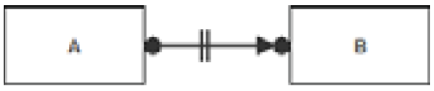
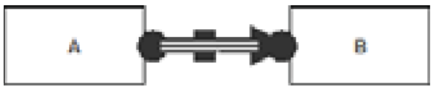
Table 3. Relation templates

name of template	***LTL semantics***	***graphical representation***
$responded\ existence(A, B)$	$\Diamond A \Rightarrow \Diamond B$	
$co\text{-}existence(A, B)$	$\Diamond A \Leftrightarrow \Diamond B$	
$response(A, B)$	$\Box(A \Rightarrow \Diamond B)$	
$precedence(A, B)$	$(\neg B\ U A) \lor \Box(\neg B)$	
$succession(A, B)$	$response(A, B) \land$ $precedence(A, B)$	
$alternate\ response(A, B)$	$\Box(A \Rightarrow \bigcirc(\neg A\ U B))$	
$alternate\ precedence(A, B)$	$precedence(A, B) \land$ $\Box(B \Rightarrow \bigcirc(precedence(A, B)))$	
$alternate\ succession(A, B)$	$alternate\ response(A, B) \land$ $alternate\ precedence(A, B)$	
$chain\ response(A, B)$	$\Box(A \Rightarrow \bigcirc B)$	
$chain\ precedence(A, B)$	$\Box(\bigcirc B \Rightarrow A)$	
$chain\ succession(A, B)$	$\Box(A \Leftrightarrow \bigcirc B)$	

3.1.3 Negation Templates

Negative Templates are negated versions of above mentioned relation templates. In Table 4 below we show LTL semantics and graphical representation of one chosen negative template from each group of templates mentioned above: existence templates, classical relation templates and chain relation templates. Figure 3 shows notations - graphical representation, for all negations templates.

Table 4. Semantics and graphical representation of some negative relation templates

name of template	***LTL semantics***	***graphical representation***
not co-existence(A, B)	$\neg(\Diamond A \wedge \Diamond B)$	
not succession(A, B)	$\Box(A \Rightarrow \neg(\Diamond B))$	
not chain succession(A, B)	$\Box(A \Rightarrow \bigcirc(\neg B))$	

If we remember *responded existence (A, B)*, we know that if event A occurs in the trace, event B has to occur too, after event B. *Not responded existence (A,B)* is then complete opposite. If event A occurs in the trace, event B can never occur. The not co-existence(A,B) specify that events A and B can never occur in the same trace. It is basically *Not responded existence (A,B)* and *Not responded existence(B,A)* applied together. *Not response (A,B)* specify that after event A event B cannot occur anymore. *Not precedence (A,B)* on the other hand specify that event A cannot occur before event B. *Not succession (A,B)* specify that B cannot be before and after any occurrence of A.

Last three not mentioned templates from Figure 3 are negations of the templates *Chain response (A, B)* and *Not chain precedence (A, B)* and *Not chain succession (A, B)*. *Not chain response (A,B)* templates specify that if event A occurs, then B should never follow A directly. *Not chain precedence (A,B)* specify that event A should never precede B directly and *Not chain succession (A,B)* is combination both of these templates.

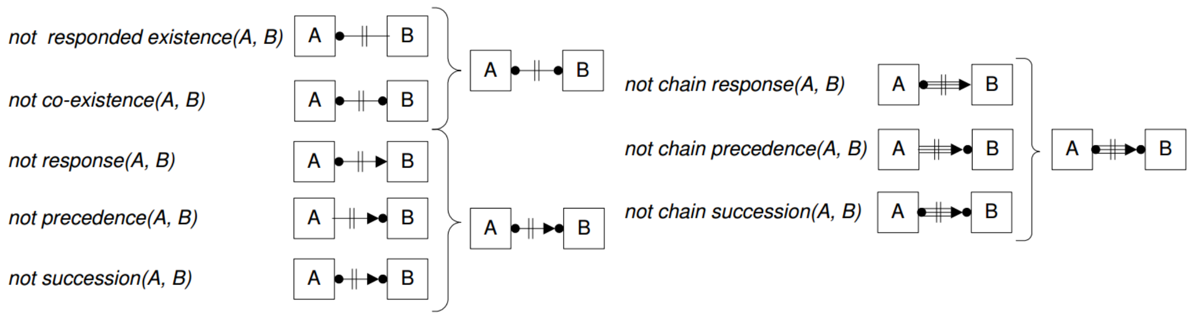


Figure 3. Notations for negation templates

3.1.4 Vacuity detection

Vacuity detection is related to Declare constraints generated by Declare templates.

As we already described in 3.1 Declare language is using the set of constraints, which must be satisfied during process executions. Kupferman and Vardi described and introduces in [14] general method for detecting vacuity.

When *Vacuity detection* is enabled, then the constraint is satisfied only when it is activated. For example constraint *Response (A, B)* is satisfied only if both A and B events are present in the trace (constraint is not trivially satisfied). If A and B are not present in the trace we consider this constraint as not satisfied in this particular trace.

On the other hand if *Vacuity detection* is not enabled, we consider constraint satisfied if it's both activated or not activated, so both trivially and not-trivially satisfied. Let us say we have a trace without presence of A and B events, our *Response (A, B)* constraint would be trivially satisfied here but not activated. Considering *Vacuity detection* is not enabled constraint would be marked as satisfied in this case.

3.2 Log representation

In process mining the important information is present in the event logs. These logs store data from the business process executions [15]. Log consists of set of *traces* where each trace represent execution of one process. If the log consists of 10 traces, then we know that this log stores information about 10 executions of the business process. Trace consists of list of events. Event in the trace represent the action of the business process model e.g. "Send invoice". Events are referred to as activities as well. Events and traces are the main structural element of the log and they both contain a set of defined attributes, when the standard are:

- **ID** – unique identifier for the element
- **Timestamp** – time and date when the element occurred
- **Name** – this is not unique attribute, it represent the name of the element, which is easily understandable to the analyst, e.g. for the trace it could be the name of the process
- **Lifecycle transition** – this represent the stage of the event's lifecycle e.g.: start, suspend, schedule, resume etc.
- **Resource** – identifier of the resource, which started the event

Events and traces can hold multiple attributes not just these standard ones mentioned above. Example attribute is cost, which can have another embedded sub-attributes like amount and currency.

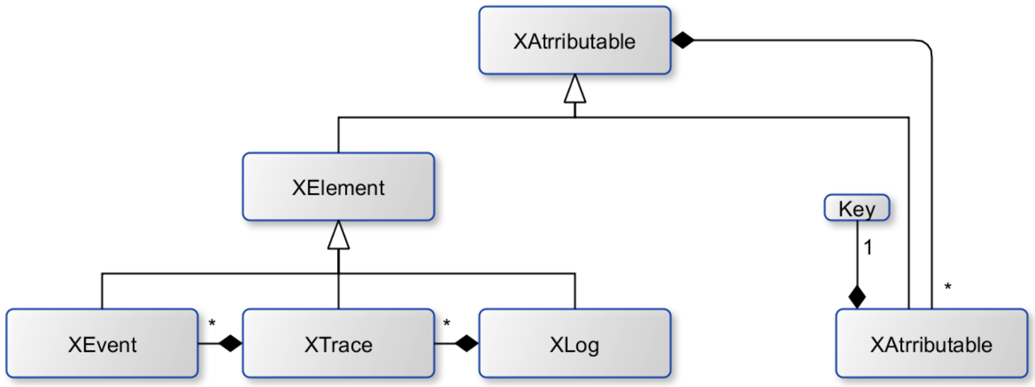


Figure 4. Structural elements of process mining event logs [15]

Traces in the log are usually not ordered, on the other hand the events in the trace are usually ordered, since they represent the process execution flow in time [15]. If there are two traces with the same order of the events in the log, they are identical.

With BPM environment there are 2 usually used formats: MXML and XES standards. Although there are another log formats defined by different vendors, in this thesis we use only MXML and XES logs as well as they are used in the process mining tools like ProM.

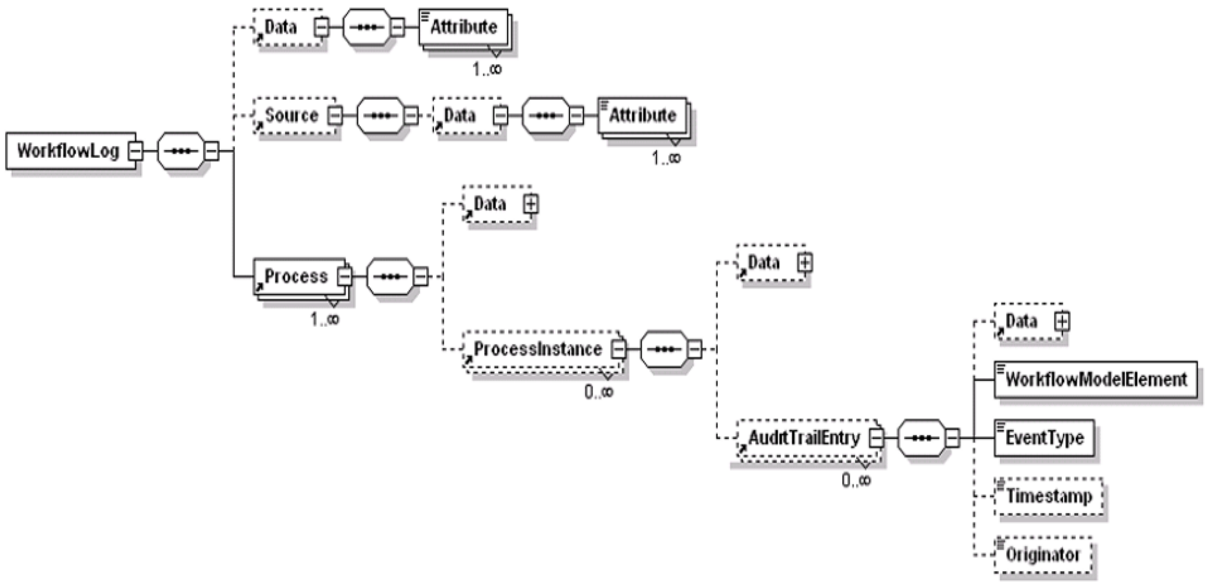
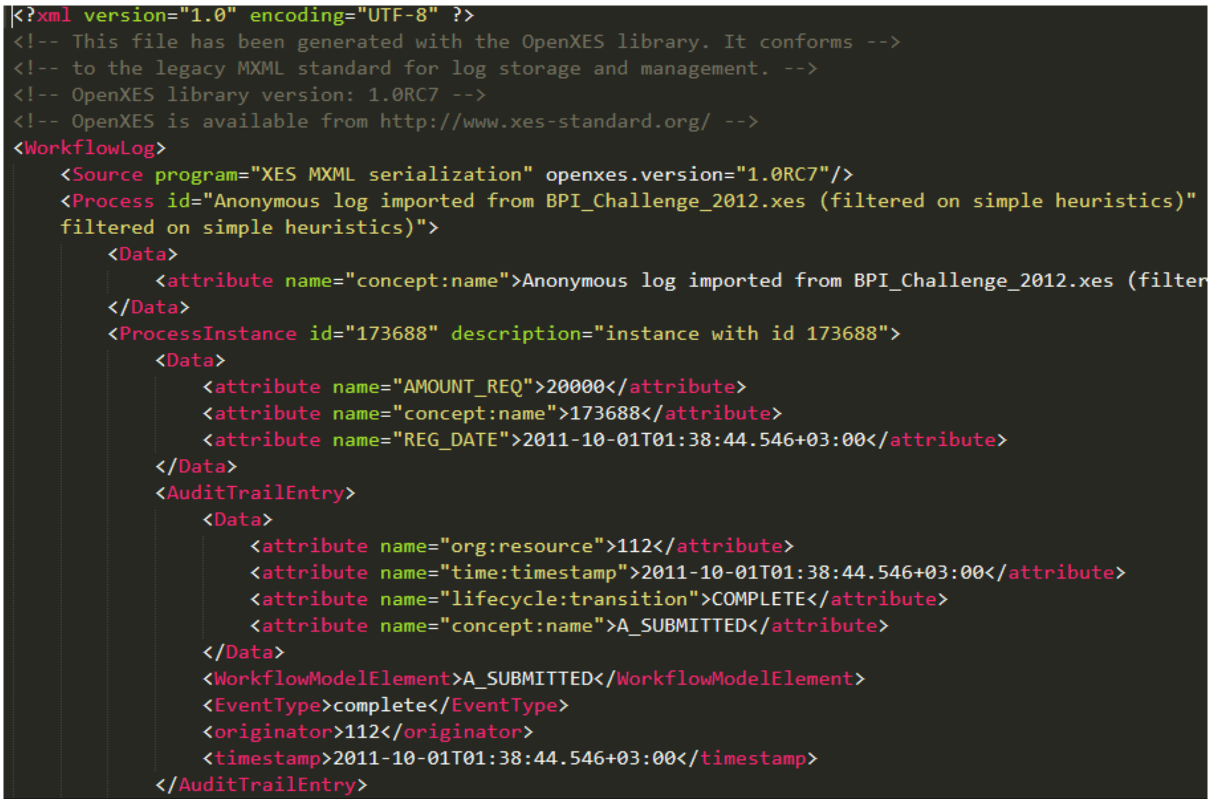


Figure 5. MXML event log file format

MXML (Minig eXtensible Markup Language) is XML-based declarative markup language format used for storing business process event logs. This language appeared in 2003 for the first time and it was adopted by ProM tools, which represent process mining community, as standard format. Figure 5 shows MXML event log file format consisting of the main “WorkFlowLog” root element, which is parent to all other elements like: Process, ProcessInstance, AuditTraitEntry etc.

ProcessInstance represent one execution of the process, therefore it is one trace in the log. AuditTrailEntry represents one event or activity in the log, where we can find multiple another stored parameters like Data, EventType, Timestamp etc. Figure 6 represents small example of MXML log.



```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- This file has been generated with the OpenXES library. It conforms -->
<!-- to the legacy MXML standard for log storage and management. -->
<!-- OpenXES library version: 1.0RC7 -->
<!-- OpenXES is available from http://www.xes-standard.org/ -->
<WorkflowLog>
    <Source program="XES MXML serialization" openxes.version="1.0RC7"/>
    <Process id="Anonymous log imported from BPI_Challenge_2012.xes (filtered on simple heuristics)"
    filtered on simple heuristics)">
        <Data>
            <attribute name="concept:name">Anonymous log imported from BPI_Challenge_2012.xes (filter
        </Data>
        <ProcessInstance id="173688" description="instance with id 173688">
            <Data>
                <attribute name="AMOUNT_REQ">20000</attribute>
                <attribute name="concept:name">173688</attribute>
                <attribute name="REG_DATE">2011-10-01T01:38:44.546+03:00</attribute>
            </Data>
            <AuditTrailEntry>
                <Data>
                    <attribute name="org:resource">112</attribute>
                    <attribute name="time:timestamp">2011-10-01T01:38:44.546+03:00</attribute>
                    <attribute name="lifecycle:transition">COMPLETE</attribute>
                    <attribute name="concept:name">A_SUBMITTED</attribute>
                </Data>
                <WorkflowModelElement>A_SUBMITTED</WorkflowModelElement>
                <EventType>complete</EventType>
                <originator>112</originator>
                <timestamp>2011-10-01T01:38:44.546+03:00</timestamp>
            </AuditTrailEntry>
```

Figure 6. Example of MXML log

Another important log format is XES. XES stands for eXtensible Event Stream and it is successor of MXML format. It is an open XML-based standard format for managing and storing event logs [16].

XES was designed for process mining as its main purpose, but authors also made it suitable for statistical analysis and data mining. In 2010, the XES was selected as a standard format for logging events by IEEE Task Force on Process Mining [17].

XES has its open-source reference implementation library called OpenXES [15].

During development of XES format, there were couple of important goals fulfilled:

- **Simplicity** – XES was designed to easily readable for humans as well as easily to be parsed.
- **Flexibility** – XES standard should represent general standard for even log data, where we can capture logs from any background.
- **Extensibility** – XES standard should be easily extended in the future. This extension should be as transparent as possible.
- **Expressivity** – XES format should allow to as little loss of information as possible., while allowing to attach human-interpretable semantics.

Since XES strives to be generic log format, only common elements, which are identifiable by any setting are defined explicitly by the standard.

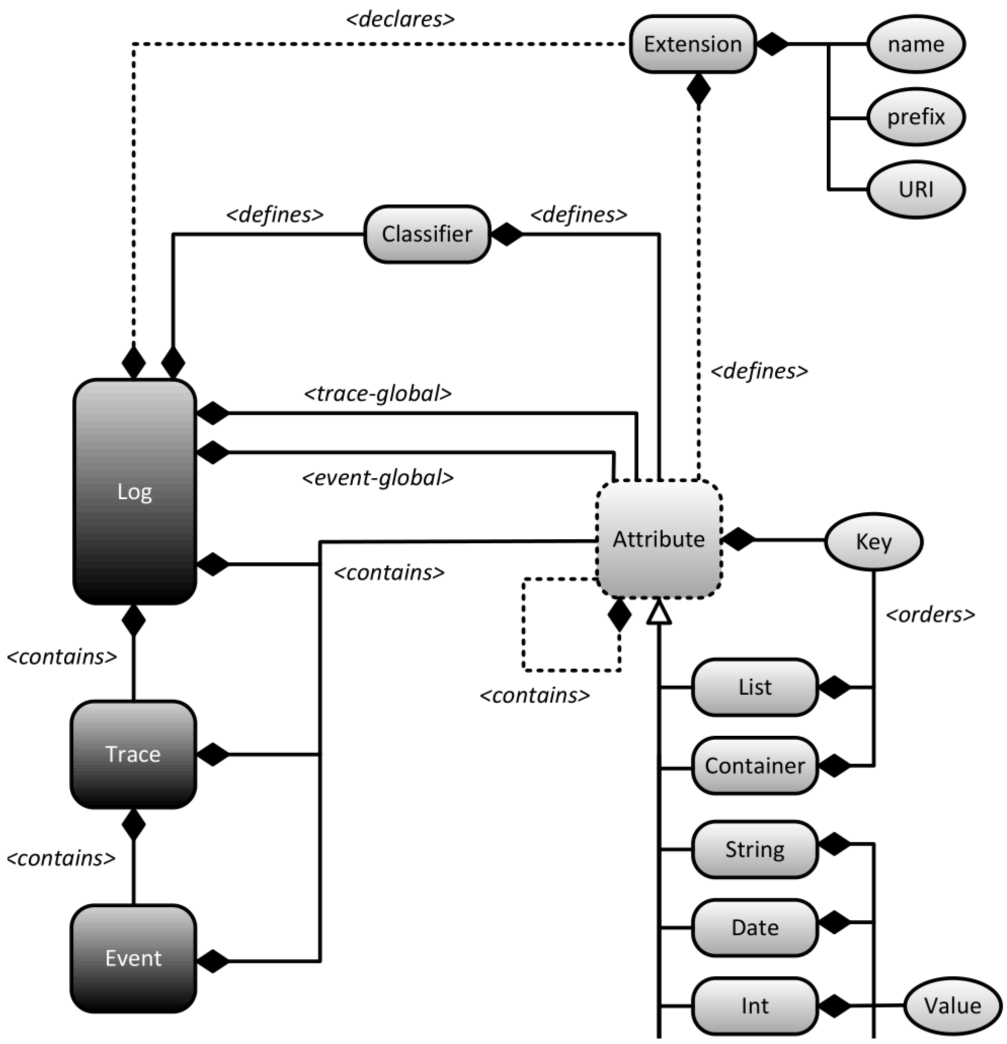


Figure 7. XES meta-model structure

Structure of XES is similar to MXML, based on XML, only that the elements have completely different representation. Looking at the Figure 7, the root element is called “Log”. Under this element we can find traces stored under element “Trace”. Event object represent event in the trace. These main three objects don’t hold any actual information, because all the actual information and detail about these objects is stored in Attributes. All of these objects can have multiple attributes.

Figure 8 shows example of XES log.

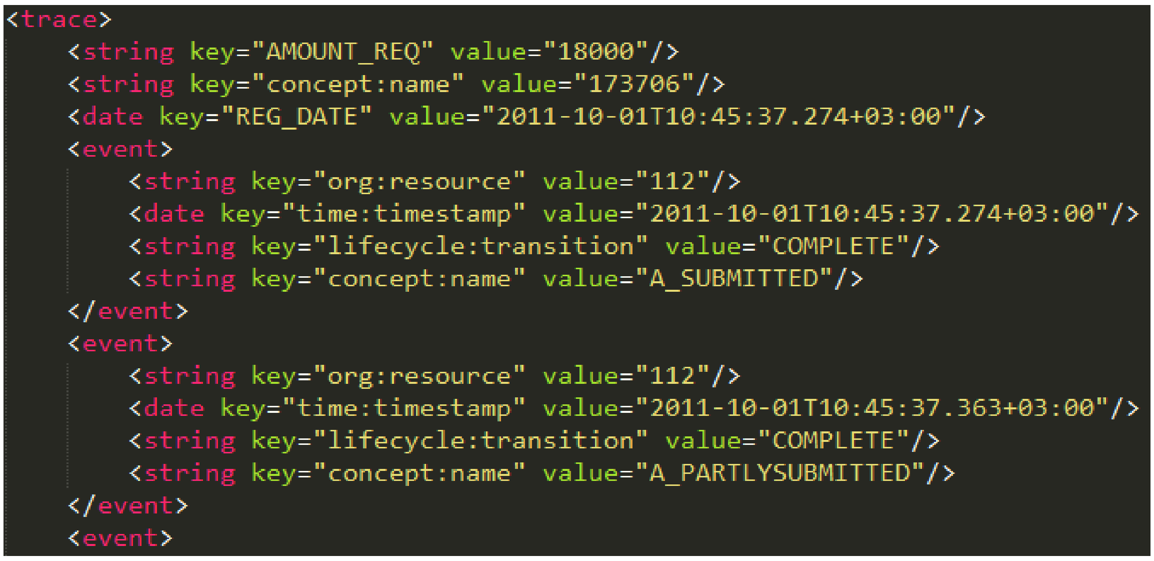

```
<trace>
    <string key="AMOUNT_REQ" value="18000"/>
    <string key="concept:name" value="173706"/>
    <date key="REG_DATE" value="2011-10-01T10:45:37.274+03:00"/>
    <event>
        <string key="org:resource" value="112"/>
        <date key="time:timestamp" value="2011-10-01T10:45:37.274+03:00"/>
        <string key="lifecycle:transition" value="COMPLETE"/>
        <string key="concept:name" value="A_SUBMITTED"/>
    </event>
    <event>
        <string key="org:resource" value="112"/>
        <date key="time:timestamp" value="2011-10-01T10:45:37.363+03:00"/>
        <string key="lifecycle:transition" value="COMPLETE"/>
        <string key="concept:name" value="A_PARTLYSUBMITTED"/>
    </event>
    <event>
```

Figure 8. XES log example

Attributes represent key-value pairs, where the Key is unique identifier within the parent of the attribute. As can be seen on the Figure 7, commonly used attributes can be Lists, Constrainers, Strings, Date, Int values etc. On figure 8 we can see the attributes like "*lifecycle:transitation*" with value "*COMPLETE*".As mentioned above, one of the main goal of XES is high flexibility, which allow us to used also nested attributes for specific dimension or perspective e.g. Trace.

3.3 Classification

As mentioned in previous sections the deviance mining produces function, called the "classifier", which can classify the trace and label it accordingly - normal or deviant as well we can use it for evaluation of the labelling accuracy of already labelled data.

Classification technique is an approach of building classification models from training input dataset. There are multiple classifiers for solving classification problem like decision tree classifiers, support vector machines, k closest neighbour classifier, rule-based classifier, naïve Bayes classifiers etc. Each classification technique adopt learning algorithm in order to construct the model, which fits the best between a class label of the input data and between the attribute set. Main purpose of each classification technique is therefor build predictive model, which can predict accurately class labels for other test unknown records.

In this subsection we will focus on chosen 3 classifiers, which we used in this thesis: decision tree classifier, k-NN classifier and Neural networks.

3.3.1 Decision tree

Decision tree classifier is a very simple and very widely used technique for classification. It tries to solve classification problem straightforward. Decisions tree learning is one of the methods most commonly used nowadays in the fields of data mining and machine learning and it is applied to broad range of tasks. Decision tree belongs to hierarchical supervised learning models. In machine learning, there are two types of decision trees: regression trees and classification trees. Regression tree analysis is when predict outcome represents real number, where we are estimating or predicting responses e.g. temperature, price of the car, etc. Classification tree analysis is when predict outcome can be class where our data belongs, it means we identify group membership – the class. For example it can help us to decide whether new observation belongs to class ’car’ or ’plane’. In our case the main focus is on classification trees.

Decisions trees can be used for multiclass classification. A decision tree is a flow-chart-like structure, where each non-leaf node indicate a test on an attribute, each branch represents the outcome of this test, and each leaf represents class label. The top-most node in the tree represents the root node.

Given the following data:

Day	***Outlook***	***Temperature***	***Humidity***	***Wind***	***Play Tennis***
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

We can construct the following decision tree (see Figure 9).

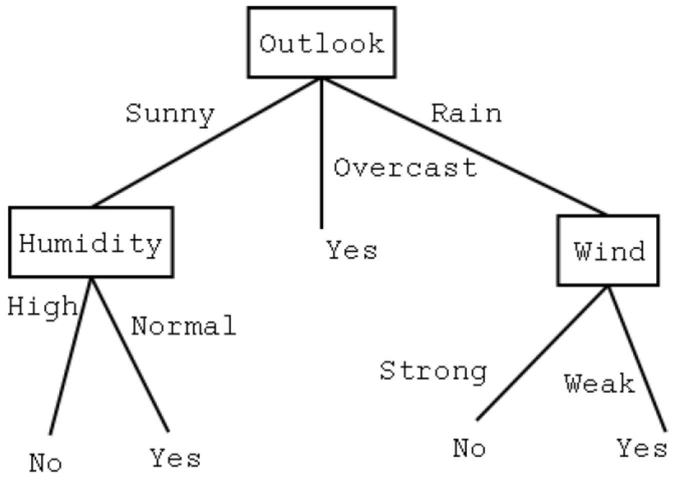


Figure 9. Example of decision tree structure.

Trees can be represent by if-then rules so they are easier to read for humans. The representation of the tree is following:

- Each internal node tests an attribute (Outlook, Humidity, Wind)
- Each branch represent the value of the attribute (High, Normal, Sunny, Rain, etc.)
- Each leaf node represents the classification assignment (Play Tennis: Yes, No)

Very simplistic description of classification by decision three be following steps:

1) Pick the best attribute (splitting our data roughly in half - Outlook)
2) Ask question (e.g. What Outlook?)
3) Follow the answer path (Sunny or Rainy or Overcast)
4) Go to 1

We continue doing the steps 1 to 4 until we get to the answer, leaf node, in the Figure 9 either YES or NO.

3.3.2 k-NN

k-NN stands for the k-Nearest Neighbours algorithm, which is a non-parametric classification and regression method. It’s one of the simplest of all machine learning algorithms.

Since k-NN can be used for both classification and regression, our focus in this thesis is classification. k-NN has on the input instance of training samples and try to produce class label on the output for unlabelled data.

k-NN is an non parametric lazy learning algorithm. Non parametric means that k-NN doesn’t make any assumption of the input data distribution. Lazy mean that that k-NN does not use the data in the input – training data to do generalization. Differently said, there is very minimal training phase, which mean that this phase is very fast.

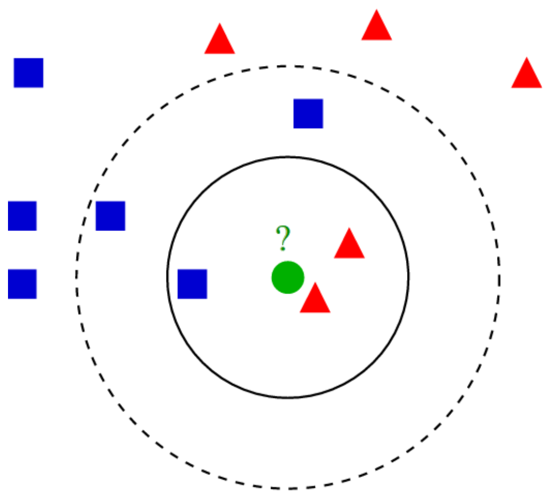


Figure 10. Example of k-NN classification

The new object is classified by the vote of the *k* closest neighbours, which are present in feature space.

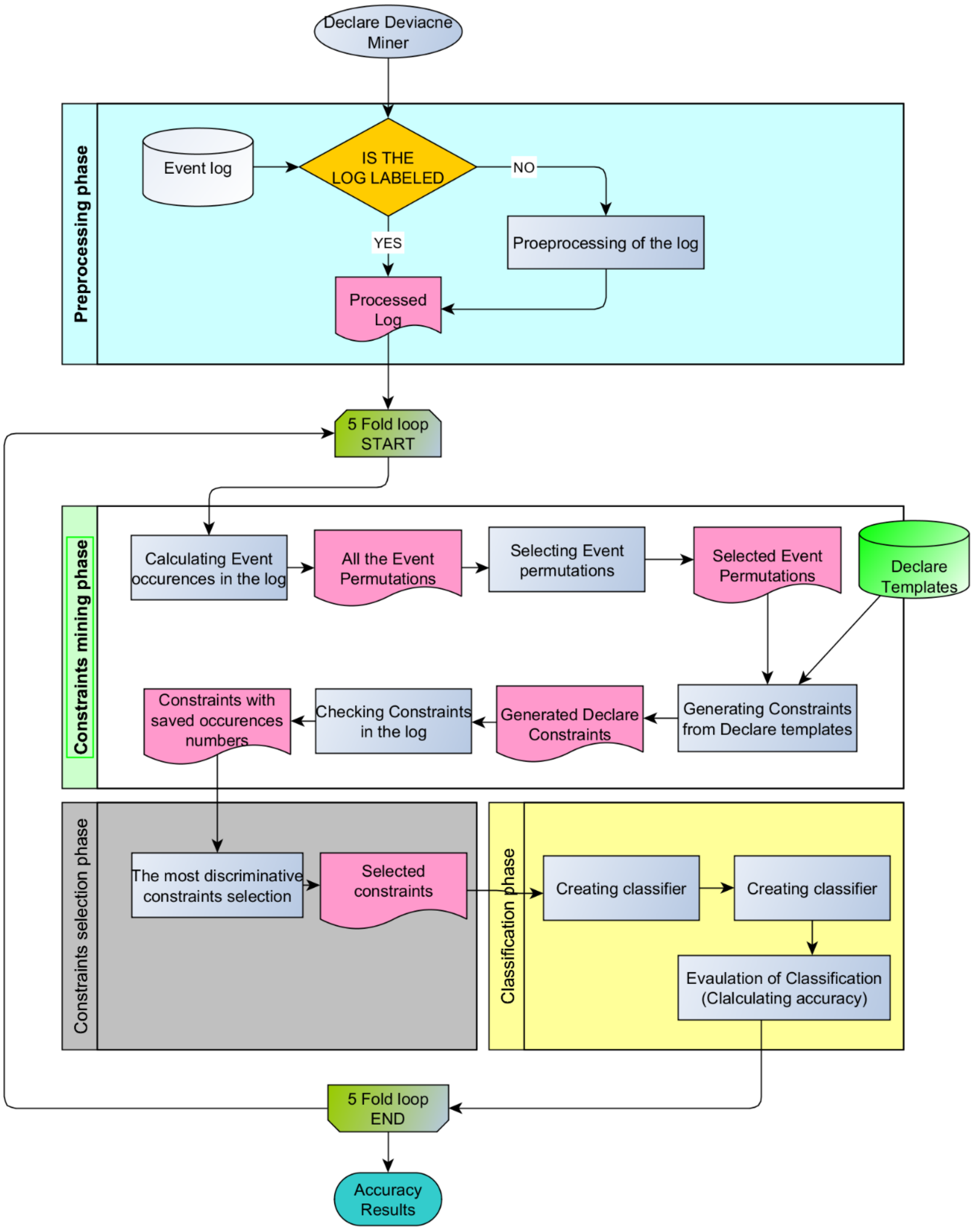
The simplest scenario is when *k = 1*. We look at the closest neighbour to our new object and just assign the class of that single nearest neighbour.

If we look at the Figure 10 then we can see the example of k-NN classification if new object if k=3 or k=5. If we have 2 classes only, then the k is odd number usually. We can see that if the k=3, 2 of the closest neighbour are from first class and 1 is from second class. Therefor we assign the first class to our object. On the other hand if k=5 we have 2 neighbours from first class and 3 from second class. Therefor we assign second class to our new object.

4 Proposed approach

In this chapter, we specify and describe our proposed approach, how are instruments, which were mentioned in previous section connected together.

Following figure represents flow chart of our proposed algorithm.



Declare Deviacne Miner
Preprocessing phase
Event log
IS THE LOG LABELED
NO
YES
Proeprocessing of the log
Processed Log
5 Fold loop START
Constraints mining phase
Calculating Event occurences in the log
All the Event Permutations
Selecting Event permutations
Selected Event Permutations
Declare Templates
Generating Constraints from Declare templates
Generated Declare Constraints
Checking Constraints in the log
Constraints with saved occurences numbers
Constraints selection phase
The most discriminative constraints selection
Selected constraints
Classification phase
Creating classifier
Creating classifier
Evaulation of Classification (Clalculating accuracy)
5 Fold loop END
Accuracy Results

Figure 11. Proposed approach flow chart

As mentioned in second section, deviance mining algorithms consist in general out of two main steps:

1. Patterns extractions step
2. Classification step

For example in frequent features mining frequent features are extracted from the log and then later on they are used to build the classifier in the second step.

In our approach we could say that our main framework consists of three main steps, considering that "Preprocessing phase" is excluded from our main algorithm:

1. Declare patterns extraction step
2. Declare Patterns selection step
3. Classification step

Later in this section we will describe the entire approach in more detail, it is just crucial to mention that we have two logs, one is the train log and one is the test log. Declare pattern extraction step is done with the train log and well as Declare Patterns selection step. The third classification step is done on the test log. Shortly said, we work with the two logs, first one is used for pattern extraction and building the classifier and the second one is used for classification itself to see how accurate our classifier is.

It is important to note that in addition to these three main steps which creates our core algorithm, there is one pre-step which is preprocessing of the events logs so they are prepared to run on our algorithm.

Our core algorithm require, aside from the event log, another 4 input parameters to run:

- Event support
- Constraint support (min support)
- Coverage
- Vacuity enabled

We will describe these input parameters as well as the proposed aproach in following subsections in more detail.

4.1 Preprocessing of the event logs

In this first stage we decided to propose techniques to preprocess event logs so they are ready to run on our core algorithm.

This stage consists of two main steps:

- Log labelling
- Log partitioning

Log labelling

Firstly we need to label the traces in the log accordingly (deviant or normal) based on different criteria.

Here is the variety of possibilities since the criterion can be temporal or not temporal. E.g. we can label the traces (which represents process execution) according to process execution time so deviant cases represent slow process execution and normal case represent expected process execution times. Another approach is to label the traces according to their outcome attribute (e.g. "invoice failed to send" or "customer didn't receive the invoice") which represent the cases with successful and failing outcomes. Another labelling can be done based on different data attributes like "Age of the patient", "Diagnose code" (e.g. patient suffered from a given cancer or not). We can also label the traces according to expected outcome of the labelling. If we want to have balance labelling in the end (similar amount of deviant and normal cases) we choose the condition for the data attribute its mean value. For example if "Age of the patient" is lower than 50 we label the trace as normal, otherwise we label the trace and deviant. Because the "Age of the patient" parameters has mean value around 50 in the log we get balanced labelling in the end. However if we change the condition to higher or lower number than 50, we will get unbalanced labelling, with either more normal or deviant cases.

Log partitioning

In second step of preprocessing, the labeled log is randomly divided into five training and five testing event logs using 80:20 ratio.

The usage of this 5 fold cross-validation format is because of getting better accuracy during classification as well as this is the classical approach

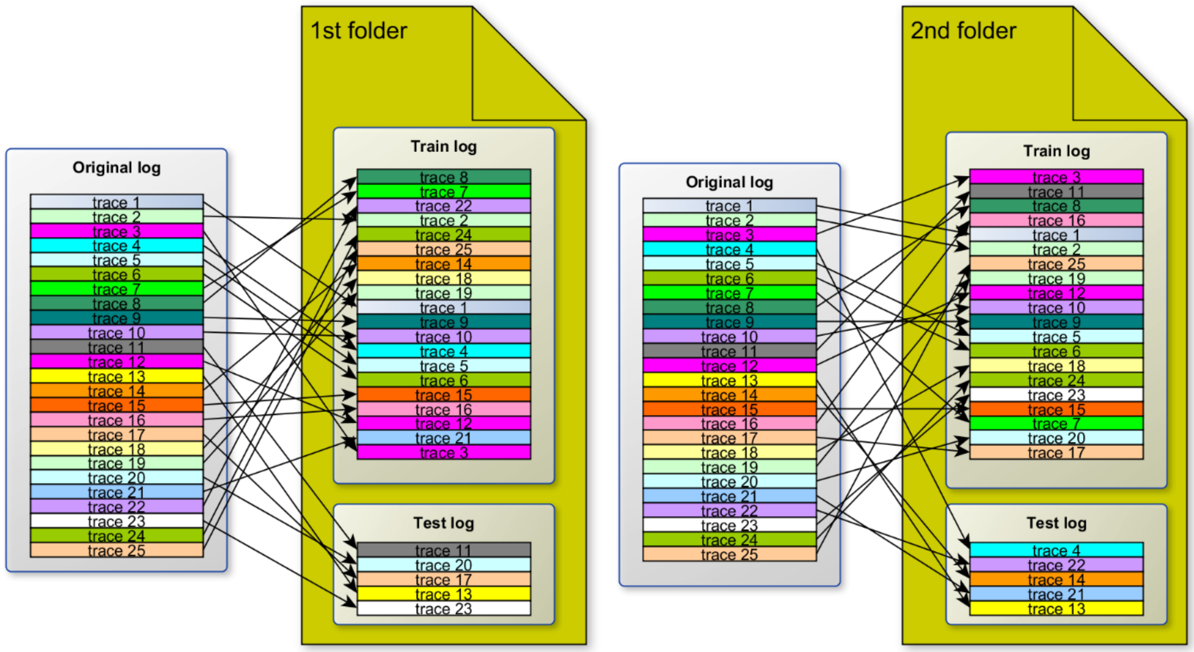


Figure 12. Explanatory figure of 5-fold, creating first 2 folds.

Figure 12 above shows how folds are being created from the log of the size of 25 traces. Firstly, we shuffle all the traces in the log randomly, for each fold, therefore we have completely different ordering of traces for each fold. Then we take 80% of the traces and save them into train log and the rest 20% are saved as a test log.

4.2 Declare constraints mining

Declare constraints mining represents one of the main parts of our core algorithm. Following figure - Figure 13 shows the dDclare constraints mining phase step by step.

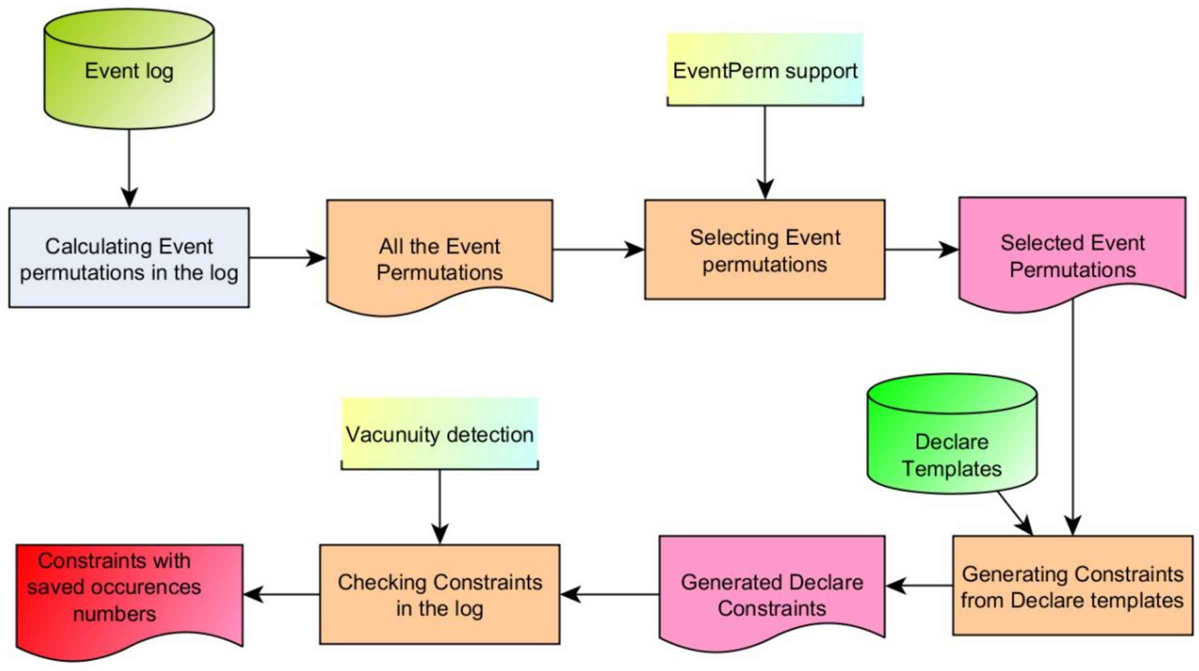

Event log
EventPerm support
Calculating Event permutations in the log
All the Event Permutations
Selecting Event permutations
Selected Event Permutations
Declare Templates
Vacuunity detection
Constraints with saved occurences numbers
Checking Constraints in the log
Generated Declare Constraints
Generating Constraints from Declare templates

Figure 13. Declare constraints mining approach

For mining Declare constraints we need to have (1) a set *T* of Declare templates, (2) the Event log *L* and the (3) input parameter *EventPermutation support*.

Firstly we mine set of unique events $E = \{e_1...e_n\}$ from the event log as well as event permutation $Ep = \{ep_1...ep_n\}$. Event permutation represent combination of the unique events in all the possible ways, so in the end it consists of unique events as well as of pairs of events which can be found in the traces of logs. We save occurrences numbers of these event permutation in the log. So example of Event occurrence Ep would look like:

{[Send_invoice]=10, [Deliver_package]=8, [Contact_customer]=3, [Send_invoice, Deliver_package]=4 ...},

where we can see that first three event permutations are single events and the fourth one is pair consisting first and second single events in this even permutations list.

After we mine all the event permutation, according to the size of the log and size of the unique events in the log we can get really high numbers of event permutations. Therefore there is need to select some amount of event permutations in order to lower the high memory consumption because of thousands of permutations, since in the following steps we are generating constraints using these permutations and we can generate multiple times more constraints from event permutations. We are talking about thousands and thousands of constraints so there was a need to come up with some heuristics in order to lower the resources exploitation.

Because of this we introduced the *EventPermutation support* parameter. Let us consider the size of the event log *S*. Size of the event log represent number of traces – process executions in the log. Let us consider that *EventPermutation support* parameter S is set to 5. This represents, that we select event permutation which are found only in 5 and more traces in the log. If we look at example of event permutation Ep above we would select only first two event permutations out of those four:

SelectedEp={[Send_invoice]=10, [Deliver_package]=8}

Once we have the list of selected event permutation, we go into the generating of the Declare constraints part of our algorithm.

In this part we have as input: Declare templates and selected event permutations (*Figure 4*). We can select all the Declare templates or just chose some of them (e.g. Absence 2 or Chain Precedence). To get higher accuracy results and more discriminative constraints it makes sense to use all the possible templates. We generate constraints for each template and each event permutation. Let us take a look at the SelectedEp list of selected event permutations above. We take the first event permutation *Send_invoice*, which is only one event (It is not a pair) therefore we can only generate Declare Constraints from the Templates which require one parameter e.g. *Absence*. We cannot generate constraint using the template which require 2 parameters like *Alternate precedence* for example because this particular template require two parameters.

If we consider to have only 3 templates like *Absence, Alternate_Precedence* and *Exactly1* on the input of our "Generating Declare constraints" step and we consider *SelectedEp* variable above as our event permutations input, then the output of this step would these four Declare constraints:

- *Absence (Send_invoice)*
- *Exactly1 (Send_invoice)*
- *Absence (Deliver_package)*
- *Exactly1 (Deliver_package)*

Once we have list of instantiated Declare constraints we proceed to checking their satisfaction in the traces of the event log. For this we used 3rd party existing ProM plugin DeclareAnalyzer[3]. On the input of the declare analyser we have the test log, our generated Declare constraints and input variable "Vacuity detection enabled" which is either true or false. (Enabled or Not Enabled).

It's crucial to mention that we check the constraints satisfaction in the positive traces first and then in negative. Therefore we divide log into two logs, one only with normal cases and one with deviant cases. Then we call Declare analyser twice on these two logs with the same constraints.

So again the input of Declare Analyzer is our train log (either normal or deviant cases), list of constraints to check and *Vacuity detection* value and the output is our constraints list with occurrences/satisfaction in traces.

The output we get from this step would look like this:

- *Absence (Send_invoice) {165, 174} {123, 567}*
- *Exactly1 (Send_invoice) {76, 98, 105, 140 } {54 55, 59}*
- *Absence (Deliver_package) {88, 89, 107, 143 } {125, 542}*
- *Exactly1 (Deliver_package) {165, 174} {}*

As we can see constraint *Absence (Send_invoice)* is satisfied in normal traces 165 and 174 and is satisfied in deviant traces 123 and 567, on the other hand constraint *Exactly1 (Deliver_package)* is not satisfied in any deviant cases, and it is satisfied in two normal traces.

[3] https://svn.win.tue.nl/repos/prom/Packages/DeclareAnalyzer/Trunk

4.3 Declare constraints selection

Selection of the most discriminative constraints is very important step in our core algorithm. We had to come up with heuristics how to define, which constraints are the most discriminative and which are not least discriminative or not discriminative.

Following figure shows Declare constraints selection algorithm as we proposed it.

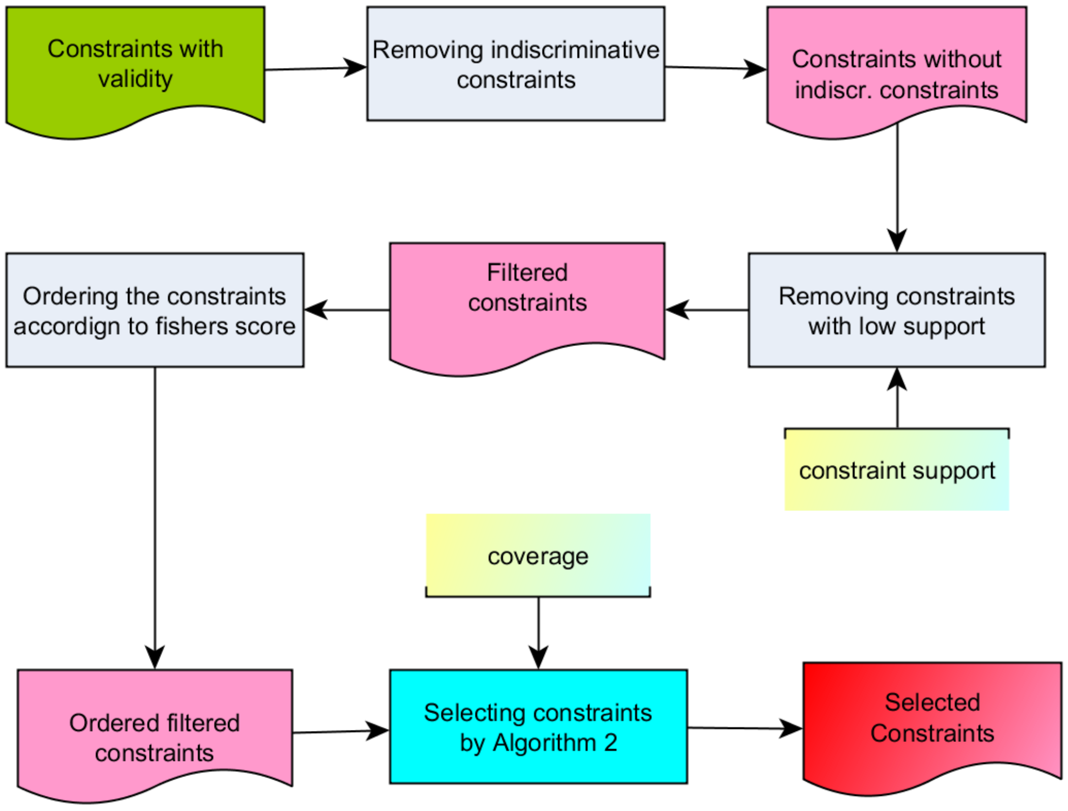

Constraints with validity
Removing indiscriminative constraints
Constraints without indiscr. constraints
Ordering the constraints accordign to fishers score
Filtered constraints
Removing constraints with low support
constraint support
coverage
Ordered filtered constraints
Selecting constraints by Algorithm 2
Selected Constraints

Figure 14. Declare constraint selection

If we look at the constraints as two value objects (negative and positive), then logically we assume that if constraint is negative, it is found in more deviant than normal cases and vice versa for positive constraint (the constraint is satisfied in more normal than deviant cases) we can say that the constraint is indiscriminative if it is satisfied in the same amount of positive and same amount of deviant cases.

Selection of the constraints is then in the following steps:

- **Removing the indiscriminative constraints** – as mentioned in the paragraph above, in this step we remove the constraints, which are satisfied in the same amount of normal and same amount of deviant cases, therefore we cannot say, that this constraints is positive or negative. For example the constraint *Absence (Send_invoice) {165, 174} {123, 567}* would be removed since it is satisfied in the two normal (traces number 165 and 174) and two deviant traces (123 and 567).

- **Removing constraints with low support** – in this part we need to have input variable *constraint support* or as in other papers mentioned in second section, *min. support*. Let us assume that the log size is 10, which means that the log consists of 10 process executions = traces. If the *constraint support*'s value is number 5, it means that all the constraints which were satisfied in the less than 5 traces, will be removed. Above mentioned constraint *Exactly1 (Deliver_package) {165, 174} {}* would be then removed from our constraints list, because it is satisfied only in two traces (2 normal and 0 deviant traces).
- **Main selection algorithm** – we consider previous two steps as "filtering steps", main selection algorithm is actual algorithm for choosing the constraints. We will describe it in more details below.

Algorithm 1 Selecting of the most discriminative Declare constraints

Procedure: Constraints selection

Inputs:

F: A set of filtered Declare constraints

TDB: Trace database

δ: Coverage threshold

Output:

Fs: A selected set of Declare constraints

```
1:   Calculate the fisher's score for all the Declare constraints in F;
2:   Sort Declare constraints in F in decreasing order of Fisher score;
2:   Start with the first constraint f0 in F;
3:   while (true)
4:     Find the next pattern f;
5:     if  f covers at least one trace in TDB
6:         Fs = Fs U {f};
7:     F = F - {f};
8:     if a trace S in TDB is covered δ times
9:         TDB = TDB – {S};
10:    if all traces are covered δ times or F = ∅
11:        break;
12:    return Fs;
```

As can be seen above Algorithm 1, for selection of the most discriminative constraints, is using *coverage threshold*, *TDB* trace database and list *F* of selected constraints as input parameters.

Firstly we need to evaluate the discriminativeness of the constraint for what we are using the score evaluation [18]. We calculate the fishers score for each of our filtered constraint using the following formula:

$$Fr = \sum_{n=0}^{\infty} \frac{\sum_{i=1}^{c} (\mu_i - \mu)^2}{\sum_{i=1}^{c} n_i \sigma_i^2}$$

where n_i represent the size of data in the class *i*, μ_i is the standard mean of constraint in the class *i*, σ_i is the standard deviant of the constraint in the class *I* and μ is the mean of the constraint's value in the whole dataset together, meaning in both classes. It's crucial to mention that the class represents the class of the trace which is either 0 or 1 (normal or deviant).

If the constraint have very similar satisfaction values within the normal class and very different values across deviant class, it has then very large Fisher score. Therefor if the occurrence frequency of our declare constraint in deviant traces is different from that in normal traces, the constraint has high Fisher's score and high discriminative value. On the other hand if the constraint has similar occurrence frequency in both normal and deviant traces, then it is not discriminative and Fisher's score has low value.

After we calculate the Fisher score for each of our constraint we order them in descending order according to the Fisher score.

Once the constraints are ordered we start taking the constraints from the top of the ordered list one by one. We take the first constraint on the top (it is the constraint with the highest Fisher score of all the constraints) and move it to selected constraints automatically. Then we check in which traces this particular constraint is satisfied. Let us say each trace has "its own coverage" value so if the constraint is satisfied in the particular trace *T* then the coverage value of this trace will increment by one. Once the coverage value of the trace equals to 5, which mean that the trace was covered 5 times by selected constraints, we remove it from our trace database *TDB*. We proceed using this approach for the rest of the constraints in the ordered list – we take the constraint from the top, mark it as selected, add to coverage value to the traces where the constraints is satisfied and we periodically check the coverage values of the traces. We continue doing this until:

A. All the constraints were used from our ordered constraints list
B. We have no traces to check (All the traces were covered 5 times by selected constraints therefor we have no more traces to check)

And in the end we get our new list of selected constraints. In Evaulation section we will describe more what kind of coverage values did we use and what input parameters are the best for getting highly discriminative Declare constraints.

4.4 Classification

The very last step of our proposed approach is classification, or in other words, evaluation of the discriminativeness of our selected Declare constraints by constructing the classifier and then classifying the test log and seeing the accuracy of labelling the traces in the test log.

A range of methods can be used to construct classifier, but we decided to use the 2 the most commonly used ones in other papers described in section 2:

- Decision Tree
- k-NN (k-Nearest neighbours)

For the classification phase we need to have our test log as well as mined selected Declare constraints as the inputs. We then take these two and represent TDB trace database in the feature space of the selected Declare constraints. TDB in feature space of selected Declare means that for each trace t in the log we have labelled sample (<c_1, c_2, … c_n>, *l*) where c1,c2 etc. represent constraints satisfaction values in the trace *t* and *l* represents label or class of the trace which is either deviant or normal.

Afterwards we train classifier from this TDB traces database. Algorithm 2 below shows and summarize our Declare constraints-base classification approach.

Algorithm 2 Declare constraints-based Classification

```
Procedure: Classifier construction
Inputs:
        Event_support: Minimum support threshold for evet permutations
        Min_support: Minimum support threshold for constraints
        $\delta$: Coverage threshold
        TDB: Trace database
Output:
        Classifier: Deviance mining classifier

1:  Let F = Mine_Declare_Constraints (TDB, event_support, min_support);
2:  Let Fs = Selected_Declare_Constraints (F, TDB, $\delta$)
3:  Transform trace database TDB into feature space of selected constraints Fs;
4:  Classifier = Train a classifier on TDB;
5:  return Classifier;
```

As mentioned above we build 3 different classifiers, from these three decision trees are natural choice since they are easily interpretable.

Let us say we have log of size 10 which means we have 10 traces in the log. Let us assume we have 70% normal and 30 % deviant therefore 7 traces are normal and 3 are deviant. From this log we can mine big list of Declare constraints (step 1 in above mentioned Algorithm 2) and using our selection algorithm we select three following constraints (step 2):

- *Absence (D) {1, 4, 7} {}*
- *Chain precedence (G, H) {1, 3, 7} {5, 8, 10}*

- *Existence (F) {1, 2 ,4, 7} {5, 8, 10}*

As we described in previous subsections "*Absence (D)* {1, 4, 7} {}" means that, this constraint in satisfied in normal traces: 1, 4 and 7 and not satisfied in any deviant trace. Afterwards we transform our trace database *TDB* into the feature space of our selected Declare constraints in the following format: (<c_1, c_2, … c_n>, *l*) for each trace t.

Feature space of our 3 selected Declare constraints in our TDB database using our example mentioned above would look like this:

trace ID	***Absence (D)***	***Chain precedence (G,H)***	***Existence (F)***	**CLASS**
1	YES	YES	YES	**NOR**
2	NO	NO	YES	**NOR**
3	NO	YES	NO	**NOR**
4	YES	NO	YES	**NOR**
5	NO	YES	YES	**DEV**
6	NO	NO	NO	**NOR**
7	YES	YES	YES	**NOR**
8	NO	YES	YES	**DEV**
9	NO	NO	NO	**NOR**
10	NO	YES	YES	**DEV**

In the last step (step 4 in the Algorithm 2 above) from these "labelled traces" or from this feature space of our selected constraints we train our classifier. In our thesis we use 3 classifiers. Figure 15 below shows example of built decision tree from the feature set above.

As can been seen from the Figure 15 the decision tree is very easy interpreted, especially in this case is very easy readable. Decision tree can be transferred into "decision rules", where in general the form of one rule would look something like this:

If *condition A* and *condition B* and *condition C* then *outcome*. In our case we can get 4 different decision rules, or paths in our decision tree:

If ***YES*** *Absence (D)* then *Normal*

If ***NO*** *Absence (D)* and **NO** *Chain Precedence (G, H)* then *Normal*

If ***NO*** *Absence (D)* and **YES** *Chain Precedence (G, H)* and **NO** *Existence (F)* then *Normal*

If ***NO*** *Absence (D)* and **YES** *Chain Precedence (G, H)* and **YES** *Existence (F)* then *Deviant*

As you can see, the analyst can look at the decision tree and understand easily when the trace is deviant. In our case constraints *Chain Precedence (G, H)* and *Existence (F)* are satisfied, but *Absence (D)* is violated.

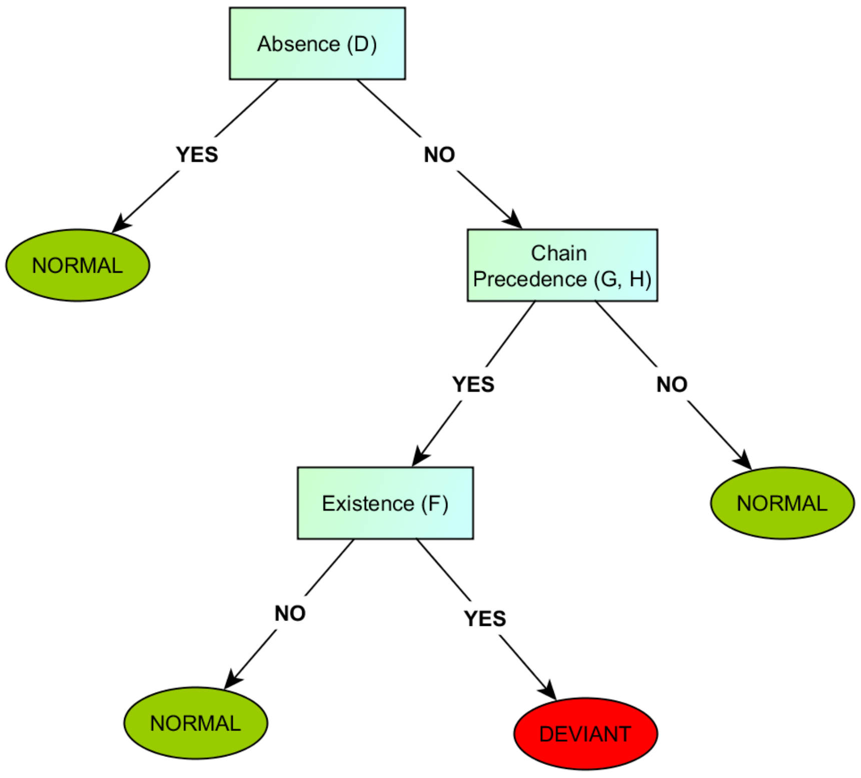


Figure 15. Example of build decision tree

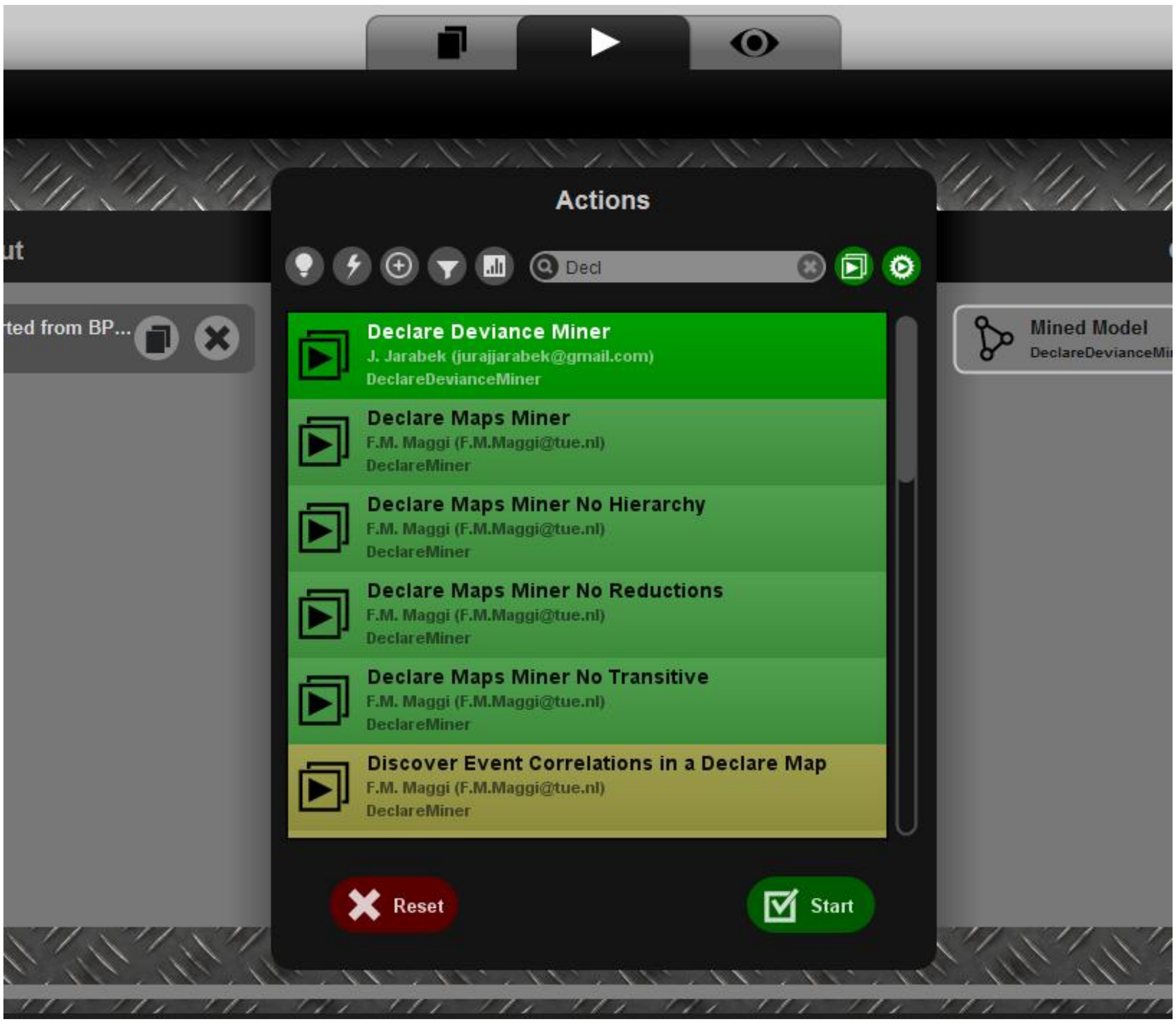
Using our classifier we assign label to the traces of the test log, in program and compare the results with the real labels of traces and calculate the labelling accuracy, the accuracy of our classifier. As well as accuracy we calculate AUC, which represents the Area Under the Curve. We will talk more about classifications and results in section 6.

5 Implementation

In this chapter, we introduce implementation of our tools as well as answer our first research question: *Is it possible to provide business analyst with understandable feedback about business process deviances?*

5.1 ProM plugin

We implemented our core algorithm as “Declare Deviance Miner” plugin in the ProM tools.



Declare Deviance Miner
J. Jarabek (jurajjarabek@gmail.com)
DeclareDevianceMiner

Declare Maps Miner
F.M. Maggi (F.M.Maggi@tue.nl)
DeclareMiner

Declare Maps Miner No Hierarchy
F.M. Maggi (F.M.Maggi@tue.nl)
DeclareMiner

Declare Maps Miner No Reductions
F.M. Maggi (F.M.Maggi@tue.nl)
DeclareMiner

Declare Maps Miner No Transitive
F.M. Maggi (F.M.Maggi@tue.nl)
DeclareMiner

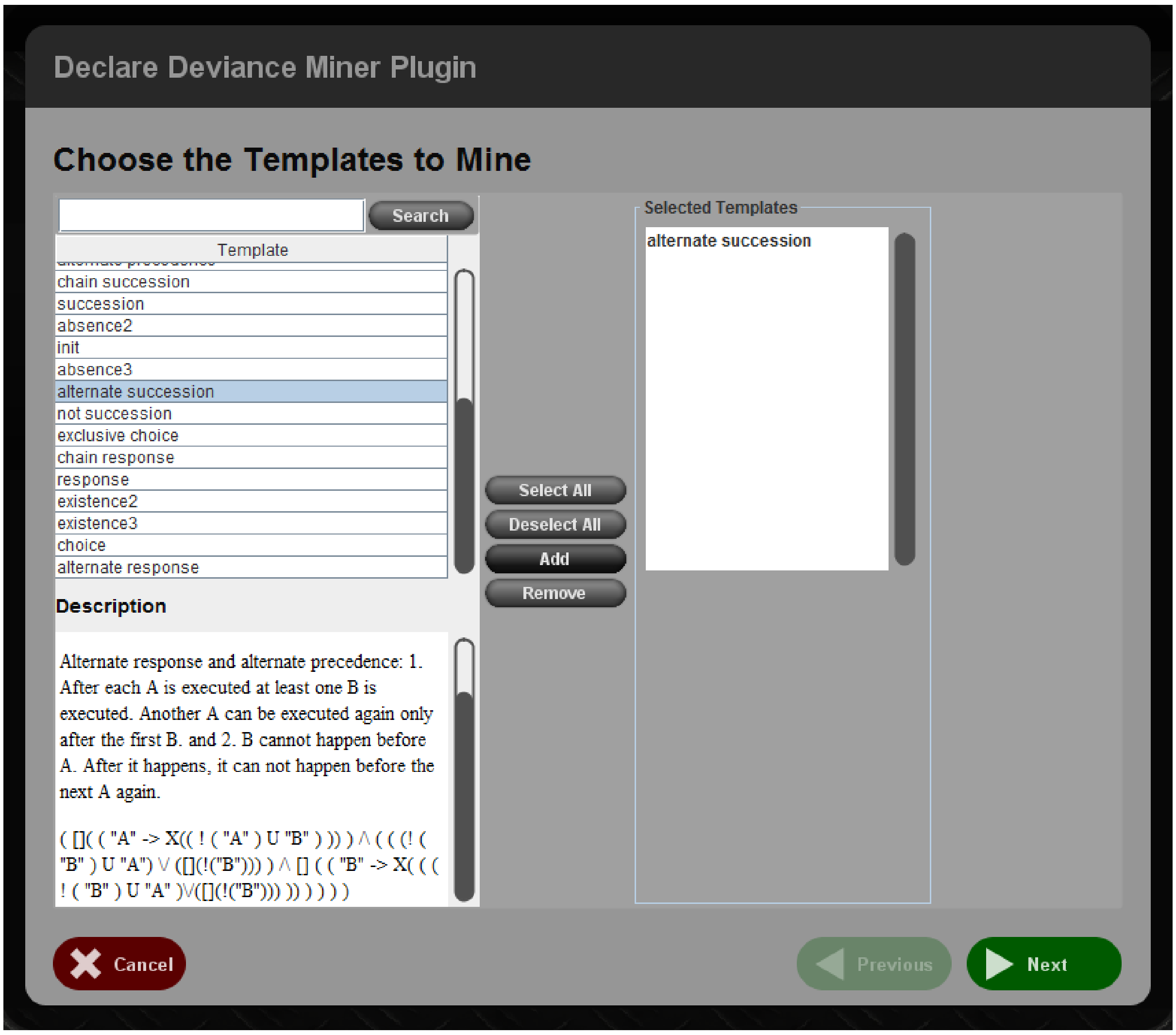
Discover Event Correlations in a Declare Map
F.M. Maggi (F.M.Maggi@tue.nl)
DeclareMiner

Figure 16. Plugins actions screen in ProM, with our plugin on the top

Our Declare Deviance Miner ProM plugin consists of three main phases mentioned in proposed approach: preprocessing phase, constraints extraction phase and constraint selection phase.

The classification phase is missing in the plugin, since we use this phase only for evaluation. As we mentioned in previous sections the output of the deviance mining should be explanatory to the analyst, so we can see the difference between deviant and normal cases. In our case, the output of the Declare Deviance Miner is set of selected Declare constraints, which hold important information, which can be shown in interpretable way to the analyst.

We tried to put effort on user friendly design, therefore the design of the plugin is very simple and explanatory.



Choose the Templates to Mine

Template

chain succession
succession
absence2
init
absence3
alternate succession
not succession
exclusive choice
chain response
response
existence2
existence3
choice
alternate response

Selected Templates

alternate succession

Description

Alternate response and alternate precedence: 1. After each A is executed at least one B is executed. Another A can be executed again only after the first B. and 2. B cannot happen before A. After it happens, it can not happen before the next A again.

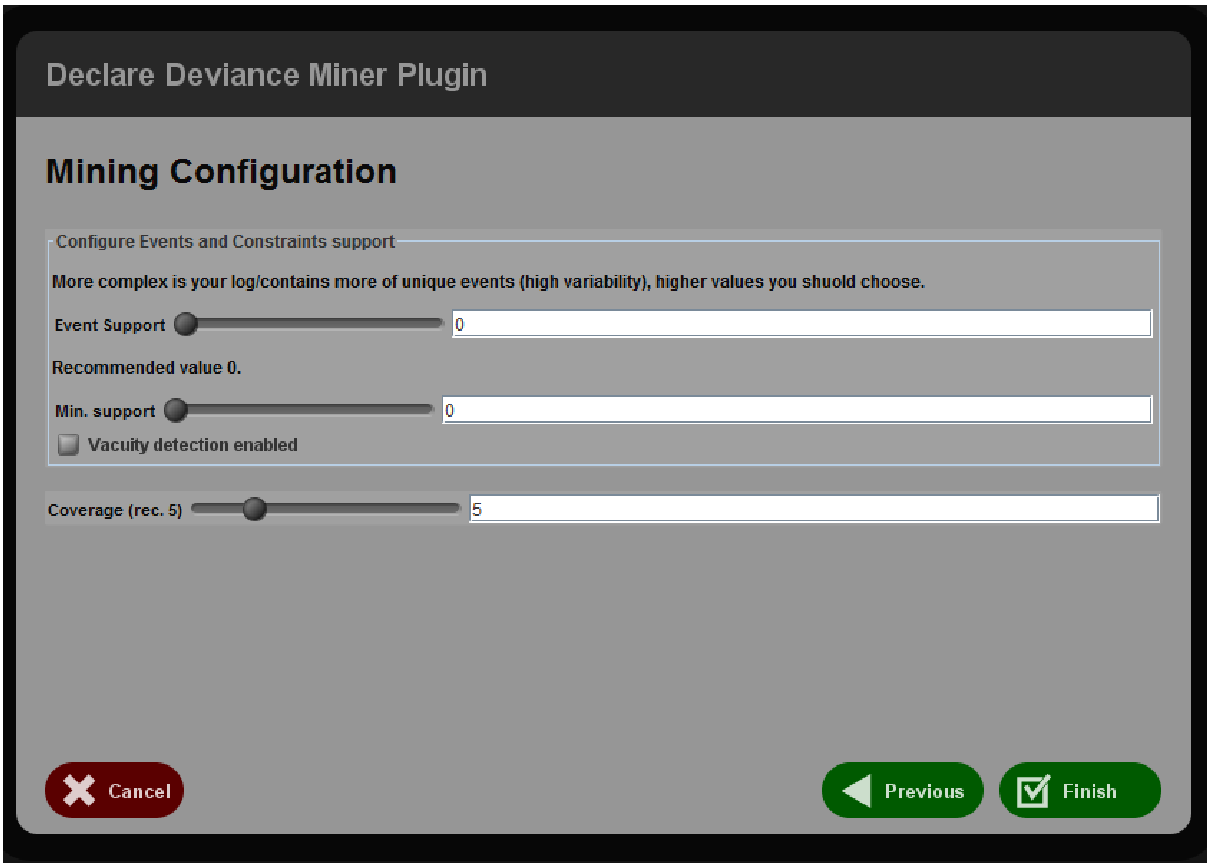
```
( []( ( "A" -> X(( ! ( "A" ) U "B" ) )) ) ∧ ( ( (! ( "B" ) U "A") ∨ ([](!("B"))) ) ∧ [] ( ( "B" -> X( ( ( ! ( "B" ) U "A" )∨([](!("B"))) )) ) ) ) )
```

Figure 17. First step in the plugin – Selecting Declare templates

After the analyst chooses our plugin (Figure 16), then the "Choose the Templates to Mine" windows is shown (see Figure 17). On this screen all Declare templates used in our implementation are shown on the left side of the screen, where analyst can choose them one by one or choose to select all of them at once by clicking on "Select All" button. There is also explanation for each Declare constraint in the left bottom box with LTL formula as well.

After selecting Declare constraints 'Mining Configuration" screen is shown (see Figure 18), where analyst chooses the mining configuration for the Plugin. As we mentioned in third section, there are only four input parameters, which our algorithm need as the input:

- **Event Support** – from 0 to 100 %
- **Min. Support or Constraint support** – from 0 to 100 %
- **Coverage** – from 1 to 20
- **Vacuity detection enabled** – true or false checkbox


Declare Deviance Miner Plugin

Mining Configuration

Configure Events and Constraints support

More complex is your log/contains more of unique events (high variability), higher values you shuold choose.

Event Support 0

Recommended value 0.

Min. support 0

Vacuity detection enabled

Coverage (rec. 5) 5

Figure 18. Mining input configruation parameters screen

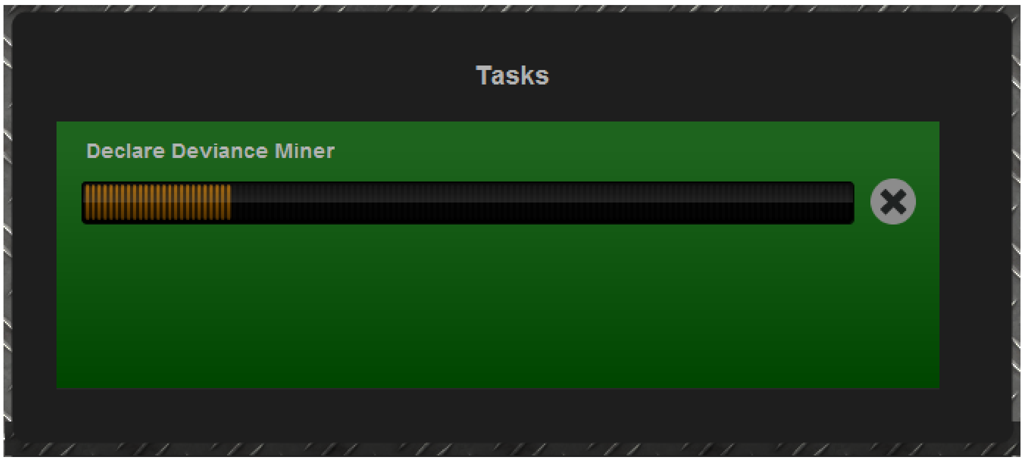


Figure 19. Loader bar showing the progress of our plugin

After the analyst clicks on finish button our core algorithm starts mining for event permutations, generating Declare constraints, calculating fisher's scores, selecting constraints etc. These tasks take some time, according to the input parameters, variability and size of the log. Therefore we decided to show loader bar to the analyst so he can see the progress of the plugin (see Figure. 19). After the plugin finishes its work, the DeclareDevianceMinerOuput object is created (Figure 20).

ProM plugins work in the way, that they take something on the input (usually Log file) and produce some output. The output of the ProM plugin can be some nice Petri Net, or some interesting process model, or just new filtered log. However, this nice Petri net or process model is just some Java object generated by plugin, which ProM doesn't know how to work with, how to show it or how to draw the image of that model etc. For this, there are ProM visualizers which know how to "handle" those objects generated by ProM plugins. They purpose is to show the interpretable results of the plugins to the analysts.

Our ProM plugin takes on the input log, usually XES log and produce DeclareDevianceMinerOutput object as our Mined Model (see Figure 19). This object is not interpretable by ProM in default, which means that there is not ProM visualizer which can work with DeclareDevianceMinerOutput objects, therefor there was a need to build our own ProM Visualizer as "add-on" in order to show interpretable results of our plugin to the analyst.

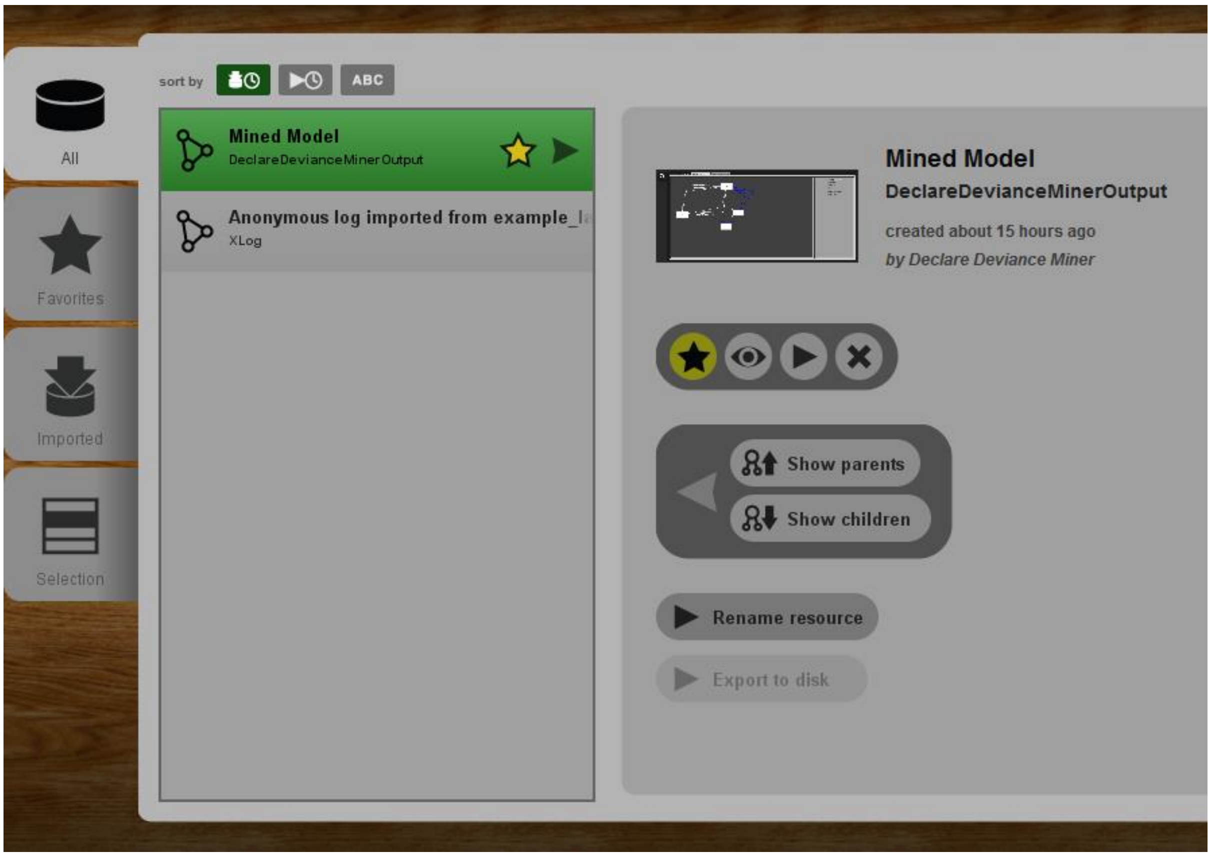
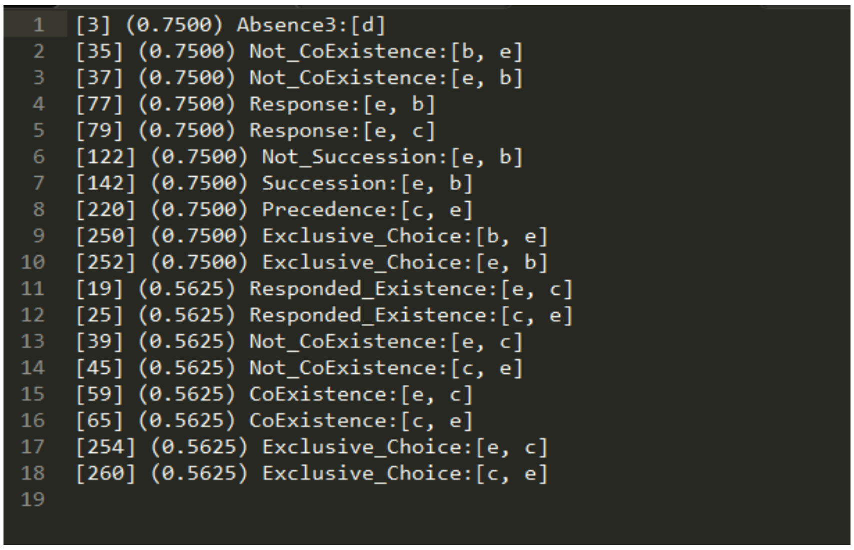

All
Favorites
Imported
Selection
sort by
ABC
Mined Model
DeclareDevianceMinerOutput
Anonymous log imported from example_l
XLog
Mined Model
DeclareDevianceMinerOutput
created about 15 hours ago
by Declare Deviance Miner
Show parents
Show children
Rename resource
Export to disk

Figure 20. ProM workspace showing our Mined Model and the log below it.

5.2 ProM Visualizer

We implemented our Deviance Miner Visualizer in the ProM as important add-on to our plugin in order to show our results to the analyst in the nice interpretable way. In addition to the Visualizer declare constraints selected by our algorithm are also printed to the output file called *selected_constraints.txt*, which is saved in the ProM folder. The example of this output file is shown in the following figure.



```
[3] (0.7500) Absence3:[d]
[35] (0.7500) Not_CoExistence:[b, e]
[37] (0.7500) Not_CoExistence:[e, b]
[77] (0.7500) Response:[e, b]
[79] (0.7500) Response:[e, c]
[122] (0.7500) Not_Succession:[e, b]
[142] (0.7500) Succession:[e, b]
[220] (0.7500) Precedence:[c, e]
[250] (0.7500) Exclusive_Choice:[b, e]
[252] (0.7500) Exclusive_Choice:[e, b]
[19] (0.5625) Responded_Existence:[e, c]
[25] (0.5625) Responded_Existence:[c, e]
[39] (0.5625) Not_CoExistence:[e, c]
[45] (0.5625) Not_CoExistence:[c, e]
[59] (0.5625) CoExistence:[e, c]
[65] (0.5625) CoExistence:[c, e]
[254] (0.5625) Exclusive_Choice:[e, c]
[260] (0.5625) Exclusive_Choice:[c, e]
```

Figure 21. Content of the selected_constraints.txt file

As can been seen the content consist of the list of selected constraints by our core algorithm. Each constraint has its own unique ID, then then fisher score and the last is the constraint itself consisting of the template name, and the parameters, which are either one event or two events.

Our ProM Visualizer has very simplistic design. It consists of tabbed window, where we can see two tabs "*Deviant constraints (deviantConstraintsAmount)*" and "*Normal constraints (normalConstraintsAmount)*" where *deviantConstraintsAmount* is the number of constraints which were satisfied in more deviant cases in the log and *normalConstraintsAmount* is the amount of constraints which were satisfied in more normal traces than deviant in the log.

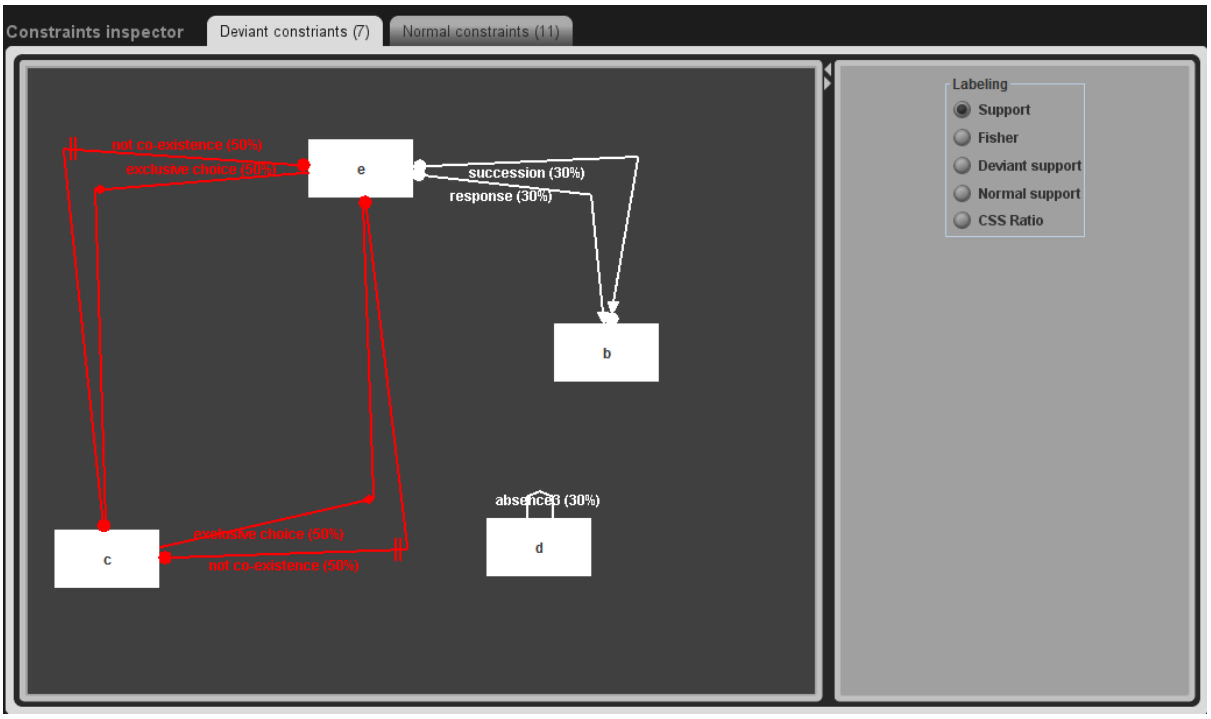


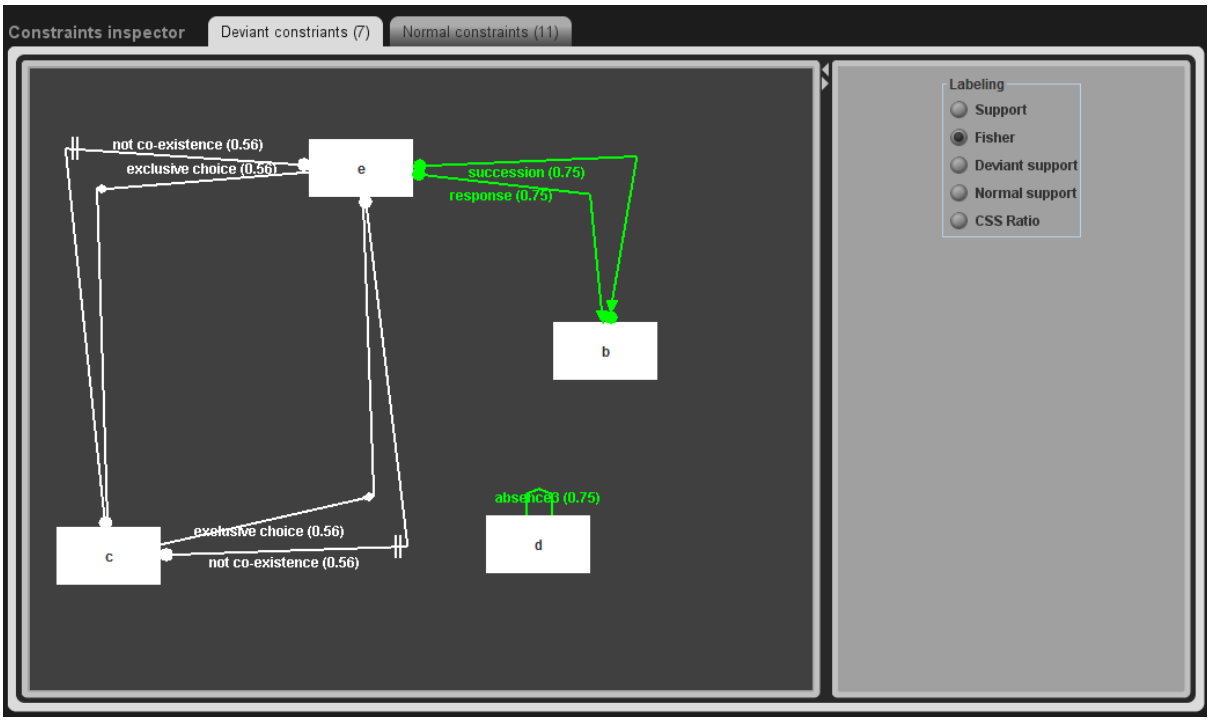
Figure 22. Visualizer showing simple plugin output

In other words, we have two Declare maps, two group of Declare constraints divided by their satisfaction numbers in the corresponding class (Normal or Deviant). As can be seen on the Figure 20 there are not just Declare maps consisting of the constraints and events, but we can also select different labelling for the constraints to show us relevant information:

- Support
- Fisher
- Deviant support
- Normal support
- CSS ratio (Cross class support)

Support is selected by default, where we label and colour the constraints by their support value. Support value as described in previous sections represent the value, in how many traces of the log the constraint is satisfied (we do not care whether the trace is deviant or normal). We show this value in percent. E.g. in figure 22 the constraint *Absence3 (d)* is satisfied in 30% of the traces, or in 30% of the log in other words. Constraint *Not Coexistence (e, c)* has support 50%, which means this constraint is satisfied in 50 % of all the traces in the log.

For colouring of the constraints lines we decided to use RGB model. For colouring the constraints lines by Support we decided to change only R value (we leave G and B values on 255), which represents red colour in RGB model. The constrains with low support are more white, constraints with high support are more red. Since on the Figure 20 we have only two values of support, we have only strong red and white. If we have more values with different range, we would see the colour more spread. If we have 50 different values, we would have 50 shades of red.


Constraints inspector | Deviant constraints (7) | Normal constraints (11)

Labeling
Support
[✓ Fisher]
Deviant support
Normal support
CSS Ratio

Figure 23. Labelling by fisher score

Another metric for labelling is fisher score (see Figure 24). Here we change only G in RGB model, so the colour for the constraints varies from white to strong green. More values of fisher score, more shades of green we get. Figure 24 shows us example of having different shades of green as well as more complex Declare map consisting of 20 deviant constraints.

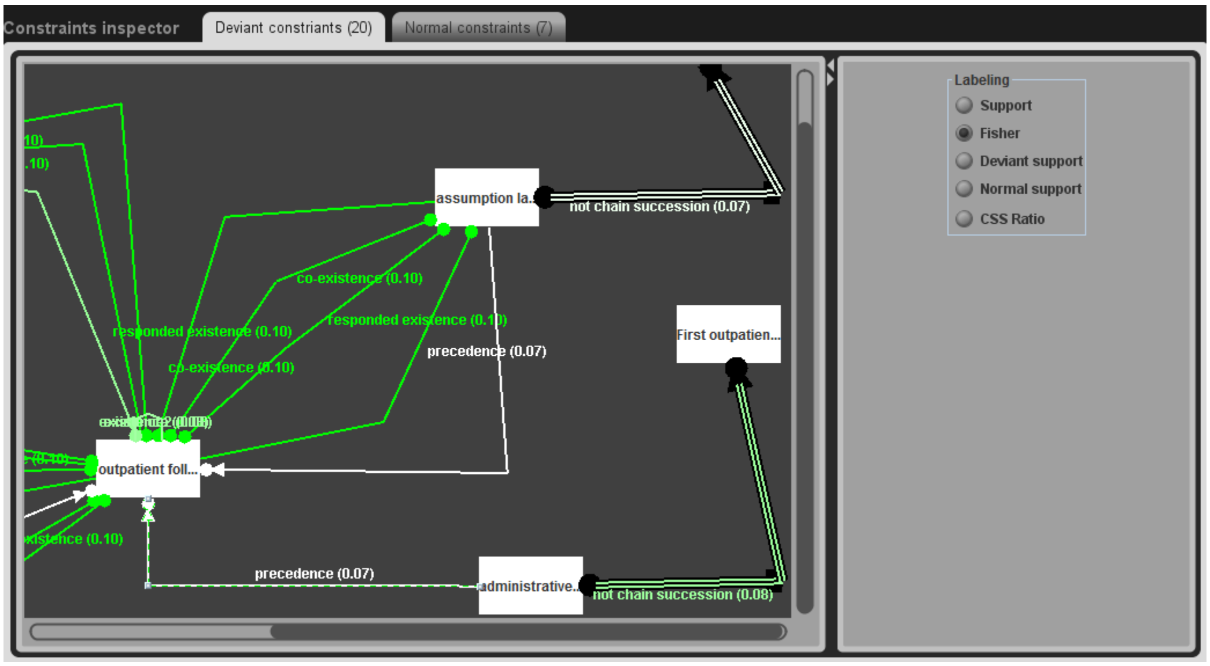


Figure 24. More complex Declare Map, with bigger variety of fisher scores

Deviant support and Normal support represent the support of the constraint in the corresponding class (Deviant or Normal). Deviant support represent in how many deviant traces of all deviant traces the constraint is satisfied, so it is shown as percentage in the end. Same goes for Normal support, but just for the normal class.

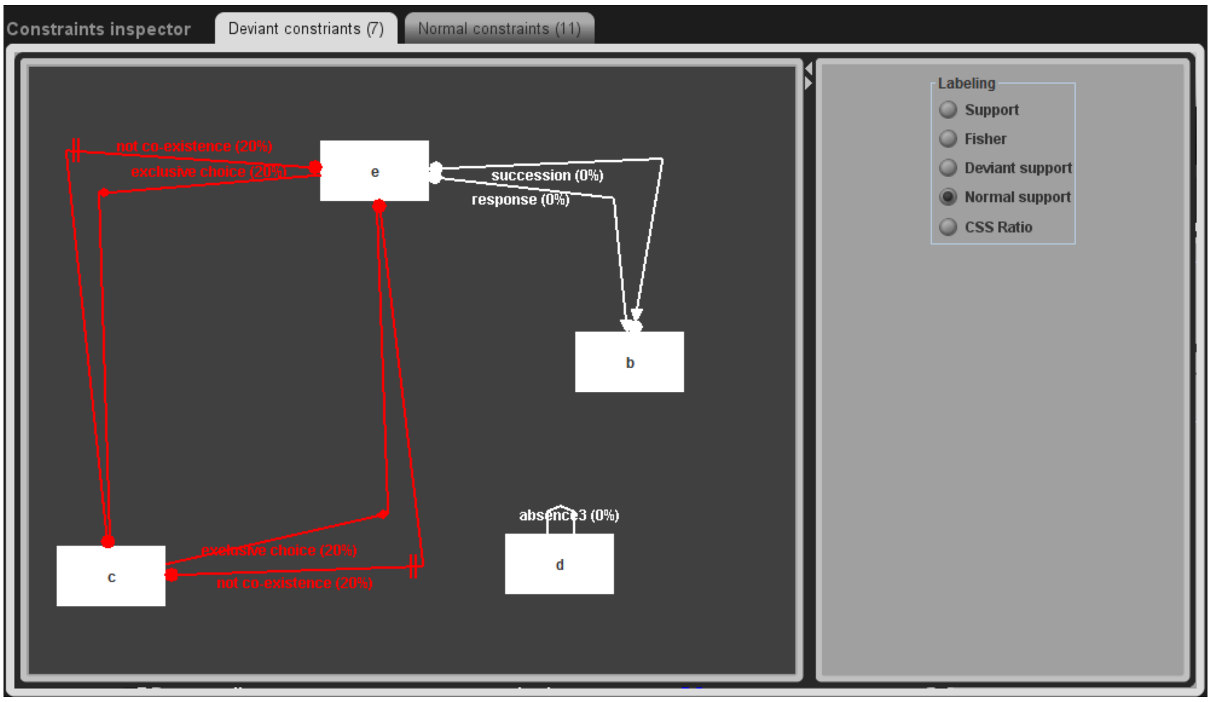
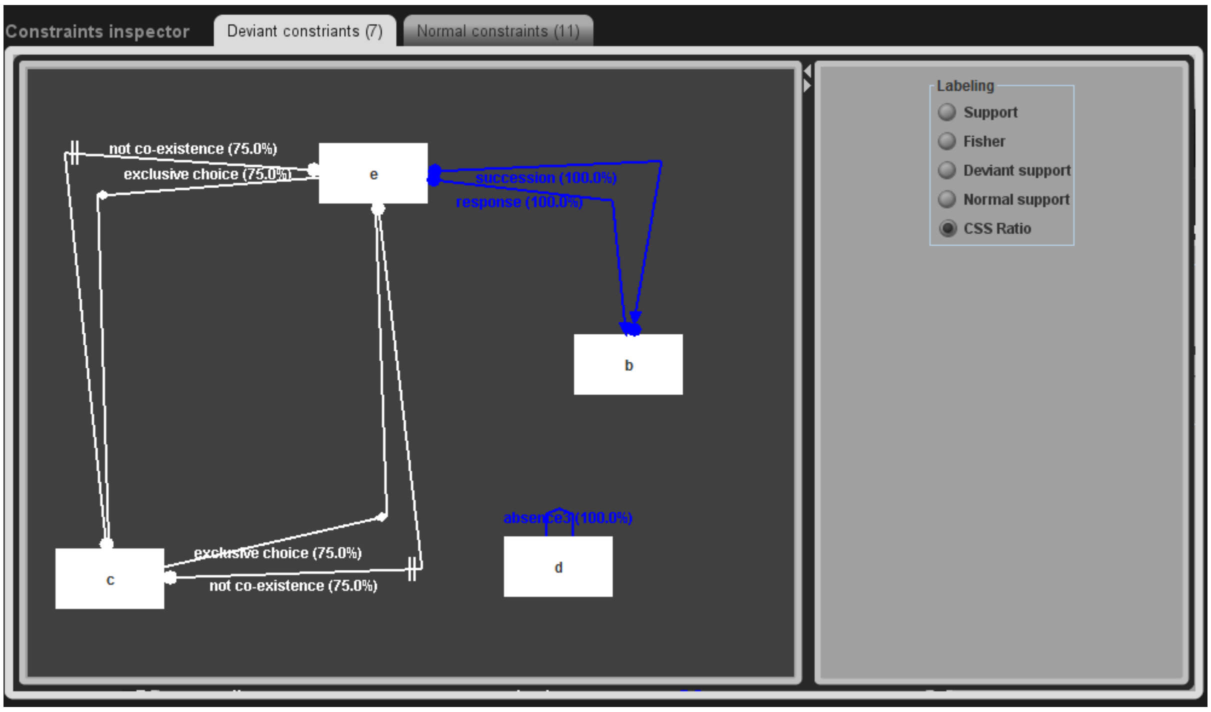


Figure 25. Labelling and colouring of the constraints by normal support

As can be seen on the Figure 25 the constraints *Succession (e, b)*, *Response (e, b)* and *Absence3 (d)* are not present in any normal trace in the log, which could also explain why they are more discriminative then the other 4 (higher fisher score) – see Figure 23.

Last to mention is CSS ratio which stands for Cross Class Support ratio. This ratio represent the ratio between constraint satisfactions in Normal and Deviant class or vice versa. The formula for this variable would be $CSS = 1 - \frac{\mathrm{Min}(deviant\_support,\ normal\_support)}{\mathrm{Max}(deviant\_support,\ normal\_support)}$. On the Figure 26 below we show labelling and colouring by odd ratio. Since already mentioned constraints *Succession (e, b)*, *Response (e, b)* and *Absence3 (d)* are present in 0% of normal traces, but they are present in some deviant traces, their odd ration is 100%.

Colouring is done here by changing only the B value from RGB model, so we get range of different shades of blue.



Labeling
- Support
- Fisher
- Deviant support
- Normal support
- [✓ CSS Ratio]

Figure 26. Labelling and colouring by CSS Ratio

5.3 Classification

We developed our core algorithm in local workspace in Eclipse as separated project, which consists of all the steps mentioned in the proposed approach section. The ProM plugin itself consists of only three mentioned phases in the proposed approach: preprocessing, constraint mining, constraint selection, but missing the classification part. The most important part in the project was classification part where we decided to implemented classification using open source library Weka[4], which consists of collection of machine learning algorithms and data mining tasks. This part was purely for evaluation purposes, which we will describe more in section 6.

[4] http://www.cs.waikato.ac.nz/ml/weka/index.html

6 Evaluation

In this chapter, we introduce our datasets, used for evaluation and we compare our accuracy results with already existing deviance mining tools, using different input configurations with our Declare Deviance Miner project so we can answer our main research question:

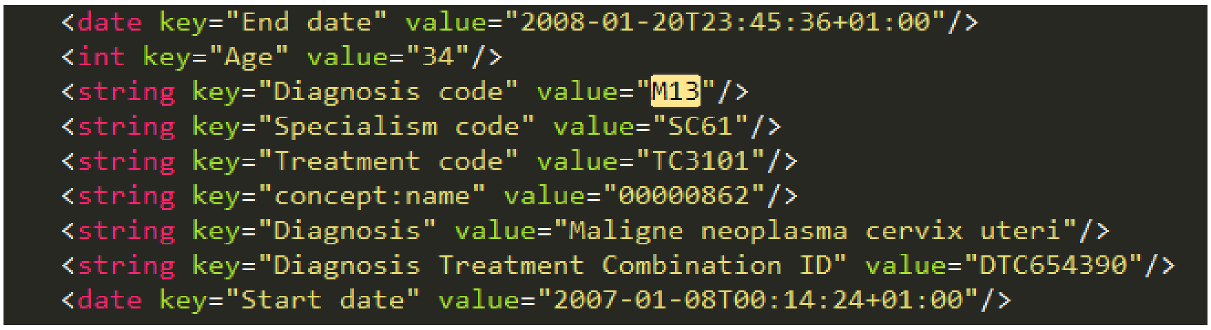
How accurate is deviance mining using our Declare Deviance Miner tools?

6.1 Datasets

We selected four BPI11 datasets for our evaluation [19]. These datasets were labelled and used in [4] as well, where authors tried to compare different deviance mining tools and their accuracy. Since we wanted to compare our accuracy results with existing tools, which were already compared in this paper, the choice of these four logs was very natural.

Table 5 below shows the Descriptive statistics for our selected four BPI11 datasets. These datasets were extracted from the original BPI11 log, which stores executions of a process connected to the treatment of the cancer or patients with cancer in one Dutch hospital.

As can be seen on the Figure 27 below, BPI11 log contains specific domain attributes, resp. each trace in the log hold attributes like *Age*, *Diagnosis code*, *Treatment coded*, *Diagnosis* etc.



```
<date key="End date" value="2008-01-20T23:45:36+01:00"/>
<int key="Age" value="34"/>
<string key="Diagnosis code" value="M13"/>
<string key="Specialism code" value="SC61"/>
<string key="Treatment code" value="TC3101"/>
<string key="concept:name" value="00000862"/>
<string key="Diagnosis" value="Maligne neoplasma cervix uteri"/>
<string key="Diagnosis Treatment Combination ID" value="DTC654390"/>
<date key="Start date" value="2007-01-08T00:14:24+01:00"/>
```

Figure 27. Example of attributes of one case in BPI11 log

In the first of the four BPI11 logs, called **BPI$_{dCC}$**, traces were labelled as deviant if the value of “Diagnosis” attribute is “cervix cancer”. The rest of the traces were labelled as normal.

In the second log, called **BPI$_{M13}$**, traces were labelled as deviant if the value of “Diagnosis code” attribute is “M13”, just like on the Figure 26 above.

In the third log, called **BPI$_{M16}$**, traces were labelled as deviant if the value of “Diagnosis code” attribute is “M16”.

In the last log, called **BPI$_{t101}$**, traces were labelled as deviant if the value of “Treatment code” attribute is “101”.

Table 5. Descriptive statistics for four BPI11 datasets

Dataset	Normal cases	Deviant cases	Total cases	Avg. length (norm.)	Avg. length (dev.)	Event classes (norm.)	Event classes (dev.)	Is ba-lanced?
BPI$_{dCC}$	917	225	1,142	109	85	100	100	No
BPI$_{dM13}$	832	310	1,142	144	74	99	99	No
BPI$_{dM16}$	926	216	1,142	127	113	99	94	No
BPI$_{t101}$	683	459	1,142	195	25	107	103	Yes

6.2 Classification accuracy

As we mentioned in previous section, multiple methods can be used in order to construct a classifier from our selected constraints or rather from the TDB features space created on these selected constraints. We decided to use Decision trees, specifically used C4.8 method implemented in Weka. We also included k-NN classifier, which is k-Nearest neighbours, where we set k=8 after trying to find the highest accuracy after some trial and error experiment.

Our classification accuracy was calculated using the standard notion of accuracy, that is: $Accurracy = \frac{tp+tn}{tp+tn+fp+fn}$, where tp and tn stands for true positives and true negatives (correctly classified normal and deviant traces), the value in denominator is all the traces in the log (true positive + true negative + false positive + false negative).

Another important value we get AUC (Area under the ROC Curve) which is a standard evaluation metric for binary classification problems. AUC is the probability that our classifier C ranks a randomly drawn positive sample higher than randomly drawn negative sample. If the AUC = 1 we have perfect classifier for which all positive samples come after all negative ones.

As mentioned in previous sections our classification is framework is based on 5-fold cross validation, which means we generate 5 folds with data split into 80% of the training and 20% testing sets. We calculate accuracy for each of these folds separately and in the end we get final accuracy by calculating the mean. We calculate standard deviation σ as well and show it in the tables below.

Our core algorithm takes four parameters on the input namely:

1. *event_support*
2. *constraint_support*
3. *coverage*
4. *vacuity_enabled*

First important challenge for us was to find the ideal input configuration in order get good accuracy results. Our hypothesis was that the lower *event_support* value as well *constraint_support* values the better accuracy results. Considering some events are rare in the log, for example only present in deviant cases, the low *event_support* allows us to use these events for generation of Declare constraints which might be very discriminative. Same goes for *constraint_support*, the lower value of *constraint_support* allows us to keep the constraints which are satisfied only in few cases, but still these constraints might be very discriminative anyway. We tried different *event_support* and *constraint_support* values in our early experiments with $BPI11_{M16}$ log. We came to conclusion that lower the *event_support* value is as well as the lower the *constraint_support* is, the better the mean accuracy of the classifier in the end for our testing dataset. Figure 28 below shows our experiments on BPI_{M16} log where we tried *event_support* values 80%, 70%, 60%, 50% and 25% for two *constraint_support* configurations: 0% and 50%. It can be seen from this graph that accuracy steadily improves with lower event support. The blue line represents *constraint_support* (in other words min. support too) 0% and the orange line represents *constraint_support* 50%.

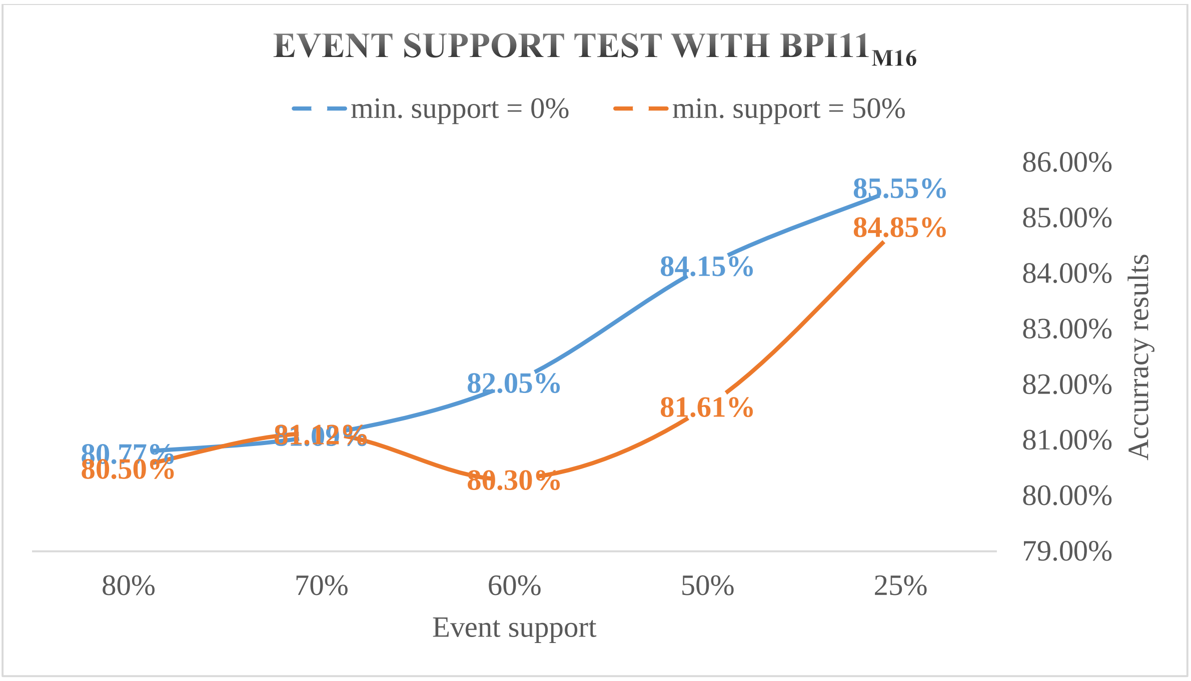


Figure 28. Event and constraint support variety of tests on BPI11$_{M16}$

Another input parameters to test was *coverage* and *vacuity_enabled*. These two variables did not influence the mean accuracy in the end that much, therefore we decided to keep coverage value 5, since this value was well tested with other tools as well after trial and error experiments in [4]. With *vacuity_enabled* = TRUE, we were able to get slightly better results, where by slightly we mean in average half percent.

Authors in [4] compared three different groups of deviance mining techniques based on the features extraction perspective, which we already mentioned in the state of the art:

- **IA (**occurrences of Individual activites in traces**) [6]** – Individual activities with their occurrence numbers in the traces represent numerical features.
- **Frequent patterns** – *Maximal repeats* (MR), *Tandem Repeats* (MR), *Alphabet Maximal Repeats* (AMR) and *Alphabet Tandem Repeats* (ATR) **[8] [9]** – frequent patterns like MR, TR etc. represents Boolean features, therefor this pattern can be either present in the trace or not. If it is present feature is then TRUE otherwise FALSE. Since in the extraction phase authors can get big amounts of MR/TR, they select only the N most frequent ones, in other words N patterns with highest support.
- **Discriminative patterns [20]**– Discriminative patterns DP or other said Iterative patterns IP represent sequence of consecutive events in the log. These iterative patterns become boollean features just like the frequent patters above, where if the feature is present in the trace then value is true, otherwise it is false.

The first method, based on occurrences of Individual activities in the trace, authors in [4] used as the baseline for their experiments. Then they tried to evaluate an add-on value of the other deviance mining techniques mentioned above. Therefor they did experiments on 6 feature sets namely: IA, IA+TR, IA+ATR, IA+MR, IA+AMR, and IA + DP.

Following tables represent the results we got with our Declare Deviance mining tools as well as the results of other deviance mining techniques.

Table 6. Classification results for BPI$_{dCC}$ dataset

Feature type	Decision Tree		k-NN	
	Accuracy(%)	AUC	Accuracy(%)	AUC
IA	78.81±2.21	0.752±0.026	79.95±1.15	0.751±0.011
IA+TR	76.97±4.14	0.736±0.061	**81.17±2.71**	0.761±0.016
IA+ATR	78.19±2.54	0.738±0.061	80.56±1.90	0.760±0.007
IA+MR	76.35±1.93	0.682±0.050	80.38±2.10	**0.773±0.013**
IA+AMR	75.66±1.78	0.687±0.038	80.21±2.00	0.771±0.013
IA+DP	78.98±2.18	**0.744±0.033**	80.65±1.12	0.771±0.045
DECLARE	**79.94±2.48**	0.586±0.108	80.03±0.03	0.724±0.030

Table 7. Classification results for BPI$_{M13}$ dataset

Feature type	Decision Tree		k-NN	
	Accuracy(%)	AUC	Accuracy(%)	AUC
IA	71.63±2.01	0.721±0.033	72.59±0.92	0.700±0.011
IA+TR	71.19±2.00	0.710±0.042	72.07±1.36	0.705±0.016
IA+ATR	72.33±1.60	0.691±0.012	71.72±0.78	0.698±0.007
IA+MR	71.98±2.59	0.722±0.026	71.28±0.91	0.692±0.013
IA+AMR	72.33±2.97	**0.728±0.029**	71.19±1.09	0.692±0.013
IA+DP	73.99±3.33	0.727±0.064	72.85±2.27	0.728±0.045
DECLARE	**75.12±2.90**	0.699±0.055	**74.25±2.18**	**0.756±0.037**

Table 8. Classification results for BPI$_{M16}$ dataset

Feature type	Decision Tree		k-NN	
	Accuracy(%)	AUC	Accuracy(%)	AUC
IA	82.49±1.06	0.759±0.058	83.27±1.81	0.774±0.031
IA+TR	83.19±0.91	0.763±0.028	83.19±1.17	0.771±0.022
IA+ATR	83.10±1.35	0.749±0.069	83.10±1.22	0.767±0.025
IA+MR	82.57±0.94	0.766±0.049	82.84±1.44	0.776±0.026
IA+AMR	82.57±0.94	0.766±0.051	82.84±1.44	0.776±0.027
IA+DP	84.06±0.79	0.736±0.076	84.68±0.61	0.803±0.008
DECLARE	**85.29±2.12**	**0.783±0.057**	**86.34±1.79**	**0.859±0.024**

Table 9. Classification results for BPI$_{T101}$ dataset

Feature type	**Decision Tree**		**k-NN**	
	Accuracy(%)	AUC	Accuracy(%)	AUC
IA	84.94±1.76	0.860±0.017	85.99±2.09	0.910±0.026
IA+TR	84.94±1.66	0.850±0.009	**87.57±2.37**	0.913±0.026
IA+ATR	85.38±1.47	0.861±0.013	86.69±2.73	0.911±0.027
IA+MR	84.76±1.60	0.859±0.017	86.43±3.28	0.912±0.026
IA+AMR	84.77±1.61	0.855±0.011	86.17±3.39	0.613±0.028
IA+DP	84.77±2.75	0.864±0.037	87.48±2.03	**0.920±0.006**
DECLARE	**85.82±1.67**	**0.887±0.013**	84.41±0.70	0.909±0.010

As we can see from the tables above, our tools give us comparable results with already existing deviance mining tools, in some cases even superior accuracy. The event logs BPI11$_{M13}$ and BPI11$_{M16}$ are example of that, where both of our classifier are winning.

We believe accuracy of our tools could be significantly improved by introducing number of satisfactions of the constraint in the trace, instead of having just Boolean values (satisfied yes or no). We would get richer information in this case and these constraints would be more discriminative than our Boolean constraints.

7 Conclusions

In this thesis we explained process deviance mining and proposed new deviance mining approaches using Declare.

Existing deviance mining techniques mine patters or constraints from the events logs based on discriminative or frequent pattern mining. Our tool mine discriminative Declare constraints.

Our first research question was, whether it is possible to provide business analyst with understandable feedback about business process deviances. We tried to address this question by explaining deviance through Declare constraints. Declare models generalize sequential behaviours in the more compact set of rules and in this sense improve understandability. In addition we implemented ProM plugin that visualize Declare models as maps in which discriminative constraints are highlighted by colour based semantics.

Our second question was, whether the accuracy obtained by explaining deviances with Declare constraints is comparable with the state of art technique. Our evaluation shows that our approach has an accuracy that is comparable and sometimes outperforms the approaches based on sequential patterns.

There are several directions for the future work how we could improve are Declare Deviance Miner:

- **Extend Declare with more patterns in the future** – There is possibility to add more Declare templates in the future, e.g. binary templates, where we would consider three events in the template rather just pairs. Longer constraints could be more discriminative therefor the accuracy could be improved.
- **Consider constraint frequencies** – Right now we only consider Boolean constraints, in other words, we save information weather the constraint is satisfied in the trace or not. In the future we could also use the number of activations of the constraint in the trace for a richer information. Instead of saying that constraints *Chain Precedence (A,B)* is satisfied or not in the trace, we can also save, that is satisfied 5 times. This could give us more discriminative power and higher fisher score as well as better accuracy results.
- **Combine Declare with sequential patterns** – We could combine Declare constraints together with sequential patterns. To explain business process deviances it is possible to use both features referring to the validity of Declare constraints in a case and features referring to the occurrence of sequential patterns in a case.
- **Combine Declare constraints by using decision tree** – We could use decision tree not just to build the classifier from our test log, but also use it for combining of patterns. Decision tree consist of the nodes and leaves, the paths from the root node to the leave represent classification rules. We could use these path for combining some constraints together (which are present in one path) and they could give us more discriminative power.
- **Plugin improvement** – There is big room for plugin improvement and functionality extensions in the future. We could show more information about the constraints to the analyst, as well as make the plugin calculations faster with optimizations.
- **More range of experiments** – There is possibility to do big range of experiments with different logs like Bpi 15, Bpi 14 or other logs and compare the results with already existing tools.

Bibliography

[1] W. van der Aalst, “Process Mining - Discovery, Conformance and Enhancement of Business Processes,” *Springer,* (2011).

[2] M. Pesic and W. van der Aalst, “A Declarative Approach for Flexible Business Processes Management,” *BPM Workshops,* vol. 4103, p. 169–180, (2006).

[3] M. Pesic, H. Schonenberg and W. Van der Aals, “DECLARE: Full Support for Loosely-Structured Processes,” *in EDOC, IEEE,* p. 287–300, (2007).

[4] H. Nguyen, M. Dumas, M. L. Rosa, F. M. Maggi and S. Suriadi, “Mining Business Process Deviance: A Quest for Accuracy. In: R. Meersman et al. (eds.): OTM 2014,” *LNCS,* vol. 8841, pp. 436 - 445, (2014).

[5] A. Partington, M. Wynn, S. Suriadi, C. Ouyang and K. J., “Process mining of clinical processes: Comparative analysis of four australian hospitals,” *ACM Trans. in Management Information System,* (in press, 2014).

[6] S. Suriadi, M. Wynn, C. Ouyang, A. ter Hofstede and N. van Dijk, “Understanding process behaviours in large insurance company in Australia: A case study. In: Salinesi, C., Norrie, M.C., Pastor, Ó. (eds.) Caise 2013,” *LNCS,* vol. 7908, p. 449–464, (2013).

[7] C. Sun, J. Du, N. Chen, S. C. Khoo and Y. Yang, “Mining explicit rules for software process evaluation. In: Proc. of ICSSP,” *ACM,* p. 118–125, (2013).

[8] R. Bose and W. van der Aalst, “Discovering signature patterns from event logs. In: Proceedings of CIDM,” *IEEE,* p. 111–118, (2013).

[9] R. Bose and W. van der Aalst, “Trace clustering based on conserved patterns: Towards achieving better process models. In: Rinderle-Ma, S., Sadiq, S., Leymann, F. (eds.) BPM 2009,” *LNBIP,* vol. 43, p. 170–181, (2010).

[10] G. Lakshmanan, S. Rozsnyai and F. Wang, "Investigating clinical care pathways correlated with outcomes. In: Daniel, F., Wang, J., Weber, B. (eds.) BPM 2013.," *LNCS,* vol. 8094, p. 323–338, (2013).

[11] J. Swinnen, B. Depaire, M. Jans and K. Vanhoof, “A process deviation analysis – A case study. In: Daniel, F., Barkaoui, K., Dustdar, S. (eds.) BPM Workshops 2011,” *Part I LNBIP,* vol. 99, p. 87–98, (2012).

[12] J. Nakatumba and W. van der Aalst, “Analyzing resource behavior using process mining. In: Rinderle-Ma, S., Sadiq, S., Leymann, F. (eds.) BPM 2009,” *LNBIP,* vol. 43, p. 69–80, (2010).

[13] J. Poelmans, G. Dedene, G. Verheyden, H. Van der Mussele, S. Viaene and E. Peters, “Combining business process and data discovery techniques for analyzing and improving

integrated care pathways. In: Perner, P. (ed.) ICDM 2010," *LNCS,* vol. 6171, p. 505–517, (2010).

[14] O. Kupferman and M. Y. Vardi, "Vacuity Detection in Temporal Model," *International Journal STTT,* vol. 4, p. 224–233, (2003).

[15] C. W. Günther and E. Verbeek, "OpenXES Developer Guide v2.0," Eindhoven, 2014.

[16] C. W. Günther and E. Verbeek, "XES Standard Definition v2.0," Eindhoven, 2014.

[17] W. v. d. Aalst, "Process Mining tools," 2011. [Online]. Available: http://www.processmining.org/logs/start. [Accessed April 2015].

[18] R. Duda, P. Hart and D. Stork, Pattern Classification (2nd Edition), Wiley Interscience, 2000.

[19] "3TU Data Center. BPI Challenge, evetn Log (2011), doi:10.4121 /uuid:d9769f3d0ab0-4fb8-0d1120ffc54".

[20] D. Lo, H. Cheng, H. J., S.-C. Khoo and C. Sun, "Classification of software behaviours for failure detection: A discriminative pattern mining aproach. In: Proc. of KDD," *ACM,* pp. 557-566, (2009).

Appendices

I. License