SAPIENZA
UNIVERSITÀ DI ROMA

Generazione di event log a conformità parziale con specifiche di processo

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di Laurea in Informatica

Ralph Angelo Tancio Almoneda
ID number 1837040

Advisor
Prof. Claudio Di Ciccio

Academic Year 2021/2022

SAPIENZA
UNIVERSITÀ DI ROMA

Event log generation with partial conformity using process specifications

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di Laurea in Informatica

Ralph Angelo Tancio Almoneda
ID number 1837040

Advisor
Prof. Claudio Di Ciccio

Academic Year 2021/2022

Event log generation with partial conformity using process specifications
Bachelor's thesis. Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: almoneda.1837040@studenti.uniroma1.it

Abstract

Uno degli elementi più fondamentali nel *process mining* è l'*event log* con il quale, insieme a modelli di processo, sono realizzati i tre tipi biù basici del *process mining*: *discovery*, *conformance checking*, ed *enhancement*. I modelli di processo dichiarativi, specificano i vincoli permessi, e ogni esecuzione che è conforme ai suddetti vincoli è considerato accettato. La possibilità di creare diversi tipi di *event log* basandosi su vari modelli di processo dichiarativi, è la chiave per valutare le tecniche di *process discovery* e verificare modelli dichiarativi già esistenti per ottenere un'approfondita comprensione di come quest'ultimi si comportano. In questo studio, presenteremo lo sviluppo di funzioni aggiunte per la generazione automatica di *event log* a conformità parziale ad un *framework* già esistente chiamato MINERful[8]. MINERful un approccio al *process mining* dichiarativo ed è anche un'applicazione a riga di comando. Gli *event log* generati utilizzando MINERful partendo da modelli di processo dichiarativi, dovranno soddisfare i vincoli forniti dall'utente. L'obbiettivo di questo studio, è di implementare funzioni che rendono possibile all'utente di scegliere quali vincoli del modello di processi dichiarativo saranno violati un'abilità utile per analisi di tipo *what-if*.

Abstract

One of the most fundamental elements in process mining is the event log from which, along with a process model, the three basic types of process mining are realized: discovery, conformance checking, and enhancement. Declarative process models specify constraints for permitted behaviour, and every execution that complies with such constraints is considered acceptable. The possibility to create different types of event logs based on various declarative process models is key to evaluating process discovery techniques and testing already existing declarative models for a deeper understanding of their behaviour. In this paper, we present the development of functions added to the already existing framework called MINERful [8] for the automated generation of event logs with partial conformity. MINERful is a declarative process mining approach and command-line application. The event logs generated using MINERful starting from declarative process models must only fulfil the constraints given by the user. This study's aim is to implement functions to make it possible to choose the constraints to be violated in a declarative process model, which has a significant value for the actuation of what-if analyses.

Contents

Chapter 1

Introduction

Event logs in the process mining context are a representation of one or more cases of a business process or can also be a documentation of several related business processes. The events in an event log are defined by a list of their attributes. The case ID, time stamps of the start and end times, and other attributes of the event are recorded by the IT system to create event logs[1]. An event log can also be viewed as a collection of cases. A case can be seen as a trace or a sequence of events. Event data may come from a large array of resources like a database system (i.e., patient data in a hospital), a CSV file or spreadsheet, a transaction journal (i.e., a trading system), a business suite or ERP system (SAP, Oracle, etc.), a message log (i.e., from IBM middleware), an open API providing data from websites or social media [2].

The events in an event log that is artificially made are created following different "rules" called constraints which are inputted by the user with a Declare specification. Using Declare, which could be expressed in a non-linear temporal logic language, is the first step to creating process models in the MINERful application[8].

My task consisted of implementing functions on the MINERful framework to enable the user to auto-generate event logs using declarative specifications with partial conformity.

The remaining part of the paper is subdivided as follows: Chapter 2 discusses the topics regarding the work, Chapter 3 the approach taken and Chapter 4 the implementation applied. Chapter 5 shows examples of the working implementation and Chapter 6 the conclusion.

[1] https://appian.com/process-mining/event-log.html
[2] Source: http://www.processmining.org/event-data.html

Chapter 2

Background

This chapter explains in more detail the topics covered during the internship which are: Process Mining, Declare, Regular Expressions, Finite State Automata and MINERful.

2.1 Process Mining

Process mining is a relatively new field of research in Informatics [3], and it's growing rapidly in significance and in development due to the fact that it is used mainly in a fast-paced, dynamic business environment. It benefits from the constant generation of data by business organizations and information systems [13]. The data taken or supplied from these systems are, more often than not, inadequate for the extraction in a log. Thus, a process mining algorithm cannot be applied to them. For data to be eligible to be used in process mining algorithms, it must fulfil at least these requirements: each case must have a *unique identifier (CaseID)* which determines the position of an event in a process instance, a description of the event occurring in the case called an *activity* which can occur multiple times in the process instance and a *timestamp* which specifies when a certain event took place, and it can be used to define the order or sequence of a process instance's events. Process instances are a specific example of what a specific process goes through, and in process mining, they are called traces. Based on this information, an event log can be generated, and each of them has a different level of maturity based on its quality.[1]

Table 2.1. Maturity levels for event logs. [1]

Level	Characterization	Example
5	Highest level: the event log is of excellent quality (i.e., trustworthy and complete) and events are well-defined. Events are recorded in an automatic, systematic, reliable, and safe manner. Privacy and security considerations are addressed adequately. Moreover, the events recorded (and all of their attributes) have clear semantics. This implies the existence of one or more ontologies. Events and their attributes point to this ontology.	Semantically annotated logs of BPM systems.
4	Events are recorded automatically and in a systematic and reliable manner, i.e., logs are trustworthy and complete. Unlike the systems operating at the level immediately below, notions such as process instance (case) and activity are supported in an explicit manner.	Events logs of traditional BPM/ workflow systems.
3	Events are recorded automatically, but no systematic approach is followed to record events. However, unlike logs at level 2, there is some level of guarantee that the events recorded match reality (i.e., the event log is trustworthy but not necessarily complete). Consider, for example, the events recorded by an ERP system. Although events need to be extracted from a variety of tables, the information can be assumed to be correct (e.g., it is safe to assume that a payment recorded by the ERP actually exists and vice versa).	Tables in ERP systems, event logs of CRM systems, transaction logs of messaging systems, event logs of high-tech systems, etc.
2	Events are recorded automatically, i.e., as a by-product of some information system. Coverage varies, i.e., no systematic approach is followed to decide which events are recorded. Moreover, it is possible to bypass the information system. Hence, events may be missing or not recorded properly.	Event logs of document and product management systems, error logs of embedded systems, worksheets of service engineers, etc.
1	Lowest level: event logs are of poor quality. Recorded events may not correspond to reality and events may be missing. Event logs for which events are recorded by hand typically have such characteristics.	Trails left in paper documents routed through the organization ("yellow notes"), paper-based medical records, etc.

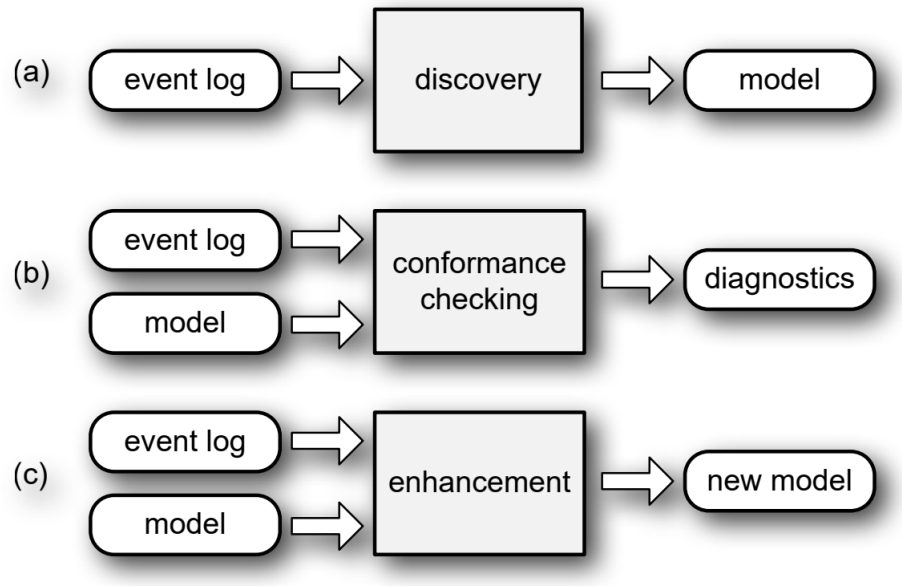


Figure 2.1. The three main use cases of process mining with input and output[1]

The three basic types of process mining are:

1. **Process Discovery**: which takes an event log and produces a model without using any a-priori information about the process itself. The generated model expresses the behaviour occuring in the data.

2. **Conformance Checking**: an existing process model is compared with an event log of the same process. Conformance checking can be used to check if reality, as recorded in the log, conforms to the model and vice versa. Note that different types of models can be considered: conformance checking can be applied to procedural models, organizational models, declarative process models, business rules/policies, laws, etc. [1]

3. **Enhancement**: similarly to conformance checking, an existing process model is compared to an event log recorded by an information system and the information recorded in the event log about the actual process is used to improve an existing model.

2.2 Declare

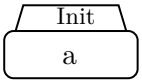
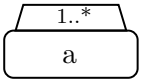
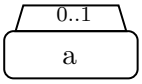
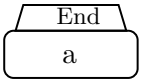
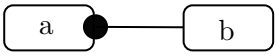
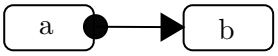
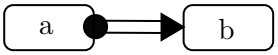
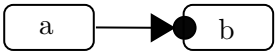
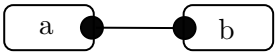
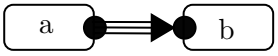
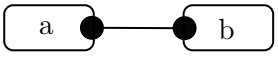
Declare[16] is a declarative process modelling language introduced by Pesic and van der Aalst in [9]. Declarative meaning every sequence of activities that leads a process execution from the start to the end does not need to be explicitly specified as opposed to imperative specification, which requires process model developers to predict all possible execution scenarios in advance and explicitly include them in the model[4].

[1] https://www.coursera.org/lecture/process-mining/4-3-introduction-to-conformance-checking-pi2L2

Process models created using Declare are based on constraints which must hold true during the enactment. They can also be seen as never to be broken rules. All sequences that respect the constraints are thus allowed. Constraints are meant to be actuated on a set of activities or tasks and mainly pertain to their temporal ordering. More specifically, Declare define a diverse set of standard templates which have been inspired by a catalogue of temporal logic patterns usually used in model checking for a variety of dynamic systems from different application domains[7].

Constraints are concrete instantiations of templates, and they can be distinguished into two main categories: existence constraints and relation constraints. Relation constraints can be further divided into three sub-groups: basic relation constraints, mutual relation constraints and negative relation constraints.[5]

Table 2.2. Notation of Declare based constraints and examples of violations and fulfillments.[5]

Template	Examples			Notation
Existence Constraints				
Init(a)	✗ bcaad	✓ adcac	✗ dabce	
AtLeastOne(a)	✗ bbcd	✓ bedcca	✓ bcaeda	
AtMostOne(a)	✗ baea	✓ beed	✓ acccd	
End(a)	✗ abcde	✓ cedea	✓ aacbaa	
Relation constraints				
RespondedExistance(a,b)	✓ bcdd	✓ bdca	✗ addcd	
Response(a,b)	✗ bdaac	✓ adbbc	✓ cddce	
AlternateResponse(a,b)	✗ aabbd	✓ aedbab	✓ beedcd	
ChainResponse(a,b)	✗ cbaab	✓ dabbab	✗ aba	
Precedence(a,b)	✗ cbdac	✓ adbbc	✓ cddcea	
AlternatePrecedence(a,b)	✓ aabd	✗ adbbab	✗ beead	
ChainPrecedence(a,b)	✗ ccbab	✗ abbab	✓ aba	
Mutual relation constraints				
CoExistence(a,b)	✗ bcdd	✓ cddc	✗ addcd	
Succession(a,b)	✗ bdaac	✓ adbbc	✓ cddce	
AlternateSuccession(a,b)	✗ aabbd	✓ addbab	✗ beedcd	
ChainSuccession(a,b)	✗ ccbab	✗ dabbab	✓ dabab	
Negative relation constraints				
NotCoExistence(a,b)	✗ bdaac	✓ adeec	✓ cddce	
NotSuccession(a,b)	✗ aabad	✓ adccda	✓ bcdaed	
NotChainSuccession(a,b)	✗ cbbab	✓ adadbb	✗ aba	

Existence templates are a set of unary templates, and they can be expressed as predicates over one variable. Looking at the first row in Table 1, Init(**a**) is an existence template which requires that **a** occurs in every trace as the first activity, same for End(**a**), but this constraint requires that the activity **a** is executed in any case as last. AtLeastOne(**a**) and AtMostOne(**a**) are also existence templates which specify, as the names suggest, that **a** occurs at least once in any process instance for the first constraint and that **a** occurs none to once for the second constraint.

RespondedExistence(**a,b**) is a relation template imposing that if **a** is executed at least once in the trace, **b** must also occur at least once whether it does it before **a** or after. Response(**a,b**) adds to the previous template the fact that **b** must be executed eventually after **a**. AlternateResponse(**a,b**) in turn adds to Response(**a,b**) more specification, which is that no other **a**'s must be present between the execution of **a** and a subsequent **b**. ChainResponse(**a,b**) directly states that whenever **a** occurs, **b** must occur immediately after. Precedence(**a,b**) requires that **a** must occur before **b**. The AlternatePrecedence(**a,b**) and ChainPrecedence(**a,b**) templates, just like the AlternateResponse(**a,b**) and ChainResponse(**a,b**) templates, follow the same logic for what they add to Precedence(**a,b**), making them stronger and stricter constraints (AlternatePrecedence(a,b): there must be no other occurrence of b between an activation of b and a precedent a, ChainPrecedence(a,b): if b occurs, a must have occurred immediately before).

The other two subgroups in the relation templates are the mutual relation templates and the negative relation templates. In the mutual relation templates group, the constraints have their operand activities as both activation and target. For example, in the CoExistence(a,b) template, if **a** is executed, then **b** must be performed as well, and if **b** occurs, then **a** must occur as well, which simply means the activation at the same time of two constraints which are RespondedExistence(**a,b**) and RespondedExistence(**b,a**). Same logic goes for Succession(a,b) which means the activation of Response(a,b) and Precedence(a,b). On the other hand, in negative relation templates like the NotCoExistence(a,b) template, if activity **a** is executed, then it is required that **b** is not performed in the same trace and vice-versa.[5]

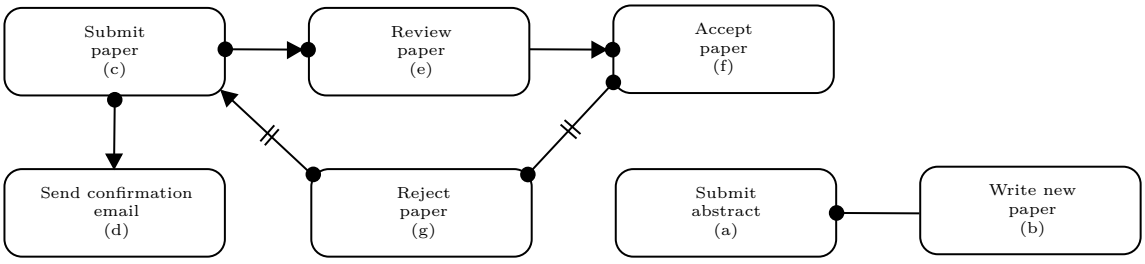


Figure 2.2. This is an example of a process model.[2]

[2]https://github.com/cdc08x/MINERful/wiki/Creating-event-logs

Example 1 (A Declare Process Specification) In Figure 1 it is shown an example of a declarative process model for the publication of a research paper. This example utilizes the Declare templates, and the activities which activate the constraints are *submit abstract*, *write new paper*, *submit paper*, *send confirmation email*, *review paper*, *accept paper* and *reject paper*. The behavior of the process model is defined by the following constraints:

C_1 *RespondedExistence(submit abstract, write new paper)*
C_2 *Response(submit paper, send confirmation email)*
C_3 *Succession(submit paper, review paper)*
C_4 *Precedence(review paper, accept paper)*
C_5 *NotSuccession(reject paper, submit paper)*
C_6 *NotCoExistence(reject paper, accept paper)*

Following the constraints from C_1 to C_6, if an abstract is submitted (a) then whenever in the same trace a new paper must be written (b) this rule can also be expressed *as RespondedExistence(a,b)* which is graphically depicted as. After a paper is submitted (c), a confirmation email must be eventually sent (d) written as *Response(c,d)* and viewed as, and it must be reviewed (e) or if a paper is reviewed then a paper (e) must have been submitted (c) previously which is represented as *Succession(c,e)*. A paper that is accepted (f) must have been previously reviewed (e) *Precedence(e,f)* and a paper cannot be rejected (g) and accepted (f) in the same trace *NotCoExistence(g,f)*. If a paper is rejected (g), it cannot be submitted (c) subsequently *NotSuccession(g,c)*.[5]

2.3 Regular Expressions

A regular expression is a pattern that is used in a regular expression engine to find a match given a text in input. A pattern is made by one or more literal characters, operators or constructs. Regular expressions elaborate text in a way that guarantees efficiency and flexibility. The comprehensive pattern-matching notation enables the user to parse quickly large amounts of text.[3].

The syntax consists of a concatenation of characters of a given alphabet, which are optionally enclosed in parentheses (and), to which operators like the concatenation or binary alternation |, and the unary Kleene star * are applied. In Table 2.3, the regular expressions also abide by the POSIX standard by using the notations of "**.**" which matches any character, [^**x**] which matches any character except x and **?** which matches none to one occurrence of the preceding pattern. To represent constraints like AtMostTwo(a) or AtLeastTwo(a) an additional notation will be used which is the parametric quantifiers n, and m, which respectively specify the minimum number of repetitions of the preceding pattern and the maximum number of repetitions of said pattern with n being an integer higher than or equal to 0 and m being an integer higher than 0. Thus the regular expression (a[^a]*){0,2} identifies

[3]Source: https://docs.microsoft.com/en-gb/dotnet/standard/base-types/regular-expressions

Table 2.3. Declare templates with regular expression translations[5]

Template	Regular Expression
Existence Constraints	
Init(a)	a.*
AtLeastOne(a)	[^a]*(a[^a]*)[^a]*
AtMostOne(a)	[^a]*(a)?[^a]*
End(a)	.*a
Relation constraints	
RespondedExistence(a,b)	[^a]*((a.*b.*)|(b.*a.*))*[^a]*
Response(a,b)	[^a]*(a.*b)*[^a]*
AlternateResponse(a,b)	[^a]*(a[^a]*b[^a]*)*[^a]*
ChainResponse(a,b)	[^a]*(ab[^a]*)*[^a]*
Precedence(a,b)	[^b]*(a.*b)*[^b]*
AlternatePrecedence(a,b)	[^b]*(a[^b]*b[^b]*)*[^b]*
ChainPrecedence(a,b)	[^b]*(ab[^b]*)*[^b]*
Mutual relation constraints	
CoExistence(a,b)	[^ab]*((a.*b.*)|(b.*a.*))*[^ab]*
Succession(a,b)	[^ab]*(a.*b)*[^ab]*
AlternateSuccession(a,b)	[^ab]*(a[^ab]*b[^ab]*)*[^ab]*
ChainSuccession(a,b)	[^ab]*(ab[^ab]*)*[^ab]*
Negative relation constraints	
NotCoExistence(a,b)	[^ab]*((a[^b]*)|(b[^a]*))?
NotSuccession(a,b)	[^a]*(a[^b]*)*[^ab]*
NotChainSuccession(a,b)	[^a]*(aa*[^ab][^a]*)*([^a]*|a)

any string that starts with an **a** followed by any character that is not **a** with a minimum of 0 times to a maximum of 2 times. The expressive strength of Regular Expressions entirely covers any regular language meaning that for every Regular Expression a corresponding Finite State Automaton (FSA) exist, which accepts all and only the matching patterns [5].

2.4 Finite State Automata

A deterministic Finite State Automaton is a 5-tuple A = (Σ, S, δ, s_0, S_f), where:

- Σ *is a finite set of symbols a.k.a an alphabet;*
- *S is a finite non-empty set of states;*

- $\delta : S \times \Sigma \rightarrow S$ *is a transition function which is a function that given a starting state and a symbol from the alphabet, returns the target state (if defined);*
- $s_0 \in S$ *is the initial state*;
- $S_f \subseteq S$ *is a non-empty set of accepting states* $s_f \in S$

For the sake of simplicity, we will omit the qualification of "deterministic" in the remainder of this paper. We will also assume that the transition function δ acts in the following way: given a finite path π of length n over Σ, $\pi = \langle\ \pi^1, .., \pi^n\ \rangle$ is a sequence of tuples $\pi^i = \langle\ s^{i-1}, \sigma^i, s^i\ \rangle \in \delta$ in which the first tuple π^1 has $s^0 = s_0$ and the starting state of π^i is the target state of π^{i-1}. Thus $\pi = \langle\ \langle\ s^0, \sigma^1, s^1\ \rangle, \langle\ s^1, \sigma^2, s^2\ \rangle, ..., \langle\ s^{n-1}, \sigma^n, s^n\ \rangle\ \rangle$. A finite string of length n ≥ 1 is accepted if given the characters c of the string, there exists a path δ with σ in every tuple replaced with a characters c of the string, $s^n \in S_f$ with $\pi^n = \langle\ s^{n-1}, c^n, s^n\ \rangle$. Thus we would call the computation π a run [7] [5].

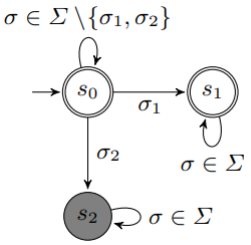


Figure 2.3. Precedence FSA[7]

Example 2 *In Figure 2.3,* $\pi^1 = \langle\ \langle\ s^0, \sigma^1, s^1\ \rangle\ ,\ \langle\ s^1, \sigma^1, s^1\ \rangle\ \rangle$, $\pi^2 = \langle\ \langle\ s^0, \sigma^2, s^2\ \rangle\ \rangle$, $\pi^3 = \langle\ \langle\ s^0, \sigma^2, s^2\ \rangle\ ,\ \langle\ s^2, \sigma^1, s^2\ \rangle\ \rangle$ *are three path examples but only* π^1 *is a run because* $s^1 \in S_f$.

2.5 MINERful

Using the declarative approach to process mining, as opposed to the imperative approach, brings quite a few advantages which are naturally carried with the declarative approach. The basic functions of declarative miners are described by making use of MINERful [8]. MINERful is a process mining tool based on the previously described declarative process modelling language Declare [9].

The main functionalities of MINERful:

- Discovery of declarative specifications of process control-flows out of event logs. Event logs can be either real or synthetic, stored as XES [15], MXML, or text files (a collection of strings, in which every character is considered as an event, every line as a trace)[4];
- Simulation of declarative specifications of process control-flows and creation of synthetic event logs. The input process specification can be given as a JSON file. The event logs can be exported as XES or MXML files[12];

[4]Source: https://github.com/cdc08x/MINERful/wiki

- Simplification, and inconsistency removal, of declarative specifications of process control-flows.

2.5.1 Discovery

The Discovery of declarative specifications based on an event log also returns metadata that was generated during the application of the algorithm. This metadata contains information about the number of cases in the log, the minimum, the maximum and average amount of events for trace, the number of events overall, and the number of symbols used which represent each event.

It also returns some support, confidence and interest factor metrics [14]. The support metric provides a view of the fulfilment of a certain constraint in percentages. It is calculated by dividing the number of occurrences of the activation by the number of traces for relation constraints, while for existence constraints, the number of traces divides the number of fulfilment. For example, if an existence constraint has a support value of 100%, it means that it is present in every trace. For relation constraints to have a support value of 100%, all the occurrences of the activation must fulfil the underlying constraint. The confidence metric is calculated with the product of the support metric of an activity or activation and the number of traces in which said activity or activation occurs[8]. The interest factor metric, just like the confidence metric, deals with the relevance of an activity or target. It describes how much a certain combination of events is repeated in all traces, and it is calculated by the product of the confidence metric with the fraction of traces in which the activity or target occurs. By using the discovery process function, the user is able to set various thresholds for each metric.

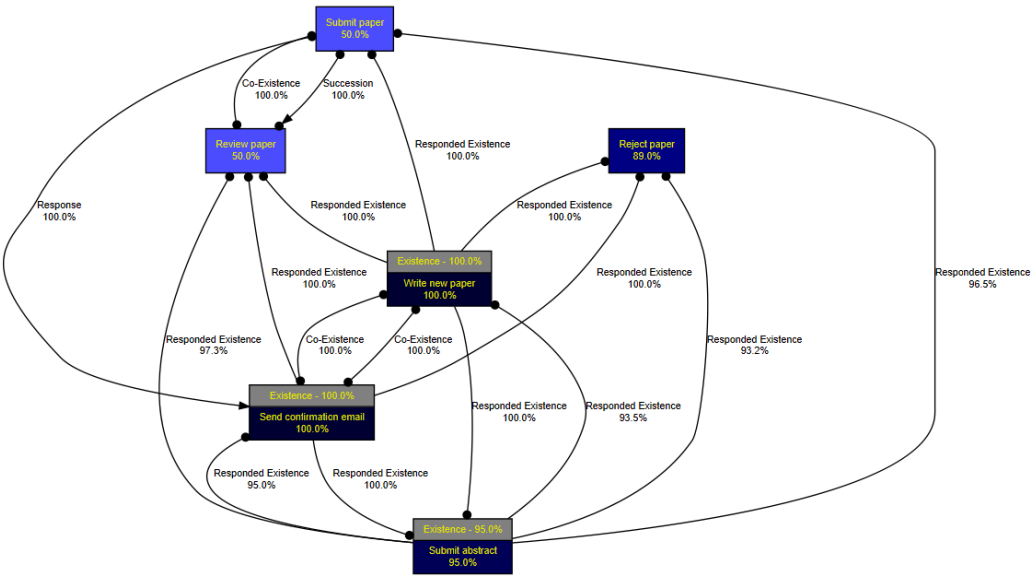


Figure 2.4. Example of a discovered process model using an event log and the application RuM.[5]

[5]https://rulemining.org

2.5.2 Simulation

Another function of MINERful is the Simulation which can be used to generate event logs based on declarative specifications, which are provided in a JSON, MXML, XES or text format and contains declarative constraints. This function is useful when it comes to analysing, testing, and refining discovery algorithms since, as opposed to real-world logs, the logs generated do not contain any **fraudulent** traces, which could greatly influence the discovery process. While using the simulation function to generate event logs, the user is able to define the number of traces that should be created and the minimum and maximum amount of events per trace. The process can be divided into the following steps:

1. **Conversion from constraints to characters**

   MINERful creates a process alphabet in which each activity recognised in the process specification is transformed into a character. Each character is unique from the other to make it possible to determine the name of an activity based on them.

2. **Constraints are translated into regular expressions**

   This is done using the regular expression templates defined in Table 2.3, and the characters of the regular expression of the constraint used are replaced with the ones made in the first step.[10]

3. **Regular expressions are then turned into FSA**

   After all of the constraints have been transformed into regular expressions, they are transformed again into finite state automata.

4. **Product of FSAs**

   Each FSA created in step 3 is then fused together by calculating the product of all the FSAs. This product is made to fulfil all the constraints and length specifications at the same time, and it is used to generate the traces in the event log.

5. **String generation**

   The strings that represent the traces in the event log are generated by traversing the FSA created in step 4 with a randomised path, and while passing through the various transitions, characters, which were the transformation of the activities, are added to the string. The length of each string in a trace is decided at random in a range specified by the user.

6. **Generating Event logs**

   The final traces that compose the event log are made by swapping the characters in each string back to their activity names. The generated event log can be exported into a text file, XES or MXML files.

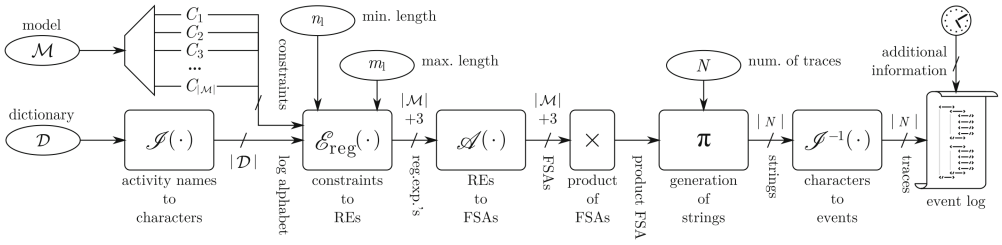


Figure 2.5. MINERful's log generation framework [5]

2.5.3 Simplification

Simplification is another function brought by MINERful, which provides different post-processing techniques to clarify better a declarative process specification. Said specifications can have two major flaws: they could lack in consistency which means having two or more constraints that cannot coexist at the same time, for example, the constraints Init(a), End(a) and AtMostOne(a) cannot be fulfilled at the same time because by specification each trace starts and ends with activity a so the activity a is already used more than once. These conflicting constraints must be detected and removed from the final process map. The other flaw could be constraint redundancy, for example, having the two constraints NotCoExistence(a,b) and NotChainSuccession(a,b) in the specification is considered a redundancy because the constraint NotChainSuccession(a,b) is already covered by the constraint NotCoExistence(a,b), if activity a and b cannot coexist, they cannot be put right after the other either. The last problem is resolved by creating another process map which shows the same language of the initial map but without the redundancies as addressed by Di Ciccio [6].

Chapter 3

Approach

The approach to generating event logs with violated traces starts by having a clear idea of what it means to negate a declarative specification. By having a working system of negative constraints, it is possible to build a new FSA. This new FSA, which is made with the product of FSAs, a process previously described in the simulation function provided by MINERful in subsection 2.5.2, will have as some of its factors, FSAs built based on the negated constraints. The constraints to be negated are user inputted, and if in a trace, a constraint is not fulfiled, then the entire trace is violated. To be able to fulfil a declarative process model, all of the constraints which compose its specification must be fulfiled.

The work I have done can be divided into three parts:

Creating new command line parameters to enable the user to negate, which traces to negate and how many to negate, which is a value that can go from 0 to the maximum number of traces also inputted by the user.

Creating new functions to make automata that support the negative constraints and to put them in the product with the other FSAs. The new negative automaton is built by having the constraints taken in input by command line or by text file and transforming them into Regular expressions. These negative regular expressions are negated from the usual ones.

Create a negative regular expression function and negative ltl function for each constraint, making sure that they are complete negatives and not conflicting negatives. Negating existence constraints are relatively easier than negating relation constraints.

If in the options, no constraint is specified, but there is an amount of traces to negate on the options, then for that amount of traces, all constraints are negated.

3.1 Negative constraints

For negative constraints, the concept is fairly simple. Depending on the constraint to negate, different approaches are needed to have an opposite result to what a

constraint normally fulfils. To negate a declarative template, one must understand its formal specification first. For example, the RespondedExistance(a,b) constraint requires that if activity a occurs, then activity b also occurs. This specification can be negated as if activity a occurs, then activity b must not occur. The main problem that presents itself by doing this is how permissive is the negated constraint. If the permissive approach is taken, then a situation like having two contradicting constraints can be fulfilled at the same time. For example, if in the specification provided by the user, the constraints Succession(a,b) and NotSuccession(a,b) are present, then in a trace like $t = \langle$ c, c, d, c $\rangle$ where neither activity a nor activity b is present, both constraints are fulfiled. Here we present a "non-permissive" approach for both existence constraints and relation constraints.[11]

3.2 Negate Regular Expressions

To make an example, the init(a) constraint, which requires that every trace starts with an a, can be negated by starting every trace with any character different from a.

3.2.1 Negate Existence constraints

In this subsection, an explanation of each negation will be provided with the initial specification, the regular expression and then the negated ones. The first six constraints have a bijective relation with other constraints present in MINERful that negate each other, while the constraint templates to negate the last six constraints are not implemented in MINERful.

- ***Absence(a)***: a doesn't occur in the trace, the regular expression is represented as [^a]* and to negate this constraint then a must occur at least once, so the negated regular expression is [^a]*(a[^a]*){1,}[^a]*
- ***AtLeast1(a)***: this constraint is the negative of the previous one so that means that to negate AtLeast1(a), Absence(a) must be fulfiled
- ***AtLeast2(a)***: activity a must occur at least 2 time, the regular expression of this constraint is [^a]*(a[^a]*){2,}[^a]*. To negate such specification, the activity a, is required to occur at most once, the regular expression is [^a]*(a[^a]*){0,1}[^a]*
- ***AtLeast3(a)***: this constraint requires the occurrence of activity a at least three times. The regular expression is: [^a]*(a[^a]*){3,}[^a]*. To negate this constraint, activity a mustn't occur more than twice and it negative regular expression is: [^a]*(a[^a]*){0,2}[^a]*
- ***AtMost1(a)***: activity a must occur zero to one time, and the negative constraint is *AtLeast2(a)*
- ***AtMost2(a)***: activity a must occur at most twice, and its negative correspondence is the constraint AtLeast3(a)

- ***AtMost3(a)***: activity a must occur at most thrice and its regular expression is [^a]*(a[^a]*){0,3}[^a]*. Thus to negate this constraint, activity a must occur at least four times which is a specification that has no constraint template in MINERful so the regular expression of such specification is: [^a]*(a[^a]*){4,}[^a]*
- ***End(a)***: activity a is required to occur at the end of each trace; the regular expression is .*a. Negating this constraint means to have as the last character of each trace any activity but a, and the regular expression is: .*[^a]
- ***Exactly1(a)***: activity a must occur exactly one time. The regular expression is: [^a]*(a[^a]*){1,1}[^a]*. The negation of such constraint, is a combination of two constraint: *AtLeast2(a)* and *Absence(a)*. The resulting regular expression of such combination is ([^a]*(a[^a]*){2,}[^a]*)|[^a]*
- ***Exactly2(a)***: activity a must occur exactly twice and the regular expression is [^a]*(a[^a]*){2,2}[^a]* and just like the previous constraint, the negation is a combination of two constraints which are: *AtLeast3(a)* and *AtMost1(a)*. The regular expression of the negated constraint thus is:

  ([^a]*(a[^a]*){3,}[^a]*)|([^a]*(a[^a]*){0,1}[^a]*)
- ***Exactly3(a)***: activity a must occur exactly three times and the regular expression is [^a]*(a[^a]*){2,2}[^a]* and just like the previous two constraints, the negative constraint's regular expression is a combination of the regular expressions of the constraints AtLeast4(a) which doesn't have a template definition in MINERful and *AtMost2(a)*. The result is:

  ([^a]*(a[^a]*){4,}[^a]*)|([^a]*(a[^a]*){0,2}[^a]*)
- ***Init(a)***: just like the End(a) constraint, the specification and the negations are really simple, respectively: any trace must start with activity a, a.*, any trace starts with any activity character except a, [^a].*

3.2.2 Negate Relation constraints

For relation constraints, negating them gets a bit more complicated than the existence ones. Firstly because they are activated with the occurrence of some activity and respond accordingly, this means that if the activation condition is not met, but it is not violated either, then the constraint is fulfilled regardless. Relation constraints are fulfilled only and only if they are fulfilled in the entire trace. So to negate them means that they have to be violated at least once in the trace. In the following list, a solution for each relation template will be provided to create their negated counterparts. Furthermore, the regular expression of the constraints which didn't appear in Table 2.3 will be provided as well.

- ***RespondedExistence(a,b)***: if activity a occurs, then b has to occur as well, whether it is before or after a is activated. With this specification, the negative specification is that b can occur if a doesn't occur in the entire trace, and if a occurs, then no b must occur whether it is before or after the activation of a. The regular expression for the negated specification is: [^ab]*((a[^b]*)|(b[^a]*))?

- ***Response(a,b)***: if activity a occurs, b has to occur in the trace as well after a. The negative specification of this constraint, let's activity b occur before the activation of a but never after. So the regular expression of the negated constraint is: [^a]*(a[^b]*){1,}[^ab]*
- ***AlternateResponse(a,b)***: If a occurs, b has to occur after a, with no a in-between which means that to negate such constraint, activity a can occur more then once before the instance of a b and the regular expression which represents this behaviour is: [^a]*(a[^b]*a[^a]*){1,}[^a]*
- ***ChainResponse(a,b)***: if a occurs, b has to occur immediately after a. So any activity other than b could occur immediately after a to negate the specification this must happen at least once in the trace. The regular expression is: [^a]*(ab[^a]*)*(a[^b]){1,}([^a]*|a*)
- ***Precedence(a,b)***: if activity b occurs, then activity a must have occurred in the trace before b. To negate this constraint means that activity a must not occur before a b at least once. An example could be a trace starting with activity b. The regular expression obtained is [^a]*([^a]*b){1,}[^b]*
- ***AlternatePrecedence(a,b)***: Each time activity b occurs, it is preceded by a with no b in between. So to negate this constraint, there must be at least another b between an occurrence of activity a and an occurrence of activity b. The regular expression to match this specification is [^a]*([^b]*b[^a]*b){1,}[^a]*
- ***ChainPrecedence(a,b)***: If activity b occurs, then a has to occur immediately beforehand. So immediately before an occurrence of activity b, there must be an activity different from a. The regular expression is [^a]*(([^a]|ab)*b){1,}([^b]*|b*)
- ***CoExistence(a,b)***: If activity a occurs then also activity b has to occur and vice-versa. to negate this constraint there needs to be a mutual exclusion of these activities. And the direct negation of such constraint is NotCoexistance(a,b) and the regular expression is: [^ab]*((a[^b]*)|(b[^a]*))?
- ***Succession(a,b)***: This constraint is the combination of the two constraints: Response(a,b) and Precedence(a,b). The regular expressions to negate this constraint is [^ab]*(a[^b]*)*[^ab]*
- ***AlternateSuccession(a,b)***: Just like the previous constraint, this one is also a combination of two constraint which are: AlternateResoponse(a,b) and AlternatePrecedence(a,b). So the negative regular expression is:

  [^ab]*(a[^ab]*(a|b)*[^ab]*b){1,}[^ab]*
- ***ChainSuccession(a,b)***: Like the previous two constraints, this one is a combination of two constraints as well. The two constraits are ChainResponse(a,b) and ChainPrecedence(a,b). The negation of this constraint, is a combination of the negation of the constraints that is made of and the regular expression is [^ab]*((a[^b]*){1,})|(([^a]*b){1,})([^ab]*|(a|b)*)

- ***Choice(a,b)***: the constraint is fulfiled if either a or b is in the trace, the regular expression of this constraint is: [^ab]*([ab][^ab]*){1,}[^ab]*. So to negate this constraint, activity a and b must never occur in the trace. The regular expression shows the absence of a and b: [^ab]*
- ***ExclusiveChoice(a,b)***: this constraint requires the activation of either a or b but not both in the same trace and the regular expression of this constraint is: ([^b]*a[^b]*) | ([^a]*b[^a]*)

To negate the following negative constraint, the only thing to change from their positive counterparts is the quantifier of the search pattern in the middle of each and every one of their regular expressions.

- ***NotRespondedExistence(a,b)***: this constraint's positive counterpart is the constraint RespondedExistence(a,b) and the corresponding modified regular expression is: [^a]*((a.*b.*)|(b.*a.*)){1,}[^a]*
- ***NotResopnse(a,b)***: this constraint's positive counterpart is the constraint Response(a,b) and the corresponding modified regular expression is:

  [^a]*(a.*b){1,}[^a]*
- ***NotChainResponse(a,b)***: this constraint's positive counter part is the constraint ChainResponse(a,b) and the corresponding modified regular expression is: [^a]*(ab[^a]*){1,}[^a]*
- ***NotPrecedence(a,b)***: this constraint's positive counter part is the constraint Precedence(a,b) and the corresponding modified regular expression is:

  [^b]*(a.*b){1,}[^b]*
- ***NotChainPrecedence(a,b)***: this constraint's positive counter part is the constraint ChainPrecedence(a,b) and the corresponding modified regular expression is: [^b]*(ab[^b]*){1,}[^b]*
- ***NotCoExistence(a,b)***: this constraint's positive counter part is the constraint CoExistence(a,b) and the corresponding modified regular expression is:

  [^ab]*((a.*b.*)|(b.*a.*)){1,}[^ab]*
- ***NotSuccession(a,b)***: this constraint's positive counter part is the constraint Succession(a,b) and the corresponding modified regular expression is:

  [^ab]*(a.*b){1,}[^ab]*
- ***NotChainSuccession(a,b)***: this constraint's positive counter part is the constraint ChainSuccession(a,b) and the corresponding modified regular expression is: [^ab]*(ab[^ab]*){1,}[^ab]*

3.3 Log generation options

The three log maker parameters that were added with their subsequent implementations were **nagtivesInLog**, **negativeConstraints** and **negativeconstraintsFile**.

NegativesInLog is the value that can be inputted by the user to specify how many traces to be negated should be present in the log. The value set for the amount of negated traces should be lower than the number of traces in the log, which is also set by the user. If not, an error may occur. This option only works if paired with the next two parameters. In the overall framework, this value is used to determine how many "negative runs"[7] (section 2.4) should the created FSA make and when it should stop.

NegativeConstraints and **negativeConstraintsFile** are similar in that they provide the means for the user to choose which constraints in the process model to negate. These parameters can work at the same time. The *negativeCoinstraintsFile* option was made for possible issues that could arise in the *negativeConstraints* option. Due to the fact that with the *negativeConstraints* option, the constraints to be negated are written on command line, the possibility of spelling mistakes rises even more so if the amount of such constraints is large.

In the **negativeConstraintsFile** option, the user specifies a text file, normally called constraints.txt, in which the constraints to be negated are written line by line to attempt to avoid the major problem of the previous parameter.

As stated previously, the last two options have the possibility to work concurrently by adding constraints to be negated in each parameter, the constraints that are repeated in both options are taken once, and the rest are put in a set union to be combined. The constraints that are not present in the process specification model are ignored and do not bring any changes to the final result of the event log generation.

3.4 Minerful Simulation Modifications

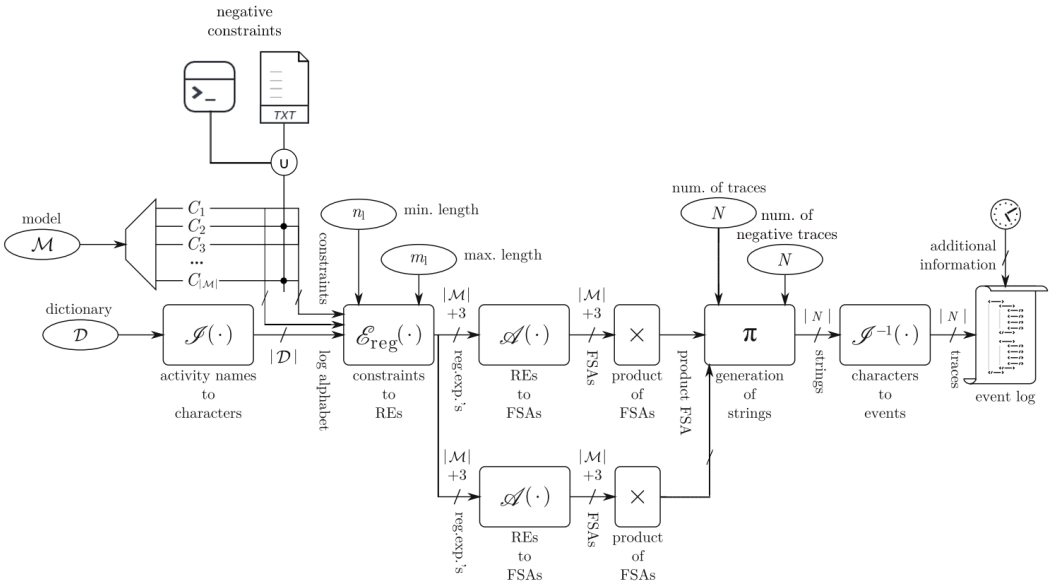


Figure 3.1. The modified event-log generation framework [8]

From the simulation steps already mentioned in subsection 2.5.2, a further step is added between step 2, in which the constraints are translated into regular expressions, and step 3, in which those regular expressions are transformed again into FSAs. In between these two steps, the regular expressions of the constraints chosen to be negated by the user are switched to their negative counterpart. Thus these negative regular expressions, along with the regular expression of the constraints that were not negated, are transformed into FSAs. Depending on the fact that the negativeInLog parameter is set or not by the user, the final FSA made by the product of constraints' FSAs may differ in architecture. If the negativeInLog value is not set, only one final FSA is built, and the chosen constraints to be negated are negated for all the traces. But if said value is set, two final FSA are used to generate the various strings which represent the final event log's traces. The first final FSA is the positive FSA which is made with the product of FSAs of each constraint without any negations whatsoever. The second final FSA is the one made with the FSAs of the negated and non-negated constraints. Depending on which trace the iteration is on, the string generated is made by traversing one of the two final FSAs with a randomised path and adding to said string activity characters. The final step is unchanged, and the generated event log conforms to the specification provided by the process model and the negated constraints.

Chapter 4

Implementation

In this chapter, we explore in more detail the work done for the implementation of the functions added to negate the various constraints and the modifications made to the existing files in the MINERful framework. The open-source code, along with the changes that have been brought to MINERful, can be viewed on Github[1].

4.1 Inputs

In order to generate an event log, a few options and specifications must be provided by the user:

- Process specification: in here are the various templates and activities that form the process model taken in input;
- the format of said process specification: which can be submitted through a JSON, XES, MXML or text file;
- the amount of traces which should be generated and put in the final event log;
- the minimum and maximum amount of events in each trace. These values are taken as an upper and lower bound in which an integer is chosen randomly for each trace to represent the number of events there should be in that trace.
- the format of the output file, which can be produced as a XES, MXML or text file.
- the location and the name of the output file which is going to be generated.
- which constraints of the process model to negate
- the amount of traces in which the chosen negated constraints should be applied.

[1] https://github.com/R4Lpf/MINERful

```
PS C:\Users\aralm\Documents\GitHub\MINERful> ./run-MINERfulEventLogMaker.sh -iMF 'processesCREA
TO/exampleprocess.json' -iME json -oLL 100 -oLm 10 -oLM 100 -oLE xes -oLF 'logsCREATO2/TEST1.xe
s' -oLN 50 -NC Succession -iNC '/home/ralph/Desktop/minerful3/constraints.txt'
```

Figure 4.1. Example of specifications for the generation of an event log.

In Figure 4.1, is provided an example of a command to generate an event log starting from a process model provided in the *exampleprocess.json* file, which has a JSON extension. This event log must have 100 traces, and every trace must have a number of events between 10 and 100. The output file is in a XES encoding, and it is named *TEST1.xes*. The template to be negated with the -NC option is Succession, while in the *constraints.txt* file, which also has constraints to be negated, is present the Response template.

```
{ "name": "Scientific paper evaluation process",
    "constraints": [
                {"template": "respondedexistence", "parameters": [["Submit abstract"],["Write new paper"]]},
                {"template": "response", "parameters": [["Submit paper"],["Send confirmation email"]]},
                {"template": "succession", "parameters": [["Submit paper"],["Review paper"]]},
                {"template": "precedence", "parameters": [["Review paper"],["Accept paper"]]},
                {"template": "notsuccession", "parameters": [["Reject paper"],["Submit paper"]]},
                {"template": "notcoexistence", "parameters": [["Accept paper"],["Reject paper"]]}
  ]
}
```

Figure 4.2. Example of process model specification in a JSON file.

In Figure 4.2, is an example of what a process model looks like in a JSON file. Which is also the written specification of the process model shown in Figure 2.2.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- This file has been generated with the OpenXES library. It conforms -->
<!-- to the XML serialization of the XES standard for log storage and -->
<!-- management. -->
<!-- XES standard version: 1.0 -->
<!-- OpenXES library version: 1.0RC7 -->
<!-- OpenXES is available from http://www.openxes.org/ -->
<log xes.version="1.0" xes.features="nested-attributes" openxes.version="1.0RC7">
    <extension name="Time" prefix="time" uri="http://www.xes-standard.org/time.xesext"/>
    <extension name="Lifecycle" prefix="lifecycle" uri="http://www.xes-standard.org/lifecycle.xesext"/>
    <extension name="Concept" prefix="concept" uri="http://www.xes-standard.org/concept.xesext"/>
    <classifier name="Event Name" keys="concept:name"/>
    <string key="concept:name" value="Synthetic log for process: Scientific paper evaluation process"/>
    <string key="lifecycle:model" value="standard"/>
    <trace>
        <string key="concept:name" value="Synthetic trace no. 50"/>
        <event>
            <string key="concept:name" value="Write new paper"/>
            <string key="lifecycle:transition" value="complete"/>
            <date key="time:timestamp" value="2021-01-29T13:08:12.378+01:00"/>
        </event>
        <event>
            <string key="concept:name" value="Submit abstract"/>
            <string key="lifecycle:transition" value="complete"/>
            <date key="time:timestamp" value="2021-01-30T11:30:09.280+01:00"/>
        </event>
        <event>
            <string key="concept:name" value="Send confirmation email"/>
            <string key="lifecycle:transition" value="complete"/>
            <date key="time:timestamp" value="2021-01-30T13:53:41.185+01:00"/>
        </event>
        <event>
            <string key="concept:name" value="Write new paper"/>
            <string key="lifecycle:transition" value="complete"/>
            <date key="time:timestamp" value="2021-01-30T22:52:47.919+01:00"/>
        </event>
        <event>
            <string key="concept:name" value="Send confirmation email"/>
            <string key="lifecycle:transition" value="complete"/>
            <date key="time:timestamp" value="2021-01-31T14:44:14.179+01:00"/>
        </event>
        <event>
            <string key="concept:name" value="Submit paper"/>
            <string key="lifecycle:transition" value="complete"/>
            <date key="time:timestamp" value="2021-02-01T00:24:28.023+01:00"/>
        </event>
        <event>
            <string key="concept:name" value="Submit paper"/>
            <string key="lifecycle:transition" value="complete"/>
```

Figure 4.3. Event log obtained with the command in Figure 4.1.

In Figure 4.3, is the final output of an even log generation with added timestamps to the created events.

4.1.1 Log Maker Parameter changes

In order to add parameters to the already existing ones in the LogMakerParameter.java file, other than the new parameters themselves and their default values, new constructors for each combination of parameters and an additional set of rules which mainly focus on the number of negated traces:

- The number of negated traces cannot exceed the number of traces already set by the user.
- The number of negated traces cannot be an integer lower than zero.

4.2 MinerFulLogMaker

In the MinerFulLogMaker.java file, the main changes were the addition of an inspection on the constraints to be negated, a new automaton for the negative constraints and a separated loop for the new automaton.

If both the parameters of the negative constraints file and the constraints inputted on the command prompt to be negated are empty, then no constraint is negated. If the constraints to be negated are specified in both options, then a set union is performed between the two sets, and if either of the two parameters is missing, then the only constraints to be negated are the ones on the available parameter.

4.2.1 Building the automatons

New functions were implemented to build the automaton with negated constraints, which is similar in structure to the original function to build the final positive automaton. This automaton is then used to run a *walker*, which generates a random path to follow to create the future strings for the traces.

```
protected Automaton buildAutomatonByBondHeuristic(String nc) {
    Collection<String> regularExpressions = null;
    Collection<Constraint> constraints = LinearConstraintsIndexFactory.getAllUnmarkedConstraintsSortedByBondsSupportFamilyConfidenceInterestFactorHierarchyLevel(this.bag);
    regularExpressions = new ArrayList<String>(constraints.size());
    String[] ncobj = Arrays.stream(nc.split(regex: ",")).map(String::trim).toArray(String[]::new);
    List<String> ncList = Arrays.asList(ncobj);
    for (Constraint con : constraints) {
        String type = con.getClass().getCanonicalName().substring(con.getClass().getCanonicalName().lastIndexOf(ch: '.') + 1);
        if (ncList.contains(type)){
            regularExpressions.add(con.getNegativeRegularExpression());
        }
        else {
            regularExpressions.add(con.getRegularExpression());
        }
    }
    return AutomatonFactory.fromRegularExpressions(regularExpressions, this.taskCharArchive.getIdentifiersAlphabet());
}

protected Automaton buildNegativeAutomatonByBondHeuristic() {
    Collection<String> regularExpressions = null;
    Collection<Constraint> constraints = LinearConstraintsIndexFactory.getAllUnmarkedConstraintsSortedByBondsSupportFamilyConfidenceInterestFactorHierarchyLevel(this.bag);
    regularExpressions = new ArrayList<String>(constraints.size());
    for (Constraint con : constraints) {
        regularExpressions.add(con.getNegativeRegularExpression());
    }
    return AutomatonFactory.fromRegularExpressions(regularExpressions, this.taskCharArchive.getIdentifiersAlphabet());
}
```

Figure 4.4. The functions used to build automatons

The *buildAutomatonByBondHeuristic()* is used to build the positive final automaton if no argument is added. If this last one is added, which is a string containing the constraints to negate, then the negative automaton is built. For each constraint present in the string argument, the function *getNegativeRegularExpression()* is called, and for the constraints that were not negated, the function called is *getRegularExpression()*.

4.2.2 Constraints files

```java
package minerful.concept.constraint.existence;

import javax.xml.bind.annotation.XmlRootElement;

import minerful.concept.TaskChar;
import minerful.concept.TaskCharSet;
import minerful.concept.constraint.Constraint;
import minerful.concept.constraint.ConstraintFamily.ExistenceConstraintSubFamily;

@XmlRootElement
public class Init extends AtLeast1 {
    @Override
    public String getRegularExpressionTemplate() {
        return "[%1$s].*";
    }

    @Override
    public String getLTLpfExpressionTemplate() {
        return "%1$s"; // a
    }

    @Override
    public String getNegativeRegularExpressionTemplate() {
        return "[^%1$s].*";
    }
```

Figure 4.5. Example of an existence template class

Each call of the *getRegularExpression()* or the *getNegativeRegularExpression()* means calling the *getRegularExpressionTemplate()* and *getNegativeRegularExpressionTemplate()* functions as well. These last two functions are present in each and every constraint template, and they return respectively the regular expressions and the negative regular expressions previously mentioned and shown in Table 2.3 and in section 3.1.

Figure 4.6. Example of a relation template class

```java
package minerful.concept.constraint.relation;

import javax.xml.bind.annotation.XmlRootElement;

import minerful.concept.TaskChar;
import minerful.concept.TaskCharSet;
import minerful.concept.constraint.Constraint;
import minerful.concept.constraint.ConstraintFamily.ConstraintImplicationVerse;

@XmlRootElement
public class Response extends RespondedExistence {

    @Override
    public String getRegularExpressionTemplate() {
        return "[^%1$s]*([%1$s].*[%2$s])*[^%1$s]*";
        // [^a]*(a.*b)*[^a]*
    }

    @Override
    public String getLTLpfExpressionTemplate() {

        return "G(%1$s -> X(F(%2$s)))"; // G(a -> X(F(b)))
    }

    @Override
    public String getNegativeRegularExpressionTemplate() {
        return "[^%1$s]*([%1$s][^%2$s]*){1,}[^%1$s%2$s]*";
    }
```

4.2.3 Traces and log generation

```java
for (int traceNum = (int) (this.parameters.tracesInLog - (int) (this.parameters.negativesInLog * (1))); traceNum < this.parameters.tracesInLog; traceNum++)
    sBuf.append(str: "<");
    walker.goToStart();
    xTrace = xFactory.createTrace();
    concExtino.assignName(
            xTrace,
            String.format(traceNameTemplate, (traceNum))
        );

    pickedTransitionChar = walker.walkOn();
    while (pickedTransitionChar != null) {
        firedTransition = processModel.getTaskCharArchive().getTaskChar(pickedTransitionChar);
        if (traceNum < MAX_SIZE_OF_STRINGS_LOG) {
            sBuf.append(firedTransition + ",");
        }

        currentDate = generateRandomDateTimeForLogEvent(currentDate);
        xEvent = makeXEvent(xFactory, concExtino, lifeExtension, timeExtension, firedTransition, currentDate);
        xTrace.add(xEvent);
        pickedTransitionChar = walker.walkOn();
    }
    this.log.add(xTrace);
    if (traceNum < MAX_SIZE_OF_STRINGS_LOG) {
        this.stringsLog[traceNum] = sBuf.substring(start: 0, Math.max(a: 1, sBuf.length() -1)) + ">";
        sBuf = new StringBuffer();
    }
}
```

Figure 4.7. The FSA made with the product of negative constraints and positive constraints in action.

In figure Figure 4.7, the traces with the negative constraints start their appearance from the difference of the number of overall traces and the number of traces to be negated. The *walker* in this code section, as stated before, uses the automaton generated with the negative constraints and takes a randomised path in it to create the traces. Just like in the positive traces, to each event is added a value to their life cycle transition and a timestamp.

Chapter 5

Validation

In this chapter, we analyse some examples to verify the correct performance of our implementation. These examples have been created to test our changes to the functions in MINERful.

The example process models are taken from: the MINERful wiki , the Generating Event Logs Through the Simulation of Declare Models paper[5] and the Declarative Process Specifications: Reasoning, Discovery, Monitoring paper [7].

To study and analyse the generated event logs, we perform one of the three main types of process mining which is conformance checking. There is a wide choice when it comes to process mining applications which allow the user to apply conformance checking to event logs with a process model. These applications vary from ProM, PM4PY and Celonis, but the process mining application that is going to be utilised is RuM [1]. The choice of using this application comes from its user-friendly interface and easier visualisation of data and end results.

[0]https://github.com/cdc08x/MINERful/wiki/Creating-event-logs
[1]https://rulemining.org

5.1 RuM

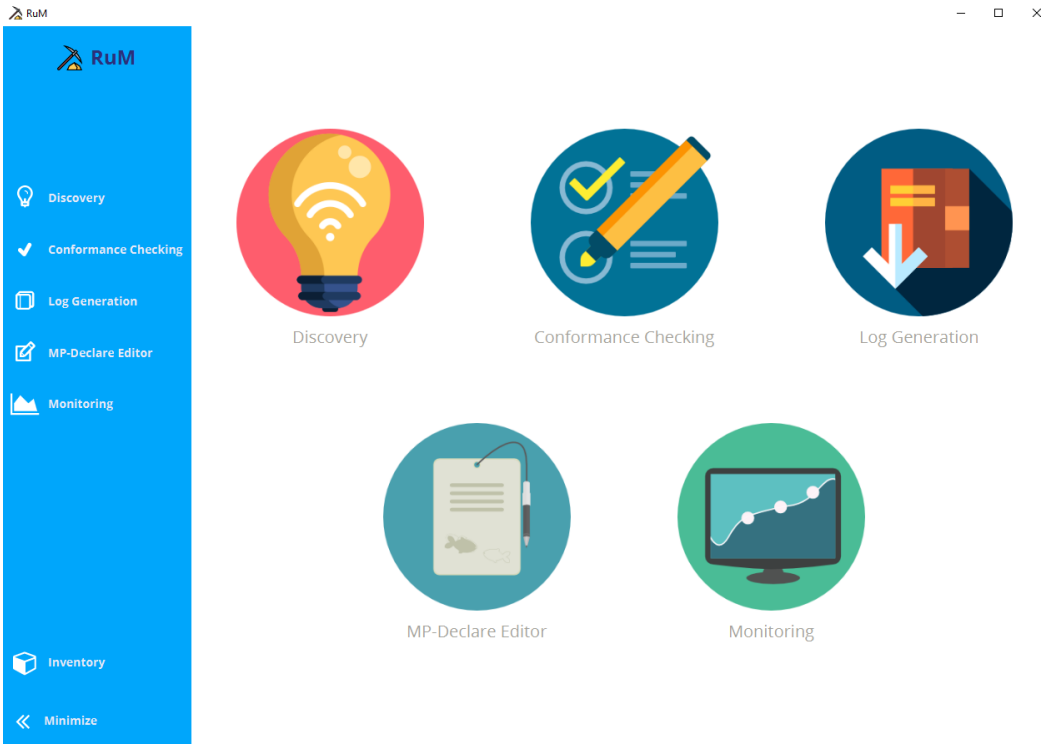


Figure 5.1. RuM's home page

RuM is the first software platform natively designed to analyze declarative, rule-based process models.[2] In the conformance checking section in RuM, the process models are provided as *decl* files while the event logs are inputted as XES files. After the conformance checking process is completed, it is extremely simple to explore which constraint or which trace is violated or fulfilled. RuM is compatible with MINERful; meaning it is able to use the MINERful method for process discovery.

As stated in the Rule Mining in Action: The RuM Toolkit paper [2], RuM implements three main techniques for conformance checking: Declare Analyzer, Declare Replayer and DataAware Declare Replayer. The first technique, supplied with a process model and an event log, returns constraint activation, violation and fulfilment for each constraint in the input log. The Declare Replayer and the DataAware Declare Replayer techniques report trace alignments. The results of a conformance checking application are grouped by either the trace number or the constraints. For each group, is displayed the name of the group along with its statistics.

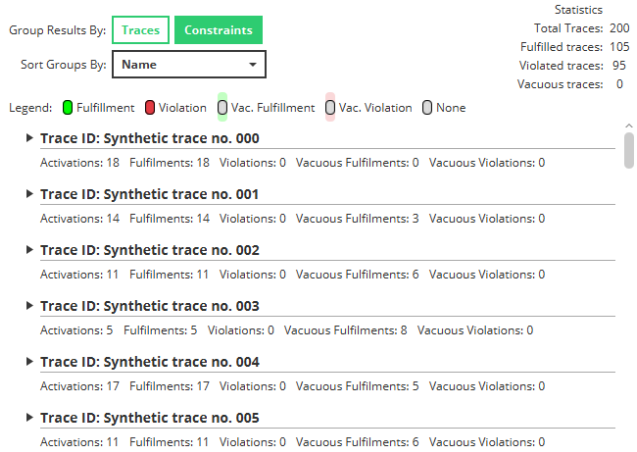


Figure 5.2. RuM's conformance checking page with traces grouping.

As we can see in Figure 5.2, the total number of traces is 200, and among them, only a little bit over half is fulfilled. The rest is violated, and if one of the traces is viewed, is it possible to analyse the behaviour of the constraints in that trace.

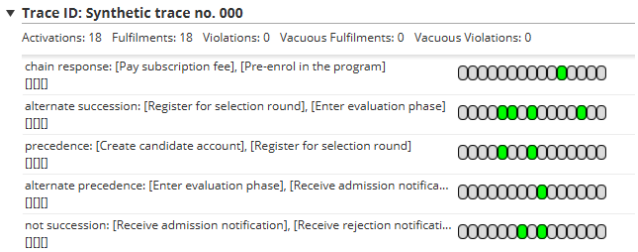


Figure 5.3. Example of what is viewed in a trace after conformance checking.

5.2 Scientific paper evaluation process

```
activity Write new paper
activity Accept paper
activity Send confirmation email
activity Submit abstract
activity Submit paper
activity Review paper
activity Reject paper
Responded Existence[Submit abstract, Write new paper] | | |
Response[Submit paper, Send confirmation email] | | |
Succession[Submit paper, Review paper] | | |
Precedence[Review paper, Accept paper] | | |
Not succession[Reject paper, Submit paper] | | |
Not co-Existence[Accept paper, Reject paper] | | |
```

Figure 5.4. Scientific paper evaluation process model.

This process model is the most used one to test the performance of the implementation previously described. Mostly using the default values for the number of traces and minimum and maximum amount of events for each trace. The first test consisted of the negation of all constraints in the model. Even though a trace is

violated by just one constraint being violated, we wanted to check the limits of the functions.Figure 5.5

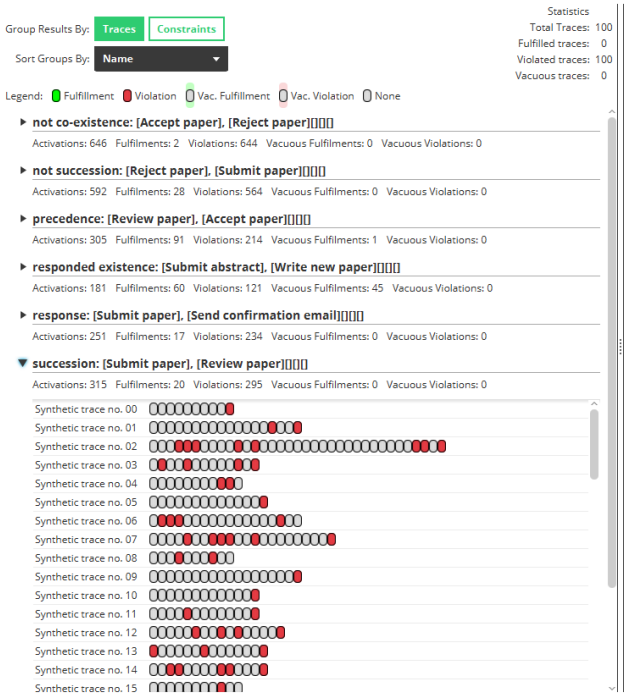


Figure 5.5. Event log with all traces and constraints negated.

So for further clarification, the negated constraint file had the following constraints to negate:

- RespondedExistence
- Response
- Succession
- Precedence
- NotSuccession
- NotCoExistence

Depending on how influential to the overall process model a constraint is, negating only one of them leads to different results. The time it takes to generate the event logs varies in function of how many activations such a constraint had.

5.3 Fracture treatment process

```
activity examine patient
activity check X ray risk
activity perform X ray
activity perform reposition
activity apply cast
activity perform surgery
activity remove cast
activity prescribe rehabilitation
Init[examine patient] | |
Alternate Precedence[check X ray risk, perform X ray] | | |
Precedence[perform X ray, perform reposition] | | |
Precedence[perform X ray, apply cast] | | |
Precedence[perform X ray, perform surgery] | | |
Succession[apply cast, remove cast] | | |
Response[perform surgery, prescribe rehabilitation] | | |
```

Figure 5.7. Fracture treatment process model.

With this process model, the goal was to understand the behaviour of our implementation with a larger amount of traces than in the last example and to understand what happens if a constraint is negated only in a limited amount of traces in a generated event log.

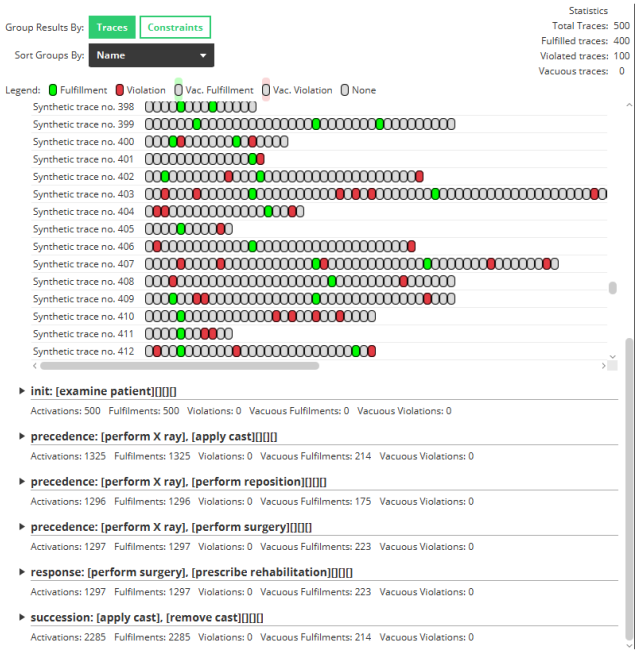


Figure 5.8. Fracture treatment conformance checking.

In Figure 5.8, the number of traces in the log is 500 and the constraint to negate is AlternatePrecedence which has to be negated in one hundred traces (trace #400 - trace #499). Even though the amount of traces increased in this example, the number of events remained in the range between ten to a hundred. The time to generate said event log increased on average to a value of 550 milliseconds.

5.4 Admission process at a university

```
activity Create candidate account
activity Register for selection round
activity Upload admission test score
activity Enter evaluation phase
activity Receive rejection notification
activity Receive admission notification
activity Pre-enrol in the program
activity Enrol in the program
activity Pay subscription fee
activity Upload Certificates
Init[Create candidate account] | |
Precedence[Create candidate account, Register for selection round] | | |
Alternate Succession[Register for selection round, Enter evaluation phase] | | |
Precedence[Upload admission test score, Enter evaluation phase] | | |
Alternate Precedence[Enter evaluation phase, Receive rejection notification] | | |
Alternate Precedence[Enter evaluation phase, Receive admission notification] | | |
Not Succession[Receive admission notification, Receive rejection notification] | | |
Precedence[Receive admission notification, Pre-enrol in the program] | | |
Precedence[Pay subscription fee, Pre-enrol in the program] | | |
Chain Response[Pay subscription fee, Pre-enrol in the program] | | |
Precedence[Pre-enrol in the program, Enrol in the program] | | |
Precedence[Upload Certificates, Enrol in the program] | | |
```

Figure 5.9. University acceptance process model.

In this last example's process model, which compared to the previous two is vaster, the conformance checking was done with generated event logs with 10000 traces. The number of events varied from one hundred events to five hundred events. Here the goal was to limit-test the functions to understand how they behave with a bigger set of data to generate and how much longer it would take to create the event log.

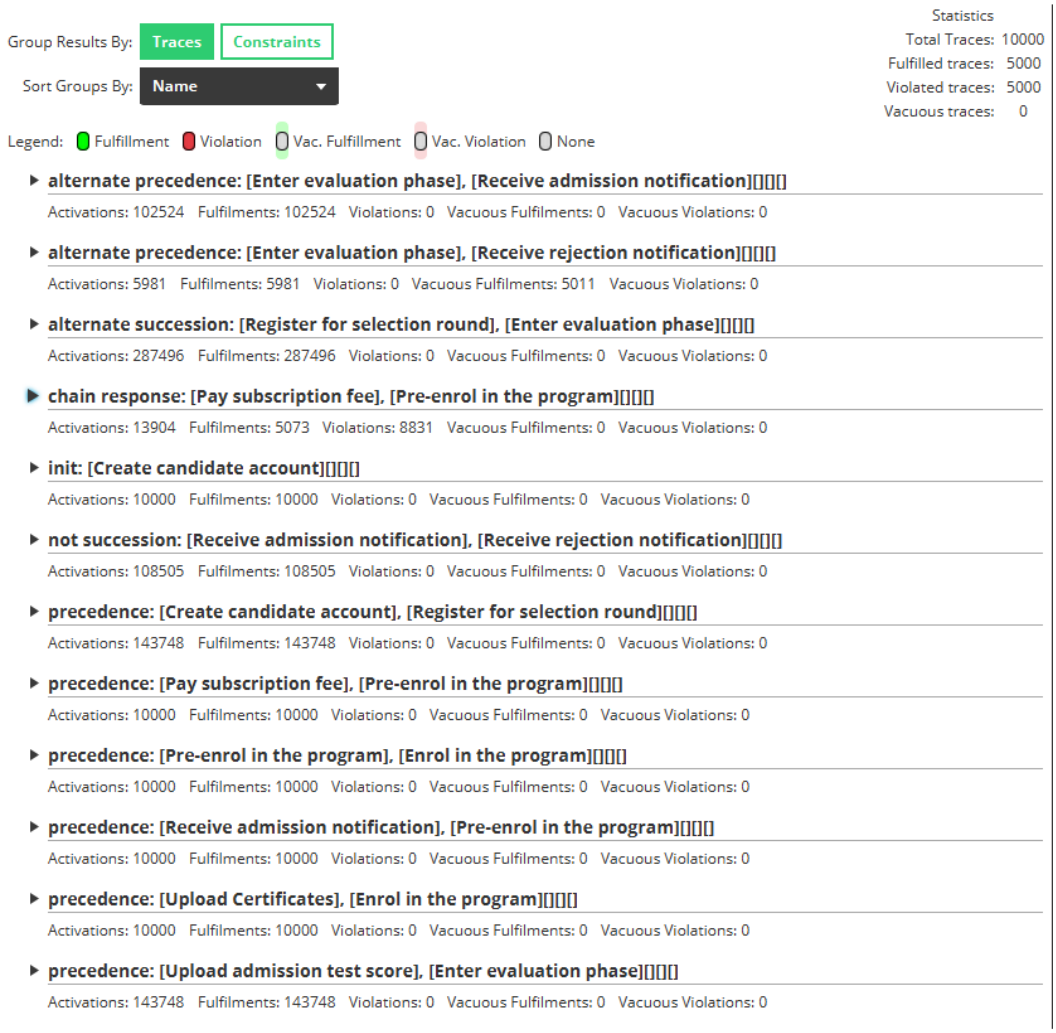


Figure 5.10. University acceptance conformance checking.

Just to create the event log once it took 8666 milliseconds to put into perspective how much time it took, the first example took an average of 150 milliseconds to create.

Chapter 6

Conclusions

Thanks to this study and internship, we have built a working event log generator using process specifications with partial conformity using MINERful functions.

The architecture to generate event logs has been presented in all of its steps which went from declarative process specification to the translation of said specification from Declare constraints into regular expressions, to a simulation of the product automatons and finally, the creation of an event log.

An understanding of what it means to negate a constraint was reached, and after the implementation, the correct functioning of our work was tested in the validation process. Three examples were used as evidence to prove the correctness of our work: the process specifications for a Scientific paper evaluation, specifications for the process of a fracture treatment and last but not least, the process specification for what happens during the admission process at a university.

Bibliography

[1] Wil van der Aalst, Arya Adriansyah, Ana Karla Alves de Medeiros, Franco Arcieri, Thomas Baier, Tobias Blickle, Jagadeesh Chandra Bose, Peter van den Brand, Ronald Brandtjen, Joos Buijs, et al. Process mining manifesto. In *International conference on business process management*, pages 169–194. Springer, 2011.

[2] Anti Alman, Claudio Di Ciccio, Dominik Haas, Fabrizio Maria Maggi, and Alexander Nolte. Rule mining with rum. In *2020 2nd International Conference on Process Mining (ICPM)*, pages 121–128. IEEE, 2020.

[3] Agnieszka Bitkowska, Piotr Sliż, Candace Tenbrink, and Aleksandra Piasecka. Application of process mining on the example of an authorized passenger car service station in poland. *Foundations of Management*, 12(1):125–136, 2020.

[4] Andrea Burattin, Fabrizio M Maggi, Wil MP van der Aalst, and Alessandro Sperduti. Techniques for a posteriori analysis of declarative processes. In *2012 IEEE 16th International Enterprise Distributed Object Computing Conference*, pages 41–50. IEEE, 2012.

[5] Claudio Di Ciccio, Mario Luca Bernardi, Marta Cimitile, and Fabrizio Maria Maggi. Generating event logs through the simulation of declare models. In *Workshop on Enterprise and Organizational Modeling and Simulation*, pages 20–36. Springer, 2015.

[6] Claudio Di Ciccio, Fabrizio Maria Maggi, Marco Montali, and Jan Mendling. Resolving inconsistencies and redundancies in declarative process models. *Information Systems*, 64:425–446, 2017.

[7] Claudio Di Ciccio and Marco Montali. Declarative process specifications: reasoning, discovery, monitoring. In *Process Mining Handbook*, pages 108–152. Springer, 2022.

[8] Claudio Di Ciccio, Mitchel HM Schouten, Massimiliano de Leoni, and Jan Mendling. Declarative process discovery with minerful in prom. *BPM (Demos)*, 1418:60–64, 2015.

[9] Maja Pesic, Helen Schonenberg, and Wil MP Van der Aalst. Declare: Full support for loosely-structured processes. In *11th IEEE international enterprise distributed object computing conference (EDOC 2007)*, pages 287–287. IEEE, 2007.

[10] Johannes Prescher, Claudio Di Ciccio, and Jan Mendling. From declarative processes to imperative models. *SIMPDA*, 1293:162–173, 2014.

[11] Tijs Slaats, Dennis MM Schunselaar, Fabrizio M Maggi, and Hajo A Reijers. The semantics of hybrid process models. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 531–551. Springer, 2016.

[12] Wil Van Der Aalst. Process mining. *Communications of the ACM*, 55(8):76–83, 2012.

[13] Wil Van Der Aalst. *Process mining: data science in action*, volume 2. Springer, 2016.

[14] Wil MP Van der Aalst, Vladimir Rubin, HMW Verbeek, Boudewijn F van Dongen, Ekkart Kindler, and Christian W Günther. Process mining: a two-step approach to balance between underfitting and overfitting. *Software & Systems Modeling*, 9(1):87–111, 2010.

[15] HMW Verbeek, Joos CAM Buijs, Boudewijn F van Dongen, and Wil MP Van Der Aalst. Xes, xesame, and prom 6. In *International conference on advanced information systems engineering*, pages 60–75. Springer, 2010.

[16] Michael Westergaard and Fabrizio Maria Maggi. Declare: A tool suite for declarative workflow modeling and enactment. *BPM (Demos)*, 820:1–5, 2011.