Apriori and Sequence Analysis for Discovering Declarative Process Models

Taavi Kala, Fabrizio M. Maggi
University of Tartu, Estonia
Email: {kala,f.m.maggi}@ut.ee

Claudio Di Ciccio
Vienna University of
Economics and Business, Austria
Email: claudio.di.ciccio@wu.ac.at

Chiara Di Francescomarino
Fondazione Bruno Kessler, Italy
Email: dfmchiara@fbk.eu

***Abstract*—The aim of process discovery is to build a process model from an event log without prior information about the process. The discovery of declarative process models is useful when a process works in an unpredictable and unstable environment since several allowed paths can be represented as a compact set of rules. One of the tools available in the literature for discovering declarative models from logs is the Declare Miner, a plug-in of the process mining tool ProM. Using this plug-in, the discovered models are represented using Declare, a declarative process modelling language based on LTL for finite traces. In this paper, we use a combination of an *Apriori algorithm* and a group of algorithms for *Sequence Analysis* to improve the performances of the Declare Miner. Using synthetic and real life event logs, we show that the new implemented core of the plug-in allows for a significant performance improvement.**

I. Introduction

Process mining is a family of techniques that allow for the analysis of business processes. Its main focus lies in the automatic retrieval and subsequent analysis of business process models from event logs. Process mining consists of discovery, enhancement and conformance checking [1]. Discovery is the extraction of process models from an event log. Enhancement is the extension or improvement of process models using information extracted from a log. Conformance checking consists in analysing whether the real executions of a process, as recorded in a log, are compliant with a process model of the same process [2].

The majority of process discovery algorithms try to construct a procedural model. However, the resulting models are often spaghetti-like and difficult to interpret especially for processes working in unstable environments. Therefore, when dealing with processes with high variability and where multiple paths are allowed, declarative process models are more effective than the imperative ones [3], [4], [5]. Instead of explicitly specifying all possible sequences of activities in a process, declarative models implicitly specify the allowed behaviour of the process with constraints, i.e., rules that must be followed during the execution. In comparison to imperative approaches, which produce "closed" models (what is not explicitly specified is forbidden), declarative languages are "open" (everything that is not constrained is allowed). In this way, models enjoy flexibility and still remain compact. An example of a declarative process modelling language is *Declare*, first introduced in [6].

The *Declare Miner* is a plug-in for the discovery of Declare models from an event log included in the process mining tool ProM.[1] It implements a two-phase approach presented in [7]. The first phase is based on the *Apriori algorithm*, developed by Agrawal and Srikant for mining association rules [8]. During this preliminary phase, the frequent sets of correlated activities are identified. A list of candidate constraints is computed on the basis of the correlated activity sets only. During the second phase, the candidate constraints are checked by replaying the log on specific automata, each accepting only those traces that are compliant to one constraint. Each constraint among the candidates becomes part of the discovered process only if the percentage of traces accepted by the related automaton exceeds a user-defined threshold. Constraints constituting the discovered process are weighted according to their support, i.e., the probability of such constraints to hold in the mined process. To filter out irrelevant constraints, more metrics are introduced, such as confidence and interest factor.

In this paper, we present an approach for the discovery of Declare models that integrates the *Apriori algorithm* presented in [8] with novel algorithms for *Sequence Analysis*, i.e., algorithms that, based on the analysis of the positioning of events in a trace, are able to understand whether a Declare constraint is satisfied in that trace or not. We evaluate the performances of our approach using synthetic logs with different characteristics and three real-life case studies and demonstrate that the original *Declare Miner* performances are significantly improved.

The paper is structured as follows. Section II introduces some background notions about process mining and Declare. Section III illustrates the proposed approach. Section IV describes the experimental evaluation. Finally, Section V discusses related work and Section VI concludes the paper and spells out directions for future work.

II. Background

In this section, we provide a brief overview about the main concepts used in this work. Section II-A gives some background about process mining. Section II-B provides some basic notions about Declare. In Section II-C, we introduce a running example we use throughout the paper.

[1] www.processmining.org

Pre-print copy of the manuscript published by IEEE (available at ieeexplore.ieee.org)
identified by DOI: 10.1109/EDOC.2016.7579378

A. Process Mining

Process mining is still a rather young research discipline which lies between data mining and computational intelligence as well as process modelling and analysis. The general idea of process mining is to discover, monitor and improve real life processes by extracting knowledge from actual event logs used in different systems that gather event data [1]. Over the last ten years, event data have become more widely available and process mining techniques have matured a lot. Different process mining algorithms have been implemented in academic and commercial systems. As there is in increasing interest from industry in process mining, a growing number of software vendors are adding functionalities that provide process mining capabilities to their software and tools.

Process Mining mainly covers four different aspects:

- *process discovery*, which takes an event log and produces a model without using any apriori information;
- *conformance checking*, which is used to compare an existing process model with an event log;
- *model extension*, which is used to extend existing models with information coming from logs;
- *model repair*, which is used to adapt an existing process model based on the behaviour recorded in an event log.

The main guiding principles and the upcoming challenges of such a recent research field have been reported in [2]. Purpose of the principles is supporting in avoiding mistakes that can be made when applying process mining in actual, real-life settings. The challenges highlight relevant open issues that are worth to be addressed in the future.

B. Declare Modelling Language

Recently, several works have investigated advantages and disadvantages of using procedural or declarative process modelling languages to describe a business process [4], [9]. The results of these studies highlighted that the dichotomy *procedural versus declarative* reflects the nature of the process. Procedural models like Petri nets, BPMN, and EPCs are more suitable to support business processes working in stable environments, in which participants have to follow predefined procedures, since they suggest step by step what to do next. In contrast, declarative process modelling languages, like Declare, provide process participants with a (preferably small) set of rules to be followed during the process execution. In this way, process participants have the flexibility to follow any path that does not violate these rules.

Declare is a declarative process modelling language introduced in [6]. A Declare model consists of a set of constraints applied to (atomic) activities. Constraints, in turn, are based on templates. Templates are abstract parametrised patterns, and constraints are their concrete instantiations on real activities. Templates have a user-friendly graphical representation understandable to the user. Their semantics can be formalized using different logics [10], the main one being LTL for finite traces. Each constraint inherits the graphical representation and semantics from its template. The major benefit of using templates is that analysts do not have to be aware of the underlying logic-based formalisation to understand the models. They work with the graphical representation of templates, while the underlying formulas remain hidden. Table I reports the main Declare templates, their formalization in LTL, their graphical representation and a textual description. The reader can refer to [5] for a full description of the language.

Here, we indicate template parameters with capital letters (see Table I) and real activities in their instantiations with lower case letters (e.g., constraint $\Box(a \rightarrow \Diamond b)$). A trace (or case) is a sequence of events like $\langle a, a, b, c\rangle$. Declare templates can be grouped in three main categories: *existence* templates (first four rows of the table), which involve only one event; *relation* templates (rows from 5 to 11), which describe a dependency between two events; and *negative relation* templates (last 5 rows), which describe a negative dependency between two events.

Consider, for example, the *response* constraint $\Box(a \rightarrow \Diamond b)$. This constraint indicates that if a *occurs*, b must eventually *follow*. Therefore, the response constraint is satisfied for traces $\langle a, a, b, c\rangle$, $\langle b, b, c, d\rangle$ and $\langle a, b, c, b\rangle$. It is not satisfied for $\langle a, b, a, c\rangle$, because the second occurrence of a is not followed by a b in such a trace. An *activation* of a constraint in a trace is an event whose occurrence imposes, because of that constraint, some obligations on another event (the rule target) in the same trace. For example, for the *response* constraint between a and b, a is an activation, because the execution of a forces b to be executed eventually. Event b is a target.

An activation of a constraint can be a *fulfilment* or a *violation* for that constraint. When a trace is perfectly compliant with respect to a constraint, every activation of the constraint in the trace leads to a fulfilment. Consider, again, the response constraint $\Box(a \rightarrow \Diamond b)$. In trace $\langle a, a, b, c\rangle$, the constraint is activated and fulfilled twice, whereas, in trace $\langle a, b, c, b\rangle$, the same constraint is activated and fulfilled only once. On the other hand, when a trace is not compliant with respect to a constraint, at least one activation leads to a violation. In trace $\langle a, b, a, c\rangle$, for example, the response constraint $\Box(a \rightarrow \Diamond b)$ is activated twice, but the first activation leads to a fulfilment (eventually b occurs), whereas the second activation leads to a violation (b does not occur subsequently). Finally, there exist cases in which the constraint is not activated at all. Consider, for instance, trace $\langle b, b, c, d\rangle$. The considered response constraint is satisfied in a trivial way because a never occurs. In this case, we say that the constraint is *vacuously satisfied* [11]. In [12], the authors introduce the notion of behavioural vacuity detection according to which a constraint is non-vacuously satisfied in a trace when it is activated in that trace.

C. Running Example

As an example of a Declare model, we consider the fracture treatment process reported in Fig. 1. It includes 8 activities: *examine patient* (a), *check X ray risk* (b), *perform X ray* (c), *perform reposition* (d), *apply cast* (e), *remove cast* (f), *perform surgery* (g), and *prescribe rehabilitation* (h). Its behavior is specified by the following constraints $C1$ - $C7$:

Pre-print copy of the manuscript published by IEEE (available at ieeexplore.ieee.org)
identified by DOI: 10.1109/EDOC.2016.7579378

TEMPLATE	FORMALIZATION	NOTATION	DESCRIPTION
existence(1,A) existence(2,A) ... existence(n,A)	$\Diamond A$ $\Diamond(A \wedge \bigcirc(existence(1,A)))$ ... $\Diamond(A \wedge \bigcirc(existence(n-1,A)))$	$n..*$ A	A has to occur at least once A has to occur at least twice ... A has to occur at least n times
absence(1,A) absence(2,A) ... absence(n,A)	$\neg\Diamond A$ $\neg existence(2,A)$ ... $\neg existence(n,A)$	$0..n$ A	A can never happen A can happen at most once ... A can happen at most n-1 times
exactly(1,A) exactly(2,A) ... exactly(n,A)	$existence(1,A) \wedge absence(2,A)$ $existence(2,A) \wedge absence(3,A)$ ... $existence(n,A) \wedge absence(n+1,A)$	n A	A has to occur exactly once A has to occur exactly twice ... A has to occur exactly n times
init(A)	A	$init$ A	Each instance has to start with the execution of A
resp. existence(A,B)	$\Diamond A \rightarrow \Diamond B$	A ● — B	If A occurs, B must occur as well
response(A,B)	$\Box(A \rightarrow \Diamond B)$	A ●→ B	If A occurs, B must eventually follow
precedence(A,B)	$\neg B \,\mathcal{W}\, A$	A →● B	B can occur only if A has occurred before
alternate response(A,B)	$\Box(A \rightarrow \bigcirc(\neg A \,\mathcal{U}\, B))$	A ⇒● B	If A occurs, B must eventually follow, without any other A in between
alternate precedence(A,B)	$(\neg B \,\mathcal{W}\, A) \wedge$ $\Box(B \rightarrow \bigcirc(\neg B \,\mathcal{W}\, A))$	A ⇒● B	B can occur only if A has occurred before, without any other B in between
chain response(A,B)	$\Box(A \rightarrow \bigcirc B)$	A ⇒● B	If A occurs, B must occur next
chain precedence(A,B)	$\Box(\bigcirc B \rightarrow A)$	A ⇒● B	B can occur only immediately after A
not resp. existence(A,B)	$\Diamond A \rightarrow \neg\Diamond B$	A ●⫽ B	If A occurs, B cannot occur
not response(A,B)	$\Box(A \rightarrow \neg\Diamond B)$	A ●→⫽ B	If A occurs, B cannot eventually follow
not precedence(A,B)	$\Box(A \rightarrow \neg\Diamond B)$	A →●⫽ B	A cannot occur before B
not chain response(A,B)	$\Box(A \rightarrow \neg\bigcirc B)$	A ⇒●⫽ B	If A occurs, B cannot occur next
not chain precedence(A,B)	$\Box(A \rightarrow \neg\bigcirc B)$	A ⇒●⫽ B	A cannot occur immediately before B

TABLE I: Graphical notation and LTL formalization of some Declare templates.

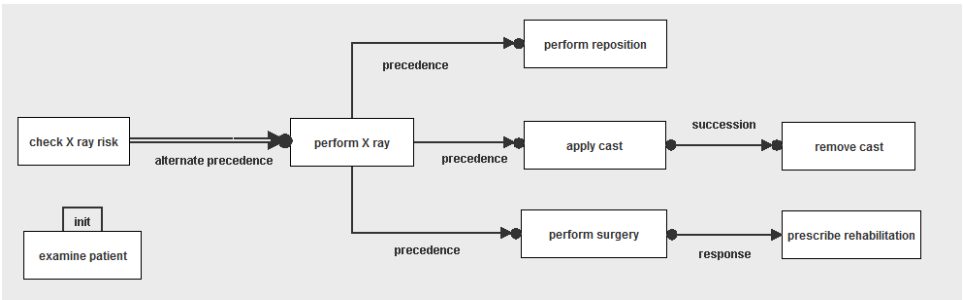


Fig. 1: The Declare model for a fracture treatment process.

1) *init(examine patient)*
2) *alternate precedence(check X ray risk; perform X ray)*
3) *precedence(perform X ray; perform reposition)*
4) *precedence(perform X ray; apply cast)*
5) *succession(apply cast; remove cast)*
6) *precedence(perform X ray; perform surgery)*
7) *response(perform surgery; prescribe rehabilitation)*

According to these constraints, every process instance starts with activity *examine patient*. Moreover, if activity *perform X ray* is performed, then *check X ray risk* must be performed before it, without other executions of *perform X ray* in between. Activities *perform reposition*, *apply cast* and *perform surgery* require that *perform X ray* is executed before they

Pre-print copy of the manuscript published by IEEE (available at ieeexplore.ieee.org)
identified by DOI: 10.1109/EDOC.2016.7579378

are executed. If *perform surgery* is performed, then prescribe rehabilitation is performed eventually after it. Finally, after every execution of *apply cast*, eventually *remove cast* is executed and, vice versa, before every execution of *remove cast*, *apply cast* must be performed.

III. Approach

The approach proposed in this paper aims at discovering Declare constraints from an event log. The idea is to identify and provide users with *frequent* constraints, i.e., constraints that are fulfilled in a percentage of traces (in a log) higher than a given threshold ($supp_{min}$). To this extent, it combines the *Apriori algorithm* technique presented in [8], and *Sequence Analysis*, i.e., a novel collection of algorithms that aim at discovering declarative constraints by analysing how events are positioned along traces.

The algorithm is composed of two phases. In the first phase, a list of frequent activity sets is generated using the *Apriori algorithm*. In the second phase, the frequent activity sets are used to generate candidate Declare constraints (by instantiating Declare templates with those activities). These candidate constraints are then pruned by only keeping those that are frequently satisfied (identified through *Sequence Analysis*).

A. Phase 1: Apriori Algorithm

The *Apriori algorithm* [8] applied in the first phase of the approach allows for the discovery of sets of activities occurring frequently in the traces composing the log (frequent itemsets). Let Σ be the set of activities available in the input event log $\mathfrak{L}$. Let $t \in \Sigma^*$ be a trace over Σ, i.e., a sequence of activities in Σ. $\mathfrak{L}$ is a multi-set over Σ^* (a trace can appear multiple times in an event log). The *support* of a set of activities is a measure that assesses the relevance of this set in an event log.

Definition 1: The support of an activity set $A \subseteq \Sigma$ in an event log $\mathfrak{L} = [t_1, t_2, \ldots, t_n]$ is the ratio of traces in $\mathfrak{L}$ that contain all the activities in A, i.e.,

$$supp(A) = \frac{|\mathfrak{L}_A|}{|\mathfrak{L}|}, \text{ where } \mathfrak{L}_A = [t \in \mathfrak{L} | \forall x \in A, x \in t]$$

An activity set is considered to be *frequent*, if its support is above a given threshold $supp_{min}$. Let A_k denote the set of all frequent activity sets of size $k \in \mathbb{N}$ and let C_k denote the set of all candidate activity sets of size k that may potentially be frequent. The *Apriori algorithm* starts by considering activity sets of size 1 and progresses iteratively by considering activity sets of increasing sizes in each iteration. The idea behind this iterative algorithm is that it holds the property that any subset of a frequent activity set must be frequent.

The set of candidate activity sets of size $k+1$, C_{k+1}, is generated by joining relevant frequent activity sets from A_k. C_{k+1} can be pruned efficiently using the property that a relevant candidate activity set of size $k+1$ cannot have an infrequent subset. The activity sets in C_{k+1} that have a support above a given threshold $supp_{min}$ constitute the frequent activity sets of size $k+1$ (A_{k+1}) used in the next iteration.

For instance, consider the running example of the fracture process depicted in Fig. 1. Let $\mathfrak{L}$ be an event log on the alphabet $\Sigma = \{a, b, c, d, e, f, g, h, i, j\}$, where $a, \ldots, h$ are the activities in the Declare model and i and j are two further activities that are not constrained by the Declare model in Fig. 1:

$$\mathfrak{L} = [\langle a, b, c, j, b, b, d, a\rangle, \langle a, b, b, c, d, a\rangle, \langle a, b, b, i, i, a, c, d\rangle, \langle a, j, j, e, e\rangle, \langle a, b, b, c, j, e, f, b\rangle]$$

and suppose that $supp_{min}$=0.5.

The *Apriori algorithm* starts considering frequent activity sets of size 1. C_1, in Fig. 2(a) (table on the left), shows the candidate activity sets of size 1 on the log $\mathfrak{L}$ and the corresponding *support* values ($supp$). A_1 (table on the right) shows the corresponding frequent activity sets (i.e., all the activity sets with a support value higher than $supp_{min}$). The candidate activity sets of size 2, C_2, are then computed starting from A_1. Relying on the property that a relevant candidate activity set cannot have an infrequent subset, only activity sets on the alphabet $\Sigma/\{d, f, g, h, i\}$ are generated. C_2, in Fig. 2(b) (table on the left), shows the candidate activity sets of size 2 and the related *support* values ($supp$). A_2 (table on the right) shows the list of the frequent activity sets (of size 2) that will become the starting point for building C_3, and so on. The *Apriori algorithm* also allows for taking into account negative events (non-occurrences). Such information might be useful for inferring, for instance, events that are mutually exclusive, e.g., if a occurs, then b does not occur.

The *Apriori algorithm* returns frequent activity sets, without specifying what kind of relation exists between activities, though. These relations are captured by Declare templates. Therefore, for any frequent activity set $\{a, b\}$, constraints such as response ($\Box(a \rightarrow \Diamond b)$) or precedence ($\neg b \,\mathcal{W}\, a$) are generated. We generate candidate constraints deriving from a Declare template with k parameters a constraint instantiated using the frequent activity sets of size k. In particular, we instantiate each template by specifying as parameters all the possible permutations of each frequent set. For instance, for the frequent activity set $\{a, b\}$, we generate the response constraints $\Box(a \rightarrow \Diamond b)$ and $\Box(b \rightarrow \Diamond a)$. It is worth noting that, for different templates, we configure the *Apriori algorithm* in different ways. For example, for relation templates, we discover frequent activity sets including pairs of positive events. On the other hand, for negative relation constraints, also negative events are taken into consideration.

B. Phase 2: Sequence Analysis

After the list of candidate constraints has been generated in Phase 1, a list of Declare constraints is extracted from it using a group of algorithms for *Sequence Analysis*. *Sequence Analysis* aims at checking, for each candidate constraint, whether a trace in the input log is compliant with the constraint, by looking at the positioning of events in the trace. Only constraints frequently fulfilled in the log, are considered relevant Declare constraints. Each Declare template demands for a

Pre-print copy of the manuscript published by IEEE (available at ieeexplore.ieee.org)
identified by DOI: 10.1109/EDOC.2016.7579378

Candidate Activity Sets

C_1	*supp*
a	100
b	80
c	80
d	60
e	40
f	20
g	0
h	0
i	20
j	60

Frequent Activity Sets

A_1	*supp*
a	100
b	80
c	80
d	60
j	60

a: Activity sets of size 1

Candidate Activity Sets

C_2	*supp*
$\{a, b\}$	80
$\{a, c\}$	80
$\{a, d\}$	60
$\{a, j\}$	60
$\{b, c\}$	80
$\{b, d\}$	60
$\{b, j\}$	40
$\{c, d\}$	60
$\{c, j\}$	40
$\{d, j\}$	20

Frequent Activity Sets

A_2	*supp*
$\{a, b\}$	80
$\{a, c\}$	80
$\{a, d\}$	60
$\{a, j\}$	60
$\{b, c\}$	80
$\{b, d\}$	60
$\{c, d\}$	60

b: Activity sets of size 2

Fig. 2: Candidate and frequent activity sets of size 1 and 2 (obtained by using $supp_{min} = 50\%$). *supp* is expressed in %.

specific *Sequence Analysis* algorithm in charge of computing the support of each candidate constraint ($supp_{constr}$).

Let $\mathfrak{L}$ be an event log on the alphabet Σ and *constr* a constraint, i.e., an instantiation of a Declare template with activities in Σ. The *support* of the constraint is a measure that assesses the relevance of the constraint in the event log.

Definition 2: The support of a constraint *constr* in an event log $\mathfrak{L} = [t_1, t_2, \ldots, t_n]$ is the ratio of traces in $\mathfrak{L}$ in which the constraint is fulfilled, i.e.,

$$supp_{constr} = \frac{|\mathfrak{L}_{constr}|}{|\mathfrak{L}|},$$

where $\mathfrak{L}_{constr} = [t \in \mathfrak{L} | \text{constr is fulfilled in } t]$

Also in this case, a constraint *constr* is considered to be *frequent* if its support is greater than the given threshold $supp_{min}$.

In particular, for each template, a specific *Sequence Analysis* algorithm has been implemented. Each of these algorithms requires as input a `candidateList`, i.e., the list of candidate constraints generated in the previous step. The event log is replayed by the algorithms and, each event in each trace of the log is processed and analysed. Based on the position in the sequence of activities in the trace, the specific *Sequence Analysis* algorithm assesses whether a candidate constraint is fulfilled or not by the trace. Once all the events in the log have been processed, only the candidate constraints with $supp_{constr}$ greater than the minimum support $supp_{min}$ are kept and presented to the user.

Algorithm 1: *Sequence Analysis* for response

Input: `candidateList`, the list of candidate constraints from Phase 1;
$e = (c, a, t)$ the current event to be processed (c is the case id of the event, a is the activity name, t is the timestamp)

```
1  if fulfilledCases is not defined then
2      define map fullfilledCases ;
3  if pendingActivations is not defined then
4      define map pendingActivations ;
5  foreach (k1, k2) in candidateList do
6      if k2 == a then
7          pendingActivations.put((k1, a),0) ;
8      else if k1 == a then
9          acts ← pendingActivations.get((a, k2)) ;
10         pendingActivations.put((a, k2), acts + 1) ;
11     if isLastEvent(e,c) then
12         acts ← pendingActivations.get((k1, k2)) ;
13     if acts == 0 then
14         cases ← fulfilledCases.get((k1, k2)) ;
15         fulfilledCases.put((k1, k2), cases + 1) ;
```

Finally, the discovered constraints can be filtered in order to leave out vacuously satisfied constraints. If vacuity detection is enabled only constraints that are activated and satisfied frequently will be discovered. If vacuity detection is disabled, also vacuously satisfied constraints will be discovered. For instance, consider again the log $\mathfrak{L}$ of the running example. By applying the *Sequence Analysis* algorithm for the precedence template to the constraint $\neg c \mathcal{W} d$, it results to be satisfied in the first four traces. Therefore, the support value for this constraint will be $supp_{constr} = 0.8$, which is greater than $supp_{min}$. Hence, the precedence constraint $\neg c \mathcal{W} d$ will be discovered by the algorithm. If vacuity detection is enabled, only the first three traces of the log $\mathfrak{L}$ are counted for $supp_{constr}$, which is, in this case, 0.6 and still higher than $supp_{min}$.

The *Sequence Analysis* algorithms for *response*, *precedence* and *existence* templates are briefly presented in the following. Note that the algorithms for negative relation templates can be trivially derived from the algorithms for the corresponding positive relation templates (for instance, the fulfillments for a succession constraint are violations for the corresponding not succession constraint and vice versa).

a) Sequence Analysis for response: The semantics of the response constraint $\Box(a \rightarrow \Diamond b)$ can be defined as "if a occurs, b must eventually follow". The pseudo-code for the *response* algorithm is reported in Algorithm 1. It takes as input the `candidateList` of activity pairs from Phase 1 and the current event to be processed. It first initializes the variables: in particular, it initializes the maps `fulfilledCases` (for storing the cases in which each candidate constraint is satisfied) and `pendingActivations` (for storing the activations of each candidate constraint that are still pending). For each pair in `candidateList`, (k_1, k_2), (i) if the current event corresponds to the second element of the activity pair (k_2), all pending activations of $response(k_1; k_2)$ are satisfied and, hence, the number of pending activations related

Pre-print copy of the manuscript published by IEEE (available at ieeexplore.ieee.org)
identified by DOI: 10.1109/EDOC.2016.7579378

Algorithm 2: *Sequence Analysis* for precedence

```
Input: candidateList, the list of candidate constraints from Phase 1;
       e = (c, a, t) the event to be processed (c is the case id of the event, a is
       the activity name, t is the timestamp)
if fulfilledCases is not defined then
    define map fullfilledCases ;
if activityFrequencies is not defined then
    define map activityFrequencies ;
if fulfilledActivatons is not defined then
    define map fulfilledActivatons ;
if activityFrequencies.constaintsKey(a) then
    freq ← activityFrequencies.get(a) ;
    activityFreqencies.put(a, freq + 1) ;
forall (k1, k2) in candidateList do
    if k2 == a && activityFrequencies.get(k1) > 0 then
        acts ← fulfilledActivations.get((a, k2)) ;
        fulfilledActivations.put((a, k2), acts + 1) ;
    if isLastEvent(e,c) then
        acts ← fulfilledActivations.get((k1, k2)) ;
        if acts == activityFrequecies.get(k2) then
            cases ← fulfilledCases.get((k1, k2)) ;
            fulfilledCases.put((k1, k2), cases + 1) ;
```

to $response(k_1; k_2)$ is set to 0; (ii) if, the current event corresponds to the first event of the pair of activities (k_1), then the number of pending activations for $response(k_1; k_2)$ is incremented by 1 (a new constraint activation has occurred that demands for an occurrence of k_2 to fulfill the constraint). When the last event is reached, if (and only if) the number of pending activations of a candidate constraint is 0, then the case is added to the map of `fulfilledCases` for that constraint.

b) Sequence Analysis for precedence: The semantics of the precedence constraint ($\neg b \,\mathcal{W}\, a$) can be defined as "b can occur only if a has occurred before". Algorithm 2 reports the pseudo-code for the *precedence* algorithm. It takes as input the `candidateList` of activity pairs from Phase 1 and the current event to be processed. It first initializes the variables: in particular, it initializes the maps `fulfilledCases` (for storing the cases in which each candidate constraint is satisfied), `activityFrequencies` (for storing the frequency of each activity in the log) and `fulfilledActivations` (for storing the activations of each candidate constraint that have been fulfilled). The frequency of the activity corresponding to event e is then increased by 1 in `activityFrequencies`. For each pair in `candidateList`, (k_1, k_2), if the current event corresponds to the second element of the activity pair (k_2) and the first element of the pair (k_1) has occurred at least once before, i.e., it has a frequency greater than 0, the number of fulfilled activations of the constraint $precedence(k_1; k_2)$ is incremented by 1. When the last event is reached, for each candidate constraint, if the number of fulfilled activations is the same as the number of occurrences of k_2 (none of these occurrences has caused a violation of the constraint), the case is added to the map of `fulfilledCases` for that constraint.

c) Sequence Analysis for existence (exactly and absence): The existence constraint $existence(n, a)$ can be described as "a is executed at least n times". Similarly, $absence(n, a)$ means "a is executed at most $n-1$ times".

Algorithm 3: *Sequence Analysis* for existence (exactly and absence)

```
Input: candidateList, the list of candidate constraints from Phase 1;
       e = (c, a, t) the event to be processed (c is the case id of the event, a is the
       activity name, t is the timestamp)
if fulfilledCases is not defined then
    define map fullfilledCases ;
if activityFrequencies is not defined then
    define map activityFrequencies ;
if activityFrequencies.containsKey(a) then
    freq ← activityFrequencies.get(a);
    activityFrequencies.put(a, freq + 1) ;
forall k in candidateList do
    if isLastEvent then
        acts ← activityFrequencies.get(k) ;
        if existenceCondition(acts) then
            cases ← fulfilledCases.get(k) ;
            fulfilledCases.put(k, cases + 1) ;
```

Finally, $exactly(n, a)$ is defined as "a is executed exactly n times". Algorithm 3 shows the pseudo-code for the *existence*, *exactly* and *absence* algorithms. Their implementations differ based on the $existenceCondition$ function. They take as input the `candidateList` from Phase 1 (which, in this case, is the list of all the activities in the input log) and the current event to be processed. It first initializes the variables: in particular, it initializes the maps `fulfilledCases` (for storing the cases in which each candidate constraint is satisfied) and `activityFrequencies` (for storing the frequency of each activity in the log). The frequency of the activity corresponding to event e is then increased by 1 in `activityFrequencies`. For each candidate constraint in `candidateList`, if the last event of the case is reached and the $existenceCondition$ is verified, the case is added to the map of `fulfilledCases` for that constraint. The $existenceCondition$ differs based on the specific template of the *Sequence Analysis*:

- $existence(a, n)$ - the frequency of a must be greater than or equal to n,
- $absence(a, n)$ - the frequency of a must be at most $n-1$,
- $exactly(a, n)$ - the frequency of a must be exactly n.

C. Implementation

The proposed approach has been implemented and is accessible through a command line interface. It is available as a JAR file, which makes it easier its integration into Java applications. The application provides the functionalities of the *Declare Miner* plug-in, though offering significantly improved performances.

The application takes as input a configuration file where to specify (i) the path of the log file to be processed; (ii) the templates specifying which type of constraints are to be discovered; (iii) the value of the minimum support used for filtering activity sets and candidate constraints ($supp_{min}$); (iv) the flag in charge of enabling or disabling vacuity detection; (v) the output path and the output file type (e.g., XML, text or an ad-hoc format for reporting the discovered constraints in a

Pre-print copy of the manuscript published by IEEE (available at ieeexplore.ieee.org)
identified by DOI: 10.1109/EDOC.2016.7579378

human readable format). In particular, the human-readable format version of the output also reports information about simple and logical sentences to convey the constraints' essence, the witnesses (cases in which each discovered constraint is activated and satisfied), counterexamples (cases in which each discovered constraint is violated) and vacuous cases.

IV. Evaluation

We evaluated the proposed approach (referred to as Seq. w/ Apriori) in terms of time performances. To this purpose, we took as benchmarks: (i) the Declare Miner, which uses the *Apriori algorithm* and automata (referred to as Aut. w/ Apriori); and (ii) the approach based on the only *Sequence Analysis* (Sequence), and we compared the execution times of these approaches. We describe the datasets in Section IV-A and the procedure we used for the evaluation in Section IV-B. Finally, we discuss the results in Section IV-C.

A. Datasets

Two types of datasets have been used for the evaluation: (i) synthetic logs, in order to compare the approaches in terms of time performances when using event logs with different characteristics; and (ii) real-life logs that were provided for the BPI Challenges in years 2012, 2013 and 2014.

1) Synthetic Logs: Synthetic logs have been generated using the generator described in [13], [14] and simulating the Declare model we have used as running example (Fig. 1). The log generator allows for the generation of logs of specified size (s), containing traces of a given length (l) and built on an alphabet of a given size ($|\Sigma|$).

The synthetic logs generated for the evaluation have size $s \in \{400, 800, 1600,\ 3200, 6400\}$, contain traces of length $l \in \{16, 24, 32, 40, 48\}$ and are built on an alphabet Σ of size $|\Sigma| \in \{8, 16, 24, 32, 40, 48\}$ (the smallest one being the alphabet of the running example $\Sigma = \{a, b, c, d, e, f, g, h\}$). In particular, the following different configurations have been applied for the generation of synthetic logs (only one of the three parameters is changed per time, while keeping the others constant):

1) $l = 24$, $|\Sigma| = 8$ and $s \in \{400, 800, 1600, 3200, 6400\}$, thus generating 5 logs of different size;
2) $s = 800$, $|\Sigma| = 8$ and $l \in \{16, 24, 32, 40, 48\}$, thus generating 5 logs with traces of different length;
3) $l = 24$, $s = 800$ and $|\Sigma| \in \{8, 16, 24, 32, 40, 48\}$, thus generating 6 logs built on alphabets of increasing size.

2) BPI Challenge Logs: The BPI Challenge logs are real-life logs used in the BPI Challenge competition. In particular, the BPI Challenge 2012 log [15] pertains to an application process for personal loans or overdrafts within a Dutch financial Institute; the BPI Challenge 2013 log [16] is related to an incident management process supported by a system called VINST in use at Volvo IT Belgium; and the BPI Challenge 2014 log [17] pertains to the management of calls or mails from customers to the Service Desk concerning disruptions of ICT-services from Rabobank Group ICT. Table II reports the statistics related to the three BPI Challenge logs.

Log	Traces	Events	Alphabet size
BPI2012	13,087	262,200	36
BPI2013	7,554	65,533	14
BPI2014	46,616	466,737	39

TABLE II: Statistics for the BPI Challenge logs.

B. Procedure

For the synthetic logs the tests have been run using configurations where $supp_{min}$ is set to 80% and with both enabled and disabled vacuity detection. For the BPI Challenge logs, instead, vacuity detection was enabled and three different values for $supp_{min}$ were considered: $supp_{min} = 80\%$, $supp_{min} = 90\%$, $supp_{min} = 100\%$. The tests have been run on a Ubuntu Linux 12.04 server machine, equipped with Intel Xeon CPU E5-2650 v2 2.60GHz, using 1 64-bit CPU core and 16GB main memory quota.

Both (i) the number of discovered constraints and (ii) the time required for the discovery have been collected for each log and configuration. The time [2] has been averaged on five runs and reported in milliseconds.

C. Results

The constraints discovered by the presented technique are exactly the same as the ones produced by the two benchmarks, while differences exist in terms of performances. In the following, we detail these differences for each of the investigated logs, by showing that Seq. w/ Apriori outperforms the other two approaches with both synthetic and real-life logs.

1) Synthetic Logs: Synthetic logs have been used to investigate the time performances of the three approaches when using event logs with different characteristics. The three analyses reported below show that overall Seq. w/ Apriori significantly improves the performances of the Aut. w/ Apriori approach and that performs better than the Sequence approach.

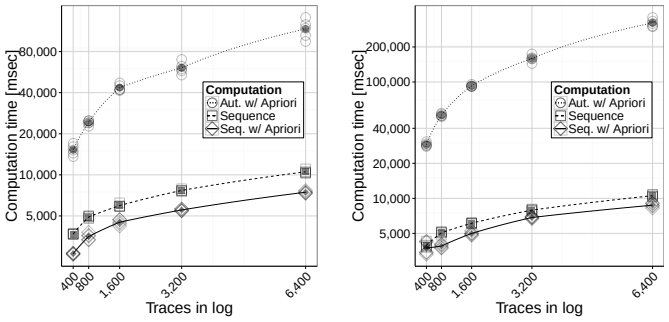
a) Varying Log Size: Fig. 3 [3] reports the plots related to the time performances of Aut. w/ Apriori, Sequence and Seq. w/ Apriori when varying the log size. In case of vacuity detection enabled (Fig. 3a), Seq. w/ Apriori and Sequence perform significantly better than Aut. w/ Apriori and Seq. w/ Apriori is faster than the other two approaches. Moreover, the difference in terms of performances increases as the size of the log increases. When vacuity detection is disabled (Fig. 3b), almost the same trend can be observed. However, in this case, while the performances of Seq. w/ Apriori and Sequence register only small variations, the performances of the Aut. w/ Apriori become much worse than the case in which vacuity detection is enabled.

b) Varying Trace Length: Fig. 4 reports the execution times of the three approaches when varying the trace lengths. As for the previous case, Seq. w/ Apriori significantly

[2]The time required for the log import has been excluded from the elaboration time.

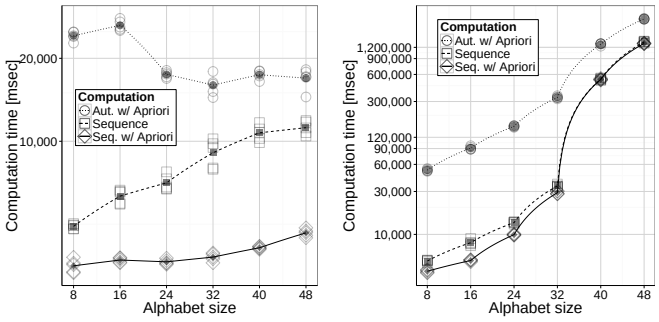
[3]Note that for a better readability of the plots, times are reported on a logarithmic scale.

Pre-print copy of the manuscript published by IEEE (available at ieeexplore.ieee.org)
identified by DOI: 10.1109/EDOC.2016.7579378



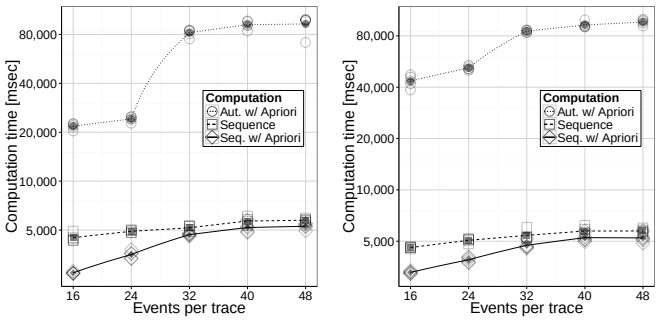
(a) With vacuity detection (b) Without vacuity detection

Fig. 3: Computation time as a function of the log size.



(a) With vacuity detection (b) Without vacuity detection

Fig. 5: Computation time as a function of the alphabet size.



(a) With vacuity detection (b) Without vacuity detection

Fig. 4: Computation time as a function of the traces length.

Log	Min.$supp$	Computation time [msec]		
		Aut.w/Apriori	Sequence	Seq.w/Apriori
BPI 2012	80%	90,255	70,759	12,360
	90%	101,811	72,471	12,898
	100%	84,581	70,304	11,634
BPI 2013	80%	6,625	11,561	3,311
	90%	6,792	11,714	3,140
	100%	6,506	10,987	2,689
BPI 2014	80%	1,388,797	177,181	27,093
	90%	707,082	179,939	28,755
	100%	51,568	175,971	16,965

TABLE III: Computation time needed to mine BPI logs.

outperforms AUT. W/ APRIORI and is faster than SEQUENCE. Also in this case, while for SEQ. W/ APRIORI and SEQUENCE there are no significant differences in terms of performances when vacuity detection is enabled (Fig. 4a) or disabled (Fig. 4b), vacuity detection has an impact on the performances in the AUT. W/ APRIORI case for short traces (traces with size up to about 25 events).

c) Varying Alphabet Size: Fig. 5 shows the time required by the three approaches when varying the alphabet size. Overall, when vacuity detection is enabled (Fig. 5a), SEQ. W/ APRIORI performs significantly better than the other two approaches for different alphabet sizes. Alphabets of large size, indeed, are very costly for SEQUENCE. On the other hand, although AUT. W/ APRIORI seems to gain in terms of performances as the alphabet size increases, it is not able to outperform the other two approaches. When vacuity detection is disabled (Fig. 5b), a different trend, instead, can be observed. In this case, the required time increases as the alphabet size increases for all the three approaches and, for small alphabets ($|\Sigma| \leq 32$), the performance of SEQ. W/ APRIORI is slightly better than the one of the SEQUENCE approach only.

2) BPI Challenges: Table III reports the execution times required to process the three real life logs for different values of $supp_{min}$ and with vacuity detection enabled. The results show that overall the time required by SEQ. W/ APRIORI is always significantly lower than the time required by the other two approaches, ranging from a minimum of c.a. 3 seconds for the BPI Challenge 2013 log to a maximum of c.a. 29 seconds for the BPI Challenge 2014 log. By looking at the other two approaches, instead, the trend is less stable. While for the BPI Challenge 2012 log and for the BPI Challenge 2014 with $supp_{min} = 80\%$ and $supp_{min} = 90\%$, SEQUENCE outperforms AUT. W/ APRIORI, for the BPI Challenge 2013 dataset and for the BPI Challenge 2014 with $supp_{min} = 100\%$, the performances of AUT. W/ APRIORI are better than those of SEQUENCE. Moreover, despite some exceptions (e.g., AUT. W/ APRIORI for the BPI Challenge 2014 dataset), the time required by all the three approaches for different values of $supp_{min}$ does not vary significantly.

V. RELATED WORK

Different approaches have been proposed so far for mining declarative process models. Some of them belong to the group of the probabilistic process mining approaches. For instance, Statistical Relational Learning has been used for learning from process traces labelled as compliant or non-compliant, declarative constraints expressed as ICs (Integrity Constraints) [18]. A logic-based approach for probabilistic process mining enhancing this approach is presented in [19].

An approach that makes use of logical programming for declarative process mining is presented in [20]. The proposed methodology is based on Inductive Logical Programming

Pre-print copy of the manuscript published by IEEE (available at ieeexplore.ieee.org)
identified by DOI: 10.1109/EDOC.2016.7579378

(ICL). The ICL algorithm, used in this approach, is adapted to the problem of learning integrity constraints in SCIFF and is able to learn a model by considering both compliant and non-compliant traces.

An algorithm to discover declarative workflows was developed in [21] using email messages as event log traces. The implemented algorithm, *MINERful*, is described as a two-step algorithm [22]. The first step aims at building a knowledge base starting from existing traces. The second step aims at computing the statistical support of constraints by querying the knowledge base.

An online process discovery technique which takes data from online streams is presented in [23]. The proposed framework is able to produce at runtime an updated picture of the process behavior in terms of Declare constraints. It also gives meaningful information about the concept drifts that occurred during the process execution to the user.

VI. Conclusion

Although existing solutions for declarative process mining are currently actively used and widely recognised, there is still room for improvement. In this work, one of the existing solutions for the discovery of declarative process models, the *Declare Miner* plug-in of the process mining tool ProM has been enhanced. In particular, the proposed solution combines the *Apriori algorithm* and a group of algorithms for *Sequence Analysis* to improve its performances. Results on both synthetic and real use cases show that the proposed approach significantly improves the performances of the original version of the Declare Miner.

In the future, we aim at investigating the possibility of using this approach for repairing existing models, as well as to extend it to support additional perspectives such as time and data.

Acknowledgment

This research has been partially carried out within the EU FP7 Programme under grant agreement 609190 (Subject-Orientation for People-Centred Production) and within the Euregio IPN12 KAOS, which is funded by the "European Region Tyrol-South Tyrol-Trentino" (EGTC) under the first call for basic research projects.

References

[1] W. M. P. van der Aalst, *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[2] W. M. P. v. d. Aalst and et al, "Process mining manifesto," in *BPM 2011 Workshops, Part I*, vol. 99. Springer-Verlag, 2012, pp. 169–194.

[3] S. Zugal, J. Pinggera, and B. Weber, "The impact of testcases on the maintainability of declarative process models," in *BMMDS/EMMSAD*, 2011, pp. 163–177.

[4] P. Pichler, B. Weber, S. Zugal, J. Pinggera, J. Mendling, and H. A. Reijers, "Imperative versus declarative process modeling languages: An empirical investigation," in *BPM Workshops*, 2011, pp. 383–394.

[5] W. M. P. van der Aalst, M. Pesic, and H. Schonenberg, "Declarative workflows: Balancing between flexibility and support," *Computer Science - R&D*, vol. 23, no. 2, pp. 99–113, 2009.

[6] M. Pesic, "Constraint-Based Workflow Management Systems: Shifting Control to Users," Ph.D. dissertation, TU/e, 2008.

[7] F. M. Maggi, R. P. J. C. Bose, and W. M. P. van der Aalst, "Efficient discovery of understandable declarative process models from event logs," in *CAiSE*, 2012, pp. 270–285.

[8] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *VLDB*. Morgan Kaufmann, 1994, pp. 487–499.

[9] H. A. Reijers, T. Slaats, and C. Stahl, "Declarative modeling—an academic dream or the future for BPM?" in *BPM 2013*, 2013, vol. 8094, pp. 307–322.

[10] M. Montali, M. Pesic, W. M. P. van der Aalst, F. Chesani, P. Mello, and S. Storari, "Declarative Specification and Verification of Service Choreographies," *ACM Transactions on the Web*, vol. 4, no. 1, 2010.

[11] O. Kupferman and M. Y. Vardi, "Vacuity detection in temporal model checking." in *CHARME 1999*, vol. 1703, 1999, pp. 82–96.

[12] A. Burattin, F. Maggi, W. van der Aalst, and A. Sperduti, "Techniques for a posteriori analysis of declarative processes," in *EDOC 2012*, 2012, pp. 41–50.

[13] C. Di Ciccio, M. L. Bernardi, M. Cimitile, and F. M. Maggi, "Generating event logs through the simulation of Declare models," in *EOMAS*, 2015.

[14] C. Di Ciccio, M. Mecella, and J. Mendling, "The effect of noise on mined declarative constraints," in *Data-Driven Process Discovery and Analysis*, 2015, vol. 203, pp. 1–24.

[15] B. van Dongen, "BPI challenge 2012," 2012. [Online]. Available: http://dx.doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f

[16] W. Steeman;, "BPI challenge 2013, closed problems," 2013. [Online]. Available: http://dx.doi.org/10.4121/uuid:c2c3b154-ab26-4b31-a0e8-8f2350ddac11

[17] B. van Dongen;, "BPI challenge 2014," 2014. [Online]. Available: http://dx.doi.org/10.4121/uuid:c3e5d162-0cfd-4bb0-bd82-af5268819c35

[18] E. Bellodi, F. Riguzzi, and E. Lamma, "Probabilistic declarative process mining," in *KSEM 2010*, vol. 6291, 2010, pp. 292–303.

[19] E. Bellodi, F. Riguzzi, and E. Lamma, "Probabilistic logic-based process mining," in *CILC2010*, vol. 598, 2010.

[20] F. Chesani, E. Lamma, P. Mello, M. Montali, F. Riguzzi, and S. Storari, "Exploiting inductive logic programming techniques for declarative process mining," *ToPNoC II*, vol. 5460, pp. 278–295, 2009.

[21] C. Di Ciccio and M. Mecella, "Mining artful processes from knowledge workers' emails," *IEEE Internet Computing*, vol. 17, no. 5, pp. 10–20, September 2013.

[22] C. Di Ciccio and M. Mecella, "On the discovery of declarative control flows for artful processes," *ACM Trans. Manage. Inf. Syst.*, vol. 5, no. 4, pp. 24:1–24:37, January 2015.

[23] F. M. Maggi, A. Burattin, M. Cimitile, and A. Sperduti, "Online process discovery to detect concept drifts in ltl-based declarative process models," in *OTM 2013 Conferences*, 2013, vol. 8185, pp. 94–111.

Pre-print copy of the manuscript published by IEEE (available at ieeexplore.ieee.org)
identified by DOI: 10.1109/EDOC.2016.7579378

This document is a pre-print copy of the manuscript
(Kala et al. 2016)
published by IEEE (available at ieeexplore.ieee.org).

The final version of the paper is identified by DOI: 10.1109/EDOC.2016.7579378

References

Kala, Taavi, Fabrizio Maria Maggi, Claudio Di Ciccio, and Chiara Di Francescomarino (2016). "Apriori and Sequence Analysis for Discovering Declarative Process Models". In: *EDOC*. Ed. by Florian Matthes, Jan Mendling, and Stefanie Rinderle-Ma. IEEE, pp. 50–58. ISBN: 978-1-4673-9885-5. DOI: 10.1109/EDOC.2016.7579378.

BibTeX

```
@InProceedings{   Kala.etal/EDOC2016:AprioriandSequenceDeclare,
  author        = {Kala, Taavi and Maggi, Fabrizio Maria and Di Ciccio,
                   Claudio and Di Francescomarino, Chiara},
  title         = {Apriori and Sequence Analysis for Discovering Declarative
                   Process Models},
  booktitle     = {EDOC},
  year          = {2016},
  pages         = {50--58},
  crossref      = {EDOC2016},
  doi           = {10.1109/EDOC.2016.7579378},
  keywords      = {Algorithm design and analysis; Business data processing;
                   Data mining}
}
@Proceedings{     EDOC2016,
  title         = {20th {IEEE} International Enterprise Distributed Object
                   Computing Conference, {EDOC} 2016, Vienna, Austria,
                   September 5-9, 2016},
  year          = {2016},
  editor        = {Florian Matthes and Jan Mendling and Stefanie
                   Rinderle{-}Ma},
  publisher     = {IEEE},
  isbn          = {978-1-4673-9885-5}
}
```