Parallel Algorithms for the Automated Discovery of Declarative Process Models

Fabrizio Maria Maggi[a], Claudio Di Ciccio[b], Chiara Di Francescomarino[c,*], Taavi Kala[a]

[a] *University of Tartu, Liivi 2, Tartu, Estonia*
[b] *Vienna University of Economics and Business, Welthandelsplatz 1, 1020 Vienna, Austria*
[c] *FBK-IRST, Via Sommarive 18, 38050 Trento, Italy*

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 modeling language based on LTL for finite traces. However, the high execution times of the Declare Miner when processing large sets of data hampers the applicability of the tool to real-life settings. Therefore, in this paper, we propose a new approach for the discovery of DECLARE models based on the combination of an *Apriori algorithm* and a group of algorithms for *Sequence Analysis* to enhance the time performance of the plug-in. The approach has been developed in a way that it is easy to be parallelized using two different partitioning methods: the search space partitioning, in which different groups of candidate constraints are processed in parallel, and the database partitioning, in which different chunks of the log are processed at the same time. The approach has been implemented in ProM in its sequential version and in two multi-threading implementations

*Corresponding author
Email addresses: fmmaggi@ut.ee (Fabrizio Maria Maggi), claudio.di.ciccio@wu.ac.at (Claudio Di Ciccio), dfmchiara@fbk.eu (Chiara Di Francescomarino), kala@ut.ee (Taavi Kala)

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

leveraging these two partitioning methods. All the new variants of the plug-in have been evaluated using a large set of synthetic and real-life event logs.

Keywords: Process Mining, Process Discovery, Declarative Process Models, Apriori Algorithm, Sequence Analysis

1. Introduction

Process mining [1] is a family of techniques that allow for the analysis of business processes using event logs. It consists of three main branches: process discovery, model enhancement and conformance checking. Process discovery deals with the extraction of process models from an event log. Model enhancement is the extension or improvement of process models using information extracted from a log. Conformance checking consists in analyzing whether the real executions of a process, as recorded in a log, are compliant with a process model representing the expected behavior of the process.

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 procedural ones [2, 3, 4]. Instead of explicitly specifying all possible sequences of activities in a process, declarative models implicitly specify the allowed behavior of the process with constraints, i.e., rules that must be followed during the execution. In comparison to procedural 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 modeling language is DECLARE, first introduced in [5]. A DECLARE model consists of a set of constraints which, in turn, are based on templates. Templates are parameterized classes of rules and constraints are their concrete instantiations.

The *Declare Miner* is a plug-in for the discovery of DECLARE models from an event log included in the process mining tool ProM. It implements the two-phase approach presented in [6]. The first phase is based on the *Apriori algorithm* developed by Agrawal and Srikant for mining association rules [7]. During this preliminary phase, the frequent sets of correlated activities are identified in the log. A list of candidate constraints is computed on

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

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 with one constraint. Each constraint among the candidates becomes part of the discovered model only if the percentage of traces accepted by the related automaton exceeds a user-defined threshold. Constraints constituting the discovered DECLARE model 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 [8], an approach for the discovery of DECLARE models enhancing the time performance of the Declare Miner has been presented. Such an approach integrates the *Apriori algorithm* and a set of 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. In this paper, we further enhance the approach presented in [8] to make it suitable for parallelization. In particular, we leverage the notions of search space partitioning and database partitioning presented in [9] using them as a basis for two multi-threading implementations of the plug-in. The difference between the two partitioning methods lies in the fact that in database partitioning the event log is divided into separate chunks and each chunk is analyzed by a different thread. In the other case (search space partitioning), the candidate DECLARE constraints to be checked are grouped per template and each template is processed separately. All three variants of the approach (the sequential one and the two additional variants leveraging search space partitioning and database partitioning) have been implemented in the *Declare Miner 2.0* plug-in in ProM and evaluated using 76 synthetic logs with different characteristics and 8 publicly available real-life logs.

The paper is structured as follows. Section 2 introduces some background notions about process mining and DECLARE. Section 3 illustrates the proposed approach. Section 4 describes the experimental evaluation. Finally, Section 5 discusses related work and Section 6 concludes the paper and spells out directions for future work.

2. Background

In this section, we provide a brief overview about the main concepts used in this work. Section 2.1 gives some background about process mining.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

Section 2.2 provides some basic notions about DECLARE. In Section 2.3, we introduce an example of a DECLARE model.

2.1. Process Mining

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

Starting point for process mining techniques is an *event log*. Each event in a log refers to an *activity* (i.e., a well-defined step in some process) and is related to a particular *trace* (i.e., a *process instance*). The events belonging to a trace are *ordered* and can be seen as one "run" of the process. Event logs may store additional information about events such as the *resource* (i.e., person or device) executing or initiating the activity, the *timestamp* of the event, or *data elements* recorded with the event.

Process mining mainly covers three different groups of techniques:

- *process discovery*, which takes an event log and produces a model without using any apriori information;
- *model enhancement*, which is used to extend or to adapt an existing process model based on the behavior recorded in an event log;
- *conformance checking*, which is used to compare an existing process model with an event log.

The main guiding principles and upcoming challenges of process mining have been reported in [10]. The former serve as a means for process miners to orient their investigations in real-life environments. The latter shed light on relevant open issues that are worth being tackled in the future. In this work, we tackle the challenge of implementing an approach dealing with logs of different sizes and characteristics (Challenge $C2$ in [10]), the

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

challenge related to the creation of representative benchmarks for process mining (Challenge $C3$), and the challenge related to the improvement of the representational bias used for process discovery (Challenge $C5$). Indeed, our extensive evaluation shows that the proposed discovery approach is able to deal with logs with diverse characteristics (Challenge $C2$). In addition, the large set of synthetic logs we generated starting from DECLARE models provide the process mining community with benchmark data characterized by high variability typical of unstable environments (Challenge $C3$). Finally, the proposed approach aims at discovering process models described using a declarative language (DECLARE), which alleviates the representational issues that procedural languages need to face in case of processes working in highly variable environments (Challenge $C5$).

2.2. The DECLARE *Modeling Language*

Recently, several works have investigated advantages and disadvantages of using procedural or declarative process modeling languages to describe a business process [2, 3, 4]. 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 modeling 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 modeling language introduced in [5]. A DECLARE model consists of a set of constraints applied to (atomic) activities. Constraints, in turn, are based on templates. Templates are abstract parameterized 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 [11], 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 formalization to understand the models. They work with the graphical representation of templates, while the underlying formulas remain hidden. Table 1 reports the main DECLARE templates, their graphical

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

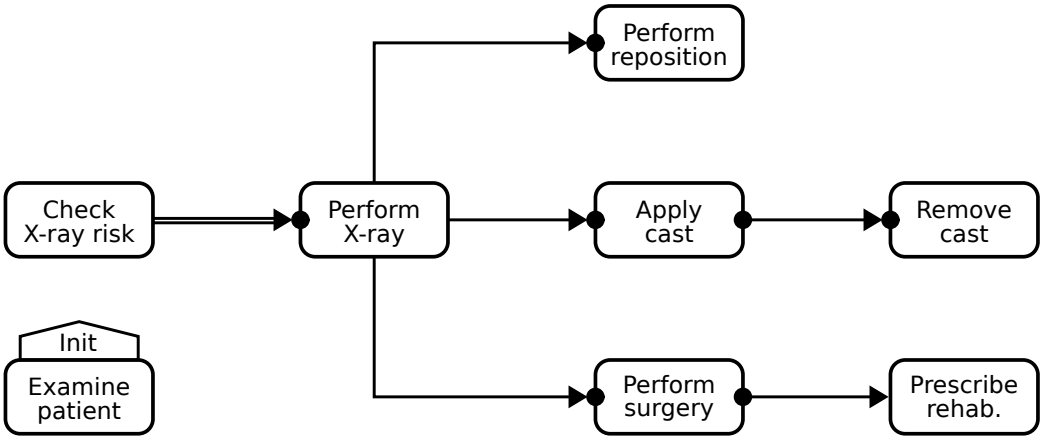


Figure 1: The DECLARE model for a fracture treatment process.

representation and a textual description. The reader can refer to [12] for a full description of the language.

Here, we indicate template parameters with capital letters (see Table 1) and real activities in their instantiations with lower case letters (e.g., constraint RESPONSE(a, b)). A trace is a sequence of events like $\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{c} \rangle$. DECLARE templates can be grouped in three main categories: *existence* templates (first 4 rows of the table), which involve only one event; *(mutual) relation* templates (rows from 5 to 15), which describe a dependency between two events; and *negative relation* templates (last 3 rows), which describe a negative dependency between two events.

Consider, for example, the RESPONSE constraint RESPONSE(a, b). This constraint indicates that "If a occurs, then b occurs after a". Therefore, the RESPONSE constraint is satisfied for traces $\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{c} \rangle$, $\langle \mathsf{b}, \mathsf{b}, \mathsf{c}, \mathsf{d} \rangle$ and $\langle \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{b} \rangle$. It is not satisfied for $\langle \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{c} \rangle$, 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 *target*) in the same trace. For example, for RESPONSE(a, 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 *fulfillment* or a *violation* for that constraint. When a trace is perfectly compliant with a constraint, every activation of the constraint in the trace leads to a fulfillment. Consider, again, the RESPONSE constraint RESPONSE(a, b). In trace $\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{c} \rangle$, the constraint

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

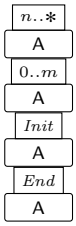
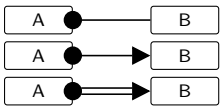
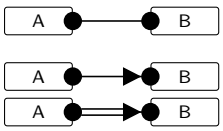
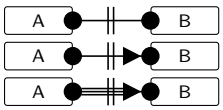
Template	Explanation	Notation
Existence templates		
Existence(n, A)	A occurs at least n times	
Absence($m+1$, A)	A occurs at most m times	
Init(A)	A is the *first* to occur	
End(A)	A is the *last* to occur	
Relation templates		
RespondedExistence(A, B)	If A occurs, then B occurs	
Response(A, B)	If A occurs, then B occurs after A	
AlternateResponse(A, B)	Each time A occurs, then B occurs afterwards, before A recurs	
ChainResponse(A, B)	Each time A occurs, then B occurs immediately after	
Precedence(A, B)	B occurs only if preceded by A	
AlternatePrecedence(A, B)	Each time B occurs, it is preceded by A and no other B can recur in between	
ChainPrecedence(A, B)	Each time B occurs, then A occurs immediately before	
Mutual relation templates		
CoExistence(A, B)	If B occurs, then A occurs, and vice versa	
Succession(A, B)	A occurs if and only if B occurs after A	
AlternateSuccession(A, B)	A and B occur if and only if the latter follows the former, and they alternate each other	
ChainSuccession(A, B)	A and B occur if and only if the latter immediately follows the former	
Negative relation templates		
NotCoExistence(A, B)	A and B never occur together	
NotSuccession(A, B)	A never occurs before B	
NotChainSuccession(A, B)	A and B occur if and only if the latter does not immediately follow the former	

Table 1: Declare templates.

is activated and fulfilled twice, whereas, in trace $\langle \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{b} \rangle$, the same constraint is activated and fulfilled only once. On the other hand, when a trace is not compliant with a constraint, at least one activation leads to a violation. In

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

trace $\langle \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{c} \rangle$, for example, the RESPONSE constraint RESPONSE(a, b) is activated twice, but the first activation leads to a fulfillment (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 $\langle \mathsf{b}, \mathsf{b}, \mathsf{c}, \mathsf{d} \rangle$. The considered RESPONSE constraint is satisfied in a trivial way in this trace, because a never occurs. In this case, we say that the constraint is *vacuously satisfied* [13]. In [14, 15], the authors introduce the notion of semantical vacuity detection according to which a constraint is non-vacuously satisfied in a trace when it is fulfilled and activated at least once in that trace.

2.3. DECLARE *Model Example*

As an example of a DECLARE model, we consider the fracture treatment process reported in Fig. 1. It includes 8 activities: Examine patient, Check X-ray risk, Perform X-ray, Perform reposition, Apply cast, Remove cast, Perform surgery, and Prescribe rehabilitation. Its behavior is specified by the following constraints 1 - 7:

1. INIT(Examine patient)
2. ALTERNATEPRECEDENCE(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 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.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

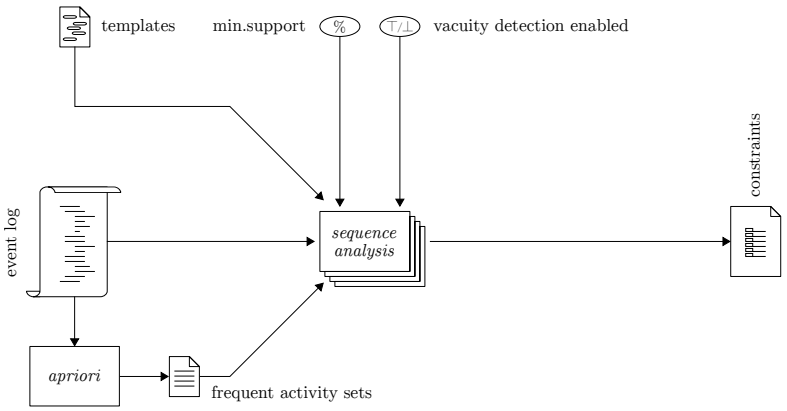


Figure 2: Architectural scheme of the approach without partitioning.

3. 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 the log) higher than a given threshold ($supp_{min}$). To this extent, it combines the *Apriori algorithm* presented in [7], and *Sequence Analysis*, i.e., a novel collection of algorithms that aim at discovering declarative constraints by analyzing how events are positioned along traces.

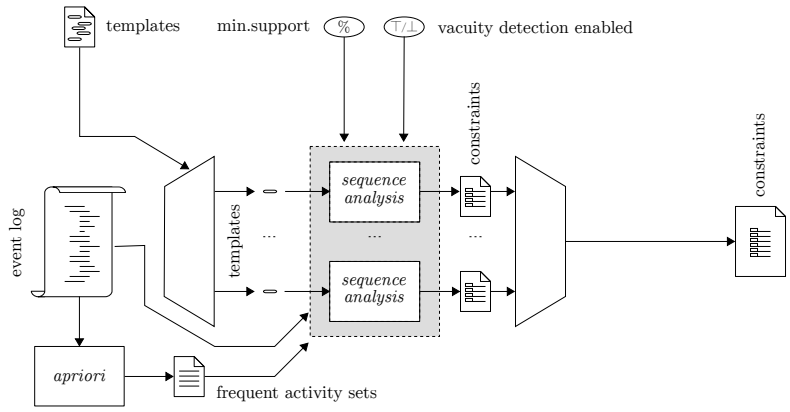
The approach is composed of two phases (Fig. 2). In the first phase, a list of frequent activity sets is derived from the log using the *Apriori algorithm*. In the second phase, the frequent activity sets are used to generate candidate DECLARE constraints (by instantiating DECLARE templates with those activities). The list of candidate constraints is then pruned by only keeping those that are frequently satisfied in the log, i.e., with a support higher than $supp_{min}$ (through *Sequence Analysis*).

As shown in detail in our experiments in Section 4, the application of the *Apriori algorithm* is less time consuming than the *Sequence Analysis* on the traces in the log. Therefore, we propose two versions of the discovery algorithm that can be used to parallelize the *Sequence Analysis* phase.

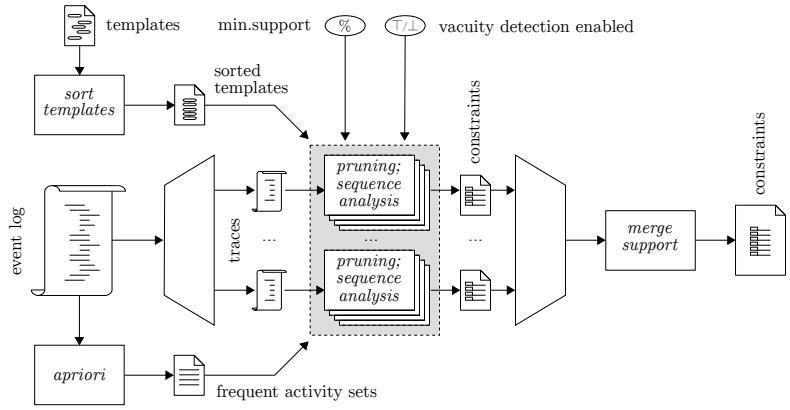
3.1. Search Space and Database Partitioning

The first version (Fig. 3(a)) of the discovery algorithm presented in this work is based on *search space partitioning* [9]. In particular, the analysis of each template is managed independently in a separate thread and the

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002



(a) search space partitioning



(b) database partitioning

Figure 3: Architectural scheme of the approach with partitioning.

results coming from each thread are finally collected to return the discovered constraints.

The second version (Fig. 3(b)) is based on a *database partitioning* [9]. In this case, the analysis of each trace is managed independently in a separate thread. In this version, we can compute the support of each candidate constraint only when all the threads have completed their execution. As shown in Section 4, in some cases this additional step can decrease the performance of this algorithm. In this version of the algorithm, since we partition per trace, it is possible, for each trace, to reduce the number of candidate constraints to be checked. In particular, two actions are performed for each trace:

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

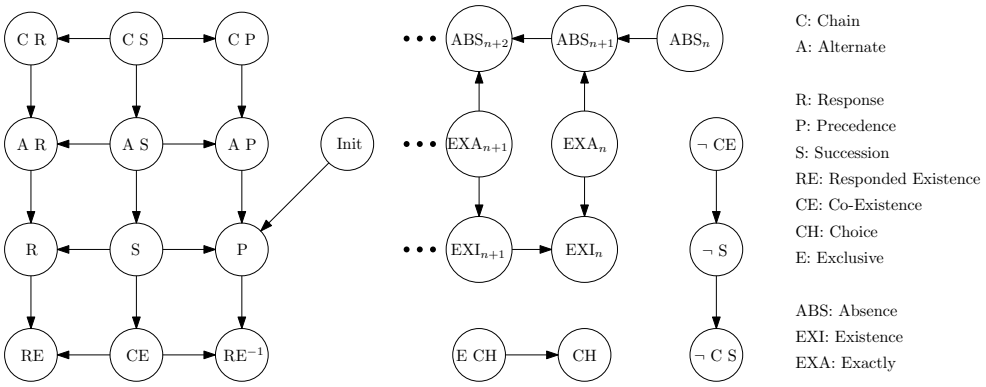


Figure 4: The hierarchy of the DECLARE constraints.

1. we remove candidates that are not activated at all;

2. we rank the constraints so as to check first the strongest ones.

In particular (action 1), for each trace, we derive the events that occur in the trace and we only check candidates whose activation occurs at least once in the trace. The ones that are not activated at all are not needed to be checked and are classified as vacuously satisfied. For instance, constraint RESPONSE(d, b), would be directly discarded for traces $\langle \mathsf{a}, \mathsf{j}, \mathsf{j}, \mathsf{e}, \mathsf{e} \rangle$ and $\langle \mathsf{a}, \mathsf{b}, \mathsf{b}, \mathsf{c}, \mathsf{j}, \mathsf{e}, \mathsf{f}, \mathsf{b} \rangle$. Indeed, the constraint is vacuously satisfied in these traces and it does not need to be checked.

In addition (action 2), for each trace, we start checking the candidates instantiations of the strongest templates according to the hierarchy presented in [16]. Figure 4 reports a detailed view of the lattice describing the order of the DECLARE templates. In the figure, each node is labeled with a shortening for a DECLARE template (for example, the label `C S` in the lattice stands for CHAINSUCCESSION). According to the lattice, for instance, CHAINRESPONSE is stronger than ALTERNATERESPONSE, which is, in turn, stronger than RESPONSE. Moreover, for the transitivity of the order, CHAINRESPONSE is also stronger than RESPONSE. When checking whether constraints are valid in a given trace, if a stronger constraint has already been found to be (non-vacuously) satisfied in the trace, it is not necessary to check also the weaker constraints, as they will also be (non-vacuously) satisfied. For instance, if CHAINRESPONSE(a, b) is verified in a trace, RESPONSE(a, b) will also hold.

Algorithm 1 and Algorithm 3 report the pseudo-code of the two versions

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

of the discovery algorithm based on search space and database partitioning, respectively. After a common part including the discovery of frequent activity sets through the *Apriori algorithm* (procedure *apriori*, line 1 in Algorithm 1 and Algorithm 3) and an initialization phase (lines 2-3) required to prepare the necessary data structures (detailed in Algorithm 2), they implement the *Sequence Analysis* differently.

In particular, in the case of search space partitioning, a replayer is created for each template (line 5 in Algorithm 1) in charge of processing one by one all the traces of the log and identifying, among the candidate constraints, the ones that are fulfilled in each trace. In the case of the database partitioning (Algorithm 3), instead, the templates are ordered according to their strength (line 4), and processed starting from the strongest ones. More specifically, for each trace and each template the list of candidate constraints is pruned by removing the candidates that are not activated at all in the current trace (line 7), as well as the ones for which there exists a stronger constraint that is satisfied in the current trace (line 12). In this way, we are able to instantiate a replayer for each trace and each template (line 13) on a smaller set of candidate constraints.

The core part of the *Sequence Analysis* is then the invocation of procedure *process* (line 8 and line 15, respectively) to analyze the traces event by event and identify the fulfilled constraints for each template. The number of traces fulfilling each constraint is computed both in case of vacuity detection enabled (lines 10-13 and lines 19-22, respectively) and disabled (lines 15-18 and lines 24-27, respectively). In particular, in the case of search space partitioning, the number of traces fulfilling a given constraint is directly retrieved from the replayers, which incrementally update the number of fulfilling traces. In the case of database partitioning, instead, the number of traces fulfilling a given constraint is obtained by combining the number of traces fulfilling the constraint with the number of traces fulfilling the stronger constraints. Finally, the list of candidate constraints is filtered based on their support (line 19 and line 28), i.e., based on the percentage of traces fulfilling them.

In Section 3.2, we explain in detail how procedure *apriori* works. The implementation of *process* is different for different templates. In Section 3.3, we provide a detailed description of the algorithms used to implement this procedure for all the standard DECLARE templates.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

Algorithm 1: Discovery algorithm with search space partitioning

Input: $\mathfrak{L} = [t_1, \ldots, t_{|\mathfrak{L}|}]$ the event log to be analyzed, consisting of traces t_i $(1 \leqslant i \leqslant |\mathfrak{L}|)$;
$\mathfrak{T} = \{\mathscr{T}_1, \ldots, \mathscr{T}_{|\mathfrak{T}|}\}$ the set of templates $\mathscr{T}_j$ $(1 \leqslant j \leqslant |\mathfrak{T}|)$;
$supp_{min}$ the minimum support;
$vacuity_?$: a flag to enable or disable vacuity detection, such that $v \in \top, \bot$.

Data: *ful*: a map associating each constraint instantiating a template $\mathscr{T}_j \in \mathfrak{T}$ to the number of traces where it is fulfilled.

```
1  {𝒜1, 𝒜2} ← apriori(𝔏, supp_min)                    /* frequent activity sets of size 1 and 2 */
2  {𝒜̂1, 𝒜̂2} ← initializeCandidates(𝒜1, 𝒜2)
3  ful ← initializeMap(𝒜̂1, 𝒜̂2, 𝔗)
4  foreach 𝒯j ∈ 𝔗 do
5      rj ← new Replayer(𝒯j, 𝒜̂k)                      /* one replayer for each template */
6      foreach ti ∈ 𝔏 do
7          foreach e_{i,h} ∈ ti do
8              rj.process(e_{i,h}, ti)      /* process event e_{i,h} in trace ti with the replayer */
9      if vacuity_? then
10         foreach (a) ∈ 𝒜̂1 do
11             ful.put( 𝒯j^1(a),   ful.get(𝒯j^1(a)) + rj.fulfillingTraces.get(a) )
12         foreach (a, b) ∈ 𝒜̂2 do
13             ful.put( 𝒯j^2(a, b),   ful.get(𝒯j^2(a, b)) + rj.fulfillingTraces.get(a, b) )
14     else
15         foreach (a) ∈ 𝒜̂1 do
16             ful.put( 𝒯j^1(a), ful.get(𝒯j^1(a)) + rj.fulfillingTraces.get(a) + rj.vacuousTraces.get(a) )
17         foreach (a, b) ∈ 𝒜̂2 do
18             ful.put( 𝒯j^2(a, b), ful.get(𝒯j^2(a, b)) + rj.fulfillingTraces.get(a, b) +
                rj.vacuousTraces.get(a, b) )
19 ful ← filterOnSupport(ful, supp_min)
```

3.2. Phase 1: Apriori Algorithm

The *Apriori algorithm* [7] 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 activity sets). This algorithm implements the procedure *apriori* reported in Algorithm 1 and Algorithm 3.

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* $\mathscr{A} \subseteq \Sigma$ *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*

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

Algorithm 2: Initialization procedures

```
1  Procedure initializeCandidates
       Input: 𝒜_1, 𝒜_2
2      𝒜̂_1 ← ∅
3      𝒜̂_2 ← ∅
4      foreach {a} ∈ 𝒜_1 do
5          𝒜̂_1 ← 𝒜̂_1 ∪ {(a)}
6      foreach {a, b} ∈ 𝒜_2 do
7          𝒜̂_2 ← 𝒜̂_2 ∪ {(a, b), (b, a)}
9      return 𝒜̂_1, 𝒜̂_2
10 Procedure initializeMap
       Input: 𝒜̂_1, 𝒜̂_2, 𝔗
11     foreach (a) ∈ 𝒜̂_1 do
12         foreach 𝒯_j^1 ∈ 𝔗 do
13             ful.put(𝒯_j^1(a), 0)                    /* existence templates */
14     foreach (a, b) ∈ 𝒜̂_2 do
15         foreach 𝒯_j^2 ∈ 𝔗 do
16             ful.put(𝒯_j^2(a, b), 0)                 /* relation templates */
18     return ful
19 Procedure initializeMaps
       Input: 𝒜̂_1, 𝒜̂_2, 𝔗
20     foreach (a) ∈ 𝒜̂_1 do
21         foreach 𝒯_j^1 ∈ 𝔗 do
22             ful.put(𝒯_j^1(a), 0)                    /* existence templates */
23             vac.put(𝒯_j^1(a), 0)
24     foreach (a, b) ∈ 𝒜̂_2 do
25         foreach 𝒯_j^2 ∈ 𝔗 do
26             ful.put(𝒯_j^2(a, b), 0)                 /* relation templates */
27             vac.put(𝒯_j^2(a, b), 0)
29     return ful, vac
```

$\mathscr{A}$, *i.e.*,

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

An activity set is considered to be *frequent* if its support is above a given threshold $supp_{min}$. Let $\mathscr{A}_k$ denote the set of all frequent activity sets of size $k \in \mathbb{N}$ and let $\mathscr{C}_k$ 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 consideration upon which this iterative

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

Algorithm 3: Discovery algorithm with database partitioning

Input: $\mathfrak{L} = [t_1, \ldots, t_{|\mathfrak{L}|}]$ the event log to be analyzed, consisting of traces t_i $(1 \leqslant i \leqslant |\mathfrak{L}|)$;
$\mathfrak{T} = \{\mathscr{T}_1, \ldots, \mathscr{T}_{|\mathfrak{T}|}\}$ the set of templates $\mathscr{T}_j$ $(1 \leqslant j \leqslant |\mathfrak{T}|)$;
$supp_{min}$ the minimum support;
$vacuity_?$: a flag to enable or disable vacuity detection, such that $v \in \top, \bot$.

Data: *ful*: a map associating each constraint instantiating a template $\mathscr{T}_j \in \mathfrak{T}$ to the number of traces where it is fulfilled;
vac: a map associating each constraint instantiating a template $\mathscr{T}_j$ to the number of traces where it is vacuously satisfied;
inheritedFromStronger: a map associating each constraint instantiating a template $\mathscr{T}_j$ to the number of traces where a stronger constraint is satisfied.

1 $\{\mathscr{A}_1, \mathscr{A}_2\} \leftarrow$ apriori$(\mathfrak{L}, supp_{min})$ `/* frequent activity sets of size 1 and 2 */`
2 $\{\widehat{\mathscr{A}_1}, \widehat{\mathscr{A}_2}\} \leftarrow$ initializeCandidates$(\mathscr{A}_1, \mathscr{A}_2)$
3 *ful*, *vac* $\leftarrow$ initializeMaps$(\widehat{\mathscr{A}_1}, \widehat{\mathscr{A}_2}, \mathfrak{T})$
4 $\mathfrak{T} \leftarrow$ sort$(\mathfrak{T})$
5 **foreach** $t_i \in \mathfrak{L}$ **do**
6 **foreach** $\mathscr{T}_j \in \mathfrak{T}$ **do**
7 $\{\widehat{\mathscr{A}_k}, \widehat{\mathscr{V}_k}\} \leftarrow$ filterVacuouslySatisfiedCandidates$(\widehat{\mathscr{A}_k}, t_i, \mathscr{T}_j)$
8 **foreach** $(a) \in \widehat{\mathscr{V}_1}$ **do**
9 *vac*.put($\mathscr{T}_j^1(a)$, *vac*.get($\mathscr{T}_j^1(a)$) + 1)
10 **foreach** $(a, b) \in \widehat{\mathscr{V}_2}$ **do**
11 *vac*.put($\mathscr{T}_j^2(a, b)$, *vac*.get($\mathscr{T}_j^2(a, b)$) + 1)
12 $\widehat{\mathscr{A}_k} \leftarrow$ filterCandidatesSatisfiedByStronger$(\widehat{\mathscr{A}_k}, ful, \mathscr{T}_j)$
13 $r_j \leftarrow$ new Replayer$(\mathscr{T}_j, \widehat{\mathscr{A}_k})$ `/* one replayer for each trace and each template */`
14 **foreach** $e_{i,k} \in t_i$ **do**
15 r_j.process$(e_{i,h}, t_i)$ `/* process event` $e_{i,h}$ `in trace` t_i `with the replayer */`
16 **foreach** $\mathscr{T}_j \in \mathfrak{T}$ **do**
17 *inheritedFromStronger*.updateTemplate($\mathscr{T}_j^k$)
18 **if** $vacuity_?$ **then**
19 **foreach** $(a) \in \widehat{\mathscr{A}_1}$ **do**
20 *ful*.put($\mathscr{T}_j^1(a)$, *ful*.get($\mathscr{T}_j^1(a)$ + r_j.*fulfillingTraces.get*(a) + *inheritedFromStronger*.get($\mathscr{T}_j^1(a)$))
21 **foreach** $(a, b) \in \widehat{\mathscr{A}_2}$ **do**
22 *ful*.put($\mathscr{T}_j^2(a, b)$, *ful*.get($\mathscr{T}_j^2(a, b)$ + r_j.*fulfillingTraces.get*(a, b) + *inheritedFromStronger*.get($\mathscr{T}_j^2(a, b)$))
23 **else**
24 **foreach** $(a) \in \widehat{\mathscr{A}_1}$ **do**
25 *ful*.put($\mathscr{T}_j^1(a)$, *ful*.get($\mathscr{T}_j^1(a)$) + r_j.*fulfillingTraces.get*(a) + *inheritedFromStronger*.get($\mathscr{T}_j^1(a)$) + *vac*.get($\mathscr{T}_j^1(a)$)
26 **foreach** $(a, b) \in \widehat{\mathscr{A}_2}$ **do**
27 *ful*.put($\mathscr{T}_j^2(a, b)$, *ful*.get($\mathscr{T}_j^2(a, b)$ + r_j.*fulfillingTraces.get*(a, b) + *inheritedFromStronger*.get($\mathscr{T}_j^2(a, b)$) + *vac*.get($\mathscr{T}_j^2(a, b)$)
28 *ful* $\leftarrow$ filterOnSupport$(ful, supp_{min})$

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

algorithm is built is that every set is always at least as frequent as its supersets (*downward closure*).

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

For instance, let $\mathfrak{L}$ be an event log on the alphabet $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}, \mathsf{f}, \mathsf{g}, \mathsf{h}, \mathsf{i}, \mathsf{j}\}$:

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

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

The *Apriori algorithm* starts considering frequent activity sets of size 1. $\mathscr{C}_1$, in Table 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*). $\mathscr{A}_1$ (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, $\mathscr{C}_2$, are then computed starting from $\mathscr{A}_1$. $\mathscr{C}_2$, in Table 2(b) (table on the left), shows the candidate activity sets of size 2 and the related *support* values (*supp*). $\mathscr{A}_2$ (table on the right) shows the list of the frequent activity sets (of size 2) that will become the starting point for building $\mathscr{C}_3$, 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., a and b never occur together.

The *Apriori algorithm* returns frequent activity sets, without specifying what kind of relation exists between activities. These relations are captured by DECLARE templates. Therefore, we generate candidate constraints to be verified over the event log as follows: for every DECLARE template, we assign its k parameters with the permutations of activities taken from the discovered frequent sets of size k. For instance, given a frequent activity set $\{\mathsf{a}, \mathsf{b}\}$ and templates RESPONSE and PRECEDENCE, the following constraints are generated: RESPONSE(a, b), RESPONSE(b, a), PRECEDENCE(a, b), and PRECEDENCE(b, a). It is worth noting that we configure the *Apriori algorithm* according to the template under analysis. For example, for relation templates, we discover frequent activity sets including only pairs of positive

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

Candidate activity sets	
$\mathscr{C}_1$	*supp* [%]
{a}	100
{b}	80
{c}	80
{d}	80
{e}	40
{f}	20
{g}	0
{h}	0
{i}	20
{j}	60

Frequent activity sets	
$\mathscr{A}_1$	*supp* [%]
{a}	100
{b}	80
{c}	80
{d}	80
{j}	60

(a) Activity sets of size 1

Candidate activity sets	
$\mathscr{C}_2$	*supp* [%]
{a, b}	80
{a, c}	80
{a, d}	80
{a, j}	60
{b, c}	80
{b, d}	80
{b, j}	40
{c, d}	80
{c, j}	40
{d, j}	40

Frequent activity sets	
$\mathscr{A}_2$	*supp* [%]
{a, b}	80
{a, c}	80
{a, d}	80
{a, j}	60
{b, c}	80
{b, d}	80
{c, d}	80

(b) Activity sets of size 2

Table 2: Candidate and frequent activity sets of size 1 and 2 ($supp_{min} = 50\%$).

(i.e., occurring) events. On the other hand, for negative relation templates, also negative (i.e, non-occurring) events are taken into account.

3.3. Phase 2: Sequence Analysis

After the list of candidate constraints has been generated in Phase 1, a list of relevant DECLARE constraints is extracted from it using a group of algorithms for *Sequence Analysis*. These algorithms implement the procedure *process* reported in Algorithm 1 and Algorithm 3 for each DECLARE template. Relevant constraints are the ones that are frequently fulfilled in the input log.

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 *constr* 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* $\mathfrak{L}$ *where the constraint is fulfilled, i.e.,*

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

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

A constraint *constr* is considered to be *relevant* if its support is greater than a given threshold $supp_{min}$.[1]

Each *Sequence Analysis* algorithm is in charge of computing the support of each candidate constraint instantiation of a given DECLARE template. To this aim, these algorithms check, for each candidate constraint, whether each trace in the input log is compliant with the constraint. Therefore, for each template, each *Sequence Analysis* algorithm requires to have access to the list of candidate constraints generated in Phase 1, $\widehat{\mathscr{A}_k}$.

The event log is replayed and, all events in each trace of the log are processed and analyzed by the algorithms. Based on their position in the trace, each specific *Sequence Analysis* algorithm assesses whether each candidate constraint is fulfilled or not in the trace. Once all 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.

The discovered constraints can also 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 presented to the user. For instance, let $\mathfrak{L}$ be an event log on the alphabet $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}, \mathsf{f}, \mathsf{g}, \mathsf{h}, \mathsf{i}, \mathsf{j}\}$:

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

and suppose that $supp_{min}$=0.7.

By applying the *Sequence Analysis* algorithm for the PRECEDENCE template to constraint PRECEDENCE(c, d), it results to be satisfied in the first four traces. Therefore, the support value for this constraint is $supp_{constr} = 0.8$, which is greater than $supp_{min}$. Constraint PRECEDENCE(c, d) will thus be discovered. If vacuity detection is enabled, only the first three traces of the

[1] Note that we use the same support threshold for activity sets (in the *Apriori algorithm*) and for constraints (in the *Sequence Analysis*). Indeed, these two notions of support are strictly correlated. Suppose, for example, that we want to discover RESPONSE constraints with vacuity detection enabled and with minimum support $supp_{min}$. In this case, we should instantiate the RESPONSE template using all the pairs of activities that occur together in a ratio of traces that is at least equal to $supp_{min}$. Indeed, the RESPONSE constraints instantiated using pairs of activities whose support is lower than $supp_{min}$ can never be (non-vacuously) satisfied in a percentage of traces equals to or higher than $supp_{min}$.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

Algorithm 4: *Sequence Analysis* for RESPONDEDEXISTENCE

Input: `e` the event to be processed;
`t` the current trace.
Data: *fulfillingTraces* a map associating each candidate constraint to the number of traces where it is fulfilled;
vacuousTraces a map associating each candidate constraint to the number of traces where it is vacuously satisfied;
$\widehat{\mathscr{A}_2}$ the set of candidate constraints.

```
1  if isFirstEvent(e, t) then
2      initialize set occurredEvents
3      initialize map pendingActivations
4  if e ∉ occurredEvents then
5      occurredEvents.add(e)
6  foreach (a, b) ∈ Â2 do
7      if e == b then                                  /* e is equal to the second activity */
8          pendingActivations.put((a, e), 0)
9      else if e == a then                             /* e is equal to the first activity */
10         if b ∉ occurredEvents then
11             pendingActivations.put((e, b), 1)
12     if isLastEvent(e,t) then
13         acts ← pendingActivations.get((a, b))
14         if acts == 0 then
15             if a ∈ t then
16                 fulfillingTraces.put((a, b), fulfillingTraces.get((a, b) + 1)
17             else
18                 vacuousTraces.put((a, b), vacuousTraces.get((a, b) + 1))
```

example log are counted for $supp_{constr}$, which is 0.6 and lower than $supp_{min}$. Therefore, in this case, constraint PRECEDENCE(`c`, `d`) will not be returned.

The *Sequence Analysis* algorithms for all the standard DECLARE templates are presented in the following.

Sequence Analysis for RESPONDEDEXISTENCE. The semantics of the RESPONDEDEXISTENCE template can be defined as "If `A` occurs, then `B` occurs". The pseudo-code for the RESPONDEDEXISTENCE algorithm is reported in Algorithm 4. It takes as input the current event `e` and the current trace `t` and, if `e` is the first event in `t`, it initializes set *occurredEvents* (containing the events already occurred at least once in the current trace) and map *pendingActivations* (containing the number of pending activations in the current trace for each candidate RESPONDEDEXISTENCE constraint) (lines 2-3). If event `e` has never occurred in `t`, `e` is added in *occurredEvents* (line 5). Line 8 of the algorithm sets to 0 the pending activations in `t` for the RESPONDEDEXISTENCE constraints having `e` as second parameter (there

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

Algorithm 5: *Sequence Analysis* for RESPONSE

Input: `e` the event to be processed;
`t` the current trace.
Data: *fulfillingTraces* a map associating each candidate constraint to the number of traces where it is fulfilled;
vacuousTraces a map associating each candidate constraint to the number of traces where it is vacuously satisfied;
$\widehat{\mathscr{A}_2}$ the set of candidate constraints.

```
1  if isFirstEvent(e, t) then
2      initialize map pendingActivations
3  foreach (a, b) ∈ Â2 do
4      if e == b then                                  /* e is equal to the second parameter */
5          pendingActivations.put((a, e), 0)
6      else if e = a then                              /* e is equal to the first parameter */
7          pendingActivations.put((e, b), 1)
8      if isLastEvent(e,t) then
9          acts ← pendingActivations.get((a, b))
10         if acts == 0 then
11             if a ∈ t then
12                 fulfillingTraces.put((a, b), fulfillingTraces.get((a, b) + 1)
13             else
14                 vacuousTraces.put((a, b), vacuousTraces.get((a, b) + 1))
```

are no longer pending activations for these constraints when `e` occurs). All the RESPONDEDEXISTENCE constraints having `e` as first parameter are activated when `e` occurs and, therefore, if the second parameter has not occurred yet, the number of pending activations for these constraints in `t` is set to 1 (indicating that there is at least 1 pending activation) (line 11). At the end of each trace, if the number of pending activations is 0 for a candidate constraint, the constraint is satisfied (or vacuously satisfied) in that trace. If the first parameter of the constraint occurs in the current trace, i.e., the constraint is activated, the number of fulfilling traces is incremented by one for that constraint (since the constraint is non-vacuously satisfied in this case) (line 16); if, instead, the constraint is not activated, the number of traces in which the constraint is vacuously satisfied is incremented by one (since the constraint is vacuously satisfied) (line 18).

Sequence Analysis for RESPONSE. The semantics of the RESPONSE template can be defined as "If `A` occurs, then `B` occurs after `A`". The pseudo-code for the RESPONSE algorithm is reported in Algorithm 5. It takes as input the current event `e` and the current trace `t` and, if `e` is the first event in `t`, it initializes map *pendingActivations* (containing the number of pending

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

activations in the current trace for each candidate RESPONSE constraint) (line 2). Line 5 of the algorithm sets to 0 the pending activations in `t` for the RESPONSE constraints having `e` as second parameter. On the other hand, all the RESPONSE constraints having `e` as first parameter are activated when `e` occurs and, therefore, the number of pending activations for these constraints in `t` is set to 1 (indicating that there is at least 1 pending activation) (line 13). At the end of each trace, if the number of pending activations is 0 for a candidate constraint, the constraint is satisfied (or vacuously satisfied) in that trace. If the first element of the constraint occurs in the current trace, i.e., the constraint is activated, the number of fulfilling traces is incremented by one for that constraint (since the constraint is non-vacuously satisfied in this case) (line 12); if, instead, the constraint is not activated, the number of traces in which the constraint is vacuously satisfied is incremented by one (since the constraint is vacuously satisfied) (line 14).

Sequence Analysis for ALTERNATERESPONSE. The semantics of the ALTERNATERESPONSE template can be defined as "Each time `A` occurs, then `B` occurs afterwards, before `A` recurs". The pseudo-code for the ALTERNATERESPONSE algorithm is reported in Algorithm 6. It takes as input the current event `e` and the current trace `t` and, if `e` is the first event in `t`, it initializes set *violatedCandidates* (containing the candidate ALTERNATERESPONSE constraints that have already been recognized as violated in the current trace) and map *pendingActivations* (containing the number of pending activations in the current trace for each candidate ALTERNATERESPONSE constraint) (lines 2-3). Line 7 of the algorithm sets to 0 the pending activations in `t` for the ALTERNATERESPONSE constraints having `e` as second parameter. Every candidate ALTERNATERESPONSE constraint having `e` as first parameter is activated when `e` occurs. Therefore, if the number of pending activation for a candidate (not yet violated) is greater than 0, the constraint is violated in the current trace. Otherwise, the number of pending activations for this constraint in `t` is set to 1 (line 13). At the end of each trace, if the number of pending activations is 0 for a (not violated) candidate constraint, the constraint is satisfied (or vacuously satisfied) in that trace. If the first parameter of the constraint occurs in the current trace, i.e., the constraint is activated, the number of fulfilling traces is incremented by one for that constraint (since the constraint is non-vacuously satisfied in this case) (line 18); if, instead, the constraint is not activated, the number of traces in which the constraint is vacuously satisfied is incremented by one

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

Algorithm 6: *Sequence Analysis* for AlternateResponse

Input: `e` the event to be processed;
`t` the current trace.
Data: *fulfillingTraces* a map associating each candidate constraint to the number of traces where it is fulfilled;
vacuousTraces a map associating each candidate constraint to the number of traces where it is vacuously satisfied;
$\widehat{\mathscr{A}_2}$ the set of candidate constraints.

```
1  if isFirstEvent(e, t) then
2      initialize set violatedCandidates
3      initialize map pendingActivations
4  foreach (a, b) ∈ Â2 do
5      if ¬violatedCandidates.contains((a, b)) then
6          if e == b then                          /* e is equal to the second parameter */
7              pendingActivations.put((a, e), 0)
8          else if e == a then                     /* e is equal to the first parameter */
9              pends ← pendingActivations.get((e, b))
10             if pends > 0 then
11                 violatedCandidates.add((e, b))
12             else
13                 pendingActivations.put((e, b), 1)
14         if isLastEvent(e, t) then
15             acts ← pendingActivations.get((a, b))
16             if acts == 0 then
17                 if a ∈ t then
18                     fulfillingTraces.put((a, b), fulfillingTraces.get((a, b) + 1)
19                 else
20                     vacuousTraces.put((a, b), vacuousTraces.get((a, b) + 1))
```

(since the constraint is vacuously satisfied) (line 20).

Sequence Analysis for ChainResponse. The semantics of the ChainResponse template can be defined as "Each time `A` occurs, then `B` occurs immediately after". The pseudo-code for the ChainResponse algorithm is reported in Algorithm 7. It takes as input the current event `e` and the current trace `t` and, if `e` is the first event in `t`, it initializes set *violatedCandidates* (containing the candidate ChainResponse constraints that have already been recognized as violated in the current trace) (line 2) and the global variable *lastEventInTrace* (containing the last event processed in the current trace) (line 3). All candidates having the previous event processed as first parameter and an event different from the current one as second parameter are recognized as violated (line 8).

If a candidate constraint has not yet been violated in the current trace

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

Algorithm 7: *Sequence Analysis* for ChainResponse

Input: `e` the event to be processed;
`t` the current trace.
Data: *fulfillingTraces* a map associating each candidate constraint to the number of traces where it is fulfilled;
vacuousTraces a map associating each candidate constraint to the number of traces where it is vacuously satisfied;
$\widehat{\mathscr{A}_2}$ the set of candidate constraints;
lastEventInTrace a variable containing the last event occurred in the current trace.

```
1  if isFirstEvent(e, t) then
2      initialize set violatedCandidates
3      initialize variable lastEventInTrace
4  foreach (a, b) ∈ Â2 do
5      if ¬isFirstEvent(e, t) then
6          if (a == lastEventInTrace) then
7              if (e ≠ b) then                    /* e is not equal to the second parameter */
8                  violatedCandidates.add((e, b))
9      if ¬violatedCandidates.contains((a, b)) then
10         if isLastEvent(e, t) then
11             if (e ≠ a) then                    /* e is not equal to the first parameter */
12                 if a ∈ t then
13                     fulfillingTraces.put((a, b), fulfillingTraces.get((a, b) + 1)
14                 else
15                     vacuousTraces.put((a, b), vacuousTraces.get((a, b) + 1))
16 lastEventInTrace = e
```

and the current event `e` is the last event of the trace and is different from the first parameter of the constraint, the constraint is satisfied (or vacuously satisfied) in the trace. If the first parameter of the constraint occurs in the current trace, i.e., the constraint is activated, the number of fulfilling traces is incremented by one for that constraint (since the constraint is non-vacuously satisfied in this case) (line 13); if, instead, the constraint is not activated, the number of traces in which the constraint is vacuously satisfied is incremented by one (since the constraint is vacuously satisfied) (line 15).

Sequence Analysis for Existence *and* Absence. The existence constraint Existence$(n, \mathtt{A})$ can be described as "`A` occurs at least n times". Similarly, Absence$(m + 1, \mathtt{A})$ means "`A` occurs at most m times". Algorithm 8 shows the pseudo-code for the Existence and Absence algorithms. Their implementations differ based on the *existenceCondition* function. The algorithm takes as input the current event `e` and the current trace `t` and, if `e` is the first event in `t`, it initializes the maps *eventCounter* (for counting the number of

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

Algorithm 8: *Sequence Analysis* for Existence and Absence

Input: `e` the event to be processed;
`t` the current trace.
Data: *fulfillingTraces* a map associating each candidate constraint to the number of traces where it is fulfilled;
$\widehat{\mathscr{A}_1}$ the set of candidate constraints.

```
1  if isFirstEvent(e, t) then
2  |   initialize map eventCounter
3  if e ∉ eventCounter then
4  |   eventCounter.put(e, 1)
5  else
6  |   eventCounter.put(e, eventCounter.get(e) + 1)
7  forall a ∈ Â1 do
8  |   if isLastEvent(e, t) then
9  |   |   acts ← eventCounter.get((a))
10 |   |   if existenceCondition(acts) then
11 |   |   |   fulfillingTraces.put((a, b), fulfillingTraces.get((a, b) + 1))
```

occurrences of each event in the current trace). At the end of each trace, if the *existenceCondition* is verified for a candidate constraint, the constraint satisfied in that trace and the number of fulfilling traces is incremented by one for that constraint (line 11).

The *existenceCondition* differs based on the specific template of the *Sequence Analysis*:

- Existence(n, A): the number of occurrences of A must be greater than or equal to n,
- Absence$(m + 1, \mathsf{A})$: the number of occurrences of A must be at most m.

The algorithms for the other templates specified in Table 1 can be easily derived from the ones described in this section. In particular, the algorithms for Precedence, AlternatePrecedence and ChainPrecedence are the same as the ones described for Response, AlternateResponse and ChainResponse, respectively. The only difference is that, for the Precedence templates, the traces in the input log have to be parsed from the end to the beginning. Similarly, the algorithms for checking the negative relation templates are the same as the ones described for the corresponding relation templates. In this case, every trace that is (non-vacuously) satisfied for a relation template is violated for the corresponding negative relation template.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

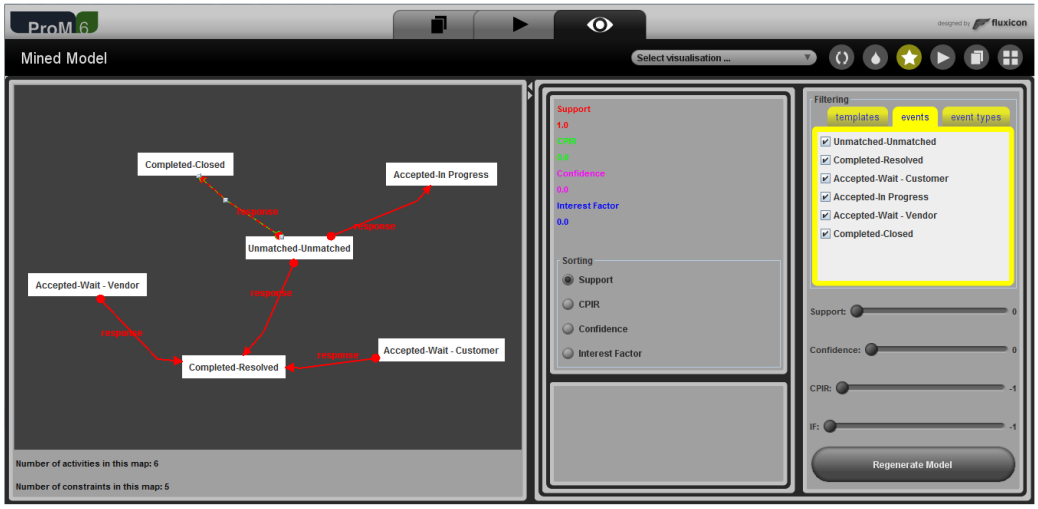


Figure 5: A screenshot of the Declare Miner 2.0 ProM plug-in.

We close this section by mentioning that we handle the specific case of empty traces outside the presented algorithms. An empty trace (vacuously) satisfies all relation, mutual relation and negative relation constraints. It satisfies ABSENCE constraints and violates any EXISTENCE constraint.

3.4. Implementation

All the algorithms presented in this work have been implemented in the *Declare Miner 2.0*, a plug-in of the process mining tool-kit ProM.[2] In the plug-in, the user can select the version that she prefers to run (sequential, with search space or with database partitioning), according to her needs. Figure 5 shows a screenshot of the *Declare Miner 2.0* plug-in.

To enhance the usability of the tool, a feature has been added, which allows for avoiding to retrieve overcomplicated models when looking for negative constructs such as NOTSUCCESSION and NOTCHAINSUCCESSION. Such a feature allows for considering NOTSUCCESSION and NOTCHAINSUCCESSION constraints activated in a trace only if both the involved activities occur in

[2] http://www.processmining.org/

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

the trace. In this way, if we have the following log

$$\mathfrak{L} = [\langle \mathsf{b}, \mathsf{a}, \mathsf{d} \rangle, \langle \mathsf{b}, \mathsf{c}, \mathsf{a} \rangle, \langle \mathsf{b}, \mathsf{d}, \mathsf{a} \rangle, \\ \langle \mathsf{b}, \mathsf{e}, \mathsf{a} \rangle, \langle \mathsf{b}, \mathsf{f}, \mathsf{a} \rangle, \langle \mathsf{b}, \mathsf{g}, \mathsf{a} \rangle, \langle \mathsf{b}, \mathsf{h}, \mathsf{a}, \mathsf{d} \rangle]$$

such a new feature allows the user to consider the NotSuccession template as interesting only when instantiated with the pair of activities (a, b), instead of considering as activated and satisfied all over the log also constraints obtained by instantiating the NotSuccession with pairs (a, c),(a, e),(a, f),(a, g),(a, h). This significantly improves the understandability of the resulting models.

Finally, the application has also been enhanced with a new feature that provides users with a textual report about the discovered Declare constraints. For instance, a Response constraint between activities Accepted-Wait - Implementation and Completed-Resolved, discovered from the BPIC2013 (incidents) log [17] will be verbalized as follows:

> *Whenever activity "Accepted-Wait - Implementation" is executed, activity "Completed-Resolved" is eventually executed afterwards.*
>
> - *cases where activity "Accepted-Wait - Implementation" is executed and the statement is valid (5,33% of cases, 403 cases in total)*
> - *cases where activity "Accepted-Wait - Implementation" is executed and the statement is not valid (0,13% of cases, 10 cases in total)*
> - *cases where activity "Accepted-Wait - Implementation" is not executed (94,53% of cases, 7141 cases in total)*

4. Evaluation

We evaluated the presented algorithms in terms of memory consumption and time performance using a wide range of synthetic and real-life logs. In the remainder of this section, we describe the event logs in Section 4.1 and the procedure we used for the evaluation in Section 4.2. Finally, we discuss the results in Section 4.3.

4.1. Event Logs

For our experimentation we used: *(i)* 76 synthetic logs with different characteristics to compare how the algorithms perform under different conditions;

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

and *(ii)* 8 real-life logs, made publicly available during the last six years (2012–2017) by the IEEE Task Force on Process Mining.[3]

4.1.1. Synthetic Logs

The synthetic logs have been generated using the generator described in [18, 19]. The log generator allows for the generation of logs obtained simulating a DECLARE model. In our experiments, we have used the DECLARE model of our running example shown in Fig. 1. Using the generator, we can create logs of a specified size (s), containing traces of a given length (l) and built on an alphabet of a given size ($|\Sigma|$). We used these parameters, characterizing the complexity of an event log, as independent variables for assessing and comparing the performance of the different algorithms. To account for the separate effect of the three dimensions under analysis, we let each variable vary individually while keeping the remaining two assigned with a default value. In particular, we assigned s with values ranging from 400 to 8 800 traces at steps of 400, l ranging from 8 to 108 events at steps of 4, and $|\Sigma|$ ranging from 8 to 116 activities at steps of 4 (the smallest one being the alphabet of the running example, $\Sigma = \{a, b, c, d, e, f, g, h\}$). We assigned as defaults $s = 1\,600$, $l = 16$, and $|\Sigma| = 16$.[4] The following configuration sets have thus been applied for the generation of the synthetic logs:

1. $s \in \{400, 800, \ldots, 8\,800\}$, $l = 16$ (default), and $|\Sigma| = 16$ (default), i.e., 22 logs of different sizes;

2. $s = 1\,600$ (default), $l \in \{8, 12, \ldots, 108\}$, and $|\Sigma| = 16$ (default), i.e., 26 logs with traces of different lengths;

3. $s = 1\,600$ (default), $l = 16$ (default), and $|\Sigma| \in \{8, 12, \ldots, 116\}$, i.e., 28 logs built on alphabets of increasing sizes.

We remark that the increase in the size of Σ introduces activities that are not subject to constraints, thus entailing a higher variability in the sequences of events in each trace. We have made the used synthetic event logs publicly

[3] Available at https://data.4tu.nl/repository/collection:event_logs_real

[4] The default values were selected to resemble the characteristics of the Sepsis2016 [20] real-life log.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

available as benchmarks,[5] to the benefit of researchers and practitioners interested in the conduction of similar performance experiments.

4.1.2. Real-Life Logs

To evaluate the performance of our approach on real-life benchmarks, we have used the event logs listed in Table 3:

- the BPI Challenge 2012 log (BPIC2012 [21]) and the BPI Challenge 2017 log (BPIC2017 [22]) pertain to an application process for personal loans or overdrafts in a Dutch financial institute;
- the BPI Challenge 2013 logs (BPIC2013 (open), BPIC2013 (incidents), BPIC2013 (closed) [17]) are related to an incident management process supported by a system called VINST in use at Volvo IT Belgium;
- the BPI Challenge 2014 log (BPIC2014 [23]) pertains to the management of calls or mails from customers to the Service Desk concerning disruptions of ICT-services from Rabobank Group ICT;
- the Traffic Fines log (Fines2015 [24]) was extracted from an information system handling road traffic fines that are processed by an Italian municipality;
- the Sepsis log (Sepsis2016 [20]) reports the trajectories of patients showing symptoms of sepsis in a Dutch hospital, from their registration in the emergency room to their discharge.

Sepsis2016 was included because of the reported flexibility of the healthcare process behind it [25]. This makes it a suitable input for declarative process discovery, because of the inherent knowledge-intensive nature of the underlying process [12, 26].

In order to prove that our approach is not only suitable for flexible scenarios, but also applicable to event logs stemming from more structured processes, we have considered other benchmarks chosen for their heterogeneous characteristics in terms of number of traces, events per trace, and alphabet sizes. In particular, BPIC2012 and BPIC2014 have a large alphabet size. BPIC2017 contains more than one million events and traces with high average

[5]The full set of synthetic logs can be found at https://github.com/cdc08x/DeclareMiner2

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

Log	Traces	Events		Alph. size
		Total	Avg. per trace	
BPIC2012	13 087	262 200	20.03	36
BPIC2013 (closed)	1 487	6 659	4.48	7
BPIC2013 (open)	819	2 350	2.87	5
BPIC2013 (incidents)	7 554	65 532	8.68	13
BPIC2014	46 616	466 737	10.01	39
Fines2015	150 370	561 469	3.73	11
Sepsis2016	1 050	15 214	14.49	16
BPIC2017	31 509	1 202 266	38.16	26

Table 3: Characteristics of the real-life logs.

length. Fines2015 contains a large amount of traces, but traces are rather short (3 to 4 events per trace).

In real-life use cases, indeed, it might not be possible to be aware of the degree of flexibility of the mined process prior to the analysis of its event logs, hence the need to make our approach capable of analyzing a wider spectrum of datasets. In addition, we remark that declarative models are reportedly effective to shed light on circumstantial information, i.e., to clarify which circumstances will cause an action to be performed [27]. Conversely, procedural models tend to obscure such a representation of facts. Therefore, the opportunity to mine declarative models should be given regardless of the nature of the underlying process, so as to provide the process analyst with a different view on details that could otherwise remain hidden.

4.2. Procedure

All the experiments have been run using all the combinations of values 80%, 90%, and 100% for $supp_{min}$ with both vacuity detection enabled and disabled. Both the multi-threading variants have been configured with 4 threads. This configuration has been chosen because, on the machine used for the experiments, the performance of the algorithms improves when increasing the number of threads from 1 to 4 and starts to degrade with more than 4 threads. The experiments have been run on a Ubuntu Linux 12.04 server machine, equipped with Intel Xeon CPU E5-2650 v2 2.60GHz, using eight 64-bit CPU cores and 16GB main memory quota. The time required for both *Apriori algorithm* and *Sequence Analysis* has been collected for all logs and configurations, averaged over three runs and reported in seconds. The

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

memory usage has been checked for the processing of every trace in the 8 real-life logs, and finally averaged. The reported measurements are in MBs.

4.3. Results

In the following, we discuss the results of our evaluation. First, we show the results obtained with the synthetic logs in terms of computation times (Section 4.3.1). Then, we show computation times and memory consumption obtained from experiments on the real-life logs (Section 4.3.2).

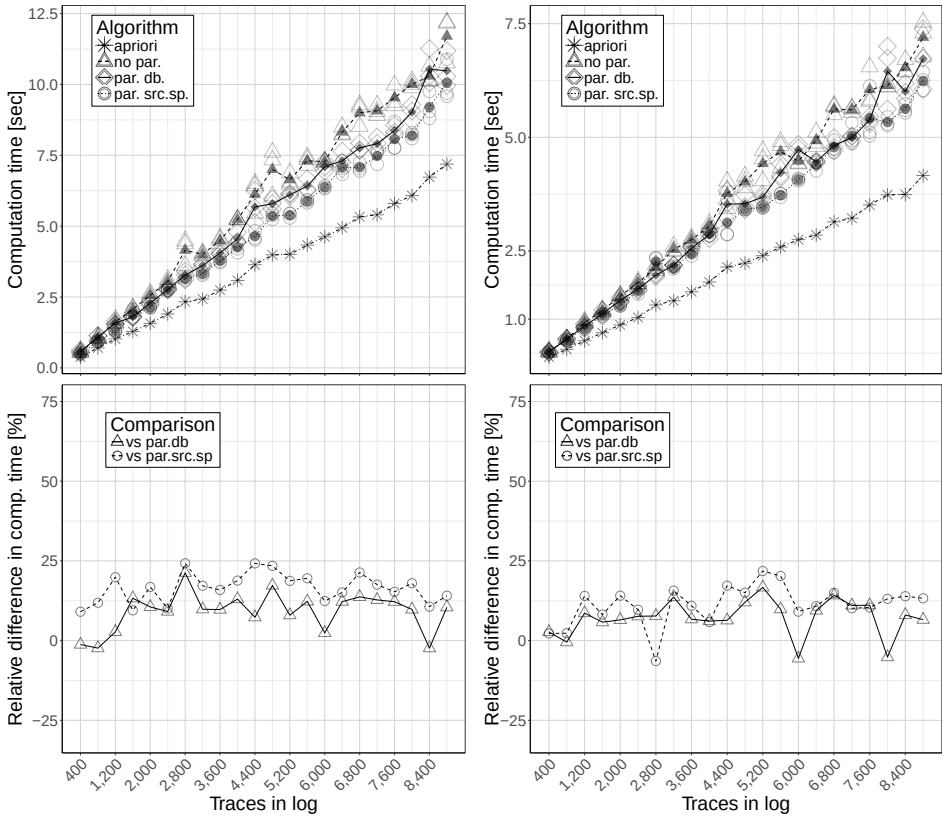
4.3.1. Synthetic Logs

The synthetic logs have been used to investigate the time performance of the 3 algorithms presented. We show the plots obtained using $supp_{min} = 90\%$ since the trends for different supports are similar with generally lower computation times for higher $supp_{min}$.[6] We report the results when varying the log size, the trace length, and the alphabet size. In particular, we show, for all configurations, both the computation time needed for the *Apriori algorithm* (`apriori`) and for *Sequence Analysis* without partitioning (`no par`), with database partitioning (`par.db`), and with search space partitioning (`par.src.sp`). In addition, we show the relative difference between the computation times needed for *Sequence Analysis* without partitioning, and with database (`vs par.db`) and search space partitioning (`vs par.src.sp`).

Varying the log size. Figure 6 reports the plots related to the time performance of the presented algorithms when varying the log size. The computation times are very low (in the order of few seconds). As expected the average times obtained with vacuity detection enabled ($\sim$ 6 sec for the log with 8 800 traces) are lower than the ones obtained with vacuity detection disabled ($\sim$ 10 sec for the log with 8 800 traces). Indeed, with vacuity detection disabled the *Apriori algorithm* returns a significantly higher number of frequent activity sets that need to be handled in the *Sequence Analysis*. However, the trends in the two cases are similar and, in both cases, `par.db` and `par.src.sp` perform slightly better than `no par`. The plots showing the relative difference of `par.src.sp` and `par.db` with respect to `no par` show that `par.src.sp` improves a bit more the performance with respect to `par.db`, especially when the vacuity detection is disabled. The time needed for the *Apriori algorithm* ($\sim$ 4 sec and

[6] The entire collection of plots can be downloaded at https://github.com/cdc08x/DeclareMiner2

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002



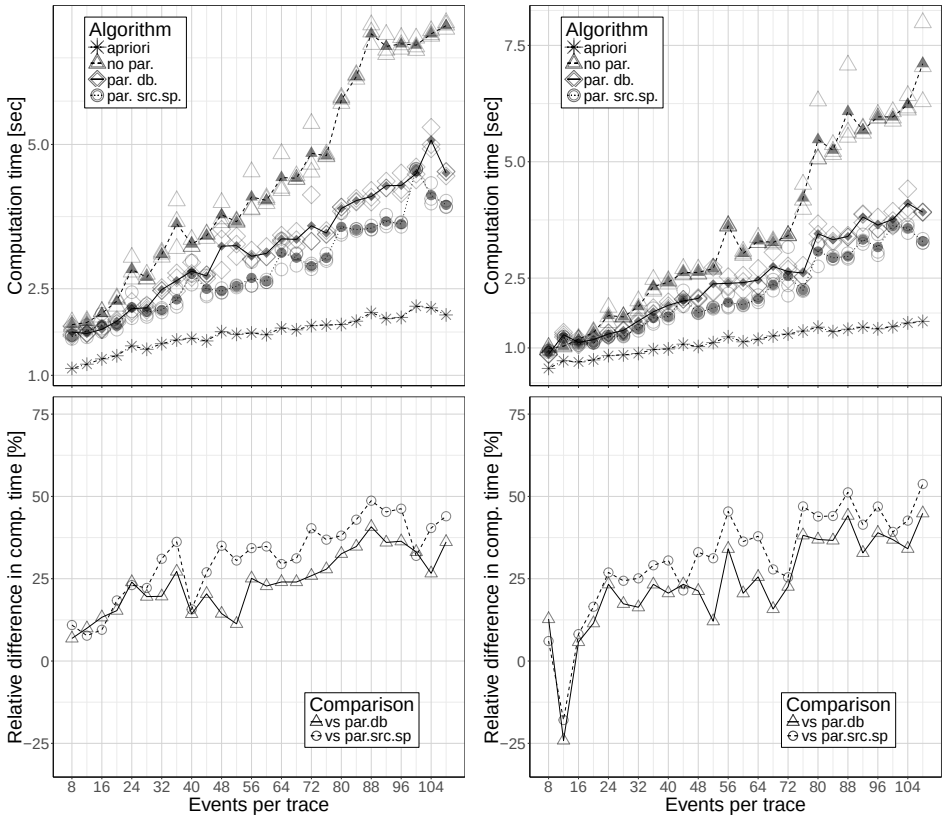
(a) Without vacuity detection (b) With vacuity detection

Figure 6: Computation time as a function of the log size.

$\sim$ 7.5 sec for the log with 8 800 traces with and without vacuity detection, respectively) is always significantly lower with respect to the one needed for *Sequence Analysis*.

Varying trace length. Figure 7 reports the plots related to the time performance of the presented algorithms when varying the trace length. The highest average computation time obtained for `par.db` and `par.src.sp` with vacuity detection enabled is $\sim$ 3 sec and with vacuity detection disabled is $\sim$ 4 sec (for traces with 108 events). The highest average computation time obtained for `no par` with vacuity detection enabled and disabled is of $\sim$ 7.5 sec and $\sim$ 8 sec, respectively. Thus, for logs containing long traces, `par.db` and `par.src.sp` perform significantly better than `no par`. The plots

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002



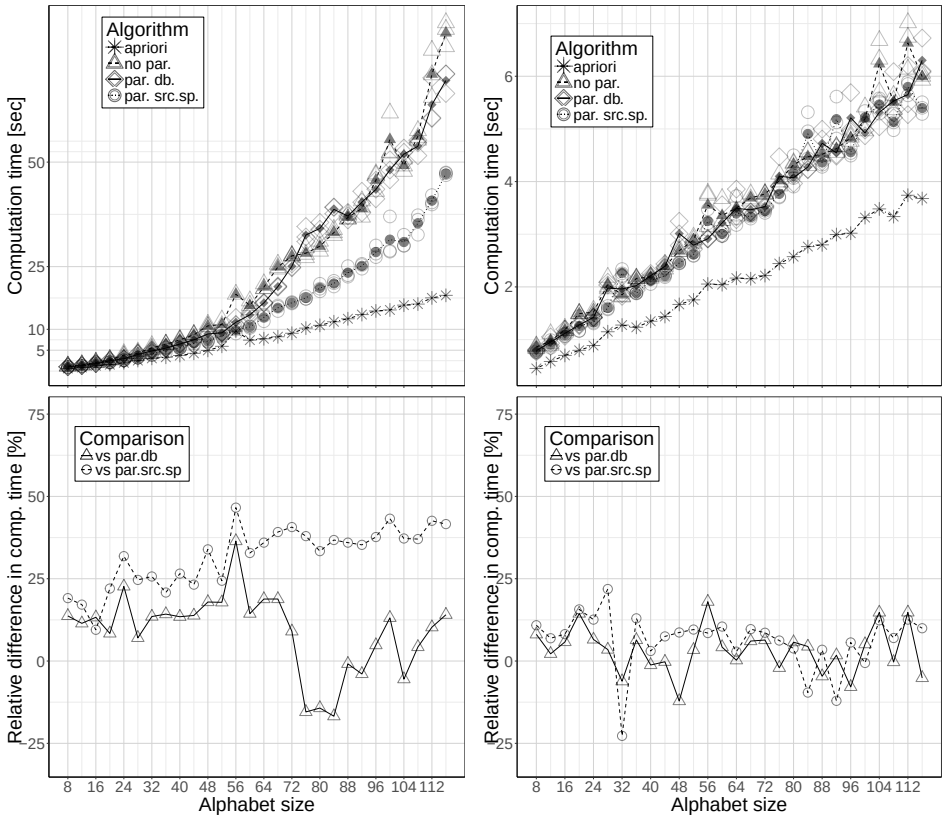
(a) Without vacuity detection (b) With vacuity detection

Figure 7: Computation time as a function of the traces length.

showing the relative difference of `par.src.sp` and `par.db` with respect to `no par` show that `par.src.sp` performs a bit better than `par.db` and reaches 50% of improvements with respect to `no par` with both vacuity detection enabled and disabled. The time needed for the *Apriori algorithm* is always significantly lower with respect to the one needed for *Sequence Analysis*.

Varying alphabet size. Figure 8 reports the plots related to the time performance of the presented algorithms when varying the alphabet size. In this case, the trends of the average computation times obtained with and without vacuity detection is significantly different. When vacuity detection is enabled `par.src.sp`, `par.db` and `no par` have similar performance (the computation time is extremely low and always lower than 7 sec). `par.db` and `no par`

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002



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

Figure 8: Computation time as a function of the alphabet size.

have a similar trend also in the case in which vacuity detection is disabled. However, in this case, `par.src.sp` performs significantly better than `no par` with an improvement of up to 50%. Also in this case, the time needed for the *Apriori algorithm* is lower with respect to the one needed for *Sequence Analysis*.

Enabling Vacuity Detection. Figure 9 shows how the time performance of the implemented algorithms improves by enabling vacuity detection. The reported times are gathered from runs on the synthetic event log having default values for s, l, and $|\Sigma|$. Computation times of `no par`, `par.db`, and `par.src.sp` are compared at different levels of $supp_{min}$ (80%, 90%, and 100%) having vacuity detection either enabled (dark grey) or disabled (light grey). All the

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

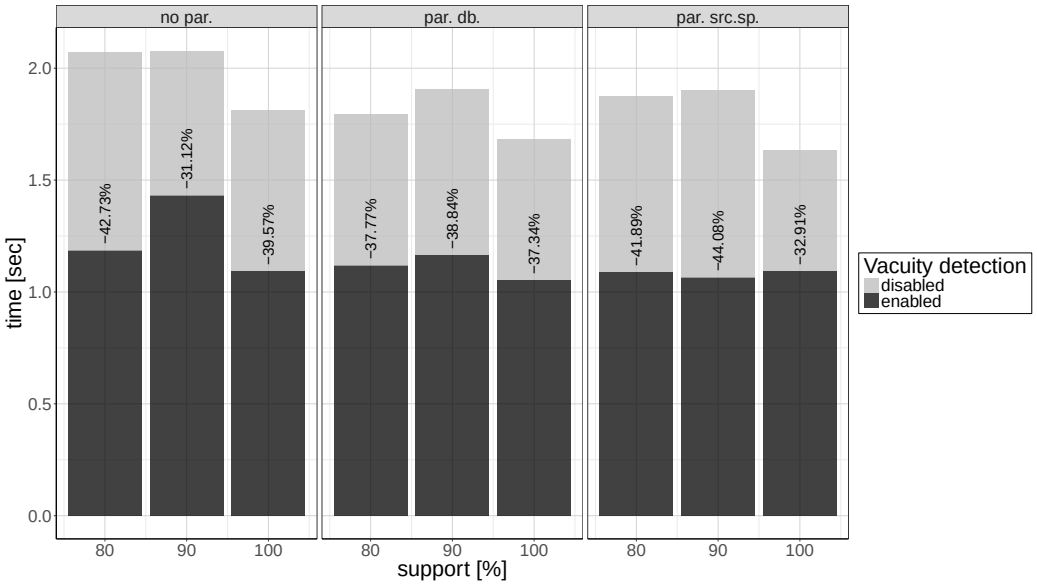


Figure 9: Computation time reduction achieved through the enablement of vacuity detection.

implemented algorithms show a similar improvement in terms of computation times when the vacuity detection is enabled. In addition, the number of constraints returned when the vacuity detection is enabled is much lower. This confirms the usefulness of this mechanism especially when dealing with real-life logs.

To summarize, the results found with the experiments on synthetic logs show that:

- the computation times are generally very low (few seconds);
- the time needed for the *Apriori algorithm* is always significantly lower with respect to the one needed for *Sequence Analysis*;
- the computation times with vacuity detection disabled are significantly higher than the ones obtained with vacuity detection enabled;
- `par.db` and `par.src.sp` perform in general better than `no par`. The highest improvement is obtained with logs containing long traces and, in general, a better improvement is obtained with vacuity detection disabled (when the computation time is higher);

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

Log	$supp_{min}$	Average computation time [sec]			
		`no par`	`par.src.sp`	`par.db`	[6]
BPIC 2012	80%	19.196	17.240	17.372	68.270
	90%	19.292	18.092	17.391	66.544
	100%	19.952	17.512	17.064	64.251
BPIC 2013 (closed)	80%	0.504	0.421	0.444	7.613
	90%	0.461	0.431	0.487	7.553
	100%	0.433	0.435	0.405	0.544
BPIC 2013 (incidents)	80%	4.243	3.769	3.673	2.968
	90%	3.972	3.634	3.889	2.759
	100%	4.080	3.991	3.755	2.401
BPIC 2013 (open)	80%	0.210	0.180	0.187	0.415
	90%	0.195	0.170	0.186	0.401
	100%	0.197	0.180	0.169	0.344
BPIC 2014	80%	65.422	59.761	62.078	1 292.303
	90%	64.771	61.852	64.066	482.475
	100%	60.897	61.056	60.353	26.052
Fines 2015	80%	66.656	59.073	58.901	33.832
	90%	67.031	57.574	60.658	25.782
	100%	62.561	59.106	59.643	24.449
Sepsis 2016	80%	1.022	0.738	0.850	62.591
	90%	0.956	0.730	0.869	43.684
	100%	0.790	0.792	0.900	7.207
BPIC 2017	80%	114.863	90.593	99.007	17 431.343
	90%	112.916	88.799	101.801	17 974.533
	100%	98.528	83.862	87.526	3 505.525

Table 4: Computation times using real-life logs (with vacuity detection).

- in general, `par.src.sp` improves more the performance of `no par` than `par.db`.

4.3.2. Real-Life Logs

The real-life logs have been used to investigate the performance of the three algorithms presented both in terms of computation time and memory consumption.

Computation time. Tables 4 and 5 report, for each of the presented algorithms, the computation times required to process the 8 real-life logs for different values of $supp_{min}$ with vacuity detection enabled and disabled, respectively. By looking at the tables, it is possible to recognize the same trends observed in the experiments with synthetic logs. The computation times are in general low (from few milliseconds for BPIC2013 (open) to few minutes

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

Log	$supp_{min}$	Average computation time [sec]			
		`no par`	`par.src.sp`	`par.db`	[6]
BPIC 2012	80%	59.638	41.677	45.443	13 630.461
	90%	50.386	39.764	41.185	8 708.702
	100%	45.655	36.593	38.455	5 169.026
BPIC 2013 (closed)	80%	0.671	0.550	0.687	39.422
	90%	0.665	0.643	0.643	33.259
	100%	0.709	0.580	0.687	14.776
BPIC 2013 (incidents)	80%	6.854	5.836	6.412	33.259
	90%	6.786	5.541	6.235	39.422
	100%	5.901	5.840	5.474	81.495
BPIC 2013 (open)	80%	0.250	0.233	0.250	10.486
	90%	0.257	0.233	0.239	6.037
	100%	0.218	0.217	0.209	0.349
BPIC 2014	80%	185.442	125.619	153.482	> 18 000.000
	90%	181.092	130.707	152.194	> 18 000.000
	100%	111.903	109.115	109.225	2 268.203
Fines 2015	80%	107.106	90.914	97.097	> 18 000.000
	90%	101.198	87.012	93.948	> 18 000.000
	100%	98.126	86.400	91.889	9 521.085
Sepsis 2016	80%	1.647	1.225	1.432	221.289
	90%	1.600	1.217	1.534	181.873
	100%	1.321	1.426	1.386	65.857
BPIC 2017	80%	394.033	259.912	298.170	> 18 000.000
	90%	361.741	255.384	288.128	> 18 000.000
	100%	270.920	227.391	230.755	> 18 000.000

Table 5: Computation times using real-life logs (without vacuity detection).

for BPIC2017). The computation times obtained with vacuity detection disabled are significantly higher than the ones obtained with vacuity detection enabled. `par.db` and `par.src.sp` perform in general better than `no par` and `par.src.sp` improves more the performance of `no par` with respect to `par.db`. The highest improvement of the parallel algorithms is obtained with BPIC2017 that contains traces with high average length. With large alphabet and log sizes the improvement is less evident. See, for example, the case of Fines2015 which contains a large amount of traces but short. Also, a better improvement is obtained when the vacuity detection is disabled. From the tables it is also possible to notice that, in general, the discovery of constraints with lower support requires higher computation time (the computation time decreases when $supp_{min}$ increases).

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

Log	$supp_{min}$	Average memory usage [MB]		
		`no par`	`par.src.sp`	`par.db`
BPIC 2012	80%	2797.896	2990.313	2987.244
	90%	3479.370	3672.308	3689.226
	100%	2814.076	2982.535	2951.414
BPIC 2013 (closed)	80%	3224.890	3402.379	3321.312
	90%	3250.195	3422.971	3320.683
	100%	3224.701	3375.245	3378.217
BPIC 2013 (incidents)	80%	3272.994	3405.096	3350.712
	90%	3336.335	3467.524	3423.487
	100%	3383.542	3582.805	3509.252
BPIC 2013 (open)	80%	3418.733	3591.149	3552.671
	90%	3452.391	3620.048	3567.557
	100%	3410.776	3594.08	3571.845
BPIC 2014	80%	2522.498	2525.335	2752.694
	90%	2554.258	2558.694	2784.968
	100%	3041.431	2812.829	3043.947
Fines 2015	80%	2757.921	2753.657	2780.860
	90%	3311.736	3031.759	2771.034
	100%	2756.915	3033.848	2743.068
Sepsis 2016	80%	3080.066	3117.173	3086.295
	90%	3085.615	3095.000	3111.557
	100%	3095.138	3097.057	3118.443
BPIC 2017	80%	3385.102	2730.640	2754.912
	90%	2776.020	2775.573	3225.580
	100%	3470.691	2816.340	3482.457

Table 6: Memory consumption using real-life logs (with vacuity detection).

Tables 4 and 5 report the execution times required to process the logs using the Declare Miner [6]. The original version of the Declare Miner attains better timings occasionally, e.g., with vacuity detection enabled and for BPIC2013 (incidents) and Fines2015 logs. However, our approach clearly improves over it for scalability: even having vacuity detection enabled, the performance of the original Declare Miner steeply decays as logs increase in the number of events per trace (see Table 3). For these logs, indeed, the performance improvement is noticeable and the ratio reaches peaks of approximately $1:50$ in the case of Sepsis2016 and $1:100$ for BPIC2017, as shown in Table 4. When vacuity detection is disabled, the performance of the original version of the Declare Miner decay dramatically, as opposed to our proposed approach. In particular, 5 hours turned out to be not sufficient to return a result when analyzing BPIC2014, Fines2015, and BPIC2017.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

Log	$supp_{min}$	Average memory usage [MB]		
		`no par`	`par.src.sp`	`par.db`
BPIC 2012	80%	3164.679	3360.663	3344.785
	90%	2507.653	2671.431	2607.790
	100%	3179.398	3343.870	3269.200
BPIC 2013 (closed)	80%	3222.504	3402.914	3326.421
	90%	3251.205	3422.295	3319.545
	100%	3233.109	3399.611	3375.306
BPIC 2013 (incidents)	80%	3311.958	3459.879	3388.076
	90%	3369.209	3523.088	3497.857
	100%	3422.844	3598.485	3541.132
BPIC 2013 (open)	80%	3410.075	3621.799	3576.108
	90%	3440.457	3608.402	3586.726
	100%	3422.180	3628.260	3569.316
BPIC 2014	80%	2492.127	2682.854	2878.689
	90%	2527.365	2521.371	2531.285
	100%	3124.654	2553.222	2552.657
Fines 2015	80%	2677.579	2685.744	3104.895
	90%	2683.610	3082.046	2895.949
	100%	2882.841	2889.041	2916.918
Sepsis 2016	80%	3082.985	3086.389	3069.019
	90%	3085.844	3097.296	3101.323
	100%	3094.253	3110.253	3139.643
BPIC 2017	80%	2648.188	2873.389	2672.466
	90%	2915.772	2912.520	2927.845
	100%	3184.292	2947.643	3429.015

Table 7: Memory consumption using real-life logs (without vacuity detection).

Memory consumption evaluation. Table 6 and Table 7 report the average memory usage (in MBs) needed for processing each trace in the real-life logs. The reported measurements are in MBs. The memory consumption is quite uniform across all the logs and ranges from 2 500 to 3 500 MBs. This suggests that, overall, the characteristics of the input logs do not affect the memory usage.

5. 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 labeled as compliant or non-compliant, declarative constraints expressed as ICs (Integrity Constraints) [28]. A logic-

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

based approach for probabilistic process mining that leverage this approach is presented in [29].

An approach that makes use of logic programming for declarative process mining is presented in [30]. The proposed methodology is based on Inductive Logic Programming. The Inductive Constraint Logic algorithm, used in that approach, is adapted to the problem of learning ICs in SCIFF and is able to learn a model by considering both compliant and non-compliant traces.

An algorithm to discover DECLARE models was developed in [31] using email messages as event log traces. The implemented algorithm, *MINERful* [32], is a two-step algorithm. The first step aims at building a knowledge base starting from the event log. The second step aims at computing the support of constraints by querying the knowledge base. In [33, 34], the authors propose an extension of the approach presented in [32] to discover target-branched DECLARE constraints, i.e., constraints in which the target parameter is replaced by a disjunction of real activities.

In [35, 36], a semantics for defining DECLARE constraints on non-atomic activities and an approach for the discovery of this type of constraints are presented. In [37, 38], the semantics of DECLARE is extended to consider metric temporal constraints and, in [39], an approach for the discovery of these constraints is presented. In [40], the semantics of DECLARE is extended to consider conditions on data. In the same paper, a technique based on daikon and decision trees is presented for the discovery of data-aware DECLARE constraints.

In [41], the authors present a mining approach that works with RelationalXES, a relational database architecture for storing event log data. The relational event data is queried with conventional SQL. Standard queries allow for the discovery of DECLARE constraints. However, they can be customized and cover process perspectives beyond control flow as shown in [42].

An on-line process discovery technique which takes data from event streams is presented in [43, 44]. The proposed approach is able to produce at runtime an updated picture of the process behavior in terms of DECLARE constraints.

6. Conclusion

The Declare Miner is a plug-in of the process mining tool ProM for the discovery of DECLARE models from logs. The high execution times of the Declare Miner when processing large sets of data hampers the applicability of the tool to real-life settings. Therefore, in this work, we have presented a

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

new approach for the discovery of DECLARE models based on a combination of an *Apriori algorithm* and a group of algorithms for *Sequence Analysis* to enhance the time performance of the plug-in. In addition, we have used the notions of search space partitioning and database partitioning as a basis for the development of two multi-threading variants of the approach. The new algorithms have been implemented and tested on synthetic and real-life logs to assess their efficiency.

In the future, we aim at investigating the possibility of using the proposed algorithms to: *(i)* support branched DECLARE; and *(ii)* consider additional perspectives other than the pure control flow, such as time and data.

Acknowledgements

The work of Chiara Di Francescomarino has been carried out within the Euregio IPN12 KAOS funded by the "European Region Tyrol-South Tyrol-Trentino" (EGTC) under the first call for basic research projects. The work of Claudio Di Ciccio has been funded by the Austrian Research Promotion Agency (FFG) grant 861213 (CitySPIN). The work of Fabrizio Maggi has partly been funded by the Estonian Research Council.

References

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

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

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

[4] H. A. Reijers, T. Slaats, C. Stahl, Declarative modeling—an academic dream or the future for BPM?, in: BPM 2013, Vol. 8094, 2013, pp. 307–322.

[5] M. Pesic, Constraint-Based Workflow Management Systems: Shifting Control to Users, Ph.D. thesis, TU/e (2008).

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

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

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

[8] T. Kala, F. M. Maggi, C. Di Ciccio, C. Di Francescomarino, Apriori and sequence analysis for discovering declarative process models, in: 20th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2016, Vienna, Austria, September 5-9, 2016, 2016, pp. 1–9.

[9] J. Adamo, Data mining for association rules and sequential patterns - sequential and parallel algorithms, Springer New York, 2001. doi:10.1007/978-1-4613-0085-4.

[10] W. M. P. van der Aalst et al., Process mining manifesto, in: BPM 2011 Workshops, Part I, Vol. 99, Springer-Verlag, 2012, pp. 169–194.

[11] M. Montali, M. Pesic, W. M. P. van der Aalst, F. Chesani, P. Mello, S. Storari, Declarative Specification and Verification of Service Choreographies, ACM Transactions on the Web 4 (1).

[12] W. M. P. van der Aalst, M. Pesic, H. Schonenberg, Declarative workflows: Balancing between flexibility and support, Computer Science - R&D 23 (2) (2009) 99–113.

[13] O. Kupferman, M. Y. Vardi, Vacuity detection in temporal model checking., in: CHARME 1999, Vol. 1703, 1999, pp. 82–96.

[14] A. Burattin, F. M. Maggi, W. M. P. van der Aalst, A. Sperduti, Techniques for a posteriori analysis of declarative processes, in: EDOC 2012, 2012, pp. 41–50.

[15] F. M. Maggi, M. Montali, C. Di Ciccio, J. Mendling, Semantical vacuity detection in declarative process mining, in: BPM, 2016, pp. 158–175.

[16] D. M. M. Schunselaar, F. M. Maggi, N. Sidorova, Patterns for a log-based strengthening of declarative compliance models, in: Integrated Formal Methods - 9th International Conference, IFM 2012, Pisa, Italy, June 18-21, 2012. Proceedings, 2012, pp. 327–342. doi:

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

10.1007/978-3-642-30729-4_23.
URL http://dx.doi.org/10.1007/978-3-642-30729-4_23

[17] W. Steeman, Real-life event logs – an incident management process: closed problems, Third International Business Process Intelligence Challenge (BPIC'13) (2013). doi:10.4121/uuid:500573e6-accc-4b0c-9576-aa5468b10cee.
URL http://dx.doi.org/10.4121/uuid:500573e6-accc-4b0c-9576-aa5468b10cee

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

[19] C. Di Ciccio, M. Mecella, J. Mendling, The effect of noise on mined declarative constraints, in: Data-Driven Process Discovery and Analysis, Vol. 203, 2015, pp. 1–24. doi:10.1007/978-3-662-46436-6_1.

[20] F. Mannhardt, Real-life event logs – sepsis cases (2016). doi:10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460.
URL http://dx.doi.org/10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460

[21] B. F. van Dongen, Real-life event logs – a loan application process, Second International Business Process Intelligence Challenge (BPIC'12) (2012). doi:10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f.
URL http://dx.doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f

[22] B. F. van Dongen, Bpi challenge 2017 (2017). doi:10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b.

[23] B. F. van Dongen, Real-life event logs – logs from ITIL processes of rabobank group ICT, Fourth International Business Process Intelligence Challenge (BPIC'14) (2014). doi:10.4121/uuid:c3e5d162-0cfd-4bb0-bd82-af5268819c35.
URL http://dx.doi.org/10.4121/uuid:c3e5d162-0cfd-4bb0-bd82-af5268819c35

[24] M. de Leoni, F. Mannhardt, Road traffic fine management process (2015). doi:10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

[25] F. Mannhardt, D. Blinde, Analyzing the trajectories of patients with sepsis using process mining, in: J. Gulden, S. Nurcan, I. Reinhartz-Berger, W. Guédria, P. Bera, S. Guerreiro, M. Fellmann, M. Weidlich (Eds.), RADAR+EMISA, Vol. 1859 of CEUR Workshop Proceedings, CEUR-ws.org, 2017, pp. 72–80.
URL http://ceur-ws.org/Vol-1859/bpmds-08-paper.pdf

[26] C. Di Ciccio, A. Marrella, A. Russo, Knowledge-intensive Processes: Characteristics, requirements and analysis of contemporary approaches, J. Data Semantics 4 (1) (2015) 29–57. doi:10.1007/s13740-014-0038-4.

[27] D. Fahland, D. Lübke, J. Mendling, H. A. Reijers, B. Weber, M. Weidlich, S. Zugal, Declarative versus imperative process modeling languages: The issue of understandability, in: T. A. Halpin, J. Krogstie, S. Nurcan, E. Proper, R. Schmidt, P. Soffer, R. Ukor (Eds.), BMMDS/EMMSAD, Vol. 29 of Lecture Notes in Business Information Processing, Springer, 2009, pp. 353–366. doi:10.1007/978-3-642-01862-6_29.

[28] E. Bellodi, F. Riguzzi, E. Lamma, Probabilistic declarative process mining, in: KSEM 2010, Vol. 6291, 2010, pp. 292–303.

[29] E. Bellodi, F. Riguzzi, E. Lamma, Probabilistic logic-based process mining, in: CILC2010, Vol. 598, 2010.

[30] F. Chesani, E. Lamma, P. Mello, M. Montali, F. Riguzzi, S. Storari, Exploiting inductive logic programming techniques for declarative process mining, ToPNoC II 5460 (2009) 278–295.

[31] C. Di Ciccio, M. Mecella, Mining artful processes from knowledge workers' emails, IEEE Internet Computing 17 (5) (2013) 10–20. doi:10.1109/MIC.2013.60.

[32] C. Di Ciccio, M. Mecella, On the discovery of declarative control flows for artful processes, ACM Trans. Manage. Inf. Syst. 5 (4) (2015) 24:1–24:37. doi:10.1145/2629447.

[33] C. Di Ciccio, F. M. Maggi, J. Mendling, Discovering target-branched declare constraints, in: Business Process Management - 12th International Conference, BPM 2014, Haifa, Israel, September 7-11, 2014. Proceedings, 2014, pp. 34–50.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

[34] C. Di Ciccio, F. M. Maggi, J. Mendling, Efficient discovery of target-branched declare constraints, Inf. Syst. 56 (2016) 258–283.

[35] M. L. Bernardi, M. Cimitile, C. Di Francescomarino, F. M. Maggi, Using discriminative rule mining to discover declarative process models with non-atomic activities, in: Rules on the Web. From Theory to Applications - 8th International Symposium, RuleML 2014, Co-located with the 21st European Conference on Artificial Intelligence, ECAI 2014, Prague, Czech Republic, August 18-20, 2014. Proceedings, 2014, pp. 281–295.

[36] M. L. Bernardi, M. Cimitile, C. Di Francescomarino, F. M. Maggi, Do activity lifecycles affect the validity of a business rule in a business process?, Inf. Syst. 62 (2016) 42–59.

[37] M. Westergaard, F. M. Maggi, Looking into the future. using timed automata to provide a priori advice about timed declarative process models, in: On the Move to Meaningful Internet Systems: OTM 2012, Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2012, Rome, Italy, September 10-14, 2012. Proceedings, Part I, 2012, pp. 250–267.

[38] F. M. Maggi, M. Westergaard, Using timed automata for *a Priori* warnings and planning for timed declarative process models, Int. J. Cooperative Inf. Syst. 23 (1).

[39] F. M. Maggi, Discovering metric temporal business constraints from event logs, in: Perspectives in Business Informatics Research - 13th International Conference, BIR 2014, Lund, Sweden, September 22-24, 2014. Proceedings, 2014, pp. 261–275.

[40] F. M. Maggi, M. Dumas, L. García-Bañuelos, M. Montali, Discovering data-aware declarative process models from event logs, in: BPM, 2013, pp. 81–96.

[41] S. Schönig, A. Rogge-Solti, C. Cabanillas, S. Jablonski, J. Mendling, Efficient and customisable declarative process mining with SQL, in: Advanced Information Systems Engineering - 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June 13-17, 2016. Proceedings, 2016, pp. 290–305.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

[42] S. Schönig, C. Di Ciccio, F. M. Maggi, J. Mendling, Discovery of multi-perspective declarative process models, in: Service-Oriented Computing - 14th International Conference, ICSOC 2016, Banff, AB, Canada, October 10-13, 2016, Proceedings, 2016, pp. 87–103.

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

[44] A. Burattin, M. Cimitile, F. M. Maggi, A. Sperduti, Online discovery of declarative process models from event streams, IEEE Trans. Services Computing 8 (6) (2015) 833–846.

Pre-print copy of the manuscript published by Elsevier
identified by DOI: 10.1016/j.is.2017.12.002

This document is a pre-print copy of the manuscript
(Maggi et al. 2018)
published by Elsevier.

The final version of the paper is identified by DOI: 10.1016/j.is.2017.12.002

References

Maggi, Fabrizio Maria, Claudio Di Ciccio, Chiara Di Francescomarino, and Taavi Kala (2018). "Parallel algorithms for the automated discovery of declarative process models". In: *Information Systems* 74, pp. 136–152. ISSN: 0306-4379. DOI: 10.1016/j.is.2017.12.002.

BibTeX

```
@Article{         Maggi.etal/IS2018:ParallelAlgorithmsDeclarativeDiscovery,
  author        = {Maggi, Fabrizio Maria and Di Ciccio, Claudio and Di
                   Francescomarino, Chiara and Kala, Taavi},
  title         = {Parallel algorithms for the automated discovery of
                   declarative process models},
  journal       = {Information Systems},
  year          = {2018},
  volume        = {74},
  pages         = {136--152},
  issn          = {0306-4379},
  doi           = {10.1016/j.is.2017.12.002},
  keywords      = {Process mining, Process discovery, Declarative process
                   models, Apriori algorithm, Sequence analysis},
  publisher     = {Elsevier}
}
```