Efficient Discovery of Understandable Declarative Process Models from Event Logs

Fabrizio M. Maggi *, R.P. Jagadeesh Chandra Bose, and Wil M.P. van der Aalst

Eindhoven University of Technology, The Netherlands.
{f.m.maggi, j.c.b.rantham.prabhakara, w.m.p.v.d.aalst}@tue.nl

Abstract. Process mining techniques often reveal that real-life processes are more variable than anticipated. Although declarative process models are more suitable for less structured processes, most discovery techniques generate conventional procedural models. In this paper, we focus on discovering *Declare* models based on event logs. A Declare model is composed of temporal constraints. Despite the suitability of declarative process models for less structured processes, their discovery is far from trivial. Even for smaller processes there are many potential constraints. Moreover, there may be many constraints that are trivially true and that do not characterize the process well. Naively checking all possible constraints is computationally intractable and may lead to models with an excessive number of constraints. Therefore, we have developed an Apriori algorithm to reduce the search space. Moreover, we use new metrics to prune the model. As a result, we can quickly generate understandable Declare models for real-life event logs.

Keywords: process mining, business process management, declarative process models

1 Introduction

The increasing availability of event data recorded by contemporary information systems makes *process mining* a valuable instrument to improve and support business processes [2]. Starting point for process mining 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 *case* (i.e., a *process instance*). The events belonging to a case are *ordered* and can be seen as one "run" of the process (often referred to as a *trace* of events). 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.

Typically, three types of process mining can be distinguished [2]: (a) *process discovery* (learning a model from example traces in an event log), (b) *conformance checking* (comparing the observed behavior in the event log with the

* This research has been carried out as a part of the Poseidon project at Thales under the responsibilities of the Embedded Systems Institute (ESI). The project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

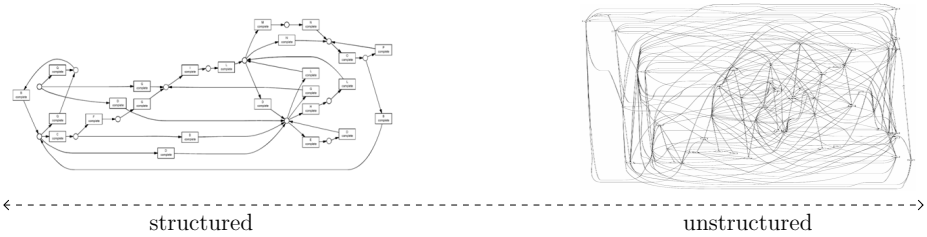


Fig. 1. Procedural models for structured and unstructured processes

modeled behavior), and (c) *model enhancement* (extending models based on additional information in the event logs, e.g., to highlight bottlenecks). In this paper, we focus on process discovery which is generally considered as the most challenging process mining task.

Fig. 1 shows two example models discovered based on two different event logs. The Petri net on the left shows a relatively structured process that is easy to understand. The Spaghetti-like model on the right is less structured and much more difficult to comprehend. In practice, process models are often less structured than anticipated. Therefore, one could argue that procedural models such as the ones depicted in Fig. 1 are less suitable. Nevertheless, almost all process discovery techniques [1,5,6,10,12,14,18] aim to discover procedural model expressed in terms of BPMN, UML activity diagrams, Petri nets, EPCs, and the like. In this paper, we use a different approach and aim to *discover declarative models* from event logs.

Fig. 2. Declare *response* constraint: $\Box(a \rightarrow \Diamond b)$

Fig. 2 shows a *Declare* model [4,13,20] consisting of only one constraint. The arrow connecting activity a to b models a so-called *response* constraint, i.e., activity a is always followed by b. This response constraint is satisfied for traces such as $\langle a, a, b, c\rangle$, $\langle b, b, c, d\rangle$, and $\langle a, b, c, a, b\rangle$, but not for $\langle a, b, a, c\rangle$ because the second a is not followed by b. The semantics of Declare constraints are rooted in Linear Temporal Logic (LTL), e.g., the response constraint can be formalized as $\Box(a \rightarrow \Diamond b)$ and checked or enforced automatically.

A Declare model consists of a set of constraints which, in turn, are based on *templates*. A template defines a particular type of constraint (like "response"). Templates have formal semantics specified through LTL formulas and are equipped with a user-friendly graphical front-end that makes the language easy to understand also for users that are not familiar with LTL [19,23]. Templates are parameterized, e.g., the response constraint in Fig. 2 is instantiated for activities a

and b. Hence, in a process model with n events there are n^2 potential response constraints.

In [17], the authors present a technique to automatically infer Declare constraints. This technique exhaustively generates all possible constraints and then checks them on the event log. This approach suffers from two important drawbacks:

- First of all, such an exhaustive approach is intractable for processes with dozens of activities. For example, in a process with 30 activities there are already 900 possible response constraints. Some of the other templates have four parameters, resulting in $30^4 = 810,000$ potential constraints for a single template. Since there are dozens of templates in the standard Declare language, this implies that the log needs to be traversed millions of times to check all potential constraints. As event logs may contain thousands or even millions of events, this is infeasible in practice.
- Second, of the millions of potential constraints, many may be trivially true. For example, the response constraint in Fig. 2 holds for any event log that does not contain events relating to activity a. Moreover, one constraint may dominate another constraint. If the stronger constraint holds (e.g., $\Box(a \rightarrow \Diamond b)$), then automatically the weaker constraint (e.g., $\Diamond a \rightarrow \Diamond b$) also holds. Showing all constraints that hold typically results in unreadable models.

This paper addresses these two problems using a two-phase approach. In the first phase, we generate the list of candidate constraints by using an Apriori algorithm. This algorithm is inspired by the seminal Apriori algorithm developed by Agrawal and Srikant for mining association rules [7]. The Apriori algorithm uses the monotonicity property that all subsets of a frequent item-set are also frequent. In the context of this paper, this means that sets of activities can only be frequent if all of their subsets are frequent. This observation can be used to dramatically reduce the number of interesting candidate constraints. In the second phase, we further prune the list of candidate constraints by considering only the ones that are relevant (based on the event log) according to (the combination of) simple metrics, such as *Confidence* and *Support*, and more sophisticated metrics, such as *Interest Factor* (IF) and *Conditional-Probability Increment Ratio* (CPIR), as explained in Section 4. Moreover, discovered constraints with high CPIR values are emphasized like highways on a roadmap whereas constraints with low CPIR values are greyed out. This further improves the readability of discovered Declare models.

The paper is structured as follows. Section 2 introduces the Declare formalism using a running example. Section 3 describes how an Apriori algorithm can be applied to generate a list of candidate constraints. Section 4 explains how metrics proposed in the literature on association rule mining can be used to evaluate the relevance of a Declare constraint. Section 5 presents experimental results comparing our new discovery algorithm with the naive approach presented in [17]. Section 6 provides an illustrative case study. Section 7 concludes the paper.

2 Declare

In this paper, we present an approach to efficiently discover understandable Declare models. Therefore, we first introduce the Declare language [4,13,20] which is grounded in Linear Temporal Logic (LTL).

LTL can be used to specify constraints on the ordering of activities (see also [11]). For instance, a constraint like "whenever activity a is executed, eventually activity b is executed" can be formally represented using LTL and, in particular, it can be written as $\Box(a \rightarrow \Diamond b)$. In a formula like this, it is possible to find traditional logical operators (e.g., implication $\rightarrow$), but also temporal operators characteristic of LTL (e.g., always $\Box$, and eventually $\Diamond$). In general, using the LTL language it is possible to express constraints relating activities (atoms) through logical operators or temporal operators. The logical operators are: implication ($\rightarrow$), conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$). The main temporal operators are: *always* ($\Box p$, in every future state p holds), *eventually* ($\Diamond p$, in some future state p holds), *next* ($\bigcirc p$, in the next state p holds), and *until* ($p \sqcup q$, p holds until q holds).

LTL constraints are not very readable for non-experts. Declare [4,13,20] provides an intuitive graphical front-end together with a formal LTL back-end. In Fig. 2 we already showed the graphical Declare representation for the *response* constraint. There are dozens of different Declare constraints possible. Each type of constraint is described using a parameterized template. Besides the response constraint, we have constraints such as *responded existence* (formally: $\Diamond a \rightarrow \Diamond b$) and *precedence* (formally: $(\neg b \sqcup a) \vee \Box(\neg b)$).

2.1 Running Example

Fig. 3 shows a simple Declare model with some example constraints for an insurance claim process. The model includes eight activities (depicted as rectangles, e.g., *Contact Hospital*) and five constraints (shown as connectors between the activities, e.g., *co-existence*).

The *responded existence* constraint specifies that if *High Medical History* is executed also *High Insurance Check* is executed in the same process instance. The *precedence* constraint indicates that, if *Receive Questionnaire Response* is executed, *Send Questionnaire* must be executed before. In contrast, the *response*

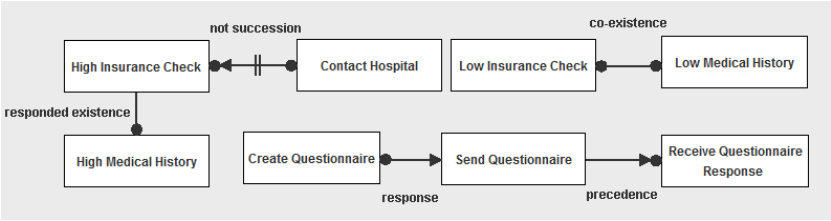


Fig. 3. Running example: Declare model consisting of five constraints

constraint indicates that if *Create Questionnaire* is executed this is eventually followed by *Send Questionnaire*. The *not succession* constraint means that *Contact Hospital* cannot be followed by *High Insurance Check*. Finally, the *co-existence* constraint indicates that if *Low Insurance Check* and *Low Medical History* occur in a process instance, they always coexist. We refer the reader to [4,13,20] for more details about the Declare language (graphical notation and LTL semantics).

2.2 Discovering Relevant Declare Constraints Using Vacuity Detection

As shown in [3], LTL constraints can be checked for a particular trace and therefore also for an entire event log. For example, one may find that responded existence constraint between *High Medical History* and *High Insurance Check* holds for 955 of 988 insurance claims. Hence, Declare models can be discovered by simply checking all possible constraints as shown in [17]. First, all possible constraints need to be constructed. Since there is a finite number of constraint templates and in a given setting there is also a finite set of activities, this is always possible. Then, the set of potential constraints can be pruned on the basis of simple metrics such as the percentage of process instances where the constraint holds, e.g., keep all constraints satisfied in at least 90% of cases.

Such an approach will result in the discovery of many constraints that are trivially valid. Consider for example a process instance $\langle a, a, b, a, b, a \rangle$. The constraint $\Box(c \rightarrow \Diamond d)$ ("whenever activity c is executed, eventually activity d is executed") holds. This constraint holds trivially because c never happens. Using the terminology introduced in [8,15], we say that the constraint is *vacuously satisfied*. In general, a formula φ is *vacuously satisfied* in a path π, if π satisfies φ and there is some sub-formula of φ that does not affect the truth value of φ in π [8]. In our example, the first term of the implication $\Box(c \rightarrow \Diamond d)$ is always false. Therefore, sub-formulas $\Diamond d$ and d do not affect the truth value of $\Box(c \rightarrow \Diamond d)$ in $\langle a, a, b, a, b, a \rangle$.

To address this issue, in [17], the authors use techniques for *LTL vacuity detection* [8,15] to discriminate between instances where a constraint is generically non-violated and instances where the constraint is non-vacuously satisfied. Only process instances where a constraint is non-vacuously satisfied are considered *interesting witnesses* for that constraint. Roughly speaking, to ensure that a process instance is an interesting witness for a constraint, it is necessary to check the validity of the constraint in the process instance with some extra conditions. The authors in [15] introduce an algorithm to compute these extra conditions. For example, for the constraint $\Box(c \rightarrow \Diamond d)$ the condition is $\Diamond c$. This means that the constraint is non-vacuously satisfied in all the process instances where "whenever activity c is executed, eventually activity d is executed" and "eventually activity c is executed". The percentage of interesting witnesses for a constraint is a much better selection mechanism than the percentage of instances for which the constraint holds.

In [21], the authors extend the list of vacuity detection conditions to ensure that if a process instance is an interesting witness for a Declare constraint, no stronger constraint holds in that process instance. Table 1 specifies the list of vacuity detection conditions for some types of Declare constraints. For instance, a response constraint is non-vacuously satisfied in all the process instances where it is satisfied ($\Box(a \rightarrow \Diamond b)$), the vacuity detection condition derived from [15] is valid ($\Diamond a$) and, in addition, constraints that are stronger than response (i.e., succession and alternate response) do not hold ($\neg((\neg b \sqcup a) \vee \Box(\neg b)) \wedge \neg(\Box(a \rightarrow \bigcirc(\neg a \sqcup b)))$). In this paper, we refer to this extended notion of vacuity detection.

For completeness we also mention the approach described in [16]. This approach also learns Declare models, but requires negative examples. This implies that everything that did not happen is assumed not to be possible. We consider this an unrealistic assumption as logs only contain example behavior.

3 Apriori Algorithm for Declare Discovery

Vacuity detection can be used to prune set of constraints, but this can only be done after generating a set of candidate constraints. As discussed in the introduction, even for smaller processes, there can be millions of potential constraints. Therefore, we adopt ideas from the well-known Apriori algorithm [7] for discovering association rules. Using an Apriori-like approach we can efficiently discover frequent sets of correlated activities in an event log.

Let Σ be the set of potential activities. Let $\mathbf{t} \in \Sigma^*$ be a trace over Σ, i.e., a sequence of activities executed for some process instance. An event log $\mathcal{L}$ is a multi-set over Σ^*, i.e., 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 (Support). *The support of an activity set $A \subseteq \Sigma$ in an event log $\mathcal{L} = [\boldsymbol{t}_1, \boldsymbol{t}_2, \ldots, \boldsymbol{t}_n]$ is the fraction of process instances in $\mathcal{L}$ that contain all of the activities in A, i.e.,*

$$supp(A) = \frac{|\mathcal{L}_A|}{|\mathcal{L}|}, \text{ where } \mathcal{L}_A = [\boldsymbol{t} \in \mathcal{L} \mid \forall_{\boldsymbol{x} \in A}\ \boldsymbol{x} \in \boldsymbol{t}]$$

An activity set is considered to be *frequent* if its support is above a given threshold $supp_{\min}$. Let $\mathcal{A}_k$ denote the set of all frequent activity sets of size $k \in \mathbb{N}$ and let $\mathcal{C}_k$ denote the set of all *candidate activity sets* of size k that may potentially be frequent. The Apriori algorithm uncovers all frequent activity sets in an event log. The algorithm starts by considering activity sets of size 1 and progresses iteratively by considering activity sets of increasing sizes in each iteration and is based on the property *that any subset of a frequent activity set must be frequent.* The set of candidate activity sets of size $k+1$, $\mathcal{C}_{k+1}$, is generated by joining relevant frequent activity sets from $\mathcal{A}_k$. This set can be pruned efficiently using the property that a relevant candidate activity set of size $k+1$ cannot have an infrequent subset. The activity sets in $\mathcal{C}_{k+1}$ that have a support above a given

Table 1. Vacuity detection conditions for some Declare constraints

template	*LTL semantics*	*vacuity detection conditions*
responded existence	$\Diamond a \rightarrow \Diamond b$	$\Diamond a \ \wedge \ \neg(\Box(a \rightarrow \Diamond b)) \ \wedge \ \neg(\Diamond a \leftrightarrow \Diamond b)$
co-existence	$\Diamond a \leftrightarrow \Diamond b$	$\Diamond a \ \wedge \ \Diamond b \ \wedge \ \neg(\Box(a \rightarrow \Diamond b) \wedge (\neg b \sqcup a) \vee \Box(\neg b))$
response	$\Box(a \rightarrow \Diamond b)$	$\Diamond a \ \wedge \ \neg((\neg b \sqcup a) \vee \Box(\neg b)) \ \wedge \ \neg(\Box(a \rightarrow \bigcirc(\neg a \sqcup b)))$
precedence	$(\neg b \sqcup a) \vee \Box(\neg b)$	$\Diamond b \ \wedge \ \neg a \ \wedge \neg(\Box(a \rightarrow \Diamond b)) \wedge$ $\neg(((\neg b \sqcup a) \vee \Box(\neg b)) \wedge \Box(b \rightarrow \bigcirc((\neg b \sqcup a) \vee \Box(\neg b))))$
not succession	$\Box(a \rightarrow \neg(\Diamond b))$	$\Diamond a \ \wedge \ \Diamond b$

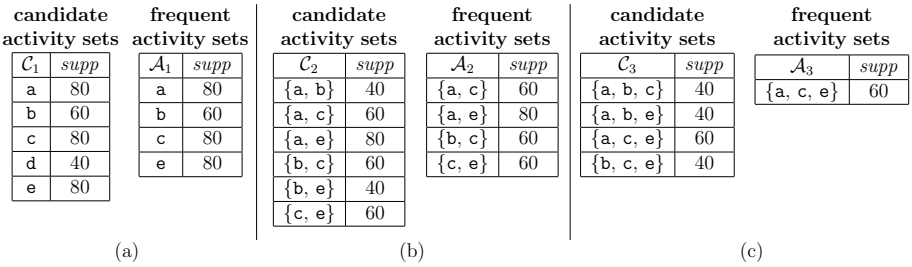
threshold $supp_{\min}$ constitute the frequent activity sets of size $k+1$ ($\mathcal{A}_{k+1}$) used in the next iteration.

We explain the Apriori algorithm with an example. Consider an event log $\mathcal{L} = [\langle e, a, b, a, a, c, e\rangle, \langle e, a, a, b, c, e\rangle, \langle e, a, a, d, d, e\rangle, \langle b, b, c, c\rangle, \langle e, a, a, c, d, e\rangle]$ defined over the set of activities $\Sigma = \{a, b, c, d, e\}$. Let us try to find frequent activity sets whose support is above 50%. The Apriori algorithm starts by first considering activity sets of size 1, i.e., the individual activities. The candidate sets in $\mathcal{C}_1$ correspond to the singletons of the elements of Σ. Fig. 4(a) depicts the candidate activity sets and their support. Among the candidate sets, activity d has a support of only 40% in the event log, which is below the specified threshold. Therefore, the frequent activity sets correspond to the singletons of the elements of $\Sigma \setminus \{d\}$. In the next iteration, the Apriori algorithm considers candidate activity sets of size 2, $\mathcal{C}_2$. Since the support of d is less than the specified threshold, all activity sets that involve d are bound to have their support less than the threshold. The Apriori algorithm elegantly captures this by deriving candidates at iteration $k+1$, $\mathcal{C}_{k+1}$, from frequent activity sets of iteration k. Fig. 4(b) depicts such candidate activity sets of size 2 along with their support values. Only 4 activity sets among $\mathcal{C}_2$ satisfy the minimum support criteria and hence are considered frequent (see $\mathcal{A}_2$ in Fig. 4(b)). Proceeding further, we get only one frequent activity set of size 3 as depicted in Fig. 4(c). The algorithm terminates after this step as no further candidates can be generated. The frequent activity sets in $\mathcal{L}$ are $\mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3$.

The Apriori algorithm returns frequent activity sets, which indicate that the activities involved in an activity set are correlated. However, it doesn't specify the type of correlation. Declare templates capture different relationships between activities. For instance, for any frequent activity set $\{a, b\}$, one can generate constraints such as the response $\Box(a \rightarrow \Diamond b)$.

In general, to discover constraints deriving from a Declare template with k parameters, we have to generate frequent activity sets of size k. Afterwards, we generate the list of candidate constraints. To do that, we instantiate the considered template by specifying as parameters all the possible permutations of each frequent set. For instance, for the frequent activity set $\{a, b\}$, we generate the response constraints $\Box(a \rightarrow \Diamond b)$ and $\Box(b \rightarrow \Diamond a)$.

Limiting ourselves to frequent activity sets drastically reduces the number of candidate constraints to be checked. For instance, if we take the example in Fig. 4(b), to generate the candidate constraints deriving from a template with



$\mathcal{C}_1$	*supp*
a	80
b	60
c	80
d	40
e	80

$\mathcal{A}_1$	*supp*
a	80
b	60
c	80
e	80

$\mathcal{C}_2$	*supp*
{a, b}	40
{a, c}	60
{a, e}	80
{b, c}	60
{b, e}	40
{c, e}	60

$\mathcal{A}_2$	*supp*
{a, c}	60
{a, e}	80
{b, c}	60
{c, e}	60

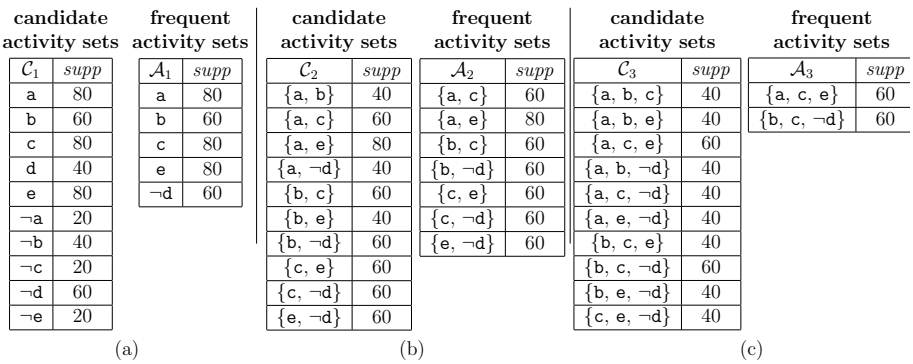
$\mathcal{C}_3$	*supp*
{a, b, c}	40
{a, b, e}	40
{a, c, e}	60
{b, c, e}	40

$\mathcal{A}_3$	*supp*
{a, c, e}	60

Fig. 4. Discovering frequent activity sets using the Apriori algorithm in the event log $\mathcal{L}$. The support values are expressed in %

2 parameters we consider all the permutations of each frequent activity set in $\mathcal{A}_2$ and we generate 12 candidate constraint (we also include pairs (a, a), (b, b), (c, c), (e, e) by considering repetitions of elements of frequent activity sets of size 1). In contrast, with the naive approach we need to consider all the dispositions of length 2 of 5 activities (a, b, c, d, e), i.e., 25 (5^2) candidate constraints. In general, to discover constraints deriving from a Declare template with k parameters from a log with n activities, the state space exploration using the naive approach always leads to n^k candidate constraints. In contrast, applying our Apriori algorithm and considering only the frequent activity sets, the number of candidate constraints depends on the support value we specify for the Apriori algorithm. This number is often significantly lower than n^k because we ignore the item sets with low support.

One could also look at negative events (non-occurrence) within the Apriori setup. Such information might be useful for inferring, for instance, events that act as mutually exclusive, e.g., if a occurs then b does not occur. To facilitate this, we can also consider the negative events $\neg a$, for all $a \in \Sigma$. Fig. 5 depicts the discovery of frequent items sets considering non-occurrence of events using the



$\mathcal{C}_1$	*supp*
a	80
b	60
c	80
d	40
e	80
¬a	20
¬b	40
¬c	20
¬d	60
¬e	20

$\mathcal{A}_1$	*supp*
a	80
b	60
c	80
e	80
¬d	60

$\mathcal{C}_2$	*supp*
{a, b}	40
{a, c}	60
{a, e}	80
{a, ¬d}	40
{b, c}	60
{b, e}	40
{b, ¬d}	60
{c, e}	60
{c, ¬d}	60
{e, ¬d}	60

$\mathcal{A}_2$	*supp*
{a, c}	60
{a, e}	80
{b, c}	60
{b, ¬d}	60
{c, e}	60
{c, ¬d}	60
{e, ¬d}	60

$\mathcal{C}_3$	*supp*
{a, b, c}	40
{a, b, e}	40
{a, c, e}	60
{a, b, ¬d}	40
{a, c, ¬d}	40
{a, e, ¬d}	40
{b, c, e}	40
{b, c, ¬d}	60
{b, e, ¬d}	40
{c, e, ¬d}	40

$\mathcal{A}_3$	*supp*
{a, c, e}	60
{b, c, ¬d}	60

Fig. 5. Discovering frequent activity sets considering negative (non-occurrence) events using the Apriori algorithm in the event log $\mathcal{L}$. The support values are expressed in %

Table 2. Association rule formulation for some Declare constraints

template	*LTL semantics*	*rule*	*antecedent*	*consequent*
responded existence	$\Diamond a \rightarrow \Diamond b$	If a occurs then b occurs	a	b
co-existence	$\Diamond a \leftrightarrow \Diamond b$	If a occurs then b occurs $\wedge$ If b occurs then a occurs	$(a \vee b)$	$(a \vee b)$
response	$\Box(a \rightarrow \Diamond b)$	If a occurs then b follows	a	b
precedence	$(\neg b \sqcup a) \vee \Box(\neg b)$	If b occurs then a precedes	b	a
not succession	$\Box(a \rightarrow \neg(\Diamond b))$	If a occurs then b does not follow	a	b

apriori algorithm on the event log $\mathcal{L}$. Note that we now have additional frequent activity sets such as $\{b, c, \neg d\}$ signifying that if b and c occurs then d does not occur.

4 Post-Processing

The set of frequent activity sets helps in reducing the preliminary set of constraints that one uncovers. In particular, the candidate constraints generated from frequent sets all involve activities that occur frequently in the log. However, this does not mean that these constraints are also frequently (non-vacuously) satisfied in the log. Let us consider, for example, $\{a, e\}$, a frequent activity set for the event log $\mathcal{L}$ mentioned earlier. One can define various Declare constraints involving these two activities, e.g., the response constraints $\Box(a \rightarrow \Diamond e)$ and $\Box(e \rightarrow \Diamond a)$. However, only the former constraint holds for all cases in $\mathcal{L}$ whereas the latter constraint is only satisfied in one of the five cases. Even more, according to the vacuity detection conditions illustrated in Table 1, $\Box(a \rightarrow \Diamond e)$ is non-vacuously satisfied in four cases, whereas $\Box(e \rightarrow \Diamond a)$ is never non-vacuously satisfied. Obviously, there is a need to further prune constraints that are less relevant, i.e., non-vacuously satisfied in a lower percentage of process instances than others. We can consider a Declare constraint as a rule or as a conjunction/disjunction of rules, e.g., $\Box(a \rightarrow \Diamond e)$ can be thought of as the rule "If a is executed, then e follows". A rule is comprised of two components, the *antecedent* part and the *consequent* part. Table 2 depicts the interpretation of Declare constraints as association rules. We can adopt various metrics proposed in the association rule mining literature to evaluate the relevance of a rule and thereby the Declare constraints. We use four metrics, i.e., *support*, *confidence*, *interest factor*, and *CPIR*. The latter three metrics are defined on the primitive measure of *support* of the antecedent, consequent, and the rule.

Definition 2 (Support (of a Declare constraint)). *The support of a Declare constraint in an event log $\mathcal{L}$ is defined as the fraction of process instances in which the constraint is non-vacuously satisfied, i.e., the percentage of interesting witnesses for that constraint in the log.*

Definition 3 (Confidence). *The confidence of a Declare constraint expressed as an association rule in an event log $\mathcal{L}$ is the ratio between the support of the*

rule and the support of the antecedent

$$conf(rule) = \frac{supp(rule)}{supp(antecedent)}$$

For the above event log $\mathcal{L}$, the support of the constraint $\Box(a \rightarrow \Diamond e)$ is 0.8 and the confidence of the constraint is 1. One can use the support and confidence metrics to prune constraints, e.g., consider only those constraints whose support is above a minimum support threshold and/or whose confidence is above a minimum confidence threshold. Confidence measures might be misleading in scenarios where the support of either the antecedent or the consequent is 1. On a general note, frequent activity sets involving an activity whose support is 1 do not reflect the real correlation between the activities. Brin et al. [9] have proposed a measure called *interest factor* to deal with such scenarios. A stronger dependency between the antecedent and the consequent is associated with a value of this measure that is further from 1.

Definition 4 (Interest Factor). *The interest factor of a Declare constraint expressed as an association rule in an event log $\mathcal{L}$ is the ratio between the support of the rule and the product of the support of the antecedent and the consequent.*

$$InterestFactor(rule) = \frac{supp(rule)}{supp(antecedent)supp(consequent)}$$

Wu et al. [22] have proposed a *conditional-probability increment ratio* (CPIR) measure that assesses whether two entities (in our case activities) a and b are positively or negatively related.

Definition 5 (CPIR). *The CPIR measure of a Declare constraint expressed as an association rule in an event log $\mathcal{L}$ is defined as*

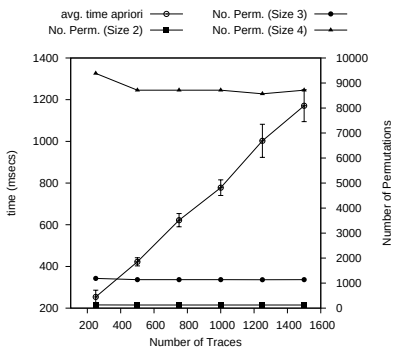
$$CPIR(rule) = \frac{supp(rule) - supp(antecedent)supp(consequent)}{supp(antecedent)(1 - supp(consequent))}$$

CPIR can be used as a measure of confidence of a constraint and can be used to mine negative constraints as well. If $CPIR$(rule) involving the activities a and b is greater than some threshold, then the constraint defined over a and b is a positive constraint of interest. If $CPIR$(rule) involving the activities a and b is negative then a and b are negatively related or in other words a and $\neg b$ are positively related.

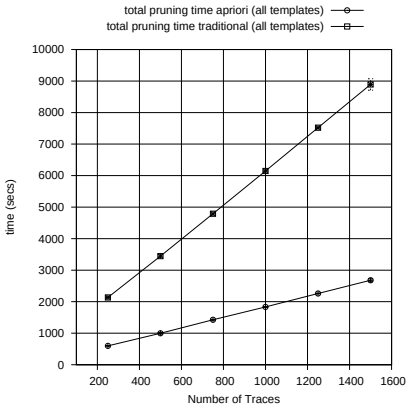
5 Experiments and Discussion

To analyze the performance of our approach, we have simulated multiple event logs of the insurance claim example with varying numbers of cases/events.

In the *first phase* of our approach, we generate a set of candidate constraints. For a Declare template with k parameters, we use the Apriori algorithm to generate frequent sets of size k. Then, we generate all permutations of each



(a) Average computation time needed for the generation of the frequent sets (with support of at least 0.4) and their permutations for varying sizes of the event log; numbers of permutations generated for frequent sets of sizes 2, 3 and 4



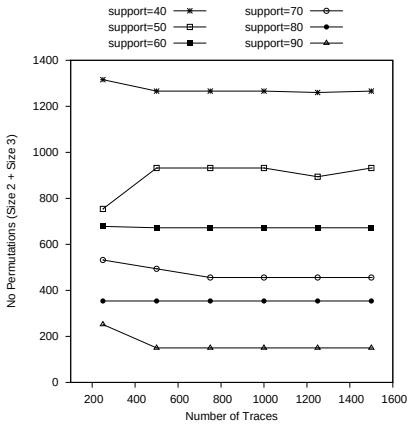
(b) Average computation time for pruning needed for the pruning phase in the naive approach and in the Apriori approach for varying sizes of the event log

Fig. 6. Experimental results for the insurance claim process

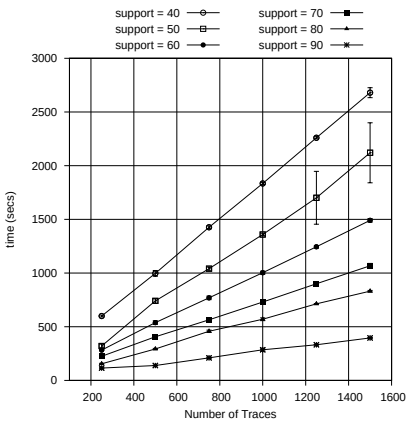
frequent set. Fig. 6(a) shows the average computation time along with the 95% confidence interval (over five independent runs) required for this first phase[1]. We generated frequent sets of different sizes (2, 3 and 4) and with a support of (at least) 0.4. We can see that the time required varies linearly with the size of the log. Fig. 6(a) also depicts the number of permutations generated. As expected, the number of permutations is close to constant and is not significantly influenced by the size of the event log. Moreover, these results show that we need to generate (on average) 126 candidate constraints to discover constraints deriving from a Declare template with 2 parameters, (on average) 1,140 candidate constraints to discover constraints deriving from a Declare template with 3 parameters, and (on average) 8,800 candidate constraints to discover constraints deriving from a Declare template with 4 parameters. Considering that the number of activities in each of the considered logs is 15, these numbers are small compared to the naive approach where 15^2 (225), 15^3 (3,375) and 15^4 (50,625) candidate constraints would be generated for constraints with 2, 3 and 4 parameters respectively.

In the *second phase* of our approach, we prune the set of candidate constraints using the metrics described in Section 4. Fig. 6(b) shows the average computation time required for pruning for the different event logs (with a 95% confidence interval computed over five independent runs). We compare the time needed for the pruning phase using the naive approach and the Apriori-based approach. In

[1] All the computation times reported in this section are measured on a Core 2 Quad CPU @2.40 GHZ 3 GB RAM.



(a) Number of permutations generated for varying sizes of the event log and for different support values



(b) Average computation time for the pruning phase for varying sizes of the event log and for different support values

Fig. 7. Analysis of the Apriori-based approach

both cases, the time required varies linearly with respect to the size of the log. However, the Apriori-based approach clearly outperforms the naive approach. The computation time for the Apriori algorithm itself is negligible (less than 0.05% of the computation time needed for pruning).

Fig. 7(a) shows the number of permutations generated for varying sizes of the event log and for different support values for the Apriori algorithm (again with 95% confidence intervals over five independent runs). As expected, the number of frequent sets does not depend on the number of process instances in the log but on the support value used by the Apriori algorithm. The number of permutations clearly increases when the support value increases. Fig. 7(b) shows the average computation time in relation to the support and number of traces. The time required varies linearly with respect to the size of the log. However, the gradient of the lines increases when the support increases.

The Declare models obtained after pruning using the naive approach and the approach based on the Apriori algorithm are always the same but the latter is more efficient.

Table 3 shows a set of four constraints discovered from one of our synthetic logs (with 250 process instances). These constraints have been discovered using a minimum value for support (0.4), a minimum value for confidence (0.9), and a minimum value for CPIR (0.03). It is important to highlight that all these metrics are important when selecting relevant constraints. Consider, for instance, the two responded existence constraints in the table. Both constraints have a confidence close to 1, but the responded existence between *Send Notification by Phone* and *Receive Questionnaire Response* has CPIR equal to 0.035, whereas the responded existence between *High Medical History* and *High Insurance Check* has CPIR

Table 3. Constraints discovered from a synthetic logs with 250 process instances

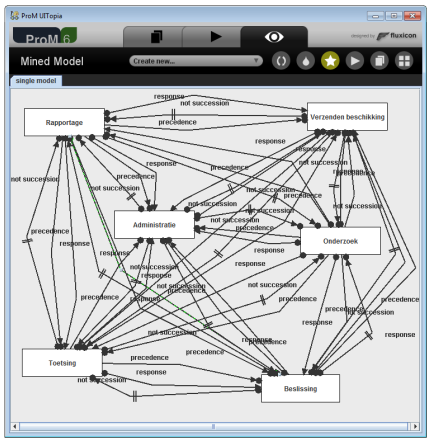
template	*LTL semantics*	*parameter a*	*parameter b*	*support*	*confidence*	*interest factor*	*CPIR*
response	$\Box(a \rightarrow \Diamond b)$	Contact Hospital	Receive Questionnaire Response	0.400	0.925	1.028	0.259
not succession	$\Box(a \rightarrow \neg(\Diamond b))$	Contact Hospital	High Insurance Check	0.432	1.000	1.633	1.000
responded existence	$\Diamond a \rightarrow \Diamond b$	Send Notification by Phone	Receive Questionnaire Response	0.712	0.903	1.000	0.035
responded existence	$\Diamond a \rightarrow \Diamond b$	High Medical History	High Insurance Check	0.496	1.000	1.633	0.999

equal to 0.999. This is due to the fact that the existence of *High Medical History* is strongly related to the existence of *High Insurance Check* (the former can only be executed if the latter has been executed), whereas in the other responded existence constraint the connection between *Send Notification by Phone* and *Receive Questionnaire Response* is less relevant.

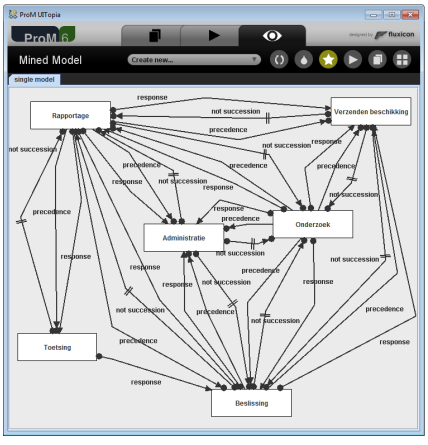
6 Case Study

After showing experimental results focusing on the performance of our new approach to discover Declare models, we now demonstrate its applicability using an event log provided by a Dutch municipality. The log contains events related to requests for excerpt from the civil registration. The event log contains 3,760 cases and 19,060 events. There are 26 different activities.

Fig. 8(a) depicts the Declare model discovered using the naive approach showing all constraints with at least 30% of interesting witnesses in the log. The resulting Spaghetti-like model has 45 constraints. The computation time to



(a) 30% of interesting witnesses



(b) 50% of interesting witnesses

Fig. 8. Declare models discovered using the naive approach

generate this model is 1,711 seconds. This model can be improved by increasing the percentage of required interesting witnesses. Fig. 8(b) depicts the resulting Declare model using the naive approach including only constraints with at least 50% of interesting witnesses. This model has 33 constraints and the computation time needed to generate it is 1,743 seconds. Note that the computational time needed to discover the Declare models in Fig. 8 is approximatively the same because in both cases 2,028 ($3 \cdot 26^2$) constraints must be checked (we search for three types of constraints with 2 parameters, i.e., precedence, response and not succession).

Fig. 9 shows the results obtained using our new approach based on the Apriori algorithm and the pruning techniques described in this paper. This model is

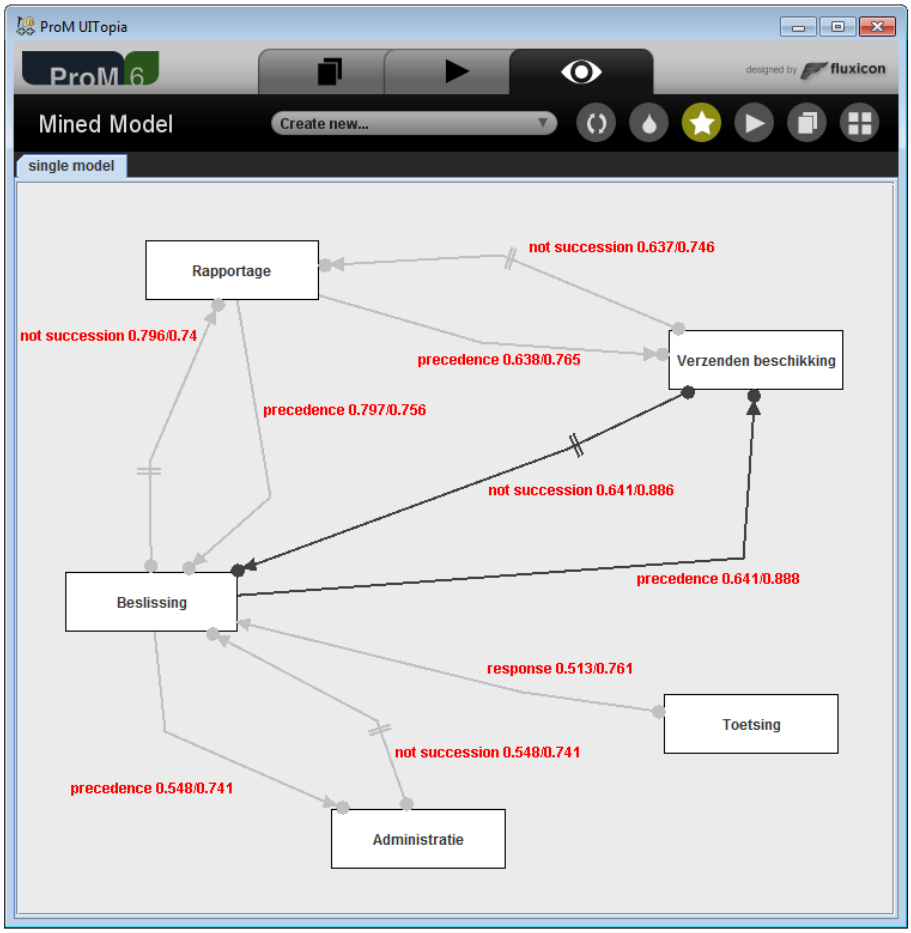


Fig. 9. Declare model discovered using the new approach. Note that the most important constraints are highlighted

composed of 9 constraints and the computation time needed to generate it is 76 seconds. The model contains only constraints with a support greater than 0.5 and CPIR value greater than 0.7.

Moreover, the discovered model emphasizes the most important constraints, just like highways are highlighted on a roadmap. Constraints with a CPIR value of at least 0.85 (considered more relevant) are indicated in black, whereas the constraints with CPIR less than 0.85 (less relevant) are indicated in gray. Each constraint is annotated with support and CPIR values (in red). The graphical feedback facilitates the interpretation of the discovered Declare model.

We evaluate the support of a Declare constraint on the basis of the vacuity detection conditions in Table 1. These conditions guarantee that a process instance is an interesting witness for a constraint if no stronger constraint holds in the same instance. This means that if two constraints hold in the log and one of them is stronger than the other, our approach will discover the stronger one.

7 Conclusion

Although real-life processes discovered through process mining are often Spaghetti-like, lion's share of process discovery algorithms try to construct a procedural model (e.g., BPMN models, EPCs, Petri nets, or UML activity diagrams). The resulting models are often difficult to interpret. Therefore, it is interesting to discover declarative process models instead.

In this paper, we present an approach to efficiently discover Declare models that are understandable. Unlike earlier approaches we do not generate all possible constraints and only check the most promising ones using an Apriori algorithm. This results in dramatic performance improvements. We also defined several criteria to evaluate the relevance of a discovered constraint. The discovered model is pruned using these criteria and the most interesting constraints are highlighted. As demonstrated using a case study, this results in understandable process models.

References

1. van der Aalst, W., Weijters, T., Maruster, L.: Workflow mining: discovering process models from event logs. Knowledge and Data Engineering pp. 1128–1142 (2004)
2. van der Aalst, W.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer-Verlag (2011)
3. van der Aalst, W., de Beer, H., van Dongen, B.: Process Mining and Verification of Properties: An Approach based on Temporal Logic. In: CoopIS 2005. pp. 130–147
4. van der Aalst, W., Pesic, M., Schonenberg, H.: Declarative Workflows: Balancing Between Flexibility and Support. Computer Science - R&D pp. 99–113 (2009)
5. van der Aalst, W., Reijers, H., Weijters, A., van Dongen, B., de Medeiros, A.A., Song, M., Verbeek, H.: Business Process Mining: An Industrial Application. Information Systems pp. 713–732 (2007)
6. Agrawal, R., Gunopulos, D., Leymann, F.: Mining Process Models from Workflow Logs. In: EDBT 1998. pp. 469–483

7. Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules. In: VLDB 1994. pp. 487–499
8. Beer, I., Eisner, C.: Efficient detection of vacuity in temporal model checking. In: Formal Methods in System Design. pp. 200–1 (2001)
9. Brin, S., Motwani, R., Silverstein, C.: Beyond Market Baskets: Generalizing Association Rules to Correlations. In: ACM SIGMOD 1997. pp. 265–276
10. Cook, J.E., Wolf, A.L.: Discovering models of software processes from event-based data. ACM Trans. on Software Engineering and Methodology pp. 215–249 (1998)
11. Damaggio, E., Deutsch, A., Hull, R., Vianu, V.: Automatic verification of data-centric business processes. In: BPM. pp. 3–16 (2011)
12. Datta, A.: Automating the Discovery of As-Is Business Process Models: Probabilistic and Algorithmic Approaches. Info. Systems Research pp. 275–301 (1998)
13. Declare (2008), http://declare.sf.net
14. Günther, C.W., van der Aalst, W.: Fuzzy Mining: Adaptive Process Simplification Based on Multi-perspective Metrics. In: BPM 2007. pp. 328–343 (2007)
15. Kupferman, O., Vardi, M.Y.: Vacuity Detection in Temporal Model Checking. International Journal on Software Tools for Technology Transfer pp. 224–233 (2003)
16. Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Inducing Declarative Logic-Based Models from Labeled Traces. In: BPM 2007. pp. 344–359 (2007)
17. Maggi, F.M., Mooij, A.J., van der Aalst, W.M.P.: User-guided discovery of declarative process models. In: CIDM 2011
18. de Medeiros, A.A., Weijters, A., van de Aalst, W.: Genetic Process Mining: An Experimental Evaluation. Data Mining and Knowledge Discovery pp. 245–304 (2007)
19. Pesic, M.: Constraint-Based Workflow Management Systems: Shifting Controls to Users. Ph.D. thesis, Beta Research School for Operations Management and Logistics, Eindhoven (2008)
20. Pesic, M., Schonenberg, H., van der Aalst, W.: DECLARE: Full Support for Loosely-Structured Processes. In: EDOC 2007. pp. 287–298
21. Schunselaar, D.M.M., Maggi, F.M., Sidorova, N.: Patterns for a Log-Based Strengthening of Declarative Compliance Models. In: iFM 2012 (2012), to appear
22. Wu, X., Zhang, C., Zhang, S.: Efficient Mining of Both Positive and Negative Association Rules. ACM Transactions on Information Systems pp. 381–405 (2004)
23. Zugal, S., Pinggera, J., Weber, B.: The impact of testcases on the maintainability of declarative process models. In: BMMDS/EMMSAD. pp. 163–177 (2011)