Efficient Discovery of Target-Branched Declare Constraints

Claudio Di Ciccio[a,*], Fabrizio Maria Maggi[b], Jan Mendling[a]

[a] *Vienna University of Business and Economics, Austria*
[b] *University of Tartu, Estonia*

Abstract

Process discovery is the task of generating process models from event logs. Mining processes that operate in an environment of high variability is an ongoing research challenge because various algorithms tend to produce spaghetti-like process models. This is particularly the case when procedural models are generated. A promising direction to tackle this challenge is the usage of declarative process modelling languages like Declare, which summarise complex behaviour in a compact set of behavioural constraints on activities. A Declare constraint is branched when one of its parameters is the disjunction of two or more activities. For example, branched Declare can be used to express rules like "in a bank, a mortgage application is always eventually followed by a notification to the applicant by phone *or* by a notification by e-mail". However, branched Declare constraints are expensive to be discovered. In addition, it is often the case that hundreds of branched Declare constraints are valid for the same log, thus making, again, the discovery results unreadable. In this paper, we address these problems from a theoretical angle. More specifically, we define the class of Target-Branched Declare constraints and investigate the formal properties it exhibits. Furthermore, we present a technique for the efficient discovery of compact Target-Branched Declare models. We discuss the merits of our work through an evaluation based on a prototypical implementation using both artificial and real-life event logs.

Keywords: Process Mining, Knowledge Discovery, Declarative Process

1. Introduction

Process discovery is the important initial step of business process management that aims at arriving at an as-is model of an investigated process [1]. Due

*Corresponding author.
E-mail address: claudio.di.ciccio@wu.ac.at
Postal address: Vienna University of Economics and Business, Institute for Information Business (Building D2, Entrance C) – Welthandelsplatz 1, A-1020 Vienna, Austria
Phone number: +43 1 31336 5222

Preprint submitted to Elsevier *28th July 2015*

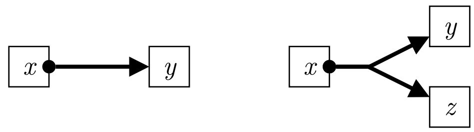
to this step being difficult and time-consuming, various techniques have been proposed to automatically discover a process model from event logs. These log data are often generated by information systems that support parts or the entirety of a process. The result is typically presented as a Petri net or a similar kind of flow chart and the automatic discovery is referred to as process mining.

While process mining has proven to be a powerful technique for structured and standardised processes, there is an ongoing debate on how processes with a high degree of variability can be effectively mined. One approach to this problem is to generate a declarative process model, which rather shows the constraints of behaviour instead of the available execution sequences. The resulting models are represented in languages like Declare. In many cases, they provide a way to represent complex, unstructured behaviour in a compact way, which would look overly complex in a spaghetti-like Petri net.

Declare is a process modelling language first introduced in [2]. The language defines a set of classes of constraints, the Declare templates, that are considered the most interesting ones for describing business processes. Templates are parameterised and constraints are instantiations of templates on real activities. For example, the *Response* constraint, stating that "activity *pay* is always eventually followed by activity *send invoice*" is an instantiation of the Declare template *Response* specifying that "an activity x is always eventually followed by an activity y". Templates have a graphical representation and formal semantics based on Linear Temporal Logic on Finite Traces (LTL_f). This allows Declare models to be verifiable and executable. Figure 1a shows the graphical representation of the *Response* template. Its LTL_f semantics is $\Box(x \rightarrow \Diamond y)$. Constraints inherit the graphical representation and the LTL_f semantics from the corresponding templates.

The current techniques for the discovery of Declare models [3, 4, 5, 6, 7] are limited to the discovery of constraints based on the standard set of Declare templates. This means that the discovered constraints will involve one activity for each parameter specified in the corresponding templates. However, as described in [2], a constraint can define more than one activity for each parameter. For example, a *Response* constraint can be used to express rules like "in a bank, a mortgage application is always eventually followed by a notification to the applicant by phone *or* by a notification by e-mail". In this rule, the "mortgage application" plays the role of the *activation*. "Notification by phone" and "notification by e-mail" constitute the so-called *targets* of the constraint. In this case, we say that the target parameter branches and, in the graphical representation, this is displayed by multiple arcs connecting the activation to the branched targets. In LTL_f semantics, a branched parameter is replaced by a disjunction of parameters. Figure 1b shows the graphical representation of the *Response* template branching on the target. Its LTL_f semantics is $\Box(x \rightarrow \Diamond(y \vee z))$.

Target-Branched Declare (TBDeclare) extends Declare by encompassing constraints that branch on target parameters, thus providing the process modellers with the possibility of defining a much wider set of constraints. In this paper, we address the problem of mining TBDeclare constraints efficiently. At the same time, the technique we propose aims at limiting the sheer amount of



(a) *Response*(x,y) (b) *Response*(x, $\{y, z\}$)

Figure 1: Declare (a) and Target-Branched Declare (b) *Response* templates.

returned constraints to the set of the most meaningful ones. To this extent, we rely on formal properties of TBDeclare, i.e., *(i)* set-dominance and *(ii)* subsumption hierarchy. Set-dominance is based on the observation that, for example, stating that “a is always eventually followed by b or c” entails that “a is always eventually followed by b, c or d”, i.e., since the set of targets for the first constraint is included in the set of targets for the second constraint, the first constraint is stronger than the second one. In this case, if both constraints hold in the provided event log, only the stronger one will be discovered. In addition, Declare constraints are not independent, but form a subsumption hierarchy. Therefore, a constraint (e.g., a is eventually followed by b or c) is redundant if a stronger constraint holds (e.g., a is directly followed by b or c). Also in this case, it is possible to keep the stronger constraint and discard the weaker one in the discovered model. The key idea of our proposed approach is to exploit set-dominance and subsumption hierarchy relationships, in combination with the use of interestingness metrics like constraint support and confidence [5], to drastically prune the set of discovered constraints. We present formal proofs to demonstrate the merits of this approach and a prototypical implementation for emphasising its feasibility and efficiency.

In this paper, we extend the work presented in [8] in four directions: *(i)* theoretical discussion, *(ii)* algorithm presentation, *(iii)* implementation improvement, and *(iv)* evaluation. From a foundational perspective, this paper formally elaborates on how the monotonicity of LTL$_f$ temporal operators can be exploited to prove set-dominance for TBDeclare. The algorithm is presented in thorough detail here: it describes all the procedures undertaken to mine the constraints, along with trailing examples. The implementation of the algorithm is also improved now, as an entirely new technique for the computation of *AlternateResponse* and *AlternatePrecedence* constraints has been devised. In this way, a major limitation of the process discovery algorithm presented in [8] is resolved. Furthermore, this has enabled us to cover a broader range of experiments including the application to an additional benchmark based on the use of the log provided for the BPI challenge 2014 [9].

Against this background, this paper is structured as follows. Section 2 introduces the essential concepts of LTL$_f$ and Declare as a background of our work. Section 3 provides the formal foundations for mining Target-Branched constraints. Section 4 defines the construction of a knowledge base from which the final constraint set is built. Section 5 describes the performance evaluation.

Section 6 investigates our contribution in the light of related work. Section 7 concludes the paper with an outlook on future research.

2. Background

Process mining is the set of techniques that aims at understanding the behaviour of a process, given as input a set of data reporting the executions of such a process, i.e., an event log L. An event log consists of a collection of traces $\vec{t_i}$, with $i \in [1, |L|]$ and $|L|$ being the size of the log, recording information about process instance executions. A trace is a sequence of events. Events are log entries specifying the execution data referred to an activity of the process. In the following, we will assume that each event is uniquely corresponding to the execution of a single activity. Therefore, we will interchangeably adopt the terms "event" and "activity" occurring in the log. The set of the activities that may occur in the log is called log alphabet. Hereinafter, the generic log alphabet is denoted as Σ. Elements of Σ will be collectively indicated as a, b, c. Denoting the set of sequences of activities as Σ^*, and indicating a multi-set as $\mathfrak{M}(\cdot)$, we have that $L \in \mathfrak{M}(\Sigma^*)$.

One of the challenges in process mining is the compact presentation of the mined behaviour. It has been observed that procedural models such as Petri nets tend to become overly complex for flexible processes that are situated in dynamic environments. Therefore, it has been argued to rather utilise declarative process modelling languages (like Declare) in such contexts, in order to facilitate a better understanding of the mined process models by humans [10, 11]. Declare has its formal foundation in linear temporal logic with finite trace semantics, which we introduce in Section 2.1. Section 2.2, then, describes Declare and how it is grounded in linear temporal logic.

2.1. Linear Temporal Logic over Finite Traces

Linear Temporal Logic (LTL) [12] is a language meant to express properties that hold true in systems that change their state over time. The behaviour of such systems is expressed in the form of a temporal structure, i.e., a transition system [13]. LTL was originally proposed in computer science as a specification language for concurrent programs. It was thought, in fact, to be adopted for the formal verification of server systems and very large system circuits, which in theory run infinitely. The states of such systems are expressed in terms of propositional formulae. The evolution is defined by transitions between states.

A typical LTL formula expressing a *fairness* condition is $\Box\Diamond\Phi$, where Φ is a propositional formula, indicating the condition to always ($\Box$) eventually ($\Diamond$) hold true. LTL$_f$ [14, 15] is the variant of LTL interpreted over finite system executions. It adopts the syntax of LTL. Formulae of LTL$_f$ are built from a set $\mathcal{A}$ of propositional symbols (*atomic propositions*) and are closed under the boolean connectives ($\neg$, unary, and $\vee$, $\wedge$, $\rightarrow$, binary) and the temporal operators $\bigcirc$ (*next*), $\Diamond$ (*eventually*), $\Box$ (*always*), unary, and $\mathcal{U}$ (*until*) and $\mathcal{W}$

(*weak until*), binary. The syntax is defined as follows.

$$
\begin{array}{rlllll}
\varphi, \psi = & \alpha & \mid \ \rho & \mid \ \lambda & & \qquad\qquad\qquad (\text{with } \alpha \in \mathcal{A}) \\
\rho = & \neg\varphi & \mid \ \varphi \vee \psi & \mid \ \varphi \wedge \psi & \mid \ \varphi \rightarrow \psi & \\
\lambda = & \bigcirc\varphi & \mid \ \Diamond\varphi & \mid \ \Box\varphi & \mid \ \varphi \,\mathcal{U}\, \psi & \mid \ \varphi \,\mathcal{W}\, \psi
\end{array}
$$

Intuitively, $\bigcirc\varphi$ means that φ holds true in the next instant in time, $\Diamond\varphi$ signifies that φ holds eventually before the last instant in time (included), $\Box\varphi$ expresses the fact that from the current state until the last instant in time φ holds, $\varphi \,\mathcal{U}\, \psi$ says that ψ holds eventually in the future and φ holds until that point, and $\varphi \,\mathcal{W}\, \psi$ relaxes $\varphi \,\mathcal{U}\, \psi$ in the sense that either from the current state until the last instant in time φ holds, or $\varphi \,\mathcal{U}\, \psi$ holds.

The semantics of LTL$_f$ is provided in terms of finite runs, i.e., finite sequences of consecutive instants in time, represented by finite words π over $2^{\mathcal{A}}$. The instant i in run π is denoted as $\pi(i)$, with $i \in [1, |\pi|]$, with $|\pi|$ being the length of the run. In the following, we indicate that, e.g., atomic proposition α is interpreted as true ($\top$) at instant i in π with $\alpha \in \pi(i)$. Conversely, if $\alpha \notin \pi(i)$, α is interpreted as false ($\bot$). Given a finite run π, we inductively define when an LTL$_f$ formula φ (respectively ψ) is true at an instant i, denoted as $\pi, i \models \varphi$ (respectively $\pi, i \models \psi$), as:

$\pi, i \models \alpha$ for $\alpha \in \mathcal{A}$, iff $\alpha \in \pi(i)$ (α is interpreted as true in $\pi(i)$);
$\pi, i \models \neg\varphi$ iff $\pi, i \not\models \varphi$;
$\pi, i \models \varphi \wedge \psi$ iff $\pi, i \models \varphi$ and $\pi, i \models \psi$;
$\pi, i \models \varphi \vee \psi$ iff $\pi, i \models \varphi$ or $\pi, i \models \psi$;
$\pi, i \models \bigcirc\varphi$ iff $\pi, i{+}1 \models \varphi$, having $i < |\pi|$;
$\pi, i \models \varphi \,\mathcal{U}\, \psi$ iff for some $j \in [i, |\pi|]$, we have that $\pi, j \models \psi$, and for all $k \in [i, j{-}1]$, we have that $\pi, k \models \varphi$.

The semantics of the remaining operators can be derived by recalling that:

$$\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi \qquad\qquad \Box\varphi \equiv \neg\Diamond\neg\varphi;$$
$$\Diamond\varphi \equiv \top \,\mathcal{U}\, \varphi; \qquad\qquad \varphi \,\mathcal{W}\, \psi \equiv \Box\varphi \vee (\varphi \,\mathcal{U}\, \psi).$$

We recall here that, given two LTL$_f$ formulas φ, ψ, $\varphi \models \psi$ (φ models ψ) iff $\forall i \in [1, \pi]$, $\pi, i \models \varphi$ entails $\pi, i \models \psi$. As clarified in [16], temporal operators enjoy the property of *monotonicity* [17]. A function $f : X \rightarrow Y$, where X and Y are partially ordered sets under the binary relation $\leqslant$, is *monotonic* iff, given $x, x' \in X$ such that $x \leqslant x'$, then $f(x) \leqslant f(x')$. f is said to be *antimonotonic* iff, given $x, x' \in X$ such that $x \leqslant x'$, then $f(x) \geqslant f(x')$, where $\geqslant$ is the inverse of $\leqslant$.

With a slight abuse of notation, given formulae φ and ψ, such that $\varphi \models \psi$, then:

1. a unary operator $\bullet$ is monotonic iff $\bullet\varphi \models \bullet\psi$;

2. a unary operator $\bullet$ is antimonotonic iff $\bullet\varphi \Dashv \bullet\psi$;

3. a binary operator $\otimes$ is monotonic iff $\varphi \models \varphi \otimes \psi$;

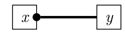
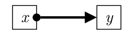
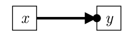
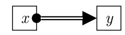
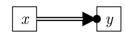
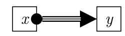
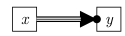
Template	**Formalisation**	**Notation**	**Activ.**	**Target**
$RespondedExistence(x, y)$	$\Diamond x \rightarrow \Diamond y$		x	y
$Response(x, y)$	$\Box(x \rightarrow \Diamond y)$		x	y
$Precedence(x, y)$	$\neg y \, \mathcal{W} \, x$		y	x
$AlternateResponse(x, y)$	$\Box(x \rightarrow \bigcirc(\neg x \, \mathcal{U} \, y))$		x	y
$AlternatePrecedence(x, y)$	$(\neg y \, \mathcal{W} \, x) \wedge \Box(y \rightarrow \bigcirc(\neg y \, \mathcal{W} \, x))$		y	x
$ChainResponse(x, y)$	$\Box(x \rightarrow \bigcirc y)$		x	y
$ChainPrecedence(x, y)$	$\Box(\bigcirc y \rightarrow x)$		y	x

Table 1: Graphical notation and LTL$_f$ formalisation of some Declare templates.

4. a binary operator $\otimes$ is antimonotonic iff $\varphi \dashv \varphi \otimes \psi$.

Monotonicity holds for propositional logic operators $\vee$, $\wedge$ and $\rightarrow$, whereas $\neg$ is antimonotonic [17]. Temporal operators $\bigcirc$, $\Diamond$, $\Box$, $\mathcal{U}$ and $\mathcal{W}$ are monotonic as well [13]. LTL$_f$ syntax and semantics will be used in the remainder of the paper.

2.2. Declare

One of the most frequently used declarative languages is Declare, first introduced in [2]. Instead of explicitly specifying the allowed sequences of events, Declare consists of a set of constraints that are applied to activities and must be valid during the process execution. Constraints are based on templates that define parametrised classes of temporal properties. Templates have a graphical representation and their semantics can be formalised using LTL$_f$. In this way, analysts work with the graphical representation of templates, while the underlying formulae remain hidden. Table 1 summarises the most commonly used Declare templates. For a complete specification see [2]. Here, we indicate template parameters with x or y symbols. Generic symbols of real activities in their instantiations (generic elements of a generic log alphabet) are indicated as $\Sigma = \{a, b, c, d, \ldots\}$. Assigned activity identifiers are denoted as sans-serif letters $\mathsf{a}, \mathsf{b}, \mathsf{c} \in \mathsf{\Sigma}$, where $\mathsf{\Sigma}$ is a set of activities that is assigned to a generic log alphabet Σ. Hence, a is a possible assignment of a. Following the same rationale, L is the symbol for the formal parameter denoting a generic log, whereas L is a concrete log. We remark here that the interpretation of Declare constraints restricts the common interpretation of LTL$_f$ in that two literals cannot be true at the same time. Furthermore, the run π on which an LTL$_f$ formula is evaluated is, in this context, a finite trace $\vec{t}$ of a log. We will univocally map atomic propositions of LTL$_f$ to the occurrence of an activity in the log alphabet ($\mathcal{A} \equiv \Sigma$).

The formulae shown in Table 1 can be readily formulated using natural language. The *RespondedExistence* template specifies that if x occurs, then y should also occur (either before or after x). The *Response* template specifies that when x occurs, then y should eventually occur after x. The *Precedence* template indicates that y should occur only if x has occurred before. Templates *AlternateResponse* and *AlternatePrecedence* strengthen the *Response* and *Precedence* templates respectively by specifying that activities must alternate without repetitions in between. Even stronger ordering relations are specified by templates *ChainResponse* and *ChainPrecedence*. These templates require that the occurrences of the two activities (x and y) are next to each other.

In order to illustrate these semantics, consider the *Response* constraint $\Box(\mathsf{a} \rightarrow \Diamond\mathsf{b})$. This constraint indicates that if a *occurs*, b must eventually *follow*. Given a log $\mathsf{L} = \left\{\vec{\mathsf{t}}_1, \vec{\mathsf{t}}_2, \vec{\mathsf{t}}_3, \vec{\mathsf{t}}_4\right\}$, where $\vec{\mathsf{t}}_1 = \langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{c} \rangle$, $\vec{\mathsf{t}}_2 = \langle \mathsf{b}, \mathsf{b}, \mathsf{c}, \mathsf{d} \rangle$, $\vec{\mathsf{t}}_3 = \langle \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{b} \rangle$, and $\vec{\mathsf{t}}_4 = \langle \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{c} \rangle$, this constraint is satisfied in $\vec{\mathsf{t}}_1$, $\vec{\mathsf{t}}_2$, and $\vec{\mathsf{t}}_3$, but not in $\vec{\mathsf{t}}_4$, because the second instance of a is not followed by a b in such trace.

An *activation* of a constraint in a trace is an event whose occurrence imposes some obligations on the occurrence of another event (the *target*) in the same trace. E.g., a is the activation and b is the target for the *Response* constraint $\Box(\mathsf{a} \rightarrow \Diamond\mathsf{b})$, because the execution of a forces b to be executed eventually. When a trace is compliant w.r.t. a constraint, every activation leads to a *fulfilment*. Consider, again, the *Response* constraint $\Box(\mathsf{a} \rightarrow \Diamond\mathsf{b})$. In trace $\vec{\mathsf{t}}_1$, the constraint is activated and fulfilled twice, whereas, in $\vec{\mathsf{t}}_3$, the same constraint is activated and fulfilled only once. On the other hand, when a trace is non-compliant, an activation can lead to a fulfilment but at least one activation in the trace leads to a *violation*. For example, in trace $\vec{\mathsf{t}}_4$, the *Response* constraint $\Box(\mathsf{a} \rightarrow \Diamond\mathsf{b})$ is activated twice: the first activation leads to a fulfilment (eventually b occurs) and the second activation to a violation (b does not occur subsequently). An algorithm to identify fulfilments and violations in a trace is presented in [18].

In the following, we will use $\mathfrak{C}$ to denote the set of Declare templates. Formally, a Declare template $\mathcal{C}/_n \in \mathfrak{C}$ is a predicate of *arity* $n \geqslant 1$, with $\mathcal{C}$ representing the *name* of the template and *arity* n specifying the number of *parameters*. For instance, the aforementioned *Response*$/_2$ constraint has arity $n = 2$ and *Response* as name. $\mathcal{C}(x, y)$ is an example of the notation we adopt to explicitly identify the two parameters (x, y) of template $\mathcal{C}$ of arity 2. In standard Declare, constraints are templates whose parameters are assigned single distinct activities. We will denote as $\mathfrak{C}^{\Sigma}$ the set of constraints that are obtained by assigning parameters of every $\mathcal{C}/_n \in \mathfrak{C}$ to n-permutations of distinct activities in the log alphabet Σ. Having, e.g., $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, $\mathfrak{C}^{\Sigma}$ would comprise *Response*(a, b), *Response*(b, a), *Response*(b, c), *Response*(c, b), *Response*(a, c), *Response*(c, a), *RespondedExistence*(a, b), *RespondedExistence*(b, a), etc. We will use $\mathcal{C}(a, b) \in \mathfrak{C}^{\Sigma}$ for indicating a generic constraint that assigns activities a and b ($a, b \in \Sigma$) to the parameters of the corresponding template $\mathcal{C}$. C is a shorthand notation that denotes a generic constraint.

In the remainder of this paper, we will focus on Declare constraints of arity 2 known as *relation constraints*. In particular, we will consider the ones that are listed in Table 1, hereinafter indicated as *unidirectional positive relation* *constraints*. The activation and target of a relation constraint C are henceforth denoted as $C|_{\bullet}$ and $C|_{\Rightarrow}$. Thus, $\mathit{Response}(a,b)|_{\bullet}$ is a and $\mathit{Response}(a,b)|_{\Rightarrow}$ is b. Vice-versa, $\mathit{Precedence}(a,b)|_{\bullet}$ is b and $\mathit{Precedence}(a,b)|_{\Rightarrow}$ is a. Table 1 reports activations and targets for all the listed templates. In the semantics of a unidirectional positive relation constraint $C = \mathcal{C}(x,y)$,

1. $\{C|_{\bullet}\} \cap \{C|_{\Rightarrow}\} = \varnothing$, and
2. $C|_{\Rightarrow}$ always falls under an even number of negations $\neg$.

In the literature, relation templates of arity 2 that do not impose rule 1 are known as "coupling constraints", whereas those that do not impose rule 2 are named "negative constraints" [19].

Table 1 contains the list of activations and targets for the templates we consider in this paper. In Section 3, we will explain how the standard Declare specification is extended towards Target-Branched Declare.

2.3. Support and confidence

To evaluate the relevance of a Declare constraint, we adopt two metrics proposed in the association rule mining literature [20]. The first one is meant to assess the reliability of a constraint w.r.t. a log, i.e., *support*. The second metric is meant to assess the relevance of a constraint w.r.t. a log, i.e., *confidence*.

The *support* of a Declare constraint C in an event log is defined as the proportion of fulfilments $\checkmark_L(C)$ of C in log L. For relation constraints, we can rely on the concept of activation. Therefore, we specify the support as the fraction of occurring activations of C that do not violate the constraint, w.r.t. the total number of activations in the log. Formally, let $\#_L(a)$ be the function $\# : \Sigma \times \mathfrak{M}(\Sigma^*) \rightarrow \mathbb{N}$, with $\mathbb{N}$ set of positive integers, that counts the occurrences of activity $a \in \Sigma$ in log $L \in \mathfrak{M}(\Sigma^*)$; let $\checkmark_L(C)$ be the function $\checkmark : \mathfrak{C}^{\Sigma} \times \mathfrak{M}(\Sigma^*) \rightarrow \mathbb{N}$ that counts the number of fulfilments of constraint $C \in \mathfrak{C}^{\Sigma}$ in log $L \in \mathfrak{M}(\Sigma^*)$. Then, the support $\mathscr{S}_L(C)$ of a constraint C in a log L is defined as a function $\mathscr{S} : \mathfrak{C}^{\Sigma} \times \mathfrak{M}(\Sigma^*) \rightarrow [0,1] \subseteq \mathbb{R}$, with $\mathbb{R}$ set of real numbers, expressed as follows:

$$\mathscr{S}_L(C) = \frac{\checkmark_L(C)}{\#_L(C|_{\bullet})} \tag{1}$$

where $C|_{\bullet}$ is the activation of constraint C.

The second metric is meant to assess the relevance of a constraint w.r.t. a log. It is named *confidence*, and scales the support of a constraint by the number of traces containing its activation. For the definition of confidence $\mathscr{C}$ of relation constraints, we rely on the notion of *activity-related log fraction* [21], i.e., the fraction of traces in which a given activity occurs at least once. Let $\varnothing_L(a)$ be the function $\varnothing : \Sigma \times \mathfrak{M}(\Sigma^*) \rightarrow \mathbb{N}$ that counts the traces of log $L \in \mathfrak{M}(\Sigma^*)$ in which activity $a \in \Sigma$ does *not* occur. Then, the activity-related log fraction is expressible as $1 - \frac{\varnothing_L(a)}{|L|}$ where $|L|$ is the number of traces in L. Therefore, given a log $L \in \mathfrak{M}(\Sigma^*)$ and a constraint $C \in \mathfrak{C}^\Sigma$, the confidence of C can be defined as a function $\mathscr{C} : \mathfrak{C}^\Sigma \times \mathfrak{M}(\Sigma^*) \to [0,1] \subseteq \mathbb{R}$, expressed as follows:

$$\mathscr{C}_L(C) = \mathscr{S}_L(C) \times \left(1 - \frac{\varnothing_L(C|_\bullet)}{|L|}\right). \tag{2}$$

3. Target-Branched Declare

In this section, we define Target-Branched Declare (TBDeclare). It extends Declare such that the target is not a single activity but a set of activities (see Table 2). This means that *Response*($\mathsf{a}, \{\mathsf{b}, \mathsf{c}\}$) is a TBDeclare constraint stating that "if a occurs, b *or* c must eventually follow". $\{\mathsf{b}, \mathsf{c}\}$ is referred to as a set-parameter. The cardinality of this set, indicating the number of branches, is called *branching factor* of the constraint. The class of TBDeclare exhibits some interesting properties, i.e., subsumption hierarchy and set-dominance. Subsumption hierarchy has already been investigated in [22] for branched Declare. In the following, we prove that the property of set-dominance holds. Then, we discuss implications of this property in terms of constraint support. These properties will be exploited in the mining algorithm.

TBDeclare template	LTL$_f$ semantics
$\mathit{RespondedExistence}(x, Y)$	$\Diamond x \to \Diamond \left(\bigvee_{i=1}^{\beta} y_i\right)$
$\mathit{Response}(x, Y)$	$\Box\left(x \to \Diamond \left(\bigvee_{i=1}^{\beta} y_i\right)\right)$
$\mathit{AlternateResponse}(x, Y)$	$\Box\left(x \to \bigcirc \left(\neg x \,\mathcal{U} \bigvee_{i=1}^{\beta} y_i\right)\right)$
$\mathit{ChainResponse}(x, Y)$	$\Box\left(x \to \bigcirc \left(\bigvee_{i=1}^{\beta} y_i\right)\right)$
$\mathit{Precedence}(Y, x)$	$\neg x \,\mathcal{W} \left(\bigvee_{i=1}^{\beta} y_i\right)$
$\mathit{AlternatePrecedence}(Y, x)$	$\mathit{Precedence}(Y, x) \wedge \Box(x \to \bigcirc \mathit{Precedence}(Y, x))$
$\mathit{ChainPrecedence}(Y, x)$	$\Box\left(\bigcirc x \to \left(\bigvee_{i=1}^{\beta} y_i\right)\right)$

Table 2: LTL$_f$ semantics for TBDeclare templates ($Y = \{y_1, \ldots, y_\beta\}$, with β *branching factor* of the constraint).

Formally, TBDeclare is a sub-class of a more general class of constraints extending standard Declare, which we will henceforth refer to as *Multi-valued Declare*. As said in Section 2.2, standard Declare imposes that template parameters are interpreted as single activities of the log alphabet Σ. Multi-valued Declare comprises the same set of templates of standard Declare, yet allowing the interpretation of parameters as elements of a boolean algebraic structure $\langle \Sigma, \star \rangle$ (a.k.a. groupoid [23]) consisting of a set of symbols Σ (i.e., the log alphabet) and a binary operator $\star$ under which the structure is closed. Thus, given $a, b \in \Sigma$, and $\rho = a \star b$, then $\rho \in \langle \Sigma, \star \rangle$. A semigroup $\langle \Sigma, * \rangle$ is a groupoid s.t. operation $*$ is associative.

Branched Declare [2] restricts the algebraic structure to a join-semilattice $\langle \Sigma, \vee \rangle$, i.e., an idempotent commutative semigroup, where $\vee$ is the join-operation [24]. For any semigroup $\langle \Sigma, * \rangle$ a natural partial-order relation can be defined [25]. A fortiori, we define it here for join-semilattices as $\rho \geqslant \rho'$ iff $\rho \vee \rho' = \rho$, for $\rho, \rho' \in \langle \Sigma, \vee \rangle$. In the domain of boolean algebras, $\geqslant$ is defined by the inverse entailment relation $\Dashv$, and the equality $=$ by the logical equivalence $\equiv$. Indeed, e.g., considering $a \vee b$ as ρ and a as ρ', we clearly have that $a \vee b \Dashv a$ as $a \vee b \vee a \equiv a \vee b$.

TBDeclare belongs to the class of unidirectional positive relation constraints that further restrict Branched Declare as follows: the interpretation of the target can be an element of $\langle \Sigma, \vee \rangle$, whereas the activation is interpretable only as a single activity in Σ. Thus, the example TBDeclare constraint given at the beginning of this section interprets the activation of *Response* as a and its target as $b \vee c$, where a, b and c are respectively assigned with a, b and c, meaning that "if a occurs, b *or* c must eventually follow." Notice that the join-operation of the join-semilattice in boolean algebra is such that semantics of Branched Declare and TBDeclare can still be expressed within LTL_f.

3.1. Set-Dominance

In this section, we formally prove that set-dominance holds for TBDeclare, mainly relying on the property of monotonicity of the LTL_f temporal operators. To this extent, we first define the class of Monotonic Branched Declare. Then, we show that two Monotonic Branched Declare constraints C and C' are such that if the assigned parameters of C are included in the assigned parameters of C', then the support of C is lower than or equal to the support of the C'. The property of monotonic non-decreasing trend of support w.r.t. the containment of set-parameters, will also be simply referred to as *set-dominance* for short. Finally, we show that for all Branched Declare constraints, the property of set-dominance holds true.

Preliminarily, we notice that, for join-semilattices, a bijective mapping μ can be established that connects elements of $\langle \Sigma, \vee \rangle$ to elements of $\langle \Sigma, \cup \rangle$ where $\cup$ is the set-union operation: $\mu : \langle \Sigma, \vee \rangle \to \langle \Sigma, \cup \rangle$. It can be shown that μ is an isomorphism preserving the partial-order $\geqslant$ defined on $\langle \Sigma, \vee \rangle$ by the partial-order given by set-containment $\supseteq$ in $\langle \Sigma, \cup \rangle$. In the following, we indicate by means of a capital letter, e.g., X or Y, a parameter that is interpreted as an element of $\langle \Sigma, \vee \rangle$. Therefore, *Response*$(x, Y)$ specifies a template where the second parameter is interpreted as an element of $\langle \Sigma, \vee \rangle$ (cf. Table 2). Without loss of generality, we identify every element ρ in $\langle \Sigma, \vee \rangle$ by its μ-mapped set $\mathrm{S} = \mu(\rho)$ in $\langle \Sigma, \cup \rangle$. Hence, *Response* having a as the activation and $b \vee c$ as the target will be denoted as *Response*(a, S) where $\mathrm{S} = b \cup c$. S will thus be also referred to as *set-parameter*. As an alternative, we will also adopt for such constraint the following notation: *Response*$(a, \{b, c\})$.

Monotonic Branched Declare is the class of Multi-valued Declare templates for which it holds true that any two constraints $\mathcal{C}(\mathrm{R}_1, \ldots, \mathrm{R}_n)$ and $\mathcal{C}(\mathrm{S}_1, \ldots, \mathrm{S}_n)$, obtained as instantiations of the same template $\mathcal{C}/_n \in \mathfrak{C}$ with set-parameters $\mathrm{R}_1, \ldots, \mathrm{R}_n$ and $\mathrm{S}_1, \ldots, \mathrm{S}_n$, with $\mathrm{S}_i \supseteq \mathrm{R}_i$ for every $i \in [1, n]$, are such that $\mathcal{C}(\mathrm{R}_1, \ldots, \mathrm{R}_n) \models \mathcal{C}(\mathrm{S}_1, \ldots, \mathrm{S}_n)$.

Theorem 1 (Set-dominance of Monotonic Branched Declare). *Given the non-empty sets of activities $R_1, \ldots, R_n$ and $S_1, \ldots, S_n$ of a log alphabet Σ such that $\Sigma \supseteq S_i \supseteq R_i$ for every $i \in [1, n]$, a log L and a Monotonic Branched Declare template $\mathcal{C}$, then the support of $C' = \mathcal{C}(S_1, \ldots, S_n)$ is greater than or equal to the support of $C = \mathcal{C}(R_1, \ldots, R_n)$, i.e., $\mathscr{S}_L(C') \geqslant \mathscr{S}_L(C)$.*

Proof 1. *Because the log L over which the support is evaluated is the same for both constraints, we focus on the number of fulfilments, namely $\checkmark_L(C)$ and $\checkmark_L(C')$, for constraints C and C'. By definition of Monotonic Branched Declare, if $S_i \supseteq R_i$ for every $i \in [1, n]$, where n is the arity of constraint template $\mathcal{C}$, then $C \models C'$. Therefore, due to the definition of model for a constraint w.r.t. a log, we have that $\checkmark_L(C) \leqslant \checkmark_L(C')$. The proof proceeds per absurdo. If $\checkmark_L(C) > \checkmark_L(C')$, there would necessarily exist at least a case that verifies C but not C'. This would contradict the fact that $C \models C'$.* □

Lemma 1 (Monotonicity of TBDeclare). *Target-Branched Declare belongs to the class of Monotonic Branched Declare, i.e., given an activity a in the log alphabet Σ, two non-empty sets of activities S and S' such that $S \subseteq S' \subseteq \Sigma$, and a TBDeclare template $\mathcal{C}$, then $\mathcal{C}(a, S) \models \mathcal{C}(a, S')$.*

Proof 2. *In the base case, $S = S' = \{b_1, \ldots, b_n\}$. Therefore, $\mathcal{C}(a, S) \equiv \mathcal{C}(a, S')$. For the proof in the inductive case $S' = S \bigcup \{b_{n+1}\}$ where $b_{n+1} \notin S$, we resort on the fact that the semantics of constraint templates of Declare are expressible by means of LTL_f. Among operators used in LTL_f, $\neg$ is known to be anti-monotonic, whereas all the other LTL_f operators are monotonic. The target of a Declare unidirectional positive relation constraint template always falls under an even number of $\neg$ operators. By definition of TBDeclare, only the target is meant to be replaced by elements of the boolean join-semilattice $\langle \Sigma, \vee \rangle$. Hence, the target set-parameter always lets the activities assigned fall under an even number of negations. This guarantees the monotonicity of the constraint, due to the principle of non-contradiction.*[1] □

The section now proceeds with the application of the inductive part of the proof to each template under examination, listed in Table 1.

RespondedExistence. $\mathit{RespondedExistence}(a, \mathrm{S}') \equiv \Diamond a \rightarrow \Diamond \left(\bigvee_{i=1}^{n} b_i \vee b_{n+1}\right)$. Recalling that, given two LTL_f formulae φ and ψ:

(a) $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$, and

(b) $\Diamond(\varphi \vee \psi) \equiv \Diamond\varphi \vee \Diamond\psi$,

[1] Given a boolean formula φ, $\neg(\neg\varphi) \equiv \varphi$.

we have that *RespondedExistence*$(a, \mathrm{S}') \equiv \neg\Diamond a \vee (\bigvee_{i=1}^{n} \Diamond b_i) \vee \Diamond b_{n+1}$. Consequently, *RespondedExistence*$(a, \mathrm{S}') \equiv$ *RespondedExistence*$(a, \mathrm{S}) \vee \Diamond b_{n+1}$. Given two LTL$_f$ formulae φ and ψ

(c) $\varphi \models \varphi \vee \psi$

due to the monotonicity of $\vee$. Therefore, Lemma 1 for *RespondedExistence* is proven.

Response. *Response*$(a, \mathrm{S}') \equiv \Box(\neg a \vee \Diamond(\bigvee_{i=1}^{n} b_i) \vee \Diamond b_{n+1})$ due to (a) and (b). We have also that:

(d) if $\varphi \models \psi$, then $\Box\varphi \models \Box\psi$

for the monotonicity of the temporal operators in LTL$_f$. Therefore, $\Box\varphi \models \Box(\varphi \vee \psi)$, because of (c). Since *Response*$(a, \mathrm{S}) \equiv \Box(\neg a \vee \Diamond(\bigvee_{i=1}^{n} b_i))$, we have that Lemma 1 holds true for *Response*.

AlternateResponse. As a consequence of the application of (a), *AlternateResponse*$(a, \mathrm{S}') \equiv \Box(\neg a \vee \bigcirc(\neg a \,\mathcal{U}\, (\bigvee_{i=1}^{n} b_i \vee b_{n+1})))$, whereas *AlternateResponse*$(a, \mathrm{S}) \equiv \Box(\neg a \vee \bigcirc(\neg a \,\mathcal{U}\, (\bigvee_{i=1}^{n} b_i)))$. Given the LTL$_f$ formulae φ, ψ and ψ',

(e) if $\psi \models \psi'$, then $\varphi \,\mathcal{U}\, \psi \models \varphi \,\mathcal{U}\, \psi'$

due to the monotonicity of the temporal operators in LTL$_f$. Therefore, we have that $(\neg a \,\mathcal{U}\, (\bigvee_{i=1}^{n} b_i)) \models (\neg a \,\mathcal{U}\, (\bigvee_{i=1}^{n} b_i \vee b_{n+1}))$, because of (c). Furthermore, given two LTL$_f$ formulae φ and ψ,

(f) if $\varphi \models \psi$, then $\bigcirc\varphi \models \bigcirc\psi$

due to the monotonicity of the temporal operators in LTL$_f$. As a consequence, $\bigcirc(\neg a \,\mathcal{U}\, (\bigvee_{i=1}^{n} b_i)) \models \bigcirc(\neg a \,\mathcal{U}\, (\bigvee_{i=1}^{n} b_i \vee b_{n+1}))$. Given the LTL$_f$ formulae φ, ψ and ψ',

(g) if $\psi \models \psi'$, then $\varphi \vee \psi \models \varphi \vee \psi'$.

This leads to the conclusion that Lemma 1 holds true for *AlternateResponse*, considering (d).

ChainResponse. Given two LTL$_f$ formulae φ and ψ, we have that:

(h) $\bigcirc(\varphi \vee \psi) \equiv \bigcirc\varphi \vee \bigcirc\psi$.

Applying (a) and (h), we have that *ChainResponse*$(a, \mathrm{S}') \equiv \Box(\neg a \vee \bigcirc(\bigvee_{i=1}^{n} b_i) \vee \bigcirc b_{n+1})$, whereas *ChainResponse*$(a, \mathrm{S}) \equiv \Box(\neg a \vee \bigcirc(\bigvee_{i=1}^{n} b_i))$. Lemma 1 is proven for *ChainResponse* then, due to (c) and (d).

Precedence. By definition of $\mathcal{W}$, we have that *Precedence*$(\mathrm{S}', a) \equiv (\Box\neg a) \vee (\neg a \,\mathcal{U}\, (\bigvee_{i=1}^{n} b_i \vee b_{n+1}))$, and *Precedence*$(\mathrm{S}, a) \equiv (\Box\neg a) \vee (\neg a \,\mathcal{U}\, (\bigvee_{i=1}^{n} b_i))$. Lemma 1 naturally extends to the case of *Precedence* by applying (c), since it is already proven that $(\neg a \,\mathcal{U}\, (\bigvee_{i=1}^{n} b_i)) \models (\neg a \,\mathcal{U}\, (\bigvee_{i=1}^{n} b_i \vee b_{n+1}))$ (see demonstration for *AlternateResponse*).

AlternatePrecedence. For what *AlternatePrecedence* is regarded, the two terms of the conjunction have to be considered separately. The first term refers to *Precedence*, and it is already proven that $\mathit{Precedence}(\mathrm{S}, a) \models \mathit{Precedence}(\mathrm{S}', a)$. The second term is $\Box(\neg a \vee \bigcirc \mathit{Precedence}(\mathrm{S}, a))$ for $\mathit{AlternatePrecedence}(\mathrm{S}, a)$ and $\Box\left(\neg a \vee \bigcirc \mathit{Precedence}(\mathrm{S}', a)\right)$ for $\mathit{AlternatePrecedence}(\mathrm{S}', a)$, due to (a). As a consequence, $\Box(\neg a \vee \bigcirc \mathit{Precedence}(\mathrm{S}, a)) \models \Box\left(\neg a \vee \bigcirc \mathit{Precedence}(\mathrm{S}', a)\right)$, due to (c) and (d). As a conclusion, since it is known that

(i) if $\varphi \models \varphi'$ and $\psi \models \psi'$ then $\varphi \wedge \psi \models \phi' \wedge \psi'$

we can conclude that Lemma 1 holds true for *AlternatePrecedence*.

ChainPrecedence. We have that $\mathit{ChainPrecedence}(a, \mathrm{S}') \equiv \Box(\neg \bigcirc a \vee (\bigvee_{\mathrm{S} \in \mathrm{S}} b_i) \vee b_{n+1})$ and $\mathit{ChainPrecedence}(a, \mathrm{S}) \equiv \Box(\neg \bigcirc a \vee (\bigvee_{\mathrm{S} \in \mathrm{S}} b_i))$, due to (a). Considering (c) and (d), it is thus proven that Lemma 1 is verified.

Following Theorem 1 describes the monotonic non-decreasing trend of the support for constraints w.r.t. set-containment of the target set of activities for TBDeclare.

Corollary 1 (Set-dominance of TBDeclare). *Given an activity a in the log alphabet Σ, two non-empty sets of activities S, S' such that $\Sigma \supseteq S' \supseteq S$, a log L and a TBDeclare template $\mathcal{C}$, then the support of $\mathcal{C}(a, S')$ is greater than or equal to the support of $\mathcal{C}(a, S)$, i.e., $\mathscr{S}_L\left(\mathcal{C}(a, S')\right) \geqslant \mathscr{S}_L(\mathcal{C}(a, S))$.*

Proof 3. *Directly follows from Theorem 1 and Lemma 1.* $\Box$

As a final remark, we highlight that the notion of support introduced in Equation (1) especially for relation constraints, is still compliant with Corollary 1 in the light of the proof of Theorem 1. In fact, it still holds that the denominator of the proportion remains the same for both constraints, as the activations are the same along the log, and the activations that do not violate $\mathcal{C}(a, \mathrm{S}')$ cannot be less than the ones of $\mathcal{C}(a, \mathrm{S})$. Otherwise, if at least a fulfilment of $\mathcal{C}(a, \mathrm{S})$ were not a fulfilment of $\mathcal{C}(a, \mathrm{S}')$, it would constitute a counterexample against Lemma 1, according to which $\mathcal{C}(a, \mathrm{S}) \models \mathcal{C}(a, \mathrm{S}')$.

In the following section, we show how the discovery algorithm exploits the fact that the support of TBDeclare is monotonously non-decreasing w.r.t. the set-containment relation of target set-parameters.

4. Discovery

This section describes MINERful for Target-Branched Declare (TB-MINERful), a three-step algorithm that, starting from an input log L, *(i)* builds a knowledge base, which keeps statistics on activity occurrences in L; *(ii)* queries the knowledge base for support and confidence of constraints in L; *(iii)* prunes constraints not having sufficient support and confidence. The input of the algorithm is a log L. Three thresholds can be specified: *(i) branching factor*, i.e., the maximum branching factor allowed for the discovered constraints, *(ii) minimum support*, and *(iii) minimum confidence*.

4.1. The Knowledge Base

The first step is the construction of a knowledge base, which keeps statistics on the occurrences of activities in the log. It comprises the 9 functions listed further below in this section. $\varnothing$ and $\#$ were already outlined in Section 2.3 and are here formally defined for the sake of completeness.

Following the same rationale of the symbology introduced in Section 2.2, set-parameters are here indicated with symbols $\mathrm{S}, \mathrm{T} \subseteq \Sigma$. $\mathrm{S} = \{b, c\}$ is possibly assigned with $\{\mathsf{b}, \mathsf{c}\}$.

Let $\mathfrak{K} = \{\ntriangleright, \ntriangleleft, \ntriangleleft\!\!\ntriangleright, \blacktriangleright\!\blacktriangleright, \blacktriangleleft\!\blacktriangleleft, \looparrowright, \looparrowleft\}$ and $\mathfrak{K}_1 = \{\varnothing, \#\}$ be two sets of functions defined as follows. In the examples, $=_\nu$ and $=_{\nu_1}$ specify the number that would be assigned to the functions respectively in $\mathfrak{K}$ and $\mathfrak{K}_1$, given a log. As an example log we use $\mathsf{L} = \{\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{c}, \mathsf{a} \rangle, \langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{c}, \mathsf{a}, \mathsf{d} \rangle\}$ defined over $\mathsf{\Sigma} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$.

$\varnothing : \Sigma \times \mathfrak{M}(\Sigma^*) \to \mathbb{N}$. Function $\varnothing(a, L)$ (hereinafter, $\varnothing_L(a)$ for short) counts the traces of $L \in \mathfrak{M}(\Sigma^*)$ in which $a \in \Sigma$ did not occur. For instance, $\varnothing_{\mathsf{L}}(\mathsf{a}) =_{\nu_1} 0$, because a occurs in every trace in L. $\varnothing_{\mathsf{L}}(\mathsf{d}) =_{\nu_1} 1$, instead.

$\# : \Sigma \times \mathfrak{M}(\Sigma^*) \to \mathbb{N}$. Function $\#(a, L)$ (hereinafter, $\#_L(a)$ for short) counts the occurrences of $a \in \Sigma$ in $L \in \mathfrak{M}(\Sigma^*)$. Therefore, $\#_{\mathsf{L}}(\mathsf{a}) =_{\nu_1} 8$.

$\ntriangleright : \Sigma \times \wp(\Sigma) \times \mathfrak{M}(\Sigma^*) \to \mathbb{N}$. [2] Function $\ntriangleright(a, \mathrm{S}, L)$ (hereinafter, $\ntriangleright_L(a, \mathrm{S})$ for short) counts the occurrences of $a \in \Sigma$ with no following $b \in \mathrm{S} = \{b_1, \ldots, b_\beta\}$ (for any $\beta \in [1, |\Sigma|]$) in the traces of $L \in \mathfrak{M}(\Sigma^*)$. In the example, $\ntriangleright_{\mathsf{L}}(\mathsf{a}, \{\mathsf{d}\}) =_\nu 4$, $\ntriangleright_{\mathsf{L}}(\mathsf{a}, \{\mathsf{b}\}) =_\nu 4$, and $\ntriangleright_{\mathsf{L}}(\mathsf{a}, \{\mathsf{b}, \mathsf{c}\}) =_\nu 2$.

$\ntriangleleft : \Sigma \times \wp(\Sigma) \times \mathfrak{M}(\Sigma^*) \to \mathbb{N}$. Function $\ntriangleleft(a, \mathrm{S}, L)$ (hereinafter, $\ntriangleleft_L(a, \mathrm{S})$ for short) counts the occurrences of $a \in \Sigma$ with no preceding $b \in \mathrm{S} = \{b_1, \ldots, b_\beta\}$ (for any $\beta \in [1, |\Sigma|]$) in the traces of $L \in \mathfrak{M}(\Sigma^*)$. Thus, e.g., $\ntriangleleft_{\mathsf{L}}(\mathsf{a}, \{\mathsf{d}\}) =_\nu 8$, $\ntriangleleft_{\mathsf{L}}(\mathsf{a}, \{\mathsf{b}\}) =_\nu 4$, and $\ntriangleleft_{\mathsf{L}}(\mathsf{a}, \{\mathsf{b}, \mathsf{c}\}) =_\nu 4$.

$\ntriangleleft\!\!\ntriangleright : \Sigma \times \wp(\Sigma) \times \mathfrak{M}(\Sigma^*) \to \mathbb{N}$. Function $\ntriangleleft\!\!\ntriangleright(a, \mathrm{S}, L)$ (hereinafter, $\ntriangleleft\!\!\ntriangleright_L(a, \mathrm{S})$ for short) counts the occurrences of $a \in \Sigma$ with no co-occurring $b \in \mathrm{S} = \{b_1, \ldots, b_\beta\}$ (for any $\beta \in [1, |\Sigma|]$) in the traces of $L \in \mathfrak{M}(\Sigma^*)$. Therefore, $\ntriangleleft\!\!\ntriangleright_{\mathsf{L}}(\mathsf{a}, \{\mathsf{d}\}) =_\nu 4$, and $\ntriangleleft\!\!\ntriangleright_{\mathsf{L}}(\mathsf{a}, \{\mathsf{b}, \mathsf{d}\}) =_\nu 0$.

$\blacktriangleright\!\blacktriangleright : \Sigma \times \Sigma \times \mathfrak{M}(\Sigma^*) \to \mathbb{N}$. Function $\blacktriangleright\!\blacktriangleright(a, b, L)$ (hereinafter, $\blacktriangleright\!\blacktriangleright_L(a, b)$ for short) counts the occurrences of $a \in \Sigma$ having $b \in \Sigma$ as the next event in the traces of $L \in \mathfrak{M}(\Sigma^*)$. Hence, e.g., $\blacktriangleright\!\blacktriangleright_{\mathsf{L}}(\mathsf{a}, \mathsf{b}) =_\nu 2$, and $\blacktriangleright\!\blacktriangleright_{\mathsf{L}}(\mathsf{a}, \mathsf{d}) =_\nu 1$.

$\blacktriangleleft\!\blacktriangleleft : \Sigma \times \Sigma \times \mathfrak{M}(\Sigma^*) \to \mathbb{N}$. Function $\blacktriangleleft\!\blacktriangleleft(a, b, L)$ (hereinafter, $\blacktriangleleft\!\blacktriangleleft_L(a, b)$ for short) counts the occurrences of $a \in \Sigma$ having $b \in \Sigma$ as the preceding event in the traces of $L \in \mathfrak{M}(\Sigma^*)$. In the example, $\blacktriangleleft\!\blacktriangleleft_{\mathsf{L}}(\mathsf{a}, \mathsf{b}) =_\nu 2$, and $\blacktriangleleft\!\blacktriangleleft_{\mathsf{L}}(\mathsf{a}, \mathsf{d}) =_\nu 0$.

[2] By $\wp(\Sigma)$, we mean the power set of Σ.

$\looparrowright : \Sigma \times \wp(\Sigma) \times \mathfrak{M}(\Sigma^*) \to \mathbb{N}$. Function $\looparrowright(a, \mathrm{S}, L)$ (hereinafter, $\looparrowright_L(a, \mathrm{S})$ for short) counts how many times, after an occurrence of $a \in \Sigma$, a repeats until the first $b \in \mathrm{S} = \{b_1, \ldots, b_\beta\}$ follows in the same trace, for all the traces of $L \in \mathfrak{M}(\Sigma^*)$. If no $b \in \mathrm{S}$ appears in the trace after a, the repetitions after a are not counted. In the example, $\looparrowright_{\mathsf{L}}(\mathsf{a}, \{\mathsf{b}\}) =_\nu 2$, $\looparrowright_{\mathsf{L}}(\mathsf{a}, \{\mathsf{c}\}) =_\nu 4$, $\looparrowright_{\mathsf{L}}(\mathsf{a}, \{\mathsf{b}, \mathsf{c}\}) =_\nu 2$, and $\looparrowright_{\mathsf{L}}(\mathsf{a}, \{\mathsf{b}, \mathsf{d}\}) =_\nu 3$.

$\looparrowleft : \Sigma \times \Sigma^\beta \times \mathfrak{M}(\Sigma^*) \to \mathbb{N}$. Function $\looparrowleft(a, \mathrm{S}, L)$ (hereinafter, $\looparrowleft_L(a, \mathrm{S})$ for short) is similar to $\looparrowright_L(a, \mathrm{S})$, but reading the traces of $L \in \mathfrak{M}(\Sigma^*)$ contrariwise. Thus, $\looparrowleft_{\mathsf{L}}(\mathsf{a}, \{\mathsf{b}\}) =_\nu 2$, $\looparrowleft_{\mathsf{L}}(\mathsf{a}, \{\mathsf{c}\}) =_\nu 0$, $\looparrowleft_{\mathsf{L}}(\mathsf{a}, \{\mathsf{b}, \mathsf{c}\}) =_\nu 0$, and $\looparrowleft_{\mathsf{L}}(\mathsf{a}, \{\mathsf{b}, \mathsf{d}\}) =_\nu 2$.

The knowledge base is thus a tuple $\mathcal{KB} = \langle \Sigma, L, \mathfrak{K}, \mathfrak{K}_1, \nu, \nu_1 \rangle$, consisting of a log alphabet Σ, a log $L \in \mathfrak{M}(\Sigma^*)$, two sets of functions $\mathfrak{K}$ and $\mathfrak{K}_1$, and two interpretation functions $\nu_1 : \mathfrak{K}_1 \times \Sigma \times L \to \mathbb{N}$ and $\nu : \mathfrak{K} \times \Sigma \times \wp(\Sigma) \times L \to \mathbb{N}$. ν_1 assigns an integer value to the functions of the knowledge base that pertain to a single activity, for all the activities in the log alphabet, on the basis of the given log. ν assigns an integer value to the functions of the knowledge base that pertain to the interplay of every activity with all subsets of other activities in the log alphabet, on the basis of the given log. Next, we discuss how the knowledge base is built based on an input log.

4.2. Building the Knowledge Base

The objective of the algorithm for building the knowledge base formally is the definition of the interpretation functions that is consistent with the given log and log alphabet. To this extent, we adopt different approaches for different functions. However, the common characteristic is that they do not need more than one parse of the traces of the log to update the knowledge base. This leads to a reduction in the computation time. In particular, it makes the algorithm linear w.r.t. the number of traces.

The rationale behind the technique is that the parsing of the log is done for counting *(i)* the occurrences and misses of single activities $a \in \Sigma$, and *(ii)* the co-occurrences and misses of pairs of activities $a, b \in \Sigma$ in each trace. Variables storing such counts will be named *(i) singleton counters* and *(ii) pairwise counters*, respectively. Singleton and pairwise counters refer to specific elements of the knowledge base. For the sake of readability, counters will be henceforth identified by a $\mathtt{N}$ (tele-type) letter, indexed by the (parametric) activities that they consider. The symbol put at the apex specifies the element of the knowledge base for which the counter is meant to be utilised. For instance, singleton counter $\mathtt{N}^{\#}_{a}$ counts the total number of occurrences of a in the log. In log $\mathsf{L} = \{\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{c}, \mathsf{a} \rangle, \langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{c}, \mathsf{a}, \mathsf{d} \rangle\}$, $\mathtt{N}^{\#}_{\mathsf{a}} = 8$. Pairwise counter $\mathtt{N}^{\not\triangleright}_{a,b}$ is dedicated to counting the occurrences of a in a trace, after which no b occurs. In log L, $\mathtt{N}^{\not\triangleright}_{\mathsf{a},\mathsf{b}} = 4$. $\mathtt{N}^{\not\triangleright}_{a,b}$ is discussed in detail in Section 4.2.2.

Pairwise counters do not take into account the relation of an activity a with *sets* of other activities, though. On the other hand, computing a value for each $\mathrm{S} \in \wp(\Sigma \backslash \{a\})$ would be impractical. Therefore, we build *differential cumulative* set-counters. They are named "cumulative" because they derive co-occurrences and misses of activity a with sets of activities $\mathrm{S} = \bigcup_{i=1}^{\beta}\{b_i : b_i \in \Sigma\backslash\{a\}\}$ with $\beta \in [1, |\Sigma|]$, starting from the values of single pairwise counters that refer to pairs of activities a, b_i. They are qualified as "differential" due to the fact that they store values by differences. In the remainder, differential cumulative set-counters will be identified by symbol $\mathtt{\Delta}$ (indicating the differential nature), put in front of the pairwise counter from which they are derived. For instance, $\mathtt{\Delta N}^{\triangleright}_{a,\mathrm{S}}$ is a differential cumulative set-counter that stores the (differential) number of cases in which a is not followed by any of the activities in S.

Co-inductively, given $\mathrm{S} \subseteq \Sigma\backslash\{a\}$, $\mathtt{\Delta N}^{\triangleright}_{a,\mathrm{S}}$ reports the difference between *(i)* the number of times in which no $b \in \mathrm{S}$ occurred and *(ii)* $\sum\limits_{\mathrm{T} \supseteq S} \mathtt{\Delta N}^{\triangleright}_{a,\mathrm{T}}$, having $\mathrm{T} \subseteq \Sigma\backslash\{a\}$. After parsing log L, we thus have the following values:
(i) $\mathtt{\Delta N}^{\triangleright}_{\mathsf{a},\{\mathsf{b}\}} = 1$, *(ii)* $\mathtt{\Delta N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{c}\}} = 1$, *(iii)* $\mathtt{\Delta N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{c},\mathsf{d}\}} = 1$, *(iv)* $\mathtt{\Delta N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{d}\}} = 1$, *(v)* $\mathtt{\Delta N}^{\triangleright}_{\mathsf{a},\{\mathsf{d}\}} = 2$.

In fact, none of the activities in $\{\mathsf{b}, \mathsf{c}\}$ occurred after a in 2 cases. It also holds true that none of the activities in $\{\mathsf{b}, \mathsf{c}, \mathsf{d}\}$ occurred after a in 1 case, and $\{\mathsf{b}, \mathsf{c}\} \subseteq \{\mathsf{b}, \mathsf{c}, \mathsf{d}\}$. Therefore,
$\mathtt{\Delta N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{c},\mathsf{d}\}} = 1$, and $\mathtt{\Delta N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{c}\}} = 1$, i.e., $\mathtt{\Delta N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{c}\}} = 2 - \mathtt{\Delta N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{c},\mathsf{d}\}}$.
By the same line of reasoning, since b did not occur after a in 4 cases, $\mathtt{\Delta N}^{\triangleright}_{\mathsf{a},\{\mathsf{b}\}} = 1$, i.e., $\mathtt{\Delta N}^{\triangleright}_{\mathsf{a},\{\mathsf{b}\}} = 4 - \mathtt{\Delta N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{c}\}} - \mathtt{\Delta N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{d}\}} - \mathtt{\Delta N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{c},\mathsf{d}\}}$.
The next section explains the procedure computing such values in detail.

The differential cumulative set-counters are used to compactly store the values to assign to interpretation functions. In the case of $\triangleright$, it is done as follows:

$$\triangleright_L(a, \mathrm{S}) =_\nu \sum_{\mathrm{T} \supseteq \mathrm{S}} \mathtt{\Delta N}^{\triangleright}_{a,\mathrm{T}}$$

In the example log, indeed,
$\triangleright_{\mathsf{L}}(\mathsf{a}, \{\mathsf{b}\}) =_\nu 4$, and $\triangleright_{\mathsf{L}}(\mathsf{a}, \{\mathsf{b}, \mathsf{c}\}) =_\nu 2$,
i.e.,
$\triangleright_{\mathsf{L}}(\mathsf{a}, \{\mathsf{b}\}) =_\nu \mathtt{\Delta N}^{\triangleright}_{\mathsf{a},\{\mathsf{b}\}} + \mathtt{\Delta N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{c}\}} + \mathtt{\Delta N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{c},\mathsf{d}\}}$, and
$\triangleright_{\mathsf{L}}(\mathsf{a}, \{\mathsf{b}, \mathsf{c}\}) =_\nu \mathtt{\Delta N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{c}\}} + \mathtt{\Delta N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{c},\mathsf{d}\}}$.

4.2.1. The main algorithm

Algorithm 1 shows the main algorithm that leads to the building of the knowledge base, based on a log alphabet Σ over a log L.

Notations and conventions. In the remainder of this section, we will assume that the concatenation operator $\circ$ is defined for sequences, i.e., given a sequence $\vec{s} = \left\langle s_1, \ldots, s_{|\vec{s}|} \right\rangle$ and an element s', then $\vec{s} \circ s' = \left\langle s_1, \ldots, s_{|\vec{s}|}, s' \right\rangle$. Since a trace of a log is defined as a sequence of events, $\circ$ also applies to appending events to traces. If we indicate with $\mathfrak{k} \in \mathfrak{K}$ the generic function of the set of functions $\mathfrak{K}$, the generic pairwise counter on activities $a, b \in \Sigma$ will be denoted as $\mathtt{N}^{\mathfrak{k}}_{a,b}$, and the generic differential cumulative set-counter on $a \in \Sigma, \mathrm{S} \subseteq \Sigma$ as $\mathtt{\Delta N}^{\mathfrak{k}}_{a,\mathrm{S}}$. As a

Algorithm 1: EVALUATEKB(Σ, L), the main algorithm for the building of the knowledge base

Input: A log alphabet Σ and a log $L = \langle \vec{t}_1, \ldots, \vec{t}_{|L|} \rangle \in \Sigma^*$, where $\vec{t}_i = \left\langle t_1, \ldots, t_{|\vec{t}|} \right\rangle$
Output: The knowledge base, whose values are assigned on the basis of Σ and L

for $i \leftarrow 1$ **to** $|L|$ **do**
    `/* Initialisation */`
    $\vec{t}_i^{\mathrm{R}} \leftarrow \vec{t}_i$ with events in reverse order
    foreach $a \in \Sigma, b \in \Sigma \setminus \{a\}$ **do** `// Reset of pairwise counters`
        $\mathtt{N}^{\mid\triangleright}_{a,b} \leftarrow 0$; $\mathtt{N}^{\triangleleft\mid}_{a,b} \leftarrow 0$; $\mathtt{N}^{\triangleleft\mid\triangleright}_{a,b} \leftarrow 0$; $\mathtt{N}^{\looparrowright}_{a,b} \leftarrow 0$; $\mathtt{N}^{\looparrowleft}_{a,b} \leftarrow 0$
    `/* Update of singleton counters */`
    foreach $a \in \Sigma : a \notin \vec{t}_i$ **do** `// Activities not occurring in trace` $\vec{t}_i$
        $\mathtt{N}^{\varnothing}_{a} \leftarrow \mathtt{N}^{\varnothing}_{a} + 1$
    for $j \leftarrow 1$ **to** $|\vec{t}_i|$ **do**
        $a \leftarrow t_{i_j}$
        $\mathtt{N}^{\#}_{a} \leftarrow \mathtt{N}^{\#}_{a} + 1$
    `/* Update of pairwise counters and differential cumulative set-counters */`
    foreach $\Delta\mathtt{N}^{\mid\triangleright}_{i_{a,S}} \in$ EVALMISSINGAFTER $(\Sigma, \vec{t}_i)$ **do** $\Delta\mathtt{N}^{\mid\triangleright}_{a,\mathrm{S}} \leftarrow \Delta\mathtt{N}^{\mid\triangleright}_{a,\mathrm{S}} \boxplus \Delta\mathtt{N}^{\mid\triangleright}_{i_{a,\mathrm{S}}}$
    foreach $\Delta\mathtt{N}^{\triangleleft\mid}_{i_{a,S}} \in$ EVALMISSINGBEFORE $(\Sigma, \vec{t}_i^{R})$ **do** $\Delta\mathtt{N}^{\triangleleft\mid}_{a,\mathrm{S}} \leftarrow \Delta\mathtt{N}^{\triangleleft\mid}_{a,\mathrm{S}} \boxplus \Delta\mathtt{N}^{\triangleleft\mid}_{i_{a,\mathrm{S}}}$
    foreach $\Delta\mathtt{N}^{\triangleleft\mid\triangleright}_{i_{a,S}} \in$ EVALMISSING $(\Sigma, \vec{t}_i)$ **do** $\Delta\mathtt{N}^{\triangleleft\mid\triangleright}_{a,\mathrm{S}} \leftarrow \Delta\mathtt{N}^{\triangleleft\mid\triangleright}_{a,\mathrm{S}} \boxplus \Delta\mathtt{N}^{\triangleleft\mid\triangleright}_{i_{a,\mathrm{S}}}$
    foreach $\Delta\mathtt{N}^{\looparrowright}_{i_{a,S}} \in$ EVALFOLLOWINGREPSINBETWEEN $(\Sigma, \vec{t}_i)$ **do** $\Delta\mathtt{N}^{\looparrowright}_{a,\mathrm{S}} \leftarrow \Delta\mathtt{N}^{\looparrowright}_{a,\mathrm{S}} \boxplus \Delta\mathtt{N}^{\looparrowright}_{i_{a,\mathrm{S}}}$
    foreach $\Delta\mathtt{N}^{\looparrowleft}_{i_{a,S}} \in$ EVALPRECEDINGREPSINBETWEEN $(\Sigma, \vec{t}_i^{R})$ **do** $\Delta\mathtt{N}^{\looparrowleft}_{a,\mathrm{S}} \leftarrow \Delta\mathtt{N}^{\looparrowleft}_{a,\mathrm{S}} \boxplus \Delta\mathtt{N}^{\looparrowleft}_{i_{a,\mathrm{S}}}$
    foreach $\mathtt{N}^{\blacktriangleright\blacktriangleright}_{i_{a,b}} \in$ EVALFOLLOWING $(\Sigma, \vec{t}_i)$ **do** $\mathtt{N}^{\blacktriangleright\blacktriangleright}_{a,b} \leftarrow \mathtt{N}^{\blacktriangleright\blacktriangleright}_{a,b} + \mathtt{N}^{\blacktriangleright\blacktriangleright}_{i_{a,b}}$
    foreach $\mathtt{N}^{\blacktriangleleft\blacktriangleleft}_{i_{a,b}} \in$ EVALPRECEDING $(\Sigma, \vec{t}_i)$ **do** $\mathtt{N}^{\blacktriangleleft\blacktriangleleft}_{a,b} \leftarrow \mathtt{N}^{\blacktriangleleft\blacktriangleleft}_{a,b} + \mathtt{N}^{\blacktriangleleft\blacktriangleleft}_{i_{a,b}}$

shorthand notation for sets of pairwise counters referring to all $a \in \Sigma, b \in \Sigma \setminus \{a\}$, we will adopt the usual pairwise counter notation, having a ${}_{\forall\forall}$ pedix in place of the referred activities (e.g., $\mathtt{N}^{\mid\triangleright}_{\forall\forall} = \bigcup_{a \in \Sigma, b \in \Sigma \setminus \{a\}} \mathtt{N}^{\mid\triangleright}_{a,b}$).

Description of the algorithm. The algorithm iterates over every trace of L. At every iteration i, the pairwise counters are reset to 0, and a variable $\vec{t}_i^{\mathrm{R}}$ keeps a clone of the trace under analysis, with events reversed in their original order. Thereafter, singleton counters are updated. For every activity $a \in \Sigma$ that does not occur in the trace under analysis, $\mathtt{N}^{\varnothing}_{a}$ is incremented by 1. In the sample trace $\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{c}, \mathsf{a} \rangle$, $\mathtt{N}^{\varnothing}_{\mathsf{a}} = 0$, because a occurs in it. $\mathtt{N}^{\varnothing}_{\mathsf{d}} = 1$ instead. For each activity a, counter $\mathtt{N}^{\#}_{a}$ is incremented by 1 every time a occurs. Thus, $\mathtt{N}^{\#}_{\mathsf{a}} = 4$ in the sample trace, since a occurs 4 times, whereas $\mathtt{N}^{\#}_{\mathsf{d}} = 0$. Consequently, we have the following, for every $a \in \Sigma$:

$$\#_L(a) =_{\nu_1} \mathtt{N}^{\#}_{a} \qquad (3)$$

and

$$\varnothing_L(a) =_{\nu_1} \mathtt{N}^{\varnothing}_{a}. \qquad (4)$$

Algorithm 2: Procedure Pairwise2Diff($\Sigma, \mathtt{N}^{\mathfrak{k}}_{\forall\forall}$), deriving differential cumulative set-counters from pairwise counters.

Input: A log alphabet Σ and a set of pairwise counters $\mathtt{N}^{\mathfrak{k}}_{\forall\forall}$
Output: A set of differential cumulative set-counters derived from $\mathtt{N}^{\mathfrak{k}}_{\forall\forall}$

foreach $a \in \Sigma$ **do**
  for $n \leftarrow 1$ **to** $\max\limits_{b \in \Sigma} \left\{ \mathtt{N}^{\mathfrak{k}}_{a,b} \subseteq \mathtt{N}^{\mathfrak{k}}_{\forall\forall} \right\}$ **do**
    $\mathrm{S}_n \leftarrow \left\{ b : \mathtt{N}^{\mathfrak{k}}_{a,b} \geqslant n \right\}$
  $\vec{n} = \{n : \mathrm{S}_n \neq \varnothing\}$ sorted by n descending
  for $i \leftarrow 1$ **to** $|\vec{n}| - 1$ **do**
    $n \leftarrow \vec{n}_i$
    $n' \leftarrow \vec{n}_{i+1}$
    $\Delta\mathtt{N}^{\mathfrak{k}}_{a,\mathrm{S}_n} \leftarrow (n - n')$
return $\bigcup\limits_{\substack{a \in \Sigma \\ n \in \vec{n}}} \Delta\mathtt{N}^{\mathfrak{k}}_{a,S_n}$

The computation of pairwise counters and corresponding differential cumulative set-counters are generally less trivial. Therefore, separate subsections follow that describe each dedicated procedure (EvalMissingAfter, EvalMissingBefore, ...). All such procedures except EvalFollowing and EvalPreceding return new sets of differential cumulative set-counters (each identified as $\Delta\mathtt{N}^{\triangleright}_{i\,a,\mathrm{S}}$, $\Delta\mathtt{N}^{\triangleleft}_{i\,a,\mathrm{S}}$, ...). Each element of these new sets are used to update the current value of the corresponding differential cumulative set-counter. We assume that all differential cumulative set-counters are initially assigned with a default value of 0. The addition operation over differential cumulative set-counters, $\boxplus$, is defined as follows:

$$\Delta\mathtt{N}^{\mathfrak{k}}_{a,\mathrm{S}} \boxplus \Delta\mathtt{N}^{\mathfrak{k}}_{i\,a',\mathrm{S}'} = \begin{cases} \Delta\mathtt{N}^{\mathfrak{k}}_{a,\mathrm{S}} + \Delta\mathtt{N}^{\mathfrak{k}}_{i\,a',\mathrm{S}'} & \text{if } a = a' \text{ and } \mathrm{S} = \mathrm{S}' \\ \Delta\mathtt{N}^{\mathfrak{k}}_{a,\mathrm{S}} & \text{otherwise} \end{cases}$$

Given an example log $\mathsf{L} = \{\langle \mathsf{a},\mathsf{a},\mathsf{b},\mathsf{a},\mathsf{c},\mathsf{a} \rangle, \langle \mathsf{a},\mathsf{a},\mathsf{b},\mathsf{a},\mathsf{c},\mathsf{a},\mathsf{d} \rangle, \langle \mathsf{c},\mathsf{a},\mathsf{a},\mathsf{d} \rangle\}$, the parsing of the first trace leads to the following values of the differential cumulative set-counters referred to activity a:
(i) $\Delta\mathtt{N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{c},\mathsf{d}\}} = 1$, *(ii)* $\Delta\mathtt{N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{d}\}} = 1$, and *(iii)* $\Delta\mathtt{N}^{\triangleright}_{\mathsf{a},\{\mathsf{d}\}} = 2$.
After the analysis of the second trace, we have:
(i) $\Delta\mathtt{N}^{\triangleright}_{\mathsf{a},\{\mathsf{b}\}} = 1$, *(ii)* $\Delta\mathtt{N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{c}\}} = 1$, *(iii)* $\Delta\mathtt{N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{c},\mathsf{d}\}} = 1$, *(iv)* $\Delta\mathtt{N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{d}\}} = 1$, and *(v)* $\Delta\mathtt{N}^{\triangleright}_{\mathsf{a},\{\mathsf{d}\}} = 2$.
Finally, the third trace leads to the following values:
(i) $\Delta\mathtt{N}^{\triangleright}_{\mathsf{a},\{\mathsf{b}\}} = 1$, *(ii)* $\Delta\mathtt{N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{c}\}} = 3$, *(iii)* $\Delta\mathtt{N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{c},\mathsf{d}\}} = 1$, *(iv)* $\Delta\mathtt{N}^{\triangleright}_{\mathsf{a},\{\mathsf{b},\mathsf{d}\}} = 1$, *(v)* $\Delta\mathtt{N}^{\triangleright}_{\mathsf{a},\{\mathsf{d}\}} = 2$.

Procedures EvalFollowing and EvalPreceding return instead sets of pairwise counters (each identified as $\mathtt{N}^{\blacktriangleleft\blacktriangleleft}_{i\,a,b}$ and $\mathtt{N}^{\blacktriangleright\blacktriangleright}_{i\,a,b}$). Therefore, the update operation is an addition. The following subsections explain in detail all the procedures that compute values for pairwise counters and differential cumulative

Algorithm 3: Procedure $\textsc{EvalMissingAfter}(\Sigma, \vec{t})$, evaluating subsequent missing occurrences of activities in a trace

Input: A log alphabet Σ and a trace $\vec{t} = \left\langle t_1, \ldots, t_{|\vec{t}|} \right\rangle$

Output: The set of differential cumulative set-counters $\Delta\mathtt{N}^{\not\rhd}$ derived from trace $\vec{t}$

for $i \leftarrow 1$ **to** $|\vec{t}|$ **do**
  $b \leftarrow t_i$
  foreach $a \in \Sigma \setminus \{b\}$ **do**
    $\mathtt{N}^{\not\rhd}_{a,b} \leftarrow 0$ // Flush operation ↓
    $\mathtt{N}^{\not\rhd}_{b,a} \leftarrow \mathtt{N}^{\not\rhd}_{b,a} + 1$

$\mathtt{N}^{\not\rhd}_{\forall\forall} \leftarrow \bigcup_{a \in \Sigma, b \in \Sigma \setminus \{a\}} \mathtt{N}^{\not\rhd}_{a,b}$

return $\textsc{Pairwise2Diff}\left(\Sigma, \mathtt{N}^{\not\rhd}_{\forall\forall}\right)$

	Trace					
	a	a	b	a	c	a
$\mathtt{N}^{\not\rhd}_{\mathsf{a,b}}$	1	2	↓	1		2
$\mathtt{N}^{\not\rhd}_{\mathsf{a,c}}$	1	2		3	↓	1
$\mathtt{N}^{\not\rhd}_{\mathsf{a,d}}$	1	2		3		4

(a) Computation of $\mathtt{N}^{\not\rhd}_{a,\cdot}$.

$\mathtt{N}^{\not\rhd}_{\mathsf{a},\cdot}$						$\Delta\mathtt{N}^{\not\rhd}_{\mathsf{a},\cdot}$
$\mathtt{N}^{\not\rhd}_{\mathsf{a,b}} =$	$1 +$	$\mathtt{N}^{\not\rhd}_{\mathsf{a,c}} =$	1	$\mathtt{N}^{\not\rhd}_{\mathsf{a,d}} =$	$1 +$	$\Rightarrow \Delta\mathtt{N}^{\not\rhd}_{\mathsf{a},\{\mathsf{b,c,d}\}} = 1$
	$1 =$				$1 +$	$\Rightarrow \Delta\mathtt{N}^{\not\rhd}_{\mathsf{a},\{\mathsf{b},\ \ \mathsf{d}\}} = 1$
					$2 =$	$\Rightarrow \Delta\mathtt{N}^{\not\rhd}_{\mathsf{a},\{\ \ \ \mathsf{d}\}} = 2$
	2				4	

(b) Computation of $\Delta\mathtt{N}^{\not\rhd}_{a,\cdot}$, given the values of $\mathtt{N}^{\not\rhd}_{a,\cdot}$.

Table 3: Computation of $\mathtt{N}^{\not\rhd}_{\mathsf{a},\cdot}$ and $\Delta\mathtt{N}^{\not\rhd}_{\mathsf{a},\cdot}$, given a sample trace: $\langle \mathsf{a, a, b, a, c, a} \rangle$.

set-counters. Each subsection concludes with the assignment of the formulation of the interpretation function, on the basis of the referring pairwise counter or differential cumulative set-counter.

4.2.2. Count of missing events after an activity

For evaluating $\not\rhd_L(a, \mathrm{S})$, procedure $\textsc{EvalMissingAfter}$ computes for every $b \in \Sigma \setminus \{a\}$ the value $\mathtt{N}^{\not\rhd}_{a,b}$. Algorithm 3 lists its pseudocode. Table 3a shows how $\mathtt{N}^{\not\rhd}_{\mathsf{a},\cdot}$ values are computed for $\langle \mathsf{a, a, b, a, c, a} \rangle$. $\mathtt{N}^{\not\rhd}_{a,b}$ is incremented by 1 every time a is read, while parsing the trace. When b is read, $\mathtt{N}^{\not\rhd}_{a,b}$ is reset to 0. The ↓ symbol indicates this operation ("flush"). At the end of the trace, the value stored in $\mathtt{N}^{\not\rhd}_{a,b}$ reports the occurrences of a after which no b occurred. In the example, we have $\mathtt{N}^{\not\rhd}_{\mathsf{a,b}} = 2$, $\mathtt{N}^{\not\rhd}_{\mathsf{a,c}} = 1$ and $\mathtt{N}^{\not\rhd}_{\mathsf{a,d}} = 4$.

The output of the procedure is a set of differential cumulative set-counters, obtained by invoking the $\textsc{Pairwise2Diff}$ procedure. Passing from pairwise counters to differential cumulative set-counters is a linear procedure, whose pseudocode is listed in Algorithm 2, for general sets of pairwise counters, and sketched in Table 3b, especially for $\Delta\mathtt{N}^{\not\rhd}$. For each $a \in \Sigma$, all pairwise counters $\mathtt{N}^{\mathfrak{t}}_{a,b}$ (for every $b \in \Sigma$) are indexed according to their value n. A set of activities S_n contains those $b \in \Sigma$ such that $\mathtt{N}^{\mathfrak{t}}_{a,b} \geqslant n$. In the example, considering a as

$\Delta\mathtt{N}^{\mathrel{|\!\!\rhd}}_{\mathsf{a},\cdot}$		$\Rightarrow$	$\mathrel{|\!\!\rhd}_{\mathsf{L}}(\mathsf{a},\cdot)$
$\{\mathsf{b},\ \mathsf{c},\ \mathsf{d}\} =$	1	$\Rightarrow$	$\mathrel{|\!\!\rhd}_{\mathsf{L}}(\mathsf{a},\{\mathsf{b},\mathsf{c},\mathsf{d}\}) =_{\nu} \mathrel{|\!\!\rhd}_{\mathsf{L}}(\mathsf{a},\{\mathsf{c},\mathsf{d}\}) =_{\nu} \mathrel{|\!\!\rhd}_{\mathsf{L}}(\mathsf{a},\{\mathsf{c}\}) = 1$
$\{\mathsf{b},\ \mathsf{d}\} =$	1	$\Rightarrow$	$\mathrel{|\!\!\rhd}_{\mathsf{L}}(\mathsf{a},\{\mathsf{b},\mathsf{d}\}) =_{\nu} \mathrel{|\!\!\rhd}_{\mathsf{L}}(\mathsf{a},\{\mathsf{b}\}) = 2$
$\{\mathsf{d}\} =$	2	$\Rightarrow$	$\mathrel{|\!\!\rhd}_{\mathsf{L}}(\mathsf{a},\{\mathsf{d}\}) = 4$

Table 4: Interpretation of $\mathrel{|\!\!\rhd}_{\mathsf{L}}(\mathsf{a},\cdot)$ for a w.r.t. all subsets of log alphabet $\Sigma = \{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d}\}$, given $\Delta\mathtt{N}^{\mathrel{|\!\!\rhd}}_{\mathsf{a},\cdot}$, for a w.r.t. $\{\mathsf{b},\mathsf{c},\mathsf{d}\}$, $\{\mathsf{b},\mathsf{d}\}$, and $\{\mathsf{d}\}$

the assignment of a, $\mathrm{S}_4 = \{\mathsf{d}\}$, $\mathrm{S}_2 = \{\mathsf{b},\mathsf{d}\}$, and $\mathrm{S}_1 = \{\mathsf{b},\mathsf{c},\mathsf{d}\}$, because $\mathtt{N}^{\mathrel{|\!\!\rhd}}_{\mathsf{a},\mathsf{d}} = 4$, $\mathtt{N}^{\mathrel{|\!\!\rhd}}_{\mathsf{a},\mathsf{b}} = 2$, and $\mathtt{N}^{\mathrel{|\!\!\rhd}}_{\mathsf{a},\mathsf{c}} = 1$. A sequence $\vec{n}$ is thus created that stores the values of pairwise counters in descending order. In the example, $\vec{n}$ is $\{4,2,1\}$. Elements of $\vec{n}$ are meant to act as an index for sets S_n. All elements in the sequence are indeed visited from the first to the second last. For each of them, a differential cumulative set-counter $\Delta\mathtt{N}^{\mathrel{|\!\!\rhd}}_{a,\mathrm{S}_n}$ is created that associates a to S_n. The value of $\Delta\mathtt{N}^{\mathrel{|\!\!\rhd}}_{a,\mathrm{S}_n}$ is assigned with $n - n'$, where n' is the following element in the list.

For example, in $\langle \mathsf{a},\mathsf{a},\mathsf{b},\mathsf{a},\mathsf{c},\mathsf{a}\rangle$, we have that $\Delta\mathtt{N}^{\mathrel{|\!\!\rhd}}_{\mathsf{a},\{\mathsf{b},\mathsf{c},\mathsf{d}\}} = 1$, $\Delta\mathtt{N}^{\mathrel{|\!\!\rhd}}_{\mathsf{a},\{\mathsf{b},\mathsf{d}\}} = 1$, and $\Delta\mathtt{N}^{\mathrel{|\!\!\rhd}}_{\mathsf{a},\{\mathsf{d}\}} = 2$. Table 3b shows the passage from $\left\{\mathtt{N}^{\mathrel{|\!\!\rhd}}_{\mathsf{a},\mathsf{b}}, \mathtt{N}^{\mathrel{|\!\!\rhd}}_{\mathsf{a},\mathsf{c}}, \mathtt{N}^{\mathrel{|\!\!\rhd}}_{\mathsf{a},\mathsf{d}}\right\}$ to $\Delta\mathtt{N}^{\mathrel{|\!\!\rhd}}_{\mathsf{a},\{\mathsf{b},\mathsf{c},\mathsf{d}\}}$, $\Delta\mathtt{N}^{\mathrel{|\!\!\rhd}}_{\mathsf{a},\{\mathsf{b},\mathsf{d}\}}$ and $\Delta\mathtt{N}^{\mathrel{|\!\!\rhd}}_{\mathsf{a},\{\mathsf{d}\}}$ for the sample trace. It is straightforward to see that the differential accumulation ($\Delta\mathtt{N}^{\mathrel{|\!\!\rhd}}_{a,\mathrm{S}}$) allows for keeping fewer values in memory (3 in the example) than the possible entries for the knowledge base ($\mathrel{|\!\!\rhd}_{L}(a,\mathrm{S})$, which amounts to 6). The memory saving is possible as we do not store information about those $\Delta\mathtt{N}^{\mathrel{|\!\!\rhd}}_{a,\mathrm{S}}$ that amount to 0 as, for instance, $\Delta\mathtt{N}^{\mathrel{|\!\!\rhd}}_{\mathsf{a},\{\mathsf{c},\mathsf{d}\}}$ in the example of Table 3b.

As previously said, the interpretation of $\mathrel{|\!\!\rhd}_{L}(a,\mathrm{S})$ can be derived from this compact data structures as follows:

$$\mathrel{|\!\!\rhd}_{L}(a,\mathrm{S}) =_{\nu} \sum_{\mathrm{T}\supseteq\mathrm{S}} \Delta\mathtt{N}^{\mathrel{|\!\!\rhd}}_{a,\mathrm{T}} \tag{5}$$

Table 4 shows the application of this derivation step for the sample trace.

4.2.3. Count of missing events before the occurrence of an activity

The technique seen for $\mathrel{|\!\!\rhd}_{L}(a,\mathrm{S})$ extends to the computation of $\mathrel{\lhd\!\!|}_{L}(a,\mathrm{S})$ with slight modifications. In fact, $\mathrel{\lhd\!\!|}_{L}(a,\mathrm{S})$ executes the procedures described above (i.e., computation of pairwise counters and derivation of differential cumulative set-counters, for every trace), although reversing the order in which the traces are parsed. We report the pseudocode in Algorithm 4 for the sake of completeness. Thus, e.g., the pairwise counter $\mathtt{N}^{\mathrel{\lhd\!\!|}}_{\mathsf{a},\mathsf{b}}$ is assigned with values in the same way in which $\mathtt{N}^{\mathrel{|\!\!\rhd}}_{\mathsf{a},\mathsf{b}}$ was computed, although parsing $\langle \mathsf{a},\mathsf{c},\mathsf{a},\mathsf{b},\mathsf{a},\mathsf{a}\rangle$ in place of $\langle \mathsf{a},\mathsf{a},\mathsf{b},\mathsf{a},\mathsf{c},\mathsf{a}\rangle$ (see Table 5). Thereafter, the differential cumulative set-counter $\Delta\mathtt{N}^{\mathrel{\lhd\!\!|}}_{a,\mathrm{S}}$ is derived from $\mathtt{N}^{\mathrel{\lhd\!\!|}}_{a,b}$, exactly as $\Delta\mathtt{N}^{\mathrel{|\!\!\rhd}}_{a,\mathrm{S}}$ is derived from $\mathtt{N}^{\mathrel{|\!\!\rhd}}_{a,b}$.

Algorithm 4: Procedure EvalMissingBefore($\Sigma, \vec{t}^{\mathrm{R}}$), evaluating preceding missing occurrences of activities in a reversed trace.

Input: A log alphabet Σ and a reversed trace $\vec{t}^{\mathrm{R}} = \left\langle t_1, \ldots, t_{|\vec{t}^{\mathrm{R}}|} \right\rangle$
Output: The set of differential cumulative set-counters $\Delta\mathtt{N}^{\nleftarrow}$ derived from reversed trace $\vec{t}^{\mathrm{R}}$

```
for i ← 1 to |t⃗^R| do
    b ← t_i
    foreach a ∈ Σ\{b} do
        N^⇷_{a,b} ← 0                      // Flush operation ↓
        N^⇷_{b,a} ← N^⇷_{b,a} + 1
N^⇷_{∀∀} ← ⋃_{a∈Σ, b∈Σ\{a}} N^⇷_{a,b}
return Pairwise2Diff(Σ, N^⇷_{∀∀})
```

	Reversed trace					
	a	c	a	b	a	a
$\mathtt{N}^{\nleftarrow}_{\mathsf{a,b}}$	1		2	↓	1	2
$\mathtt{N}^{\nleftarrow}_{\mathsf{a,c}}$	1	↓	1		2	3
$\mathtt{N}^{\nleftarrow}_{\mathsf{a,d}}$	1		2		3	4

(a) Computation of $\mathtt{N}^{\nleftarrow}_{\mathsf{a},\cdot}$.

$\mathtt{N}^{\nleftarrow}_{\mathsf{a},\cdot}$						$\Delta\mathtt{N}^{\nleftarrow}_{\mathsf{a},\cdot}$
$\mathtt{N}^{\nleftarrow}_{\mathsf{a,b}} =$	2	$\mathtt{N}^{\nleftarrow}_{\mathsf{a,c}} =$	2 +	$\mathtt{N}^{\nleftarrow}_{\mathsf{a,d}} =$	2 +	$\Rightarrow \Delta\mathtt{N}^{\nleftarrow}_{\mathsf{a},\{\mathsf{b,c,d}\}} = 2$
			1 =		1 +	$\Rightarrow \Delta\mathtt{N}^{\nleftarrow}_{\mathsf{a},\{\mathsf{c,d}\}} = 1$
					1 =	$\Rightarrow \Delta\mathtt{N}^{\nleftarrow}_{\mathsf{a},\{\mathsf{d}\}} = 1$
			3		4	

(b) Computation of $\Delta\mathtt{N}^{\nleftarrow}_{\mathsf{a},\cdot}$, given the values of $\mathtt{N}^{\nleftarrow}_{\mathsf{a},\cdot}$.

Table 5: Computation of $\mathtt{N}^{\nleftarrow}_{\mathsf{a},\cdot}$ and $\Delta\mathtt{N}^{\nleftarrow}_{\mathsf{a},\cdot}$, given a sample trace: $\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{c}, \mathsf{a} \rangle$.

After a trace has been completely parsed, $\Delta\mathtt{N}^{\nleftarrow}_{a,\mathrm{S}}$ is $\boxplus$-added to the differential cumulative set-counter. As a consequence, we have that:

$$\nleftarrow_L (a, \mathrm{S}) =_\nu \sum_{T \supseteq S} \Delta\mathtt{N}^{\nleftarrow}_{a,\mathrm{T}} \tag{6}$$

4.2.4. Count of missing events in the same trace in which an activity occurs

For what $\nleftrightarrow_L (a, \mathrm{S})$ is concerned, its computation is based on the differential cumulative set-counter $\Delta\mathtt{N}^{\nleftrightarrow}_{a,\mathrm{S}}$, in turn derived from pairwise counter $\mathtt{N}^{\nleftrightarrow}_{a,b}$. The pseudocode is listed in Algorithm 5 (procedure EvalMissing). $\mathtt{N}^{\nleftrightarrow}_{a,b}$ stores for each trace and each $a \in \Sigma$ either 0, if b occurs in the trace at least once, or the number of occurrences of a, if b did not occur in the trace. Referring to a and trace $\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{c}, \mathsf{a} \rangle$, we have that: *(i)* $\mathtt{N}^{\nleftrightarrow}_{\mathsf{a,b}} = 0$, *(ii)* $\mathtt{N}^{\nleftrightarrow}_{\mathsf{a,c}} = 0$, and *(iii)* $\mathtt{N}^{\nleftrightarrow}_{\mathsf{a,d}} = 4$. The accumulation of $\mathtt{N}^{\nleftrightarrow}_{a,b}$ in $\Delta\mathtt{N}^{\nleftrightarrow}_{a,\mathrm{S}}$ is performed in the same way seen for $\Delta\mathtt{N}^{\nrightarrow}_{a,\mathrm{S}}$ and $\Delta\mathtt{N}^{\nleftarrow}_{a,\mathrm{S}}$. It follows that:

$$\nleftrightarrow_L (a, \mathrm{S}) =_\nu \sum_{T \supseteq S} \Delta\mathtt{N}^{\nleftrightarrow}_{a,\mathrm{T}} \tag{7}$$

Algorithm 5: Procedure EVALMISSING$(\Sigma, \vec{t})$, evaluating missing co-occurrences of activities in a trace

Input: A log alphabet Σ and a trace $\vec{t} = \left\langle t_1, \ldots, t_{|\vec{t}|} \right\rangle$

Output: The set of differential cumulative set-counters $\Delta\mathtt{N}^{\nleftrightarrow}$ derived from trace $\vec{t}$

- **foreach** $a \in \Sigma, b \in \Sigma \setminus \{a\}$ **do** $?^{\nleftrightarrow}_{a,b} \leftarrow \top$
- **for** $i \leftarrow 1$ **to** $|\vec{t}|$ **do**
  - $a \leftarrow t_i$
  - **foreach** $a \in \Sigma \setminus \{b\}$ **do**
    - **if** $?^{\nleftrightarrow}_{a,b} = \top$ **then**
      - $\mathtt{N}^{\nleftrightarrow}_{a,b} \leftarrow \mathtt{N}^{\nleftrightarrow}_{a,b} + 1$
    - **if** $?^{\nleftrightarrow}_{b,a} = \top$ **then**
      - $\mathtt{N}^{\nleftrightarrow}_{b,a} \leftarrow 0$ `// Flush operation ↓`
      - $?^{\nleftrightarrow}_{b,a} \leftarrow \bot$
- $\mathtt{N}^{\nleftrightarrow}_{\forall\forall} \leftarrow \bigcup_{a \in \Sigma, b \in \Sigma \setminus \{a\}} \mathtt{N}^{\nleftrightarrow}_{a,b}$
- **return** PAIRWISE2DIFF$\left(\Sigma, \mathtt{N}^{\nleftrightarrow}_{\forall\forall}\right)$

4.2.5. Count of repeated occurrences of an activity before other events

For the computation of $\looparrowright_L (a, \mathrm{S})$, we here present a far more efficient calculation as opposed to the one used in [8]. The new algorithm follows the general framework seen so far (computation of values for pairwise counters $\mathtt{N}^{\looparrowright}_{a,b}$ first, then derivation of differential cumulative set-counters $\Delta\mathtt{N}^{\looparrowright}_{a,\mathrm{S}}$). Its pseudocode is reported in Algorithm 6 (procedure EVALFOLLOWINGREPSINBETWEEN). The input traces are sliced into sub-traces, at every new occurrence of a following the first one. Given, e.g., the sample trace $\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{c}, \mathsf{a}, \mathsf{d} \rangle$, it is sliced into the following sub-traces (see Table 6): *(i)* $\langle \mathsf{a} \rangle$, *(ii)* $\langle \mathsf{a}, \mathsf{b} \rangle$, *(iii)* $\langle \mathsf{a}, \mathsf{c} \rangle$, and *(iv)* $\langle \mathsf{a}, \mathsf{d} \rangle$. Thereafter, pairwise counter $\mathtt{N}^{\looparrowright}_{a,b}$ is computed for every sub-trace except the last one (sub-trace iv, $\langle \mathsf{a}, \mathsf{d} \rangle$, in the example). The calculation of $\mathtt{N}^{\looparrowright}_{a,b}$ is similar to the one of $\mathtt{N}^{\nrightarrow}_{a,b}$ for entire traces (see Section 4.2.2), with one exception: not all activities $b \in \Sigma$ are considered, but only those that occur in the trace under analysis. For instance, on a trace like $\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{c}, \mathsf{a} \rangle$, $\mathtt{N}^{\looparrowright}_{a,b}$ would not be computed for $b \in \{\mathsf{d}\}$. In the example of Table 6, $\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{c}, \mathsf{a}, \mathsf{d} \rangle$, sub-trace ii, $\langle \mathsf{a}, \mathsf{b} \rangle$, leads to the following values of $\mathtt{N}^{\looparrowright}_{a,b}$: $\mathtt{N}^{\looparrowright}_{\mathsf{a},\mathsf{b}} = 0$, $\mathtt{N}^{\looparrowright}_{\mathsf{a},\mathsf{c}} = 1$, and $\mathtt{N}^{\looparrowright}_{\mathsf{a},\mathsf{d}} = 1$. The rationale is, that for every pair of a's in the trace, the b event which misses in-between will eventually occur after at least two occurrences of a. Therefore, a is repeated at least twice before b. In fact, the last sub-trace is not considered in the computation of $\mathtt{N}^{\looparrowright}_{a,b}$, because the missing b represents an event which does not occur at all after a. However, this case is already covered by $\mathtt{N}^{\nrightarrow}_{a,b}$.

A new value for differential cumulative set-counter $\Delta\mathtt{N}^{\looparrowright}_{a,\mathrm{S}}$ is aggregated from $\mathtt{N}^{\looparrowright}_{a,b}$ for each $b \in \mathrm{S}$ at every slicing point, i.e., before the next occurrence of a. Thereafter, it is $\boxplus$-added to the preceding values. In the example, $\Delta\mathtt{N}^{\looparrowright}_{\mathsf{a},\{\mathsf{b},\mathsf{c},\mathsf{d}\}} = 1$ is calculated for subtrace i, $\langle \mathsf{a} \rangle$. Then, from subtrace ii ($\langle \mathsf{a}, \mathsf{b} \rangle$), $\Delta\mathtt{N}'^{\looparrowright}_{\mathsf{a},\{\mathsf{c},\mathsf{d}\}} = 1$ is

Algorithm 6: Procedure EvalFollowingRepsInBetween($\Sigma, \vec{t}$), evaluating missing co-occurrences of activities in a trace

Input: A log alphabet Σ and a trace $\vec{t} = \left\langle t_1, \ldots, t_{|\vec{t}|} \right\rangle$

Output: The set of differential cumulative set-counters $\Delta\mathtt{N}^{\looparrowright}$ derived from trace $\vec{t}$

- **foreach** $a \in \Sigma$ **do**
  - $?^{\looparrowright}_{a} \leftarrow \bot$ `// A flag checking whether` a `already occurred in the trace`
  - $\vec{s}_a \leftarrow \langle\rangle$ `// A sub-trace appending events after` a
- $S^i \leftarrow \{\}$ `// Stores pairs that index with activity` a `the activity sets before which` a `recurred`
- **for** $i \leftarrow 1$ **to** $|\vec{t}|$ **do**
  - $a \leftarrow t_i$
  - **if** $?^{\looparrowright}_{a} = \top$ **then**
    - `/* Increment the pairwise counters for activities not in the subtrace */`
    - **foreach** $b \in \Sigma \backslash \{a\}$ **do**
      - **if** $b \notin \vec{s}_a$ **then**
        - $\mathtt{N}^{\looparrowright}_{a,b} \leftarrow \mathtt{N}^{\looparrowright}_{a,b} + 1$
    - `/* Derive differential cumulative set-counters */`
    - **foreach** $\Delta\mathtt{N}^{\looparrowright}_{\vec{s}\,a,S} \in \textsc{Pairwise2Diff}\left(\Sigma, \bigcup_{b \in \Sigma \backslash \{a\}} \mathtt{N}^{\looparrowright}_{a,b}\right)$ **do**
      - $S^i \leftarrow S^i \cup \langle a, \mathrm{S}\rangle$
      - $\Delta\mathtt{N}^{\looparrowright}_{i\,a,\mathrm{S}} \leftarrow \Delta\mathtt{N}^{\looparrowright}_{i\,a,\mathrm{S}} \boxplus \Delta\mathtt{N}^{\looparrowright}_{\vec{s}\,a,\mathrm{S}}$
    - **foreach** $b \in \Sigma \backslash \{a\}$ **do** $\mathtt{N}^{\looparrowright}_{a,b} \leftarrow 0$; `// Reset the pairwise counters`
    - $\vec{s}_a \leftarrow \langle\rangle$ `// Reset the substring related to` a
  - **else**
    - $?^{\looparrowright}_{a} \leftarrow \top$
    - $\vec{s}_a \leftarrow \vec{s}_a \circ a$
  - **foreach** $b \in \Sigma \backslash \{a\}$ **do**
    - **if** $?^{\looparrowright}_{b} = \top$ **then**
      - $\vec{s}_b \leftarrow \vec{s}_b \circ a$
- **return** $\bigcup_{\langle a,S\rangle \in \mathcal{S}^i} \Delta\mathtt{N}^{\looparrowright}_{i\,a,S}$

computed. $\Delta\mathtt{N}^{\looparrowright}_{''\mathsf{a},\{\mathsf{b},\mathsf{d}\}} = 1$ stems from sub-trace iii, i.e., $\langle \mathsf{a}, \mathsf{c}\rangle$. It follows that:

$$\looparrowright_L (a, \mathrm{S}) =_\nu \sum_{\mathrm{T} \supseteq \mathrm{S}} \Delta\mathtt{N}^{\looparrowright}_{a,\mathrm{T}} \tag{8}$$

4.2.6. Count of repeated occurrences of an activity before other events on reversed traces

The calculation of $\looparrowleft_L (a, \mathrm{S})$ executes the operations described for $\looparrowright_L (a, \mathrm{S})$, reversing the order in which the trace is parsed. Thus, the pairwise counter $\mathtt{N}^{\looparrowleft}_{a,b}$ is assigned with values in the same way in which $\mathtt{N}^{\looparrowright}_{a,b}$ was computed, but parsing, e.g., $\langle \mathsf{d}, \mathsf{a}, \mathsf{c}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{a}\rangle$ in place of $\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{c}, \mathsf{a}, \mathsf{d}\rangle$ (see Table 7). Traces are divided into sub-traces at every occurrence of the activation. In the example, the reversed trace $\langle \mathsf{d}, \mathsf{a}, \mathsf{c}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{a}\rangle$ is thus sliced into: *(i)* $\langle \mathsf{d}, \mathsf{a}, \mathsf{c}\rangle$, *(ii)* $\langle \mathsf{a}, \mathsf{b}\rangle$, *(iii)* $\langle \mathsf{a}\rangle$, and *(iv)* $\langle \mathsf{a}\rangle$. The differential cumulative set-counter $\Delta\mathtt{N}^{\looparrowleft}_{a,\mathrm{S}}$ is derived from $\mathtt{N}^{\looparrowleft}_{a,b}$ exactly as $\Delta\mathtt{N}^{\looparrowright}_{a,\mathrm{S}}$ stems from $\mathtt{N}^{\looparrowright}_{a,b}$. Each time a sub-trace is parsed,

Sliced trace	$\mathtt{N}^{\looparrowright}_{\mathsf{a,b}}$	$\mathtt{N}^{\looparrowright}_{\mathsf{a,c}}$	$\mathtt{N}^{\looparrowright}_{\mathsf{a,d}}$	$\Delta\mathtt{N}^{\looparrowright}_{\mathsf{a},\cdot}$
$\langle \mathsf{a} \rangle \hookleftarrow$	1	1	1	$\Delta\mathtt{N}^{\looparrowright}_{\mathsf{a},\{\mathsf{b,c,d}\}} = 1$
$\langle \mathsf{a}, \mathsf{b} \rangle \hookleftarrow$		1	1	$\Delta\mathtt{N}'^{\looparrowright}_{\mathsf{a},\{\quad \mathsf{c,d}\}} = 1$
$\langle \mathsf{a}, \mathsf{c} \rangle \hookleftarrow$	1		1	$\Delta\mathtt{N}''^{\looparrowright}_{\mathsf{a},\{\mathsf{b},\quad \mathsf{d}\}} = 1$
$\langle \mathsf{a}, \mathsf{d} \rangle$				

Table 6: Computation of $\mathtt{N}^{\looparrowright}_{\mathsf{a},\cdot}$ and $\Delta\mathtt{N}^{\looparrowright}_{\mathsf{a},\cdot}$, given a sample trace: $\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{c}, \mathsf{a}, \mathsf{d} \rangle$. The $\hookleftarrow$ symbol indicates the point in which the trace has been split (i.e., before the next occurrence of a).

Sliced trace	$\mathtt{N}^{\looparrowleft}_{\mathsf{a,b}}$	$\mathtt{N}^{\looparrowleft}_{\mathsf{a,c}}$	$\mathtt{N}^{\looparrowleft}_{\mathsf{a,d}}$	$\Delta\mathtt{N}^{\looparrowleft}_{\mathsf{a},\cdot}$
$\langle \mathsf{d}, \mathsf{a}, \mathsf{c} \rangle \hookleftarrow$	1		1	$\Delta\mathtt{N}^{\looparrowleft}_{\mathsf{a},\{\mathsf{b},\quad \mathsf{d}\}} = 1$
$\langle \mathsf{a}, \mathsf{b} \rangle \hookleftarrow$		1	1	$\Delta\mathtt{N}'^{\looparrowleft}_{\mathsf{a},\{\quad \mathsf{c,d}\}} = 1$
$\langle \mathsf{a} \rangle \hookleftarrow$	1	1	1	$\Delta\mathtt{N}''^{\looparrowleft}_{\mathsf{a},\{\mathsf{b,c,d}\}} = 1$
$\langle \mathsf{a} \rangle$				

Table 7: Computation of $\mathtt{N}^{\looparrowleft}_{\mathsf{a},\cdot}$ and $\Delta\mathtt{N}^{\looparrowleft}_{\mathsf{a},\cdot}$, given a sample trace, $\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{c}, \mathsf{a}, \mathsf{d} \rangle$, which is reversed into $\langle \mathsf{d}, \mathsf{a}, \mathsf{c}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{a} \rangle$. The $\hookleftarrow$ symbol indicates the point in which the trace has been split (i.e., before the next occurrence of a).

$\mathtt{N}^{\looparrowleft}_{a,b}$ is reset to 0 for every $a, b \in \Sigma$. After the next sub-trace has been parsed, $\Delta\mathtt{N}^{\looparrowright}_{a,\mathrm{S}}$ is $\boxplus$-added by the differential cumulative set-counter. The last sub-trace in the reversed trace ($\langle a \rangle$, in the example) is not considered in the computation. Therefore, we have that:

$$\looparrowleft_L (a, \mathrm{S}) =_\nu \sum_{\mathrm{T} \supseteq \mathrm{S}} \Delta\mathtt{N}^{\looparrowleft}_{a,\mathrm{T}} \tag{9}$$

4.2.7. Count of events immediately following the occurrence of an activity

In order to compute the value of $\blacktriangleright\!\blacktriangleright_L (a, b)$, the pairwise counter $\mathtt{N}^{\blacktriangleright\blacktriangleright}_{a,b}$ is utilised. For each trace, $\mathtt{N}^{\blacktriangleright\blacktriangleright}_{a,b}$ stores the occurrences of b immediately following a, as described in Algorithm 7 (procedure EvalFollowing). In $\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{c}, \mathsf{a} \rangle$, e.g., $\mathtt{N}^{\blacktriangleright\blacktriangleright}_{\mathsf{a,b}} = 1$, $\mathtt{N}^{\blacktriangleright\blacktriangleright}_{\mathsf{a,c}} = 1$ and $\mathtt{N}^{\blacktriangleright\blacktriangleright}_{\mathsf{a,d}} = 0$. Traces in event logs are defined as *sequences* of events. As such, two events cannot be contemporary. Therefore, only one event can immediately follow the occurrence of an activity a. Owing to this, our technique does not require the usage of differential cumulative set-counters here:

$$\blacktriangleright\!\blacktriangleright_L (a, b) =_\nu \mathtt{N}^{\blacktriangleright\blacktriangleright}_{a,b} \tag{10}$$

The same observation holds true for the computation of $\mathtt{N}^{\blacktriangleleft\blacktriangleleft}_{a,b}$.

Algorithm 7: Procedure EVALMISSING$(\Sigma, \vec{t})$, evaluating missing co-occurrences of activities in a trace

Input: A log alphabet Σ and a trace $\vec{t} = \left\langle t_1, \ldots, t_{|\vec{t}|} \right\rangle$

Output: All values of $\not\Rightarrow$, interpreted over Σ and $\vec{t}$

for $i \leftarrow 1$ **to** $|\vec{t}|$ **do**
  if $i > 1$ **then** $a \leftarrow b$
  $b \leftarrow t_i$
  if $i > 1$ **then** $\mathtt{N}^{\blacktriangleright\blacktriangleright}_{a,b} \leftarrow \mathtt{N}^{\blacktriangleright\blacktriangleright}_{a,b} + 1$

$\mathtt{N}^{\blacktriangleright\blacktriangleright}_{\forall\forall} \leftarrow \bigcup\limits_{a \in \Sigma, b \in \Sigma \setminus \{a\}} \mathtt{N}^{\blacktriangleright\blacktriangleright}_{a,b}$

return $\mathtt{N}^{\blacktriangleright\blacktriangleright}_{\forall\forall}$

Target-Branched Declare constraint	Support
RespondedExistence(a, S)	$1 - \frac{\not\Leftrightarrow_L(a,\mathrm{S})}{\#_L(a)}$
Response(a, S)	$1 - \frac{\not\Rightarrow_L(a,\mathrm{S})}{\#_L(a)}$
AlternateResponse(a, S)	$1 - \frac{\not\Rightarrow_L(a,\mathrm{S}) + \looparrowright_L(a,\mathrm{S})}{\#_L(a)}$
ChainResponse(a, S)	$\frac{\sum_{b \in \mathrm{S}} \blacktriangleright\blacktriangleright_L(a,\mathrm{S})}{\#_L(a)}$
Precedence(S, a)	$1 - \frac{\not\Leftarrow_L(a,\mathrm{S})}{\#_L(a)}$
AlternatePrecedence(S, a)	$1 - \frac{\not\Leftarrow_L(a,\mathrm{S}) + \looparrowleft_L(a,\mathrm{S})}{\#_L(a)}$
ChainPrecedence(S, a)	$\frac{\sum_{b \in \mathrm{S}} \blacktriangleleft\blacktriangleleft_L(a,\mathrm{S})}{\#_L(a)}$

Table 8: Target-Branched Declare constraints and support functions.

4.2.8. Count of events immediately preceding the occurrence of an activity

The computation of $\blacktriangleleft\blacktriangleleft_L(a, b)$ takes advantage of pairwise counter $\mathtt{N}^{\blacktriangleleft\blacktriangleleft}_{a,b}$. For each trace, $\mathtt{N}^{\blacktriangleleft\blacktriangleleft}_{a,b}$ stores the occurrences of b immediately preceding a. Instructions of EVALPRECEDING are the same as EVALFOLLOWING (Algorithm 7) but applied to a reversed trace. In $\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{c}, \mathsf{a}, \mathsf{d} \rangle$, e.g., $\mathtt{N}^{\blacktriangleleft\blacktriangleleft}_{\mathsf{a},\mathsf{b}} = 1$, $\mathtt{N}^{\blacktriangleleft\blacktriangleleft}_{\mathsf{a},\mathsf{c}} = 1$ and $\mathtt{N}^{\blacktriangleleft\blacktriangleleft}_{\mathsf{a},\mathsf{d}} = 0$. To determine these values, the same technique adopted for $\mathtt{N}^{\blacktriangleright\blacktriangleright}_{a,b}$ can be utilised after reversing the trace. In the sample trace, $\langle \mathsf{d}, \mathsf{a}, \mathsf{c}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{a} \rangle$ would be parsed in place of $\langle \mathsf{a}, \mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{c}, \mathsf{a}, \mathsf{d} \rangle$:

$$\blacktriangleleft\blacktriangleleft_L(a, b) =_\nu \mathtt{N}^{\blacktriangleleft\blacktriangleleft}_{a,b} \tag{11}$$

4.3. Querying the Knowledge Base

Once the knowledge base is built, the support of constraints can be calculated. Table 8 lists the functions adopted to this end for each TBDeclare constraint. All queries build upon a Laplacian concept of probability with support being computed as the number of supporting cases divided by the total number of cases. In particular, the total number of cases is the count of occurrences of the activation $a \in \Sigma$ in the log $\#_L(a)$. For *ChainResponse*(a, S), supporting cases are those occurrences of a immediately followed by some $b \in \mathrm{S}$, i.e., $\blacktriangleright\!\blacktriangleright_L(a, b)$. Supporting cases can be summed up because if a is followed by a given $b \in \mathrm{S}$ in a trace, it cannot be immediately followed by any other event $c \in \mathrm{S}$. In other words, the two cases are mutually exclusive. However, this assumption does not hold true, e.g., for *Response*(a, S). Therefore, in this case, we consider the non-supporting cases, when a is *not* followed by any of the $b \in \mathrm{S}$, i.e., $\not\triangleright_L(a, \mathrm{S})$. We get that $P(E) = 1 - P(\overline{E})$ with $P(E)$ being the probability of E and $\overline{E}$ its negation. Hence, the support of *Response*(a, S) is $1 - \frac{\not\triangleright_L(a,\mathrm{S})}{\#_L(a)}$. Likewise, the support of *RespondedExistence*(a, S) is computed on the basis of the non-supporting cases. The support of *AlternateResponse*(a, S) is based on the cases when either *(i)* a is not followed by any $b \in \mathrm{S}$ ($\not\triangleleft\!\!\!\not\triangleright_L(a, \mathrm{S})$), or *(ii)* a occurs more than once before the first occurrence of $b \in \mathrm{S}$ ($\looparrowright_L(a, \mathrm{S})$). The two conditions are mutually exclusive. Therefore, it is appropriate to sum them up. Analogous considerations lead to the definition of support functions for *Precedence*(S, a), *AlternatePrecedence*(S, a) and *ChainPrecedence*(S, a).

4.4. Pruning the Returned Constraints

The power-set of activities in the log alphabet amounts to $2^{|\Sigma|-1}$. Therefore, if we name the number of TBDeclare templates as N, up to $N \times 2^{|\Sigma|-1}$ constraints can potentially hold true. When a maximum limit of the branching factor β to the cardinality of the set is imposed, this number is reduced to

$$|\Sigma| \times N \times \sum_{i=1}^{\min\{\beta\,,\,|\Sigma|-1\}} \binom{|\Sigma|-1}{i}$$

However, even with branching factor set to 3 and $|\Sigma| = 10$, already 3,087 constraints have to be evaluated. A model including such a number of constraints would be hardly comprehensible for humans [26, 27]. In order to reduce this number, we adopt pruning based on set-dominance and on hierarchy subsumption.

4.4.1. Pruning Based on Set-Dominance.

The idea of this pruning approach is that if, e.g., *Response*$(a, \{b, c\})$ and *Response*$(a, \{b, c, d\})$ have the same support, the first is more informative than the second. Indeed, stating that "if a is executed then either b or c would eventually follow", entails that also "either b, c or d would eventually follow". In general terms, the support of TBDeclare constraints that are instantiations of the same template and share the activation increases according to the set-containment relation of target activities (see Corollary 1). To this end, the mining algorithm distributes the discovered constraints, along with their computed support, on a structure like the Hasse Diagram of Figure 2. This is a Direct-acyclic graph, such that a breadth-first search can be implemented. For each constraint, the pruning technique visits the nodes, from the biggest in size

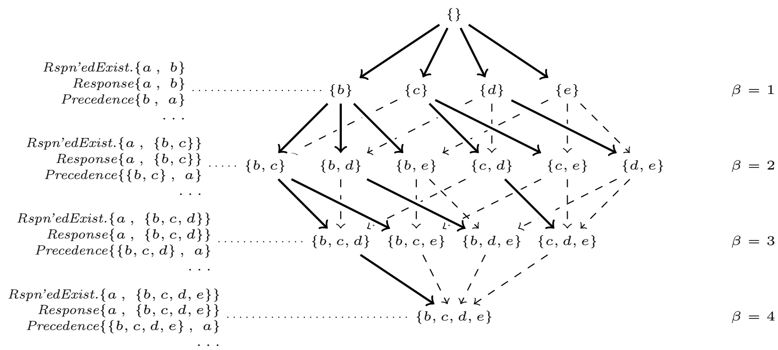


Figure 2: A Hasse Diagram representing the Partial Order set containment relation. Containing sets are at the head of connecting arcs, contained sets are at the tail.

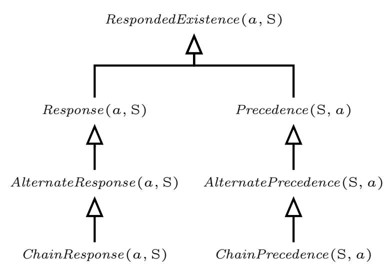


Figure 3: Diagram showing the subsumption hierarchy relation. Constraints that are subsumed are at the tail.

to the smallest. For instance, it can start from *Response*$(a, \{b, c, d, e\})$, i.e., the sink node, if the branching factor is equal to the size of the log alphabet. Given the current node, it checks whether in one of the parent nodes a constraint is stored (i.e., *Response*$(a, \{b, c, d\})$, *Response*$(a, \{b, c, e\})$, *Response*$(a, \{b, d, e\})$, *Response*$(a, \{c, d, e\})$) with greater or equal support. If so, it marks the current as redundant, and proceeds the visit towards the parent nodes that are not already marked as redundant. Otherwise, it marks all the ancestors as redundant. The parsing ends when either *(i)* the visit reaches the root node or *(ii)* no parent, which is not already marked as redundant, is available for the visit.

4.4.2. Pruning Based on Hierarchy Subsumption.

As investigated in [4, 22, 28], Declare constraints are not independent, but partially form a subsumption hierarchy. We consider a constraint $\mathcal{C}(a, \mathrm{S})$ subsumed by another constraint $\mathcal{C}'(a, \mathrm{S})$ when all the traces that comply with $\mathcal{C}(a, \mathrm{S})$ also comply with $\mathcal{C}'(a, \mathrm{S})$. *Response*$(a, \mathrm{S})$, e.g., is subsumed by *RespondedExistence*(a, S). Figure 3 depicts the subsumption hierarchy for TBDeclare constraints. It follows that a subsumed constraint always has a support which is less than or equal to the subsuming one. This pruning technique aims at keeping those constraints that are the most restrictive, among the most supported. Therefore, it labels as redundant every constraint C which is at the same time *(i)* subsumed by another constraint C', and *(ii)* having a lower support than C'. Therefore, if, e.g, given a log L defined over a log alphabet Σ s.t. $a \in \Sigma$ and $\mathrm{S} \subseteq \Sigma$, $\mathscr{S}_L(\mathit{RespondedExistence}(a, \mathrm{S})) > \mathscr{S}_L(\mathit{Response}(a, \mathrm{S}))$, then $\mathit{Response}(a, \mathrm{S})$ is marked as redundant. However, if $\mathscr{S}_L(\mathit{RespondedExistence}(a, \mathrm{S})) = \mathscr{S}_L(\mathit{Response}(a, \mathrm{S}))$, then $\mathit{Response}(a, \mathrm{S})$ is preferred. This is due to the fact that more restrictive constraints hold more information than the less restrictive ones. The pruning approach is based on the monotone non-decrement of support (cf. Figure 3). It operates as follows. Starting from the root of the hierarchy tree, if a constraint has a support equal to one of the children, it is marked as redundant and the visit proceeds with the children. If a child has a support which is lower than the parent, it is marked as redundant. All its children will be automatically marked as redundant as well, as they cannot have a higher support.

Both pruning techniques complement one another in reducing the set of the discovered constraints.

5. Experiments and Evaluation

In this section, we investigate the efficiency and effectiveness of our approach. In particular, we compare the performances of the new proposed algorithms w.r.t. the ones described in [8]. Section 5.1 shows the results obtained by applying the proposed technique to synthetic logs. Section 5.2 validates our approach by using event logs from a process to solve disruptions of ICT-services in the Rabobank Netherlands Group ICT and from a loan application process of a Dutch financial institute. All experiments were run on a server machine equipped with Intel Xeon CPU E5-2650 v2 2.60GHz, using 1 64-bit CPU core and 16GB main memory quota.

5.1. Evaluation Based on Simulation

To test the effectiveness and the efficiency of our approach, we have defined a simple Declare model including the following constraints:

- *ChainPrecedence*({a,b},c)
- *ChainPrecedence*({a,b,d},c)
- *AlternateResponse*(a,{b,c})
- *RespondedExistence*(a,{b,c,d,e})
- *Response*(a,{b,c})
- *Precedence*({a,b,c,d},e)

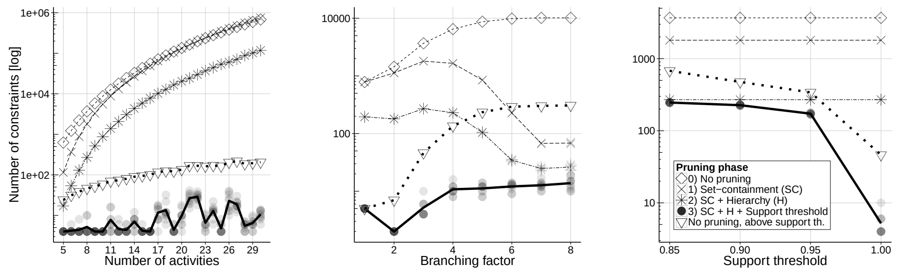
and we have simulated it to generate a compliant event log as described in [4]. In our experiments, we focus on different characteristics of the discovery task including average length of the traces, number of traces, and number of activities. Moreover, we consider characteristics of the discovered model including minimum support and maximum branching factor. In our experiments, we have run the algorithm varying the value of one variable at a time. The remaining variables were fixed and corresponding to 4 and 25 for minimum and maximum trace length respectively, 10,000 for log size, 8 for log alphabet size, 1.0 for support threshold, and 3 for branching factor. Each configuration has been averaged over 10 randomly generated logs.

Branch.F	Supp.T	Equal	Restr.	None	Branch.F	Supp.T	Equal	Restr.	None
1	0.85	0	1	13	5	0.85	2	1	85.6
	0.9	0	1	12.6		0.9	2	1	86.9
	0.95	0	1	9.1		0.95	2	1	81.7
	1	0	0	0		1	2	1	17
2	0.85	2	4.1	95.9	6	0.85	2	1	28.3
	0.9	2	3.4	73.9		0.9	2	1	25.8
	0.95	2	2	69.3		0.95	2	1	22.9
	1	2	0	0		1	2	1	15.8
3	0.85	2	3	232.2	7	0.85	2	1	23.2
	0.9	2	3	209.1		0.9	2	1	19.4
	0.95	2	2.8	159.7		0.95	2	1	18.8
	1	2	1	2.4		1	2	1	16.8
4	0.85	2	1	203.7	8	0.85	2	1	24.5
	0.9	2	1	202.2		0.9	2	1	21.1
	0.95	2	1	186.9		0.95	2	1	18.5
	1	2	1	10		1	2	1	15.1

Table 9: Summary of matching constraints in the mined process.

Effectiveness. First, we demonstrate the effectiveness of our approach by investigating the reduction effect of the proposed pruning techniques. In particular, we analyse the trend of the variable "number of discovered constraints" as a function of log alphabet size, branching factor, and support threshold, in logarithmic scale.

Figure 4a shows the trend of the number of discovered constraints by varying the log alphabet size. Different curves refer to different configurations of the miner: without any pruning (diamonds); with set-containment-based pruning (crosses); with set-containment- and hierarchy-based pruning (asterisks); with set-containment- and hierarchy-based pruning and support threshold (points); with support threshold only (triangles). This plot provides evidence that as the number of activities in the log alphabet increases, the number of discovered con-



(a) Number of discovered constraints as function of the log alphabet size

(b) Number of discovered constraints as function of the branching factor

(c) Number of discovered constraints as function of the support threshold

Figure 4: Effectiveness tests performed on synthetic logs.

straints increases as well. However, we discover a lower increase of constraints as we proceed further in the sequence of pruning techniques. Moreover, there is a significant difference between the number of discovered constraints with filtering based on the minimum support threshold only, and based on the pruning techniques presented in this paper. This improvement yields a reduction ratio of 94.84% (205.6 versus 10.6, on average), for a log alphabet size of 30.

Figure 4b shows the trend of the number of discovered constraints by varying the branching factor. Without pruning, or with the simple filtering by minimum support threshold, the number of discovered constraints increases as the number of branches increases. On the other hand, when we apply the set-dominance and hierarchy-based pruning techniques, the number of discovered constraints is approximately constant up to a branching value of 3. After this value, the number of constraints decreases. When we apply all the proposed pruning techniques together, the number of constraints eventually increases. In addition, the number of constraints obtained by applying set-dominance and subsumption hierarchy converges to the number of constraints discovered when all the pruning techniques are applied together. The difference between the number of discovered constraints with support threshold and the number of discovered constraints after using the pruning techniques presented in this paper is quantified (branching factor of 8) in a reduction ratio of 95.51% (307.7 versus 13.8, on average).

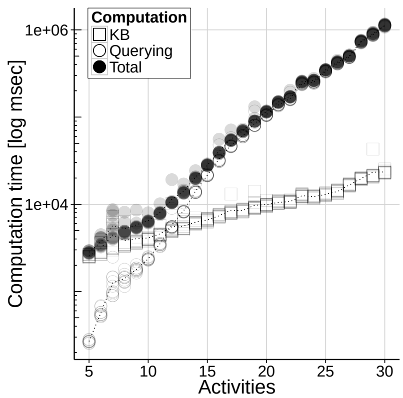
The plot in Figure 4c confirms that for any threshold between 0.85 and 1.0, the number of constraints discovered by applying all the pruning techniques is lower than the one obtained by applying the support threshold filtering only. The reduction ratio is indeed 88.74% (46.2 versus 5.2, on average) when the threshold is set to 1.0.

An additional experiment to test the effectiveness of our approach is illustrated in Table 9. Here, for different values of branching factor (ranging from 1 to 8) and support threshold (ranging from 0.85 to 1), we evaluate the capability of the discovery algorithm to rediscover the model that was used for log generation. In particular, for each combination of branching factor and support threshold, we have generated 10 random logs starting from the model described at the beginning of this section. Then, we have considered the average number of constraints correctly discovered (column *Equal* in the table) and the average number of discovered constraints that strengthen one of the constraints of the original model (column *Restr.* in the table).[3] In column *None* in the table, we show the average number of additional constraints discovered. These constraints are characteristic of each specific (random) log but still compliant with the original model. Note that the branching factor affects the number of constraints correctly discovered since, for example, if we specify a maximum branching factor equal to 2, it will be impossible to discover a constraint with 3 branches.

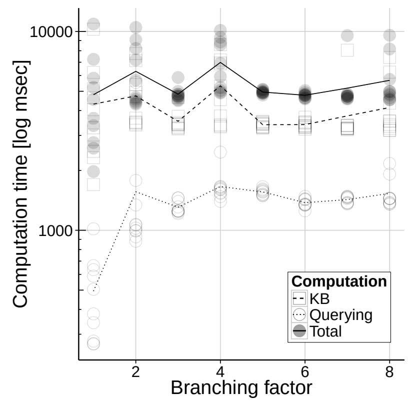
The constraints correctly discovered with branching factor equal to 2 and support threshold equal to 1 are *ChainPrecedence*({a,b},c) and *AlternateResponse*(a,{b,c}). This model contains the only constraints that can be correctly discovered using a branching factor of 2. Indeed, the third constraint with 2 branches in the original model is *Response*(a,{b,c}), which is entailed by *AlternateResponse*(a,{b,c}). The constraints correctly discovered with branching factor equal to 3 and support equal to 1 are, again, *ChainPrecedence*({a,b},c) and *AlternateResponse*(a,{b,c}). However, in this case, also *Precedence*({a,b,d},e), restriction of *Precedence*({a,b,c,d},e), is discovered. This result improves the original models that contains a redundancy. Indeed, in all cases in which e is preceded by c, it is also preceded by a or by b due to *ChainPrecedence*({a,b},c). Starting from a branching factor of 3 up to a branching factor of 8, these 3 constraints are always part of the discovered models. This confirms the effectiveness of the proposed approach since this set of constraints corresponds to the original set after removing redundancies.

Efficiency. Figure 5 shows the efficiency of our approach by plotting the computation time as a function of log alphabet size, branching factor, log size, and average trace size. Figure 5a shows the trend of the computation time (in logarithmic scale) by varying the log alphabet size. Different curves refer to the computation time for *(i)* the knowledge base construction, *(ii)* the querying on the knowledge base, and *(iii)* to the total computation time. Notice that there is a break point, when the log alphabet is composed of 12 activities: there the query time becomes higher than the knowledge base construction time. In Figure 5b, we can see that the computation time (here displayed in logarithmic scale) does not depend on the branching factor. It is approximately constant and higher for querying the knowledge base. Figure 5c shows the trend of the computation time by varying the log size, whereas Figure 5d depicts the trend of the computation time by varying the average trace size (both displayed in linear scale). In both cases, the query time clearly outperforms the knowledge

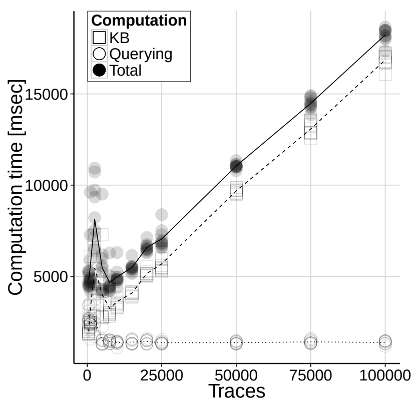
[3]Note that there is also the possibility, in some cases, that the discovered model contains constraints that are entailed by one of the constraints of the original model. However, this happens very rarely using randomly generated logs and it never occurred in our experiments.



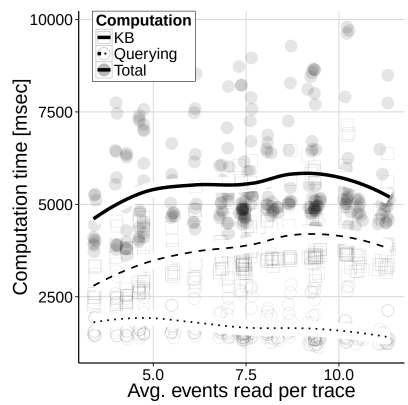
(a) Computation time v. log alphabet



(b) Computation time v. branching factor



(c) Computation time v. the number of traces in the log

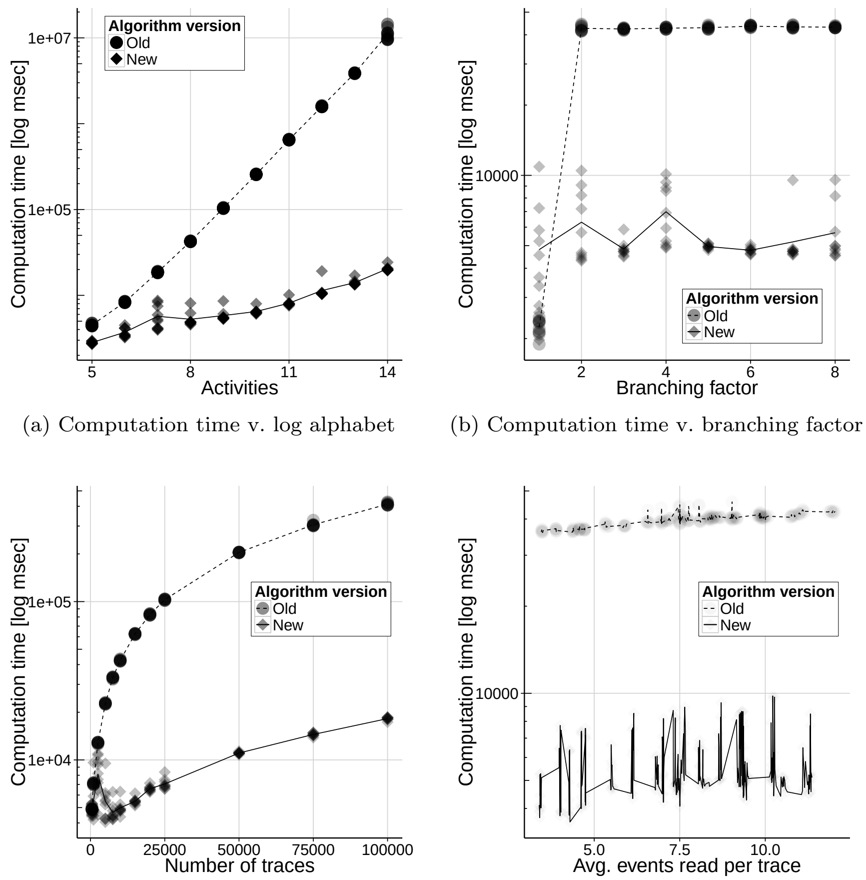


(d) Computation time v. trace length

Figure 5: Efficiency tests performed on synthetic logs, comparing the computation time needed for building the knowledge base ("KB" time), and for deriving the constraints ("querying" time).

base construction time. Generally speaking, the only factor that makes queries less efficient than the knowledge base construction is the size of the alphabet.

In Figure 6, we compare the time performances of the new version of the discovery algorithm, w.r.t. the version presented in [8]. In particular, we plot the computation time as a function of log alphabet size, branching factor, log size, and average trace size, in logarithmic scale. For all these parameters, the plots highlight the dramatic reduction of the computation time when using the new proposed approach. The main factor that contributed to the performance improvement is the new algorithm adopted for the computation of



(a) Computation time v. log alphabet

(b) Computation time v. branching factor

(c) Computation time v. the number of traces in the log

(d) Computation time v. trace length

Figure 6: Efficiency tests performed on synthetic logs, comparing the time performances of the new version of the discovery algorithm versus the version of [8].

AlternateResponse and *AlternatePrecedence* constraints.

5.2. Evaluation Based on Real Data

In this section, we validate our approach using real-life logs. The results are described in the following sections. The first log we use has been provided by a Dutch financial institute. The second log has been provided by the Rabobank Netherlands Group ICT.

5.2.1. A Dutch Financial Institution

We have evaluated the applicability of our approach using a real-life event log provided for the BPI challenge 2012 [29]. The event log pertains to an application process for personal loans or overdrafts of a Dutch financial institution. It contains 262 200 events distributed across 24 different possible activities and includes 13 087 cases.

In this case, it is possible to prune the list of discovered constraints in order to obtain a compact set of constraint, which is understandable for human analysts. By applying the miner with a support threshold equal to 1, confidence threshold set to 0.8, and branching factor 5, we obtain the following 11 constraints:

ChainResponse(A_SUBMITTED, A_PARTLYSUBMITTED)
AlternateResponse(A_SUBMITTED, {A_PREACCEPTED,A_DECLINED,A_CANCELLED})
AlternateResponse(A_SUBMITTED, {A_PREACCEPTED,A_DECLINED,W_Afhandelen leads})
AlternateResponse(A_SUBMITTED, {W_Completeren aanvraag,A_DECLINED,A_CANCELLED})
AlternateResponse(A_SUBMITTED, {W_Completeren aanvraag,A_DECLINED,W_Afhandelen leads})
ChainPrecedence(A_SUBMITTED, A_PARTLYSUBMITTED)
AlternateResponse(A_PARTLYSUBMITTED, {A_PREACCEPTED,A_DECLINED,A_CANCELLED})
AlternateResponse(A_PARTLYSUBMITTED, {A_PREACCEPTED,A_DECLINED,W_Afhandelen leads})
ChainResponse(A_PARTLYSUBMITTED, {A_PREACCEPTED,A_DECLINED,W_Afhandelen leads,W_Beoordelen fraude})
AlternateResponse(A_PARTLYSUBMITTED, {W_Completeren aanvraag,A_DECLINED,A_CANCELLED})
AlternateResponse(A_PARTLYSUBMITTED, {W_Completeren aanvraag,A_DECLINED,W_Afhandelen leads})

This results are in line with what described in the report published by the winners of the BPI challenge 2012 [30]. For example, one of the results discussed in this report is that each case starts with an application of a customer where an application is first submitted and, immediately after, partly submitted. In addition, over 13 087 cases in the log, in 4 852 cases, an application partly submitted is immediately pre-accepted, in 3 429 cases it is immediately declined and in the remaining cases is followed up (through activities corresponding to events $A\_Afhandelenleads$ or $A\_Beoordelenfraude$). This is in line with the $ChainResponse$ constraints discovered.

5.2.2. Rabobank

The case study we illustrate in this section has been provided for the BPI challenge 2014 by the Rabobank Netherlands Group ICT [9]. The log we use pertains to the management of calls or mails from customers to the Service Desk concerning disruptions of ICT-services. The log contains 46 616 cases, 466 737 events referring to 39 different activities. There are 242 originators and domain specific event attributes like `KM number`, `Interaction ID` and `IncidentActivity_Number`.

By applying the miner with a support threshold equal to 1, confidence threshold set to 0.8, and branching factor 5, we obtain the following 18 constraints:

Precedence({*Reassignment, Operator Update, Update from customer, Open*},*Assignment*)

RespondedExistence(*Assignment*,{*Reassignment, Closed, Pending vendor*})
RespondedExistence(*Assignment*,{*Reassignment, Open*})
RespondedExistence(*Assignment*,{*Operator Update, Update from customer, Open*})
RespondedExistence(*Assignment*,{*Update from customer, Caused By CI, Open*})
RespondedExistence(*Assignment*,{*Update from customer, Description Update, Open*})
RespondedExistence(*Assignment*,{*Update from customer, Update, Open*})
RespondedExistence(*Assignment*,{*Update from customer, OO Response, Open*})
RespondedExistence(*Assignment*,{*Closed, Status Change*})
RespondedExistence(*Assignment*,{*Closed, External Vendor Assignment*})
RespondedExistence(*Assignment*,{*Closed, Pending vendor, Vendor Reference*})
RespondedExistence(*Assignment*,{*Closed, Open*})
RespondedExistence(*Assignment*,{*Caused By CI, Resolved, Open*})
Response(*Open*,{*Reassignment, Closed, Pending vendor*})
Response(*Open*,{*Assignment, Closed, Pending vendor*})
Response(*Open*,{*Closed, Status Change*})
Response(*Open*,{*Closed, External Vendor Assignment*})
Response(*Open*,{*Closed, Pending vendor, Vendor Reference*})

From further analysis of the log (see also `http://www.win.tue.nl/bpi/2014/challenge`), it is possible to verify that these results reflect the reality. For example, over 46 616 cases in the log, only in 449 cases an opened incident is not eventually closed. These 449 cases always contain a status change and an external vendor assignment. Only 447 of them contain a pending vendor and the remaining 2 as well as a status change and an external vendor assignment both contain an assignment, a reassignment and a vendor reference. This is in line with the list of *Response* constraints discovered.

6. Related Work

Process Mining [31] is the set of techniques for the extraction of process descriptions, stemming from a set of recorded real executions (*event logs*). ProM [32] is one of the most used plug-in based software environments for implementing process mining techniques. Process Mining mainly covers three different aspects: process discovery, conformance checking and operational support. The first aims at discovering the process model from logs. Control-flow mining in particular focuses on the causal and sequential relations among activities. The second focuses on the assessment of the compliance of a given process model with event logs, and the possible enhancement of the process model in this regard. The third is finally meant to assist the enactment of processes at run-time, based on given process models.

From [33] onwards, many techniques have been proposed for the control-flow mining: pure algorithmic (e.g., α algorithm, drawn in [34] and its evolution α^{++} [35]), heuristic (e.g., [36]), genetic (e.g., [37]), etc. A very smart extension to the previous research work was achieved by the two-steps algorithm proposed in [38]. Differently from the former approaches, which typically provide a single process mining step, it splits the computation in two phases: *(i)* the configurable mining of a Transition System (TS) representing the process behavior and *(ii)* the automated construction of a Petri net bisimilar to the TS [39, 40]. In the field of conformance checking, [41, 42, 43] have proposed techniques capable of realigning procedural process models to logs.

The need for flexibility in the definition of some types of process, such as the knowledge-intensive processes [44], has led to an alternative to the classical "procedural" approach: the "declarative" approach. Rather than using a procedural language for expressing the allowed sequences of activities ("closed" models), it is based on the description of workflows through the usage of constraints: the idea is that every task can be performed, except what does not respect such constraints ("open" models). The work of van der Aalst et al. [27] showed how the declarative approach (such as the one adopted by Declare [45]) could help in obtaining a fair trade-off between flexibility in managing collaborative processes and support in controlling and assisting the enactment of workflows. The original semantics of Declare used in these works is based on LTL_f. Other semantics for Declare have been proposed in [46] (based on the Event Calculus) and in [47, 48, 49] (based on Dynamic Condition Response Graphs). Very recent investigations have compared the procedural and the declarative paradigms and discussed the possibility of adopting hybrid approaches based on both procedural and declarative models [50, 51, 11, 52, 53, 54, 55].

Our work contributes to the area of declarative process mining. In this context, Maggi et al. [5] first proposed an unsupervised algorithm for mining Declare processes. They based the discovery of constraints on the replay of the log on specific automata, each accepting only those traces that are compliant to one constraint. Candidate constraints are generated considering all the instantiations of Declare templates with activities that occur in the log. Each constraint among the candidates becomes part of the discovered process only if the percentage of traces accepted by the related automaton exceeds a user-defined threshold. In order to remove irrelevant constraints from the output set, the authors apply vacuity detection techniques [56]. Constraints are considered as vacuously satisfied when no trace in the log violates them, yet no trace shows the effect of their application either. A vacuously satisfied constraint is, e.g., that every request is eventually acknowledged, in a process instance that does not contain requests.

[6] describes an evolution of [5], with the adoption of a two-phase approach. The first phase is based on the Apriori algorithm, developed by Agrawal and Srikant for mining association rules [20]. During this preliminary phase, the frequent sets of correlated activities are identified. The candidate constraints are computed on the basis of the correlated activity sets only. During the second phase, the candidate constraints are checked as in [5]. Therefore, the search space for the second phase is reduced. In output, constraints constituting the discovered process are weighted according to their support, i.e., the probability of such constraints to hold in the mined process. To filter out irrelevant constraints, more metrics are introduced, such as confidence and interest factor. Both the concepts of support and confidence have been adopted in this paper.

In [28], Maggi et al. refined the technique of [6] by pruning returned constraints on the basis of three main methods: *(i)* the removal of weaker constraints entailed by stronger constraints; *(ii)* the reparation of predefined basic Declare models; *(iii)* an ontology-guided search for constraints, linking activities that either belong to different groups of interest, or to the same group. The last two require the user input, whereas the first does not. A technique for pruning an existing Declare model based on event correlations has been presented in [57].

All the aforementioned "automata based" methods have been implemented in [58]. Unfortunately, none of these methods turned out to be practical in our context. This is mainly due to their checking algorithm, based on the replay of the log on one automaton for each candidate constraint. In TBDeclare, the search space of candidate constraints would be much too vast to make this approach feasible.

[59, 60, 61] describe the usage of inductive logic programming techniques to mine models expressed as a SCIFF [62] first-order logic theory, consisting of a set of implication rules named Social Integrity Constraints (IC's for short). To complete the Declare discovery, the learned theory is automatedly translated into Declare notation. [63, 64] extend this technique by weighting in a second phase the constraints with a probabilistic estimation. The learned IC's are indeed translated from SCIFF, discovered by DPML, into Markov Logic formulae [65]. Their probabilistic-based weighting is computed by the Alchemy tool [64]. Both the techniques in [59] and [64] rely on the availability of compliant and non-compliant traces of execution, w.r.t. the process to mine. As in the aforementioned "logic-based" approaches, we preferred to elaborate a technique which avoided the replay of every trace on automata in the log. On the other hand, we had to deal with traces which were not labeled in advance. Therefore, our technique does not require the user's specification of positive and negative past executions.

The third branch of Declare mining algorithms, alternative to the automata- and logic-based, is the one that started with [3]. It is based on a two-step approach. The first step computes statistic data describing the occurrences of activities and their interplay in the log. The second one checks the validity of Declare constraints by querying such a statistic data structure (knowledge base). [4] extends such an approach by weighing each constraint with reliability and interest metrics, such as support and confidence. [21] shows the boost in performance that such algorithm allowed, w.r.t. the automata-based approaches and [66] reports on its application in the context of highly flexible processes [44]. Although fast, these algorithms do not broaden the spectrum of returned constraints to TBDeclare. Therefore, we extended these works with a wider range of constraints and an efficient implementation algorithm.

Recently, [67] have proposed a framework for discovering general LTL_f rules in event logs. Though more flexible than other approaches, it reveals not suitable for TBDeclare, due to a deep increase of computation time, as soon as disjunction among variables are introduced. An efficient approach for the discovery of Declare models at runtime has been presented in [68]. However, this technique only allows for the discovery of standard Declare constraints.

Various conceptual extensions of Declare have been proposed in the literature, partially with accompanying mining algorithms. In [69], the authors define *Timed Declare*, an extension of Declare based on a metric temporal logic semantics allowing for the specification of required delays and deadlines. The approach relies on timed automata to monitor metric dynamic constraints. In [70], such semantics is used for the discovery of metric temporal Declare constraints. [71] presents an approach for the discovery of Declare rules characterizing the lifecycle of non-atomic activities in a log. In [72], the authors propose an approach for monitoring data-aware Declare constraints at run-time, based on the data-aware semantics for Declare presented in [46, 73]. In the work proposed in [7], an alternative data-aware semantics for Declare has been introduced by using a first-order variant of LTL to specify data-aware patterns. Such extended patterns are used in [7] as the target language for a process discovery algorithm, which produces data-aware Declare constraints from raw event logs. The data-aware semantics for Declare has been further extended in [74]. In [75], a semantics for Declare based on metric first order temporal logics allows for combining data and temporal perspectives. Our work is complementary to these works. It is an avenue of future research to integrate TBDeclare with these perspectives.

7. Conclusion

In this paper, we have defined the class of Target-Branched Declare, which exhibits interesting properties in terms of set-dominance. We exploit these properties for the definition of an efficient mining approach. Furthermore, we specify pruning rules in order to arrive at a compact rule set. Our technique is evaluated for efficiency and effectiveness using simulated data and the case of the BPI Challenges of 2012 and 2014. In future research, we aim to further study broader classes of branched Declare. At this stage, we have focused on target-branched constraints. It is an open question how our results can be translated to the class of activation-branched constraints. Furthermore, we also plan to extend our technique towards the coverage of the entire Declare language.

Acknowledgements

The research work of Claudio Di Ciccio and Jan Mendling has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement 318275 (GET Service).

References

[1] M. Dumas, M. L. Rosa, J. Mendling, H. A. Reijers, Fundamentals of Business Process Management, Springer, 2013.

[2] M. Pesic, Constraint-based workflow management systems: Shifting control to users, Ph.D. thesis, Technische Universiteit Eindhoven (10 2008).
URL http://repository.tue.nl/638413

[3] C. Di Ciccio, M. Mecella, Mining constraints for artful processes, in: W. Abramowicz, D. Kriksciuniene, V. Sakalauskas (Eds.), International Conference on Business Information Systems, Vol. 117 of Lecture Notes in Business Information Processing, Springer, 2012, pp. 11–23. doi:10.1007/978-3-642-30359-3_2.
URL http://dx.doi.org/10.1007/978-3-642-30359-3

[4] C. Di Ciccio, M. Mecella, A two-step fast algorithm for the automated discovery of declarative workflows, in: Symposium on Computational Intelligence and Data Mining, IEEE, 2013, pp. 135–142. doi:10.1109/CIDM.2013.6597228.
URL http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6588692

[5] F. M. Maggi, A. J. Mooij, W. M. P. van der Aalst, User-guided discovery of declarative process models, in: CIDM, IEEE, 2011, pp. 192–199.
URL http://dx.doi.org/10.1109/CIDM.2011.5949297

[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: J. Ralyté, X. Franch, S. Brinkkemper, S. Wrycza (Eds.), CAiSE, Vol. 7328 of Lecture Notes in Computer Science, Springer, 2012, pp. 270–285.
URL http://dx.doi.org/10.1007/978-3-642-31095-9_18

[7] F. M. Maggi, M. Dumas, L. García-Bañuelos, M. Montali, Discovering data-aware declarative process models from event logs, in: Daniel et al. [76], pp. 81–96. doi:10.1007/978-3-642-40176-3_8.

[8] C. Di Ciccio, F. M. Maggi, J. Mendling, Discovering target-branched declare constraints, in: S. W. Sadiq, P. Soffer, H. Völzer (Eds.), Business Process Management, Vol. 8659 of Lecture Notes in Computer Science, Springer, 2014, pp. 34–50. doi:10.1007/978-3-319-10172-9_3.

[9] 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

[10] 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.), Enterprise, Business-Process and Information Systems Modeling, 10th International Workshop,

BPMDS 2009, and 14th International Conference, EMMSAD 2009, held at CAiSE 2009, Amsterdam, The Netherlands, June 8-9, 2009. Proceedings, Vol. 29 of Lecture Notes in Business Information Processing, Springer, 2009, pp. 353–366.

[11] H. A. Reijers, T. Slaats, C. Stahl, Declarative modeling-an academic dream or the future for bpm?, in: Daniel et al. [76], pp. 307–322. `doi:10.1007/978-3-642-40176-3_26`.
URL `http://dx.doi.org/10.1007/978-3-642-40176-3_26`

[12] A. Pnueli, The temporal logic of programs, in: 18th Annual Symposium on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), IEEE, 1977, pp. 46–57.

[13] E. M. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, 2001.

[14] G. De Giacomo, M. Y. Vardi, Linear temporal logic and linear dynamic logic on finite traces, in: F. Rossi (Ed.), IJCAI, IJCAI/AAAI, 2013, pp. 854–860.
URL `http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6997`

[15] G. De Giacomo, R. De Masellis, M. Montali, Reasoning on LTL on finite traces: Insensitivity to infiniteness, in: C. E. Brodley, P. Stone (Eds.), Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada., AAAI Press, 2014, pp. 1027–1033.
URL `http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8575`

[16] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang, Symbolic model checking: $10^2$0 states and beyond, Inf. Comput. 98 (2) (1992) 142–170. `doi:10.1016/0890-5401(92)90017-A`.
URL `http://dx.doi.org/10.1016/0890-5401(92)90017-A`

[17] D. Gries, Monotonicity in calculational proofs, in: E. Olderog, B. Steffen (Eds.), Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel), Vol. 1710 of Lecture Notes in Computer Science, Springer, 1999, pp. 79–85. `doi:10.1007/3-540-48092-7_4`.
URL `http://dx.doi.org/10.1007/3-540-48092-7_4`

[18] A. Burattin, F. M. Maggi, W. M. P. van der Aalst, A. Sperduti, Techniques for a posteriori analysis of declarative processes, in: C. Chi, D. Gasevic, W. van den Heuvel (Eds.), 16th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2012, Beijing, China, September 10-14, 2012, IEEE, 2012, pp. 41–50. `doi:10.1109/EDOC.2012.15`.
URL `http://doi.ieeecomputersociety.org/10.1109/EDOC.2012.15`

[19] C. Di Ciccio, M. Mecella, J. Mendling, The effect of noise on mined declarative constraints, in: P. Ceravolo, R. Accorsi, P. Cudre-Mauroux (Eds.), Data-Driven Process Discovery and Analysis, Vol. 203 of Lecture Notes in Business Information Processing, Springer Berlin Heidelberg, 2015, pp. 1–24. doi:10.1007/978-3-662-46436-6_1.
URL http://dx.doi.org/10.1007/978-3-662-46436-6_1

[20] R. Agrawal, R. Srikant, Fast algorithms for mining association rules in large databases, in: J. B. Bocca, M. Jarke, C. Zaniolo (Eds.), VLDB, Morgan Kaufmann, 1994, pp. 487–499.
URL http://www.vldb.org/conf/1994/P487.PDF

[21] 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.

[22] D. M. M. Schunselaar, F. M. Maggi, N. Sidorova, Patterns for a log-based strengthening of declarative compliance models, in: J. Derrick, S. Gnesi, D. Latella, H. Treharne (Eds.), IFM, Vol. 7321 of Lecture Notes in Computer Science, Springer, 2012, pp. 327–342. doi:10.1007/978-3-642-30729-4_23.

[23] B. A. Hausmann, O. Ore, Theory of quasi-groups, American Journal of Mathematics 59 (4) (1937) 983–1004.
URL http://www.jstor.org/stable/2371362

[24] G. Birkhoff, Lattice theory, in: Colloquium Publications, 3rd Edition, Vol. 25, Amer. Math. Soc., 1967.

[25] H. Mitsch, A natural partial order for semigroups, Proceedings of the American Mathematical Society 97 (3) (1986) 384–388.
URL http://www.jstor.org/stable/2046222

[26] J. Mendling, M. Strembeck, J. Recker, Factors of process model comprehension - findings from a series of experiments, Decision Support Systems 53 (1) (2012) 195–206.

[27] 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. doi:10.1007/s00450-009-0057-9.
URL http://dx.doi.org/10.1007/s00450-009-0057-9

[28] F. M. Maggi, R. P. J. C. Bose, W. M. P. van der Aalst, A knowledge-based integrated approach for discovering and repairing declare maps, in: C. Salinesi, M. C. Norrie, O. Pastor (Eds.), CAiSE, Vol. 7908 of Lecture Notes in Computer Science, Springer, 2013, pp. 433–448. doi:10.1007/978-3-642-38709-8_28.

[29] 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

[30] A. D. Bautista, L. Wangikar, S. M. K. Akbar, Process mining-driven optimization of a consumer loan approvals process - the BPIC 2012 challenge case study, in: Business Process Management Workshops - BPM 2012 International Workshops, Tallinn, Estonia, September 3, 2012. Revised Papers, 2012, pp. 219–220.

[31] W. M. P. van der Aalst, Process Mining: Discovery, Conformance and Enhancement of Business Processes, Springer, 2011. doi:10.1007/978-3-642-19345-3.

[32] W. M. P. van der Aalst, B. F. van Dongen, C. W. Günther, A. Rozinat, E. Verbeek, T. Weijters, ProM: The process mining toolkit, in: A. K. A. de Medeiros, B. Weber (Eds.), BPM (Demos), Vol. 489 of CEUR Workshop Proceedings, CEUR-WS.org, 2009.
URL http://ceur-ws.org/Vol-489/paper3.pdf

[33] R. Agrawal, D. Gunopulos, F. Leymann, Mining process models from workflow logs, in: H.-J. Schek, G. Alonso, F. Saltor, I. Ramos (Eds.), Advances in Database Technology – EDBT'98, Vol. 1377 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 1998, pp. 467–483, 10.1007/BFb0101003.
URL http://dx.doi.org/10.1007/BFb0101003

[34] W. M. P. van der Aalst, T. Weijters, L. Maruster, Workflow mining: Discovering process models from event logs, IEEE Trans. Knowl. Data Eng. 16 (9) (2004) 1128–1142.
URL http://csdl.computer.org/comp/trans/tk/2004/09/k1143abs.htm

[35] L. Wen, W. M. P. van der Aalst, J. Wang, J. Sun, Mining process models with non-free-choice constructs, Data Min. Knowl. Discov. 15 (2) (2007) 145–180.
URL http://dx.doi.org/10.1007/s10618-007-0065-y

[36] A. J. M. M. Weijters, W. M. P. van der Aalst, Rediscovering workflow models from event-based data using little thumb, Integrated Computer-Aided Engineering 10 (2) (2003) 151–162.
URL http://iospress.metapress.com/content/8puq22eumrva7vyp/

[37] A. K. A. de Medeiros, A. J. M. M. Weijters, W. M. P. van der Aalst, Genetic process mining: an experimental evaluation, Data Min. Knowl. Discov. 14 (2) (2007) 245–304. doi:10.1007/s10618-006-0061-7.
URL http://dx.doi.org/10.1007/s10618-006-0061-7

[38] W. M. P. van der Aalst, , V. Rubin, E. Verbeek, B. F. van Dongen, E. Kindler, C. W. Günther, Process mining: a two-step approach to balance between underfitting and overfitting, Software and Systems Modeling 9 (2010) 87–111, 10.1007/s10270-008-0106-z.
URL http://dx.doi.org/10.1007/s10270-008-0106-z

[39] J. Cortadella, M. Kishinevsky, L. Lavagno, A. Yakovlev, Deriving petri nets from finite transition systems, IEEE Trans. Comput. 47 (8) (1998) 859–882. doi:10.1109/12.707587.
URL http://dx.doi.org/10.1109/12.707587

[40] J. Desel, W. Reisig, The synthesis problem of petri nets, Acta Informatica 33 (1996) 297–315, 10.1007/s002360050046.
URL http://dx.doi.org/10.1007/s002360050046

[41] D. Fahland, W. M. P. van der Aalst, Repairing process models to reflect reality, in: A. P. Barros, A. Gal, E. Kindler (Eds.), BPM, Vol. 7481 of Lecture Notes in Computer Science, Springer, 2012, pp. 229–245. doi:10.1007/978-3-642-32885-5_19.
URL http://dx.doi.org/10.1007/978-3-642-32885-5

[42] D. Fahland, W. M. P. van der Aalst, Model repair - aligning process models to reality, Inf. Syst. 47 (2015) 220–243. doi:10.1016/j.is.2013.12.007.

[43] F. M. Maggi, D. Corapi, A. Russo, E. Lupu, G. Visaggio, Revising process models through inductive learning, in: Business Process Management Workshops - BPM 2010 International Workshops and Education Track, Hoboken, NJ, USA, September 13-15, 2010, Revised Selected Papers, 2010, pp. 182–193.

[44] 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.

[45] M. Pesic, H. Schonenberg, W. M. P. van der Aalst, Declare: Full support for loosely-structured processes, in: EDOC, IEEE Computer Society, 2007, pp. 287–300.
URL http://doi.ieeecomputersociety.org/10.1109/EDOC.2007.25

[46] F. Chesani, P. Mello, M. Montali, P. Torroni, Verification of choreographies during execution using the reactive event calculus, in: WS-FM, 2008, pp. 55–72.

[47] T. T. Hildebrandt, R. R. Mukkamala, T. Slaats, Designing a cross-organizational case management system using dynamic condition response graphs, in: Proceedings of the 15th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2011, Helsinki, Finland, August 29 - September 2, 2011, 2011, pp. 161–170.

[48] T. T. Hildebrandt, R. R. Mukkamala, T. Slaats, F. Zanitti, Contracts for cross-organizational workflows as timed dynamic condition response graphs, J. Log. Algebr. Program. 82 (5-7) (2013) 164–185.

[49] T. T. Hildebrandt, R. R. Mukkamala, Declarative event-based workflow as distributed dynamic condition response graphs, in: K. Honda, A. Mycroft (Eds.), PLACES, Vol. 69 of EPTCS, 2010, pp. 59–73.
URL http://dx.doi.org/10.4204/EPTCS.69.5

[50] G. Greco, A. Guzzo, D. Saccà, Simulations on workflow management systems: A framework based on event choice datalog, Intelligenza Artificiale 5 (2) (2011) 189–206. doi:10.3233/IA-2011-0022.
URL http://dx.doi.org/10.3233/IA-2011-0022

[51] C. Haisjackl, I. Barba, S. Zugal, P. Soffer, I. Hadar, M. Reichert, J. Pinggera, B. Weber, Understanding declare models: strategies, pitfalls, empirical results, Software & Systems Modeling (2014) 1–28doi:10.1007/s10270-014-0435-z.
URL http://dx.doi.org/10.1007/s10270-014-0435-z

[52] F. M. Maggi, T. Slaats, H. A. Reijers, The automated discovery of hybrid processes, in: S. W. Sadiq, P. Soffer, H. Völzer (Eds.), Business Process Management - 12th International Conference, BPM 2014, Haifa, Israel, September 7-11, 2014. Proceedings, Vol. 8659 of Lecture Notes in Computer Science, Springer, 2014, pp. 392–399. doi:10.1007/978-3-319-10172-9_27.
URL http://dx.doi.org/10.1007/978-3-319-10172-9_27

[53] N. C. Silva, C. A. L. de Oliveira, F. A. L. A. Albino, R. M. F. Lima, Declarative versus imperative business process languages - A controlled experiment, in: S. Hammoudi, L. A. Maciaszek, J. Cordeiro (Eds.), ICEIS 2014 - Proceedings of the 16th International Conference on Enterprise Information Systems, Volume 3, Lisbon, Portugal, 27-30 April, 2014, SciTePress, 2014, pp. 394–401. doi:10.5220/0004896203940401.
URL http://dx.doi.org/10.5220/0004896203940401

[54] P. Pichler, B. Weber, S. Zugal, J. Pinggera, J. Mendling, H. A. Reijers, Imperative versus declarative process modeling languages: An empirical investigation, in: Business Process Management Workshops - BPM 2011 International Workshops, Clermont-Ferrand, France, August 29, 2011, Revised Selected Papers, Part I, 2011, pp. 383–394.

[55] C. Haisjackl, S. Zugal, P. Soffer, I. Hadar, M. Reichert, J. Pinggera, B. Weber, Making sense of declarative process models: Common strategies and typical pitfalls, in: Enterprise, Business-Process and Information Systems Modeling - 14th International Conference, BPMDS 2013, 18th International Conference, EMMSAD 2013, Held at CAiSE 2013, Valencia, Spain, June 17-18, 2013. Proceedings, 2013, pp. 2–17.

[56] O. Kupferman, M. Y. Vardi, Vacuity detection in temporal model checking, STTT 4 (2) (2003) 224–233. doi:10.1007/s100090100062. URL dx.doi.org/10.1007/s100090100062

[57] R. P. J. C. Bose, F. M. Maggi, W. M. P. van der Aalst, Enhancing declare maps based on event correlations, in: Business Process Management - 11th International Conference, BPM 2013, Beijing, China, August 26-30, 2013. Proceedings, 2013, pp. 97–112.

[58] F. M. Maggi, Declarative process mining with the declare component of ProM, in: M.-C. Fauvet, B. F. van Dongen (Eds.), BPM (Demos), Vol. 1021 of CEUR Workshop Proceedings, CEUR-WS.org, 2013. URL http://ceur-ws.org/Vol-1021/paper_8.pdf

[59] E. Lamma, P. Mello, M. Montali, F. Riguzzi, S. Storari, Inducing declarative logic-based models from labeled traces, in: G. Alonso, P. Dadam, M. Rosemann (Eds.), BPM, Vol. 4714 of Lecture Notes in Computer Science, Springer, 2007, pp. 344–359. doi:10.1007/978-3-540-75183-0_25. URL http://dx.doi.org/10.1007/978-3-540-75183-0_25

[60] E. Lamma, P. Mello, F. Riguzzi, S. Storari, Applying inductive logic programming to process mining, in: H. Blockeel, J. Ramon, J. W. Shavlik, P. Tadepalli (Eds.), ILP, Vol. 4894 of Lecture Notes in Computer Science, Springer, 2007, pp. 132–146. doi:10.1007/978-3-540-78469-2_16. URL http://dx.doi.org/10.1007/978-3-540-78469-2_16

[61] F. Chesani, E. Lamma, P. Mello, M. Montali, F. Riguzzi, S. Storari, Exploiting inductive logic programming techniques for declarative process mining, T. Petri Nets and Other Models of Concurrency 2 (2009) 278–295. URL http://dx.doi.org/10.1007/978-3-642-00899-3_16

[62] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, P. Torroni, Verifiable agent interaction in abductive logic programming: The sciff framework, ACM Trans. Comput. Log. 9 (4) (2008) 29:1–29:43. doi:10.1145/1380572.1380578.

[63] E. Bellodi, F. Riguzzi, E. Lamma, Probabilistic logic-based process mining, in: W. Faber, N. Leone (Eds.), CILC, Vol. 598 of CEUR Workshop Proceedings, CEUR-WS.org, 2010. URL http://ceur-ws.org/Vol-598/paper17.pdf

[64] E. Bellodi, F. Riguzzi, E. Lamma, Probabilistic declarative process mining, in: Y. Bi, M.-A. Williams (Eds.), KSEM, Vol. 6291 of Lecture Notes in Computer Science, Springer, 2010, pp. 292–303. doi:10.1007/978-3-642-15280-1_28. URL http://dx.doi.org/10.1007/978-3-642-15280-1_28

[65] M. Richardson, P. Domingos, Markov logic networks, Machine Learning 62 (1-2) (2006) 107–136. doi:10.1007/s10994-006-5833-1.
URL http://dx.doi.org/10.1007/s10994-006-5833-1

[66] 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.

[67] M. Räim, C. Di Ciccio, F. M. Maggi, M. Mecella, J. Mendling, Log-based understanding of business processes through temporal logic query checking, in: R. Meersman, H. Panetto, T. S. Dillon, M. Missikoff, L. Liu, O. Pastor, A. Cuzzocrea, T. Sellis (Eds.), International Conference on Cooperative Information Systems, On the Move to Meaningful Internet Systems Confederated International Conferences, Vol. 8841 of Lecture Notes in Computer Science, Springer, 2014, pp. 75–92. doi:10.1007/978-3-662-45563-0_5.
URL http://dx.doi.org/10.1007/978-3-662-45563-0

[68] F. M. Maggi, A. Burattin, M. Cimitile, A. Sperduti, Online process discovery to detect concept drifts in ltl-based declarative process models, in: On the Move to Meaningful Internet Systems: OTM 2013 Conferences - Confederated International Conferences: CoopIS, DOA-Trusted Cloud, and ODBASE 2013, Graz, Austria, September 9-13, 2013. Proceedings, 2013, pp. 94–111.

[69] M. Westergaard, F. M. Maggi, Looking into the future. using timed automata to provide a priori advice about timed declarative process models, in: R. Meersman, H. Panetto, T. S. Dillon, S. Rinderle-Ma, P. Dadam, X. Zhou, S. Pearson, A. Ferscha, S. Bergamaschi, I. F. Cruz (Eds.), 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, Vol. 7565 of Lecture Notes in Computer Science, Springer, 2012, pp. 250–267. doi:10.1007/978-3-642-33606-5_16.
URL http://dx.doi.org/10.1007/978-3-642-33606-5_16

[70] 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.

[71] M. L. Bernardi, M. Cimitile, C. D. 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.

[72] M. Montali, F. M. Maggi, F. Chesani, P. Mello, W. M. P. van der Aalst, Monitoring business constraints with the event calculus, ACM TIST 5 (1) (2013) 17.

[73] M. Montali, F. Chesani, F. M. Maggi, P. Mello, Towards data-aware constraints in declare, in: SAC, ACM Press and Addison Wesley, 2013, pp. 1391–1396.

[74] R. D. Masellis, F. M. Maggi, M. Montali, Monitoring data-aware business constraints with finite state automata, in: International Conference on Software and Systems Process 2014, ICSSP '14, Nanjing, China - May 26 - 28, 2014, 2014, pp. 134–143.

[75] A. Burattin, F. M. Maggi, A. Sperduti, Conformance checking based on multi-perspective declarative process models, CoRR abs/1503.04957.

[76] F. Daniel, J. Wang, B. Weber (Eds.), Business Process Management - 11th International Conference, BPM 2013, Beijing, China, August 26-30, 2013. Proceedings, Vol. 8094 of Lecture Notes in Computer Science, Springer, 2013. doi:10.1007/978-3-642-40176-3.