Discovering
Target-Branched Declare Constraints

Claudio Di Ciccio[1], Fabrizio Maria Maggi[2], and Jan Mendling[1]

[1] Vienna University of Business and Economics, Austria,
{claudio.di.ciccio,jan.mendling}@wu.ac.at
[2] University of Tartu, Estonia,
f.m.maggi@ut.ee

Abstract. Process discovery is the task of generating 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 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. However, Declare constraints with branching are expensive to be calculated.In addition, it is often the case that hundreds of branching 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-world event logs.

Keywords: Process Mining; Discovery; Declarative Processes

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 [8]. Due 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 from 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 power 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. However, simple branching statements like "if you do a, you will do eventually either b or c" cannot be easily mined for Declare models.

In this paper, we address the problem of mining Declare branching constraints. We define the class of Target-Branched Declare and devise efficient mining algorithms for it. The key idea is to exploit dominance relationships, which help to drastically prune the search space. We present formal proofs to demonstrate its merits. A prototypical implementation is used for performance analysis, emphasising feasibility and efficiency for our approach.

Against this background, this paper is structured as follows. Section 2 introduces the essential concepts of Declare. 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 on Mining Declarative Process Models

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 a dynamic environment. Therefore, it has been argued to rather utilise declarative models in such a context, in order to facilitate better understanding of the mined process by humans [9,22].

One of the most frequently used declarative languages is Declare introduced by Pesic and van der Aalst in [26]. Instead of explicitly specifying the sequence of events, Declare consists of a set of constraints that are applied to activities. Constraints, in turn, are based on templates that define parametrised classes of properties. Templates have a graphical representation and their semantics can be formalised using formal logics [21,7], the main one being Linear Temporal Logic over finite traces (LTL_f). In this way, analysts work with the graphical representation of templates, while the underlying formulas remain hidden. Table 1 summarises important Declare templates. For a complete specification see [26]. Here, we indicate template parameters with x or y symbols and real activities in their instantiations with a, b or c letters.

The formulas 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. The 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

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

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

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 semantics, consider the *Response* constraint $\Box(a \rightarrow \Diamond b)$. This constraint indicates that if a *occurs*, b must eventually *follow*. Therefore, this constraint is satisfied for traces such as $\mathbf{t}_1 = \langle a, a, b, c \rangle$, $\mathbf{t}_2 = \langle b, b, c, d \rangle$, and $\mathbf{t}_3 = \langle a, b, c, b \rangle$, but not for $\mathbf{t}_4 = \langle a, b, a, c \rangle$ because, in this case, the second instance of a is not followed by a b.

An *activation* of a constraint in a trace is an event whose occurrence imposes some obligations on other *target* events in the same trace. E.g., a is an activation and b is a target for the *Response* constraint $\Box(a \rightarrow \Diamond b)$, because the execution of a forces b to be executed eventually. When a trace is compliant with respect to a constraint, every activation of it leads to a fulfillment. Consider, again, the *Response* constraint $\Box(a \rightarrow \Diamond b)$. In trace $\mathbf{t}_1$, the constraint is activated and fulfilled twice, whereas, in $\mathbf{t}_3$, the same constraint is activated and fulfilled only once. On the other hand, when a trace is not compliant, an activation of it can lead to a fulfillment but also at least to one activation violation. In trace $\mathbf{t}_4$, the *Response* constraint $\Box(a \rightarrow \Diamond b)$ is activated twice: the first activation leads to a fulfillment (eventually b occurs) and the second activation to a violation (b does not occur subsequently). An algorithm to check fulfillments and violations is presented in [2]. To judge the relevance of constraints, we adopt *support* and *confidence* from data mining [1]. The support of a Declare constraint in an event log is defined as the fraction of activations of the constraint that lead to a fulfillment. The confidence of a Declare constraint is the product between the support of the rule and the support of the activation, i.e., the percentage of traces in which the activation occurs.

In spite of its advantages, one of the conceptual limitations of mining Declare constraints at this stage is the lack of support for branching. Branching as supported in the synthesis approach for behavioural profiles [28,24] and for the alpha algorithm [25] try to explicit mine for statements like “if you do a, you will (eventually) do either b or c”. Such exclusiveness statements are typically used in experiments on process model understanding, see [18], because of their practical importance. Therefore, we investigate how Declare can be enriched

with branching constraints in such a way that mining can still be conducted efficiently.

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. This means that *Response*$(a, \{b, c\})$ is a TBDeclare constraint stating that "if a occurs, b *or* c must eventually follow". In TBDeclare, a constraint template maps to a LTL$_f$ formula, and a constraint is its interpretation over a log (see Table 2). The models of a constraint are therefore traces that comply with the formula. We consider the class of TBDeclare for the reason that it exhibits interesting properties. First, we prove that a property of set-dominance holds. Then, we discuss implications of this for support. These properties will be exploited in the mining algorithm.

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

Table 2: LTL$_f$ semantics for Target-Branched Declare constraints, given an activity x and a set of activities $Y = \{y_i | i > 0\}$

3.1 Set-Dominance

In this subsection, we identify that the inclusion property of two branching sets translates into the inclusion of their fulfilment of a constraint template.

Lemma 1. *Given a task x in the process alphabet Σ, two non-empty sets of tasks Y and Y' such that $Y \subseteq Y' \subseteq \Sigma$, and a TBDeclare constraint template $\mathcal{C}$, then $\mathcal{C}(x, Y) \models \mathcal{C}(x, Y')$.*

Proof (sketch). In the base case, $Y = Y' = \{y_1, \ldots, y_n\}$. Therefore, $\mathcal{C}(x, Y) \equiv \mathcal{C}(x, Y')$.

If $Y' = Y \bigcup \{y_{n+1}\}$, with $y_{n+1} \notin Y$, the demonstration proceeds by proving the statement for each constraint template.

RespondedExistence$(x, Y') \equiv \Diamond x \rightarrow \Diamond \left(\bigvee_{i=1}^{n} y_i \vee y_{n+1}\right)$. Recalling that, given two non-negated literals φ and ψ:

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

we have that $\mathit{RespondedExistence}(x, Y') \equiv \neg\Diamond x \vee \bigvee_{i=1}^{n} \Diamond y_i \vee \Diamond y_{n+1}$. Consequently, $\mathit{RespondedExistence}(x, Y') \equiv \mathit{RespondedExistence}(x, Y) \vee y_{n+1}$. Given a formula Φ and a non-negated literal ψ, $\Phi \models \Phi \vee \psi$. Therefore, Lemma 1 for *RespondedExistence* is proven. The argument for the other templates has been established in a similar way, which is here omitted for space reasons. □

3.2 Support Monotone Non-Decrement w.r.t. Set-Dominance

Given a constraint C and a log L, the support function $\mathcal{S}(C, L)$ returns the number of cases in which the constraint is verified (C_L^+) over the number of cases in which the constraint is activated along the log (C_L^T):

$$\mathcal{S}(C, L) = \frac{C_L^+}{C_L^T}$$

Theorem 1 describes the monotonic non-decreasing trend of support for constraints with respect to set-containment of the target set of activities.

Theorem 1. *Given a task a in the process alphabet Σ, two non-empty sets of tasks Y and Y' such that $Y \subseteq Y' \subseteq \Sigma$, a log L and a TBDeclare constraint template $\mathcal{C}$, then $\mathcal{S}(\mathcal{C}(x, Y), L) \leqslant \mathcal{S}(\mathcal{C}(x, Y'), L)$.*

Proof. In the following, we name the number of cases in which $\mathcal{C}(x, Y)$ and $\mathcal{C}(x, Y')$ are verified as, resp., C_L^+ and $C_L'^+$. In the light of Lemma 1, if $Y \subseteq Y'$ then $\mathcal{C}(x, Y) \models \mathcal{C}(x, Y')$. Therefore, due to the definition of model for a constraint w.r.t. a log, we have $C_L^+ \leqslant C_L'^+$. Since a is the activation for both constraints, the cases in which they are activated are the same, accounting to C_L^T. As a consequence, $\frac{C_L^+}{C_L^T} \leqslant \frac{C_L'^+}{C_L^T}$. □

4 Discovery

This section describes MINERful for Target-Branched Declare (TB-MINERful), a three step algorithm for: *(i)* building a knowledge base, which keeps statistics on task occurrences; *(ii)* querying the knowledge base for support and confidence of constraints; *(iii)* pruning constraints not having sufficient support and confidence. The input of the algorithm is a log L based on a log alphabet Σ. Three thresholds can be specified: *(i) branching factor*, limiting the size of the activity sets for discovered constraints, *(ii) support*, and *(iii) confidence*.

4.1 The Knowledge Base

The first step is the construction of a knowledge base keeping statistics on task occurrences in the log. It consists of 9 functions listed below along with a semi-formal definition. We indicate parameters for constraints as x, y, z. Y, Z are

set-parameters. Consider, e.g., a set of activities $\Sigma = \{a, b, c, d\}$ (log alphabet). While a, b, c, d refers to activity instantiations, a possible instantiation of Y is $\{b, c\}$. As example log we use $L = \{\langle a, a, b, a, c, a\rangle, \langle a, a, b, a, c, a, d\rangle\}$.

- $\overset{\nrightarrow}{\gamma_0}(x)$ counts the traces where x did not occur. For instance, $\overset{\nrightarrow}{\gamma_0}(a) = 0$ for L, because a occurs in every trace. $\overset{\nrightarrow}{\gamma_0}(d) = 1$ instead.
- $\Gamma(x)$ counts the occurrences of x. Therefore, $\Gamma(a) = 8$ in L.
- $\overset{\twoheadrightarrow}{\delta_0}(x, Y)$ counts the occurrences of x with no following $y \in Y$ in the traces. In the example, e.g., $\overset{\twoheadrightarrow}{\delta_0}(a, \{d\}) = 4$, $\overset{\twoheadrightarrow}{\delta_0}(a, \{b\}) = 4$, and $\overset{\twoheadrightarrow}{\delta_0}(a, \{b, c\}) = 2$.
- $\overset{\twoheadleftarrow}{\delta_0}(x, Y)$ counts the occurrences of x with no preceding $y \in Y$ in the traces. Thus, e.g., $\overset{\twoheadleftarrow}{\delta_0}(a, \{d\}) = 8$, $\overset{\twoheadleftarrow}{\delta_0}(a, \{b\}) = 4$, and $\overset{\twoheadleftarrow}{\delta_0}(a, \{b, c\}) = 4$.
- $\overset{\leftrightarrow}{\delta_0}(x, Y)$ counts the occurrences of x with no $y \in Y$ in the traces. Therefore, $\overset{\leftrightarrow}{\delta_0}(a, \{d\}) = 1$, and $\overset{\leftrightarrow}{\delta_0}(a, \{b, d\}) = 0$ in L.
- $\overrightarrow{\delta_1}(x, y)$ counts the occurrences of x having y as the next event. Hence, $\overrightarrow{\delta_1}(a, b) = 2$, $\overrightarrow{\delta_1}(a, d) = 1$.
- $\overleftarrow{\delta_1}(x, y)$ counts the occurrences of x having y as the preceding event. In L, $\overrightarrow{\delta_1}(a, b) = 2$ and $\overrightarrow{\delta_1}(a, d) = 0$.
- $\overset{\hookrightarrow}{\beta}(x, Y)$ counts how many times x repeats until the first $y \in Y$. If no $y \in Y$ appears in the trace, the count is not further considered. In the example, $\overset{\hookrightarrow}{\beta}(a, \{b\}) = 2$, $\overset{\hookrightarrow}{\beta}(a, \{c\}) = 4$, $\overset{\hookrightarrow}{\beta}(a, \{b, c\}) = 2$, and $\overset{\hookrightarrow}{\beta}(a, \{b, d\}) = 3$.
- $\overset{\hookleftarrow}{\beta}(x, Y)$ is similar to $\overset{\hookrightarrow}{\beta}(x, Y)$, but reading the trace contrariwise. Thus, $\overset{\hookleftarrow}{\beta}(a, \{b\}) = 2$, $\overset{\hookleftarrow}{\beta}(a, \{c\}) = 0$, $\overset{\hookleftarrow}{\beta}(a, \{b, c\}) = 0$, and $\overset{\hookleftarrow}{\beta}(a, \{b, d\}) = 2$.

Next, we discuss how this knowledge base is built based on an input log.

4.2 Building the Knowledge Base

Here, we define an algorithm for building the knowledge base, which requires one run over the traces to update it. This makes the algorithm linear w.r.t. the number of traces and their length.

For evaluating $\overset{\twoheadrightarrow}{\delta_0}(x, Y)$, the technique executes two steps for each string. As a first step, it computes for every activity $y \in \Sigma \backslash \{x\}$ the value to accumulate in $\overset{\twoheadrightarrow}{\delta_0}(x, y)$, i.e., $\mathtt{N}^{\delta_0}_{x,y}$. We will also refer to $\mathtt{N}^{\delta_0}_{x,y}$ as a pairwise counter. Table 3a shows how this is achieved for $\langle a, a, b, a, c, a\rangle$. $\mathtt{N}^{\delta_0}_{x,y}$ is incremented by 1 every time x is read, while parsing the trace. When y is read, $\mathtt{N}^{\delta_0}_{x,y}$ is reset to 0. The $\downarrow$ symbol indicates this operation ("flush"). At the end of the trace, the value stored in $\mathtt{N}^{\delta_0}_{x,y}$ reports the occurrences of x after which no y occurred. Pairwise counters do not take into account the relation of x with *sets* of activities, though. On the other hand, computing a value for each $Y \in \mathcal{P}(\Sigma \backslash \{a\})$ would be impractical. Therefore, we build *differential cumulative* set-counters, $\Delta\mathtt{N}^{\delta_0}_{x,Y}$. If

	Trace					
	a	a	b	a	c	a
$\mathtt{N}^{\delta_0}_{a,b}$	1	2	↓	1		2
$\mathtt{N}^{\delta_0}_{a,c}$	1	2		3	↓	1
$\mathtt{N}^{\delta_0}_{a,d}$	1	2		3		4

(a) Computation of $\Delta\mathtt{N}^{\delta_0}_{a,\cdot}$

$\mathtt{N}^{\delta_0}_{a,\cdot}$						$\Delta\mathtt{N}^{\delta_0}_{a,\cdot}$
$\mathtt{N}^{\delta_0}_{a,b} =$	1+	$\mathtt{N}^{\delta_0}_{a,c} =$	1	$\mathtt{N}^{\delta_0}_{a,d} =$	1+	$\Rightarrow \Delta\mathtt{N}^{\delta_0}_{a,\{b,c,d\}} = 1$
	1				1+	$\Rightarrow \Delta\mathtt{N}^{\delta_0}_{a,\{b,\quad d\}} = 1$
					2	$\Rightarrow \Delta\mathtt{N}^{\delta_0}_{a,\{\quad d\}} = 2$

(b) Computation of $\Delta\mathtt{N}^{\delta_0}_{a,\cdot}$, given the values of $\mathtt{N}^{\delta_0}_{a,\cdot}$

Table 3: Computation of $\mathtt{N}^{\delta_0}_{a,\cdot}$ and $\Delta\mathtt{N}^{\delta_0}_{a,\cdot}$, given a sample trace: $\langle a, a, b, a, c, a \rangle$

$\Delta\mathtt{N}^{\delta_0}_{a,\cdot}$					$\Rightarrow$	$\overrightarrow{\delta_0}(a,\cdot)$
$\{b,$	$c,$	$d\}$	$=$	1	$\Rightarrow$	$\overrightarrow{\delta_0}(a,\{b,c,d\}) = \overrightarrow{\delta_0}(a,\{c,d\}) = \overrightarrow{\delta_0}(a,\{c\}) = 1$
$\{b,$		$d\}$	$=$	1	$\Rightarrow$	$\overrightarrow{\delta_0}(a,\{b,d\}) = \overrightarrow{\delta_0}(a,\{b\}) = 2$
$\{$		$d\}$	$=$	2	$\Rightarrow$	$\overrightarrow{\delta_0}(a,\{d\}) = 4$

Table 4: Computation of $\overrightarrow{\delta_0}(a,\cdot)$, given $\Delta\mathtt{N}^{\delta_0}_{a,\cdot}$

$Y \subseteq Z$, $\Delta\mathtt{N}^{\delta_0}_{x,Z}$ reports the number of times in which none of its elements occurred in the trace after x. $\Delta\mathtt{N}^{\delta_0}_{x,Y}$ reports only the difference between *(i)* the number of times in which no $y \in Y$ occurred, and *(ii)* $\Delta\mathtt{N}^{\delta_0}_{x,Z}$. Therefore, in $\langle a, a, b, a, c, a \rangle$, we have that $\Delta\mathtt{N}^{\delta_0}_{a,\{b,c,d\}} = 1$, $\Delta\mathtt{N}^{\delta_0}_{a,\{b,d\}} = 1$, and $\Delta\mathtt{N}^{\delta_0}_{a,\{d\}} = 2$. Passing from pairwise counters to differential cumulative set-counters is a linear operation: Table 3b sketches the technique. From this data structure, $\overrightarrow{\delta_0}(x,Y)$ can be extracted as follows:

$$\overrightarrow{\delta_0}(x,Y) = \sum_{Z \supseteq Y} \Delta\mathtt{N}^{\delta_0}_{x,Z}$$

Table 4 shows the extraction for the example trace. It is straightforward to see that the differential accumulation ($\Delta\mathtt{N}^{\delta_0}_{x,Y}$) allows for keeping fewer values in memory (3 in the example) than the possible entries for the knowledge base ($\overrightarrow{\delta_0}(x,Y)$, which amounts to 6). Every time a new trace is parsed, $\mathtt{N}^{\delta_0}_{x,y}$ is reset to 0 for each $x, y \in \Sigma$. At the end of the analysis of every subsequent trace, values for a new structure $\Delta\mathtt{N}^{\delta_0'}_{x,Y}$ are calculated. Thereupon, they are added to the preceding results. It might happen that a new Z set was not considered in $\Delta\mathtt{N}^{\delta_0}_{x,\cdot}$ for previous traces, but a new $\Delta\mathtt{N}^{\delta_0'}_{a,Z}$ is computed. In such case, $\Delta\mathtt{N}^{\delta_0}_{x,Z}$ is considered as 0 by the default and the new value in $\Delta\mathtt{N}^{\delta_0'}_{x,Z}$ is added. This technique extends to the computation of $\overleftarrow{\delta_0}(x,Y)$ and $\overleftrightarrow{\delta_0}(x,Y)$ with slight modifications. The values of the remaining functions are also determined in a similar way. However, the detailed descriptions are here omitted for the sake of space.

4.3 Querying the Knowledge Base

Once the knowledge base is built, the support of constraints can be calculated. Table 5 lists the functions adopted to this extent, for each TBDeclare constraint. All queries build upon a Laplacian concept of probability with support being

TBDeclare constraint	Support
RespondedExistence(x, Y)	$1 - \frac{\overleftrightarrow{\delta}_0(x,Y)}{\Gamma(x)}$
Response(x, Y)	$1 - \frac{\overset{\twoheadrightarrow}{\delta}_0(x,Y)}{\Gamma(x)}$
AlternateResponse(x, Y)	$1 - \frac{\overset{\twoheadrightarrow}{\delta}_0(x,Y) + \overset{\hookrightarrow}{\beta}(x,Y)}{\Gamma(x)}$
ChainResponse(x, Y)	$\frac{\sum_{y \in Y} \overrightarrow{\delta}_1(x,y)}{\Gamma(x)}$
Precedence(Y, x)	$1 - \frac{\overset{\twoheadleftarrow}{\delta}_0(x,Y)}{\Gamma(x)}$
AlternatePrecedence(Y, x)	$1 - \frac{\overset{\twoheadleftarrow}{\delta}_0(x,Y) + \overset{\hookleftarrow}{\beta}(x,Y)}{\Gamma(x)}$
ChainPrecedence(Y, x)	$\frac{\sum_{y \in Y} \overleftarrow{\delta}_1(x,y)}{\Gamma(x)}$

Table 5: Target-Branched Declare constraints and support functions

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 in the log, $\Gamma(x)$. For *ChainResponse*(x, Y), supporting cases are those occurrences of a immediately followed by some $y \in Y$, i.e., $\overrightarrow{\delta}_1(x, y)$. Supporting cases can be summed up because if x is followed by a given $y \in Y$ in a trace, it cannot be immediately followed by any other event $z \in Y$. In other words, the two cases are mutually exclusive. However, this assumption does not hold true, e.g., for *Response*(x, Y). Therefore, we consider the non-supporting cases, when x is *not* followed by any of the $y \in Y$, i.e., $\overset{\twoheadrightarrow}{\delta}_0(x, Y)$. 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*(x, Y) is $1 - \frac{\overset{\twoheadrightarrow}{\delta}_0(x,Y)}{\Gamma(x)}$. Likewise, the support of *RespondedExistence*(x, Y) is computed on the basis of the non-supporting cases. The support of *AlternateResponse*(x, Y) is then based on the cases when either *(i)* x is not followed by any $y \in Y$ ($\overleftrightarrow{\delta}_0(x, Y)$), or *(ii)* x occurs more than once before the first occurrence of $y \in Y$ ($\overset{\hookrightarrow}{\beta}(x, Y)$). The two conditions are mutually exclusive. Therefore, it is appropriate to sum them up. Similar considerations lead to the definition of support functions for *Precedence*(Y, x), *AlternatePrecedence*(Y, x) and *ChainPrecedence*(Y, x).

Confidence is computed as the constraint's support multiplied by the fraction of traces where the activation occurs. Therefore, given a TBDeclare constraint $\mathcal{C}(x, Y)$, a log L, and the support function $\mathcal{S}(\mathcal{C}(x, Y), L)$, the confidence of $\mathcal{C}(x, Y)$ w.r.t. L, $\mathcal{L}(\mathcal{C}(x, Y), L)$, is defined as

$$\mathcal{L}(\mathcal{C}(x, Y), L) = \mathcal{S}(\mathcal{C}(x, Y), L) \times \left(1 - \frac{\overset{\nearrow\!\!\!\!\!/}{\gamma}_0(x)}{\Gamma(x)}\right)$$

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 constraint templates as N, up to $N \times 2^{|\Sigma|-1}$ constraints can potentially hold true. When a maximum limit to the cardinality of the set is imposed, the number is reduced to

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

However, even with branching factor set to 3 and $|\Sigma| = 10$, already 3087 constraints have to be evaluated. A model including such a number of constraints would be hardly comprehensible for humans [18,26]. In order to reduce this number, we adopt pruning based on set-dominance and on hierarchy subsumption.

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", implies 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 Theorem 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 1. 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 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.

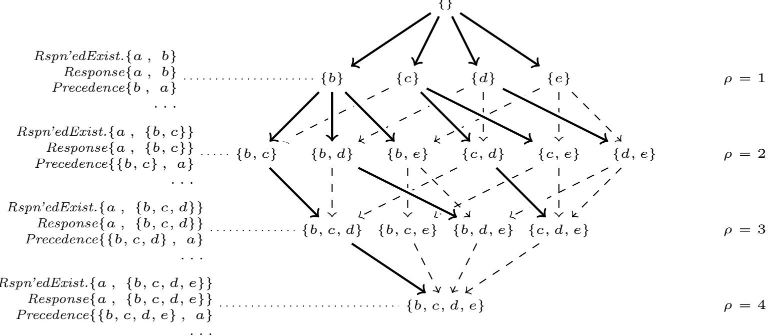


Fig. 1: A Hasse Diagram representing the Partial Order set containment relation. Containing sets are at the head of connecting arcs, contained sets are at the tail.

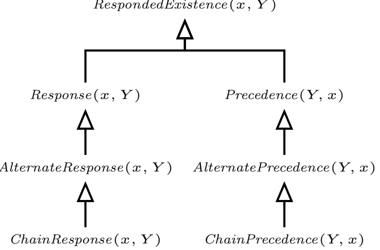
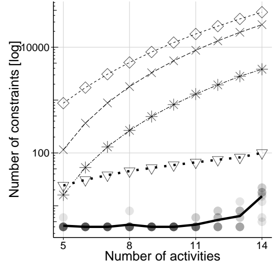


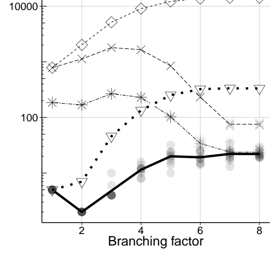
Fig. 2: Diagram showing the subsumption hierarchy relation. Constraints that are subsumed are at the tail.

Pruning Based on Hierarchy Subsumption. As investigated in [7,23,13], Declare constraints are not independent, but partially form a subsumption hierarchy. We consider a constraint $\mathcal{C}(x,Y)$ subsumed by another constraint $\mathcal{C}'(x,Y)$ when all the traces that comply with $\mathcal{C}(x,Y)$ also comply with $\mathcal{C}'(x,Y)$. $\mathit{Response}(x,Y)$, e.g., is subsumed by $\mathit{RespondedExistence}(x,Y)$. Figure 2 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, $\mathcal{S}\left(\mathit{RespondedExistence}(x,Y),L\right) > \mathcal{S}\left(\mathit{Response}(x,Y),L\right)$, then $\mathit{Response}(x,Y)$ is marked as redundant. However, if $\mathcal{S}\left(\mathit{RespondedExistence}(x,Y),L\right) = \mathcal{S}\left(\mathit{Response}(x,Y),L\right)$, then $\mathit{Response}(x,Y)$ 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 2). 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.

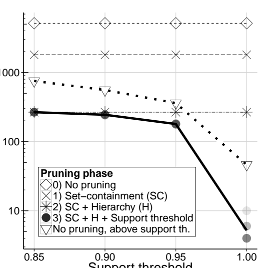
Both pruning techniques complement one another in reducing the constraint set.



(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 branching factor

Fig. 3: Effectiveness tests performed on synthetic logs.

5 Experiments and Evaluation

In this section, we investigate the efficiency and effectiveness of our approach. Section 5.1 shows results obtained by applying the proposed technique on synthetic logs. Section 5.2 demonstrates the effectiveness of our approach for event logs 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 32GB 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:

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

and we have simulated it to generate a compliant event log as described in [7]. 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 number of branches. 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 resp. minimum and maximum trace length, 10,000 for log size, 8 for log alphabet size, 1.0 for support threshold, and 3 for branching factor.

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.

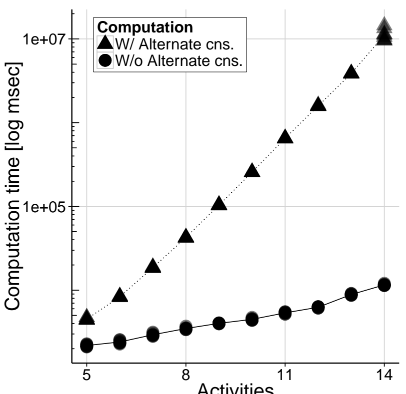
Figure 3a shows the trend (in logarithmic scale) of the number of discovered constraints by varying the log alphabet size. Different curves refer to different configurations of the miner: without any pruning (diamonds); with set-containment-based pruning (crosses); with set-containment- and hierarchy-based pruning (asterisks); with set-containment- and hierarchy-based pruning, along with 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 constraints 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, and based on the pruning techniques presented in this paper. This improvement yields a reduction ratio of 84% (100.3 v. 15.2, on average).

Figure 3b shows the trend (in logarithmic scale) of the number of discovered constraints by varying the branching factor. Here, the trend of the number of discovered constraints is different for different configurations. 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 increases 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 constraint with support threshold and the number after using the pruning techniques presented in this paper is quantified (branching factor of 8) in a reduction ratio of 88% (46.2 v. 5.2, on average).

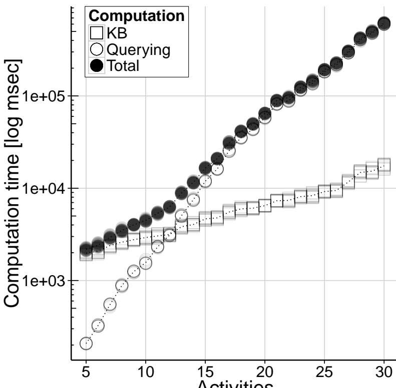
The plot in Figure 3c confirms that for any threshold between 0.85 and 1.0, the number of constraints discovered by applying all the pruning techniques is lower than the one obtained by applying the support-threshold filtering. The reduction ratio is indeed 93% (331.8 v. 22, on average), when the threshold is set to 1.0.

Efficiency: Second, we focus on time efficiency of our approach. We observe that efficiency strongly depends on the template. In particular, the "alternate" templates are less performative. Figure 4a shows this by plotting the computation time as function of the log alphabet size (in logarithmic scale). When the alternate templates are included in the evaluation, the computation time grows exponentially with the growth of the alphabet size.

As a next step, we therefore exclude the alternate templates and get the computation time as a function of log alphabet size (Figure 4b), log size (Figure 5a), and average trace size (Figure 5b). Figure 4b shows the trend (in logarithmic scale) of the computation time by varying the log alphabet size. Different curves



(a) Efficiency test results, including and excluding alternate templates in the evaluation



(b) Efficiency test results, w.r.t. the different phase of the algorithm

Fig. 4: Efficiency tests performed on synthetic logs, considering computation time as function of the log alphabet size.

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 in which the query time becomes higher than the knowledge base construction time. Figure 5a shows the trend (in logarithmic scale) of the computation time by varying the log size, whereas Figure 5b depicts the trend (in logarithmic scale) of the computation time by varying the average trace size. In both cases the query clearly outperforms the knowledge base construction time.

5.2 Evaluation Based on Real Data

We have evaluated the applicability of our approach using real-world event logs provided for the BPI challenge 2012 [27]. The event log pertains to an application process for personal loans or overdrafts of a Dutch bank. It contains 262,200 events distributed across 24 different possible event names and includes 13,087 cases.

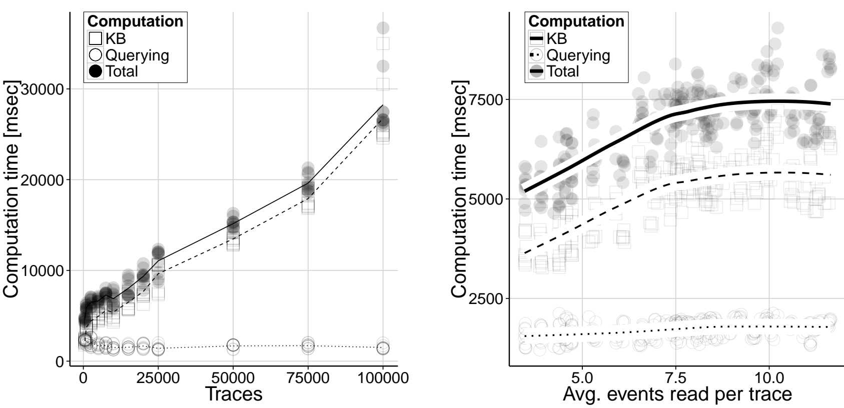
In this case, it is possible to prune the list of discovered constraints in order to obtain a compact set of constraint, which is understandable for human analysts. By applying the miner with a support equal to 1, confidence equal to 0.85, and branching factor 5, we obtain the following 11 constraints:

ChainResponse(A_SUBMITTED, A_PARTLYSUBMITTED)

ChainPrecedence(A_SUBMITTED, A_PARTLYSUBMITTED)

Response(A_SUBMITTED, {A_PREACCEPTED,A_DECLINED,A_CANCELLED})

Response(A_SUBMITTED, {A_PREACCEPTED,A_DECLINED,W_Afhandelen leads})



(a) Computation time as function of the log size

(b) Computation time as function of the average trace size

Fig. 5: Efficiency tests performed on synthetic logs.

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

These results have been derived with a computation time of 7.2 sec for the construction of the knowledge base, and 25.98 min for constraint mining.

6 Related Work

Several analysis tools for Declare are available in the literature. Some of them have been implemented as plug-ins of the process mining tool ProM [12].

Some approaches focus on the run-time monitoring of compliance specifications defined through Declare. For example, in [16,11], the authors propose a technique for monitoring Declare models based on finite state automata. In [29], the authors define *Timed Declare*, an extension of Declare that relies on timed automata. In [19], the EC is used for defining a data-aware semantics for Declare. In [20], the authors propose an approach for monitoring data-aware Declare constraints at run-time based on this semantics. This approach also allows the verification of metric temporal constraints.

Other works [10,3,5,7,17,15,14] focus on the discovery of Declare models. The algorithms proposed in [5,17,15] are suitable for discovering standard Declare

models, also for highly flexible processes [6,4], but cannot be used for dealing with Target-Branched Declare. From this perspective, the approaches proposed in [10,3] are more flexible and allow for the specification of rules that go beyond the traditional Declare templates. However, these approaches can be hardly used in real-world settings since they are based on supervised learning techniques requiring negative examples. In the work proposed in [14], a first-order variant of LTL is used to specify a limited version of data-aware patterns. Such extended patterns are used as the target language for a process discovery algorithm, which produces data-aware Declare constraints from raw event logs. Also in this case Target-Branched Declare is not supported.

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 2012 challenge. In future research, we aim to investigate potential for improving efficiency. We also plan to extend our technique towards the coverage of data, in order to discern which condition leads to a specific choice.

References

1. Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *VLDB*, pages 487–499. Morgan Kaufmann, 1994.
2. A. Burattin, F.M. Maggi, W.M.P. van der Aalst, and A. Sperduti. Techniques for a Posteriori Analysis of Declarative Processes. In *EDOC*, pages 41–50, 2012.
3. F. Chesani, E. Lamma, P. Mello, M. Montali, F. Riguzzi, and S. Storari. Exploiting Inductive Logic Programming Techniques for Declarative Process Mining. *ToPNoC*, 5460:278–295, 2009.
4. Claudio Di Ciccio, Andrea Marrella, and Alessandro Russo. Knowledge-Intensive Processes: An Overview of Contemporary Approaches. In *KiBP*, pages 33–47, 2012.
5. Claudio Di Ciccio and Massimo Mecella. Mining Constraints for Artful Processes. In *BIS*, LNBIP 117, pages 11–23, 2012.
6. Claudio Di Ciccio and Massimo Mecella. Mining Artful Processes from Knowledge Workers' Emails. *IEEE Internet Computing*, 17(5):10–20, 09 2013.
7. Claudio Di Ciccio and Massimo Mecella. A Two-Step Fast Algorithm for the Automated Discovery of Declarative Workflows. In *CIDM*, pages 135–142. 2013.
8. Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management*. Springer, 2013.
9. Dirk Fahland, Daniel Lübke, Jan Mendling, Hajo Reijers, Barbara Weber, Matthias Weidlich, Stefan Zugal. Declarative versus Imperative Process Modeling Languages: The Issue of Understandability. In *BPMDS*, pages 353–366, 2009.

10. Evelina Lamma, Paola Mello, Fabrizio Riguzzi, and Sergio Storari. Applying Inductive Logic Programming to Process Mining. In *Inductive Logic Programming*, LNCS 4894, pages 132–146, 2008.
11. F. M. Maggi, M. Westergaard, M. Montali, and W. M. P. van der Aalst. Runtime Verification of LTL-based Declarative Process Models. In *RV 2011*, LNCS 7186, pages 131–146.
12. Fabrizio Maria Maggi. Declarative Process Mining with the Declare Component of ProM. In *BPM (Demos)*, CEUR 1021, 2013.
13. Fabrizio Maria Maggi, R. P. Jagadeesh Chandra Bose, and Wil M. P. van der Aalst. A Knowledge-Based Integrated Approach for Discovering and Repairing Declare Maps. In *CAiSE*, LNCS 7908, pages 433–448, 2013.
14. Fabrizio Maria Maggi, Marlon Dumas, Luciano García-Bañuelos, and Marco Montali. Discovering Data-Aware Declarative Process Models from Event Logs. In *BPM*, LNCS 8094, pages 81–96, 2013.
15. F.M. Maggi, J.C. Bose, and W.M.P. van der Aalst. Efficient Discovery of Understandable Declarative Models from Event Logs. In *CAiSE*, pages 270–285, 2012.
16. F.M. Maggi, M. Montali, M. Westergaard, and W.M.P. van der Aalst. Monitoring Business Constraints with Linear Temporal Logic: An Approach Based on Colored Automata. In *BPM 2011*, LNCS 6896, pages 132–147, 2011.
17. F.M. Maggi, A.J. Mooij, and W.M.P. van der Aalst. User-Guided Discovery of Declarative Process Models. In *CIDM*, pages 192–199. IEEE, 2011.
18. Jan Mendling, Mark Strembeck, and Jan Recker. Factors of Process Model Comprehension - Findings from a Series of Experiments. *Decision Support Systems*, 53(1):195–206, 2012.
19. Marco Montali, Federico Chesani, Fabrizio Maria Maggi, and Paola Mello. Towards Data-Aware Constraints in Declare. In *SAC*, pages 1391–1396, 2013.
20. Marco Montali, Fabrizio Maria Maggi, Federico Chesani, Paola Mello, and Wil M. P. van der Aalst. Monitoring Business Constraints with the Event Calculus. *ACM TIST*, 5(1):17, 2013.
21. Marco Montali, Maja Pesic, Wil M. P. van der Aalst, Federico Chesani, Paola Mello, and Sergio Storari. Declarative Specification and Verification of Service Choreographies. *ACM Transactions on the Web*, 4(1), 2010.
22. Hajo A. Reijers, Tijs Slaats, and Christian Stahl. Declarative Modeling–An Academic Dream or the Future for BPM? In *BPM*, LNCS 8094, pages 307–322, 2013.
23. Dennis M. M. Schunselaar, Fabrizio Maria Maggi, and Natalia Sidorova. Patterns for a Log-Based Strengthening of Declarative Compliance Models. In *IFM*, LNCS 7321, pages 327–342, 2012.
24. Sergey Smirnov, Matthias Weidlich, and Jan Mendling. Business Process Model Abstraction Based on Synthesis from Well-Structured Behavioral Profiles. *Int. J. Cooperative Inf. Syst.*, 21(1):55–83, 2012.
25. W. M. P. van der Aalst, T. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE TKDE*, 16(9):1128–1142, 2004.
26. Wil M. P. van der Aalst, Maja Pesic, and Helen Schonenberg. Declarative Workflows: Balancing between Flexibility and Support. *CSRD*, 23(2):99–113, 2009.
27. B.F. van Dongen. BPI Challenge 2012, 2012.
28. Matthias Weidlich, Artem Polyvyanyy, Nirmit Desai, Jan Mendling, and Mathias Weske. Process Compliance Analysis Based on Behavioural Profiles. *Inf. Syst.*, 36(7):1009–1025, 2011.
29. Michael Westergaard and Fabrizio Maria Maggi. Looking into the Future: Using Timed Automata to Provide A Priori Advice about Timed Declarative Process Models. In *OTM*, LNCS 7565, pages 250–267, 2012.