Ensuring Model Consistency in Declarative Process Discovery*

Claudio Di Ciccio[1], Fabrizio Maria Maggi[2], Marco Montali[3], and Jan Mendling[1]

[1] Vienna University of Economics and Business, Austria
{claudio.di.ciccio,jan.mendling}@wu.ac.at
[2] University of Tartu, Estonia
f.m.maggi@ut.ee
[3] Free University of Bozen-Bolzano, Italy
montali@inf.unibz.it

Abstract. Declarative process models define the behaviour of business processes as a set of constraints. Declarative process discovery aims at inferring such constraints from event logs. Existing discovery techniques verify the satisfaction of candidate constraints over the log, but completely neglect their interactions. As a result, the inferred constraints can be mutually contradicting and their interplay may lead to an inconsistent process model that does not accept any trace. In such a case, the output turns out to be unusable for enactment, simulation or verification purposes. In addition, the discovered model contains, in general, redundancies that are due to complex interactions of several constraints and that cannot be solved using existing pruning approaches. We address these problems by proposing a technique that automatically resolves conflicts within the discovered models and is more powerful than existing pruning techniques to eliminate redundancies. First, we formally define the problems of constraint redundancy and conflict resolution. Thereafter, we introduce techniques based on the notion of an *automata-product monoid* that guarantee the consistency of the discovered models and, at the same time, keep the most interesting constraints in the pruned set. We evaluate the devised techniques on real-world benchmarks.

1 Introduction

The compact and correct representation of behaviour observed in event data of a business process is one of the major concerns of process mining. Various techniques have been defined for generating models that balance criteria such as fitness and completeness. Mutual strengths and weaknesses of declarative and

* The research of Claudio Di Ciccio and Jan Mendling has received funding from the EU Seventh Framework Programme under grant agreement 318275 (GET Service). The research of Fabrizio Maria Maggi has received funding from the Estonian Research Council and by ERDF via the Estonian Centre of Excellence in Computer Science.

procedural models are discussed in terms of capturing the behaviour of the log in a structured and compact way.

One of the advantages of procedural models such as Petri nets is the rich set of formal analysis techniques available. These techniques can, for instance, identify redundancy in terms of implicit places or inconsistencies like deadlocks. In turn, novel declarative modelling languages like DECLARE have hardly anything to offer as counterparts. This is a problem for several reasons. First, we are currently not able to check the consistency of a generated constraint set. Many algorithms that generate DECLARE models work with confidence and support, often set to values smaller than 1 such that potentially inconsistent constraint sets are returned. Second, it is currently unclear whether a given constraint set is minimal. Since there are constraint types that imply one another, it is possible that constraint sets are generated that are partially redundant. The lack of formal techniques for handling these two issues is unsatisfactory from both a research and a practical angle. It is also a roadblock for conducting fair comparisons in user experiments when a Petri net without deadlocks and implicit places is compared with a constraint set of unknown consistency and minimality.

In this paper, we address the need for formal analysis of DECLARE models. We define the notion of an *automata-product monoid* as a formal notion for analysing consistency and local minimality, which is grounded in automata multiplication. Based on this structure, we devise efficient analysis techniques. Our formal concepts have been implemented as part of a process mining tool, which we use for our evaluation. Using event log benchmarks, we are able to show that inconsistencies and redundancies are indeed likely to occur and that our technique generates constraints sets that are not only consistent, but also substantially smaller than sets provided by prior algorithms.

The paper is structured as follows. Section 2 introduces the problem of inconsistencies and redundancies. In this context, the major concepts of DECLARE are revisited. Section 3 frames the problem. Section 4 defines our formal notion of an automata-product-space, which offers the basis to formalise techniques for checking consistency and local minimality. Section 5 gives an overview of our implementation and the results of our evaluations based on benchmarking data. Section 6 discusses our contributions in the light of related work. Section 7 concludes the paper.

2 Background

This section describes the consistency and minimality problem and revisits the DECLARE concepts.

2.1 The Consistency Problem

In order to illustrate the problem of potential inconsistencies and redundancies, we utilise the event log set of the BPI Challenge 2012 [**?**]. The event log pertains to an application process for personal loans or overdrafts of a Dutch bank. It

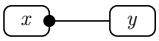
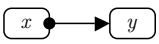
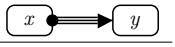
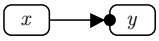
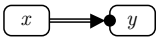
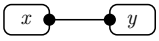
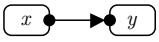
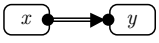
		Template	Regular expression	Notation
Existence templates	Cardinality templates	*Participation*(x)	`[^x]*(x[^x]*)+[^x]*`	
		AtMostOne(x)	`[^x]*(x)?[^x]*`	
	Position templates	*Init*(x)	`x.*`	
		End(x)	`.*x`	
Relation templates	Forward-unidirectional relation templates	*RespondedExistence*(x, y))	`[^x]*((x.*y.*)|(y.*x.*))*[^x]*`	
		Response(x, y)	`[^x]*(x.*y)*[^x]*`	
		AlternateResponse(x, y)	`[^x]*(x[^x]*y[^x]*)*[^x]*`	
		ChainResponse(x, y)	`[^x]*(xy[^x]*)*[^x]*`	
	Backward-unidirectional relation templates	*Precedence*(x, y)	`[^y]*(x.*y)*[^y]*`	
		AlternatePrecedence(x, y)	`[^y]*(x[^y]*y[^y]*)*[^y]*`	
		ChainPrecedence(x, y)	`[^y]*(xy[^y]*)*[^y]*`	
	Coupling templates	*CoExistence*(x, y)	`[^xy]*((x.*y.*)|(y.*x.*))*[^xy]*`	
		Succession(x, y)	`[^xy]*(x.*y)*[^xy]*`	
		AlternateSuccession(x, y)	`[^xy]*(x[^xy]*y[^xy]*)*[^xy]*`	
		ChainSuccession(x, y)	`[^xy]*(xy[^xy]*)*[^xy]*`	
	Negative templates	*NotChainSuccession*(x, y)	`[^x]*(aa*[^xy][^x]*)*([^x]*|x)`	
		NotSuccession(x, y)	`[^x]*(x[^y]*)*[^xy]*`	
		NotCoExistence(x, y)	`[^xy]*((x[^y]*)|(y[^x]*))?`	

Table 1: Semantics of Declare templates as POSIX regular expressions [**?**]

contains 262,200 events distributed across 24 different possible tasks and 13,087 traces. In general, an *event log* L is as a collection of traces $\boldsymbol{t}_i$ with $i \in [1, |L|]$, which in turn are finite sequences of events $e_{i,j}$ with $i \in [1, |L|]$ and $j \in [1, |\boldsymbol{t}_i|]$. Each event refers to a task. The *log alphabet* $\mathfrak{A}$ is the set of symbols identifying all possible tasks and we write $\mathtt{a}, \mathtt{b}, \mathtt{c}$ to refer to them.

Process mining tools such as MINERful [**?**] and Declare Maps Miner [**?**] generate declarative process models in DECLARE from event logs. In essence, these models define a set of declarative constraints that collectively determine the allowed and forbidden traces. Each constraint is defined using a *template* that captures the semantics of the constraint using generic parameters. We generically refer to parameters of templates as x, y, z. Table 1 summarises the available templates. A template is then instantiated by assigning parameters to actual tasks. For instance, *Response*$(\mathtt{a}, \mathtt{b})$ is a constraint imposing that if `a` is executed, then `b` must be eventually executed in the future. In this example, `a` and `b` are the assigned parameters of *Response*(x, y). We define $\mathfrak{C}$ as the set of templates

and refer to $\mathfrak{C}_{\mathfrak{A}}$ as the set of constraints constructed by considering all possible parameter assignments of the templates in $\mathfrak{C}$ to the tasks in $\mathfrak{A}$.

The main idea of declarative process mining is that overfitting of the discovered models can be avoided by defining thresholds for parameters such as support. The support (*supp*) of a constraint is defined as the number of traces verifying the constraint divided by the total number of traces in the event log. Additional metrics are confidence (*conf*) and interest factor (*IF*), which scale the support by the percentage of traces in which the constraint is triggered, resp. both parameters occur. By choosing a support threshold smaller than 100%, we can easily obtain constraint pairs that are supported by different parts of the log and such that the first contradicts the second. E.g., when using MINERful on the BPIC 2012 event log with a support threshold of 75%, it returns the constraints *NotChainSuccession*(A_PREACCEPTED, W_Completeren aanvrag) and *ChainResponse*(A_PREACCEPTED, W_Completeren aanvrag), which have an empty set of traces that fulfil both. In fact, the first constraint imposes that A_PREACCEPTED can never be directly followed by W_Completeren aanvrag, whereas the second one requires that if A_PREACCEPTED is executed, W_Completeren aanvrag must immediately follow. Clearly, such inconsistent constraint pairs should not be returned. Models with inconsistencies cannot be used for simulation nor execution, and process analysts might be confused by these results.

2.2 The Minimality Problem

The second problem next to consistency is minimality. As observed in [**?**,**?**], DECLARE templates can be organised in a hierarchy of constraints, depending on a notion of subsumption. Technically, given the names N_1 and N_2 of two templates $\mathcal{C}, \mathcal{C}' \in \mathfrak{C}$ of the same arity, we say that $\mathcal{C}$ *is subsumed by* $\mathcal{C}'$, written $N_1 \sqsubseteq N_2$, if for every trace $\boldsymbol{t}$ over $\mathfrak{A}$ and every parameter assignment σ from the parameters of $\mathcal{C}$ to tasks in $\mathfrak{A}$, whenever $\boldsymbol{t}$ complies with the instantiation of $\mathcal{C}$ determined by σ, then $\boldsymbol{t}$ also complies with the instantiation of $\mathcal{C}'$ determined by σ. For binary constraints, we write $N_1 \sqsubseteq N_2^{-1}$ if the subsumption holds by inverting the parameters of $\mathcal{C}'$ w.r.t. those in $\mathcal{C}$, i.e., by considering templates $N_1(x, y)$ and $N_2(y, x)$.

For example, *RespondedExistence*(a, b) states that if a occurs in a trace, then b has to occur in the same trace (either before or after a). *Response*(a, b) thus enforces *RespondedExistence*(a, b) by stating that not only must b be executed, but also that it must *follow* a. By generalising, we have then *Response* $\sqsubseteq$ *RespondedExistence*. By the same line of reasoning, we have that *Precedence* $\sqsubseteq$ *RespondedExistence*$^{-1}$.

Based on the concept of subsumption, we can define the notion of relaxation, $\mathscr{R}$. $\mathscr{R}$ is a unary operator that returns the direct parent in the subsumption hierarchy of a given template. If there exists no parent for the given template, then $\mathscr{R}$ returns a predicate that would hold true for any possible trace, i.e., $\top$.

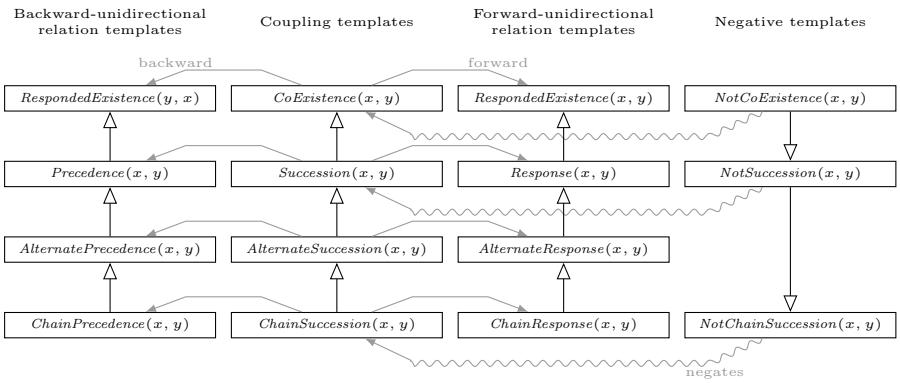


Fig. 1: The subsumption map of DECLARE relation templates

Formally, given a template $\mathcal{C} \in \mathfrak{C}$, we have:

$$\mathscr{R}(\mathcal{C}) = \begin{cases} \mathcal{C}' & \text{if } \textit{(i)}\ \mathcal{C}' \in \mathfrak{C} \setminus \{\mathcal{C}\},\ \textit{(ii)}\ \mathcal{C} \sqsubseteq \mathcal{C}',\ \text{and} \\ & \textit{(iii)}\ \nexists \mathcal{C}'' \in \mathfrak{C} \setminus \{\mathcal{C}, \mathcal{C}'\} \text{ s.t. } \mathcal{C} \sqsubseteq \mathcal{C}'' \sqsubseteq \mathcal{C}' \\ \top & \text{otherwise} \end{cases}$$

We extend the relaxation operator and the subsumption relation also to the domain of constraints: Hence, e.g., $\mathscr{R}(\textit{Response}(\mathsf{a}, \mathsf{b})) = \textit{RespondedExistence}(\mathsf{a}, \mathsf{b})$. Figure 1 depicts the subsumption hierarchy for relation templates. The forward and backward components are specified for coupling templates, and the negative templates are linked with their negated counterparts. Note that, in addition to the specified template subsumption, also $\textit{Init}(x)$ and $\textit{End}(x)$ are subsumed by $\textit{Participation}(x)$.

When using MINERful on BPIC 2012 with a support threshold of 75%, it returns the constraints $\textit{ChainResponse}(\mathsf{A\_SUBMITTED}, \mathsf{A\_PARTLYSUBMITTED})$ and $\textit{NotChainSuccession}(\mathsf{A\_SUBMITTED}, \mathsf{A\_ACCEPTED})$. The latter constraint is clearly redundant, because the former requires the first task following A_SUBMITTED to be A_PARTLYSUBMITTED. Therefore, no other task but A_PARTLYSUBMITTED can directly follow. A fortiori, A_SUBMITTED and A_ACCEPTED cannot be in direct succession. Clearly, such redundant constraint pairs should not be returned. Models that are not minimal are difficult to understand for the process analysts. Also, redundant constraints do not provide any additional information about the permitted behaviour.

3 Framing the Problem

In Section 2, we have informally introduced the issues of consistency and redundancy in declarative process discovery. We now specify the problem more precisely. Our goal is to define *effective* post-processing techniques that, given

a previously discovered DECLARE model M possibly containing inconsistencies and redundancies, manipulate it by removing inconsistencies and reducing redundancies, but still retaining as much as possible its original structure. In this respect, the post-processing is completely agnostic to the process mining algorithm used to generate the model, as well as to the input event log.

This latter assumption makes it impossible to understand how much a variant of the discovered model "fits" with the log. However, we can at least assume that each single constraint in M retains the support, confidence, and interest factor that were calculated during the discovery phase. These values can be used to decide which constraints have to be preferred, and ultimately decide whether a variant M' of M has to be preferred over another one M''. Still, notice that by no means such values can be composed to calculate a global support/confidence/interest factor for the whole model M'. This is only possible if the original log is considered. To see this, consider the case of two constraints C_1, C_2, with support $s_1, s_2 < 100\%$. When the two constraints are considered together, the global support could range from 0 to the minimum of s_1 and s_2, and the exact value could only be determined by computing it directly over the log.

In principle, we could obtain an optimal solution by exhaustive enumeration, executing the following steps. 1. The vocabulary Σ of M is extracted. 2. The set $\mathfrak{C}_\Sigma$ of all possible candidate constraints is built. 3. The set $\mathcal{P}^{\mathfrak{C}_\Sigma}$ of all possible subsets of $\mathfrak{C}_\Sigma$, i.e., of all possible DECLARE models using constraints in $\mathfrak{C}_\Sigma$, is computed. 4. A set $\mathcal{K}$ of candidate models is obtained from $\mathcal{P}^{\mathfrak{C}_\Sigma}$, by filtering away those models that are inconsistent or contain redundant constraints. 5. A ranking of the models in $\mathcal{K}$ is established, considering their similarity to the original, discovered model M.

However, this exhaustive enumeration is in general unfeasible, given the fact that it requires to iterate over the exponentially many models in $\mathcal{P}^{\mathfrak{C}_\Sigma}$, a too huge state space. Consequently, we devise a heuristic algorithm that mediates between optimality of the solution, and performance. In summary, its main features are:

- It produces as output a consistent variant of the initial model M. This is a strict, necessary requirement.
- The algorithm works in an incremental fashion, i.e., it constructs the variant of M by iteratively selecting constraints, and once a constraint is added, it is never retracted from the model. This is done by iterating through candidate constraints in decreasing order of "suitability" w.r.t. the input log, which is computed by considering the support/confidence/interest factor of such constraints. On the one hand, this drives our algorithm to favour more suitable constraints, and remove less suitable constraints in the case of an inconsistency. On the other hand, this has a positive effect on performance, and also guarantees that the algorithm is deterministic.
- Due to incrementality, the algorithm is not guaranteed to produce a final variant that is optimal in size, but we obtain a local minimum. However, our experimental findings show that the algorithm is able to significantly reduce the number of redundant constraints.

4 The approach

This section describes how we tackle the problem of finding a non-redundant consistent DECLARE model in a way that reduces the intractable theoretical complexity. First, we present the algebraic structure on top of which the check of redundancies and conflicts is performed: It bases upon the mapping of the conjunction of DECLARE constraints to the product of finite state automata (FSAs). Thereafter, we define and discuss the algorithm that allows us to pursue our objective. In particular, we rely on the associativity of the product of FSAs. This property allows us to check every constraint one at a time and include it in a temporary solution. This is done by saving the product of the constraints checked so far with the current one. For the selection of the next candidate constraint to check, we make use of a greedy heuristic, that explores the search space by gathering at every step the constraint that has the highest support, or is most likely to imply the highest number of other constraints. The algorithm proceeds without visiting the same node in the search space twice.

4.1 Declare models as automata

As already shown in [**?**], DECLARE constraints can be formulated as regular expressions (REs) over the log alphabet. The assumption is that every task in the log alphabet is bi-univocally identified by a character. Thus, traces can be assimilated to finite sequences of characters (i.e., strings) and regular languages represent the traces allowed by a DECLARE model.

Using the `POSIX` wildcards, we can express, e.g., $\mathit{Init}(\mathtt{a})$ as `a.*`, and $\mathit{Response}(\mathtt{a},\mathtt{b})$ as `[^a]*(a.*b)*[^a]*`. The comprehensive list of transpositions for DECLARE templates is listed in Table 1 and explained in [**?**]. Henceforth, we will refer to such mapping as $\mathscr{E}_{\mathrm{Reg}}(C)$, which takes as input a constraint C and returns the corresponding RE: E.g., $\mathscr{E}_{\mathrm{Reg}}(\mathit{Response}(\mathtt{a},\mathtt{b}))$=`[^a]*(a.*b)*[^a]*`. Defining the operations of conjunction between DECLARE constraints ($\wedge$) and intersection between REs (&&), $\mathscr{E}_{\mathrm{Reg}}$ is a monoid homomorphism w.r.t. $\wedge$ and &&. In other words, given two constraints C and C', $\mathscr{E}_{\mathrm{Reg}}(C \wedge C') = \mathscr{E}_{\mathrm{Reg}}(C)$ && $\mathscr{E}_{\mathrm{Reg}}(C')$, preserving closure, associativity and the identity element (resp., $\top$ and `.*`).

Since regular grammars are recognisable through REs [**?**], an RE can always be associated to a deterministic labelled FSA, which accepts all and only those finite strings that match the RE. Formally, an FSA is a tuple $\mathcal{S} = \langle \Sigma, S, s_0, \delta, S^f \rangle$, where: Σ is the alphabet; S is the finite non-empty set of states; $s_0 \in S$ is the initial state; $\delta : S \times \Sigma \to S$ is the transition function; $S^f \subseteq S$ is the set of final states. Naming as $\mathscr{A}$ the operation leading from an RE to an FSA, we thus have that a DECLARE constraint can be associated with its corresponding FSA, $A^C = \mathscr{A}(\mathscr{E}_{\mathrm{Reg}}(C))$. Henceforth, we also call A^C the C-automaton. We remark that, by applying $\mathscr{A}$ to the RE of a conjunction of constraints, we obtain an FSA that exactly corresponds to the product $\times$ of the FSAs for the individual constraints [**?**]: $\mathscr{A}(\mathscr{E}_{\mathrm{Reg}}(C \wedge C')) = \mathscr{A}(\mathscr{E}_{\mathrm{Reg}}(C)) \times \mathscr{A}(\mathscr{E}_{\mathrm{Reg}}(C'))$. Also, we recall that the identity element for FSAs is a single-state automaton whose unique

state is both initial and accepting, and has a self-loop for each character in the considered alphabet.

Given a model $M = \{C_1, \ldots, C_{|M|}\}$, we can therefore implicitly describe the set of traces that comply with M as the language accepted by the product of all C_i-automata (for $i \in [1, |M|]$). The language accepted by an FSA A will be denoted as $\mathscr{L}(A)$. In the light of this discussion, our approach searches a solution to the problem of finding a non-redundant consistent DECLARE model within the **automata-product monoid**, i.e., the associative algebraic structure with identity element (the universe-set of FSAs) and product operation $\times$. For the automata-product monoid, the property of commutativity also holds.

4.2 The algorithm

Algorithm 1 outlines the pseudocode of our technique. Its input is a DECLARE model, M, intended as a set of constraints $C_1, \ldots, C_{|M|}$. For every $C \in M$, we assume that its support, confidence and interest factor are given too, which is the usual condition when M is the output of mining algorithms such as Declare

Algorithm 1: Procedure makeConsistent(M), returning the suboptimal solution to the problem of finding a minimal set of non-conflicting constraints in a DECLARE model.

Input: A log alphabet $\mathfrak{A}$, and a DECLARE model M defined over $\mathfrak{A}$. M is a set of constraints for which support, confidence and interest factor are given
Output: Set of non-conflicting constraint M^{R}

1 $M' \leftarrow$ removeSubsumptionHierarchyRedundancies(M)
2 $M^{\mathrm{S}} \leftarrow \{C \in M' : supp(C) = 1.0\}$ // Non-conflicting constraints
3 $M^{\mathrm{U}} \leftarrow M' \setminus M^{\mathrm{S}}$ // Potentially conflicting constraints
4 $A \leftarrow \langle \mathfrak{A}, \{s_0\}, s_0, \{\bigcup_{\sigma \in \mathfrak{A}} \langle s_0, \sigma, s_0 \rangle, \{s_0\}\} \rangle$ // Automaton accepting any sequence of tasks
5 $M^{\mathrm{R}} \leftarrow \emptyset$ // Set of returned constraints
6 $M^{\mathrm{V}} \leftarrow \emptyset$ // Set of checked constraints
/* Pruning of redundant constraints from the set of non-conflicting ones */
7 $M^{\mathrm{S}}_{\mathrm{list}} \leftarrow$ sortBySupportCategoryConfidenceIF(M^{S})
8 **foreach** $C_i^{\mathrm{M^S}} \in M^{\mathrm{S}}_{\mathrm{list}}$, with $i \in [1, |M^{\mathrm{S}}_{\mathrm{list}}|]$ **do**
9 $M^{\mathrm{V}} \leftarrow M^{\mathrm{V}} \bigcup \{C_i^{\mathrm{M^S}}\}$ // Record that $C_i^{\mathrm{M^S}}$ has been checked
10 $A^{C_i^{\mathrm{M^S}}} \leftarrow \mathscr{A}\left(\mathscr{E}_{\mathrm{Reg}}\left(C_i^{\mathrm{M^S}}\right)\right)$ // Build the constraint-automaton of $C_i^{\mathrm{M^S}}$
11 **if** $\mathscr{L}(A) \supset \mathscr{L}\left(A^{C_i^{\mathrm{M^S}}}\right)$ **then** // If $C_i^{\mathrm{M^S}}$ is not redundant
12 $A \leftarrow A \times A^{C_i^{\mathrm{M^S}}}$ // Merge the $C_i^{\mathrm{M^S}}$-automaton with the main FSA
13 $M^{\mathrm{R}} \leftarrow M^{\mathrm{R}} \bigcup \{C_i^{\mathrm{M^S}}\}$ // Include $C_i^{\mathrm{M^S}}$ in the set of returned constraints
/* Pruning of conflicting constraints */
14 $M^{\mathrm{U}}_{\mathrm{list}} \leftarrow$ sortBySupportCategoryConfidenceIF(M^{U})
15 **foreach** $C_i^{\mathrm{M^U}} \in M^{\mathrm{U}}_{\mathrm{list}}$, with $i \in [1, |M^{\mathrm{U}}_{\mathrm{list}}|]$ **do**
16 resolveConflictAndRedundancy$\left(A, M^{\mathrm{R}}, C_i^{\mathrm{M^U}}, M^{\mathrm{V}}\right)$
17 **return** removeSubsumptionHierarchyRedundancies(M^{R})

Algorithm 2: Procedure resolveConflictAndRedundancy $\left(A, M^{\mathrm{R}}, C, M^{\mathrm{V}}\right)$, adding constraint C to the set of constraint M^{R}, if it has not already been checked (and thus included in set M^{V}), and is not conflicting with the already added constraints, as verified over the corresponding FSA A.

Input: An FSA A, a set of non-conflicting constraints M^{R}, a constraint C, and a list of already checked constraints M^{V}

```
1  if C ∉ M^V then                                          // If C was not already checked
2      M^V ← M^V ⋃ {C}                                      // Record that C has been checked
3      A^C ← 𝒜(ℰ_Reg(C))                                    // Build the C-automaton
4      if ℒ(A) ⊃ ℒ(A^C) then                                // If C is not redundant
5          if ℒ(A × A^C) ≠ ∅ then                           // If C is not conflicting
6              A ← A × A^C                                  // Merge the C-automaton with the main FSA
7              M^R ← M^R ⋃ {C}                              // Include C in the set of returned constraints
8          else                                             // Otherwise, resolve the conflict
9              if ℛ(C) ≠ ⊤ then                             // If a relaxation of C, i.e., ℛ(C), exists
10                 resolveConflictAndRedundancy(A, M^R, ℛ(C), M^V)
11             if C is a coupling constraint then
12                 resolveConflictAndRedundancy(A, M^R, fw(C), M^V)
13                 resolveConflictAndRedundancy(A, M^R, bw(C), M^V)
```

Maps Miner or MINERful. Table 2a shows an example of M, defined on the log alphabet $\{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$. We also assume that the same metrics are defined for those constraints that are not in M, yet are either their subsuming, negated, forward or backward version. Again, this is common in the output of the aforementioned algorithms. For the sake of readability, these additional constraints are not reported in Table 2a. Table 2b shows the output that corresponds to the post-processing of Table 2a. Constraints that are considered as redundant are coloured in grey. Struck-out constraints are those that are in conflict with the others and thus dropped from the returned set.

Given M, the first operation "removeSubsumptionHierarchyRedundancies" prunes out redundant constraints based on the subsumption hierarchy. The procedure considers a removal of the subsuming constraints such that their support is less than or equal to the subsumed one, and the elimination of forward and backward constraints if the related coupling constraint has an equivalent support. Detail of this operations have already been described in [**?**]. The usefulness of this procedure resides in the fact that it reduces the number of candidate constraints to be considered, thus reducing the number of iterations performed by the algorithm. In Table 2b, this operation is responsible for the dropping of *Participation*(a), due to the fact that *Init*(a) is known to hold true.

Thereafter, we partition M into two subsets, i.e.: *(i)* M^{S}, consisting of those constraints that are verified over the entire event log (i.e., having a support of 1.0), and *(ii)* M^{U}, containing the remaining constraints. The reason for doing this is that the former is guaranteed to have no conflict: Given the fact that

Constraint	*Supp*	*Conf*	*IF*
Init(a)	1.0	1.0	1.0
Participation(a)	1.0	1.0	1.0
CoExistence(a, d)	1.0	1.0	1.0
End(d)	1.0	1.0	1.0
NotChainSuccession(b, d)	1.0	0.9	0.8
NotChainSuccession(a, d)	0.75	0.5	0.5
ChainResponse(b, c)	1.0	0.9	0.8
NotChainSuccession(a, b)	0.9	0.7	0.6
NotChainSuccession(a, c)	0.8	0.7	0.6
ChainResponse(b, a)	0.75	0.9	0.9

(a) Input

	i	Constraint	*Supp*	*Conf*	*IF*
$M^{\mathrm{S}}_{\mathrm{list}}$	1	*Init*(a)	1.0	1.0	1.0
	2	*End*(d)	1.0	1.0	1.0
	3	*CoExistence*(a, d)	1.0	1.0	1.0
	4	*ChainResponse*(b, c)	1.0	0.9	0.8
	5	*NotChainSuccession*(b, d)	1.0	0.9	0.8
$M^{\mathrm{U}}_{\mathrm{list}}$	1	*NotChainSuccession*(a, b)	0.9	0.7	0.6
	2	*NotChainSuccession*(a, c)	0.8	0.7	0.6
	3	~~*NotChainSuccession*(a, d)~~	~~0.75~~	~~0.5~~	~~0.5~~
	4	*AlternateResponse*(b, a)	0.75	0.9	0.9

(b) Processed output

Table 2: An example of input constraint set processing

constraints are mined using the alphabet of the event log, those that have a support of 1.0 can be conjoined, giving raise to a *consistent* constraint model.

Even though constraints in M^{S} are guaranteed to be conflict-free, they could still contain redundancies. Therefore, the following part of the algorithm is dedicated to the elimination of redundant constraints from this set. To check redundancies, we employ the characterisation of constraints in terms of FSAs. Instead, constraints in M^{U} may contain both redundancies and inconsistencies. Table 2b presents the partition of M into M^{S} and M^{U}.

First, we initialise an FSA A to be the identity element w.r.t. automata product. In other words, A is initialised to accept any sequence of events that map to a task in the log alphabet. This automata incrementally incorporates those constraints that are maintained in the filtered model. To set up the redundancy elimination in M^{S} as well as the redundancy and inconsistency elimination in M^{U}, we then order their constitutive constraints according the the following criteria (in descending order of priority): *(i)* descending support (this is trivial for M^{S}, since all constraints have a support of 1.0);*(ii)* category – consider first existence constraints, then positive relation constraints, and finally negative constraints; *(iii)* descending confidence; *(iv)* descending interest factor. This ranking is of utmost importance, as it determines the priority with which constraints are analysed. The priority, in turn, implicitly defines the "survival expectation" of a constraint, as constraints that come later in the list are more likely to be pruned if they are either redundant or conflicting.

We briefly explain the reason for this multi-dimensional ranking. Support is the first criterion adopted, because we prefer to preserve those constraints that are satisfied in the most part of the log. The category criterion is instead driven by the expertise acquired in the last years in the context of DECLARE mining [**?**,**?**]. In particular, we tend to preserve those constraints that have the potential of inducing the removal of a massive amount of other constraints, due to redun-

dancy. As an example, consider the case of the *Init* template: Given $\rho \in \mathfrak{A}$, if $Init(\rho)$ holds true, then also the relation constraint $Precedence(\rho, \sigma)$ is guaranteed to hold true, for every $\sigma \in \mathfrak{A} \setminus \{\rho\}$. This means that, in the best case, $|\mathfrak{A}|-1$ constraints will be removed because they are all redundant with $Init(\rho)$. Similarly, consider the positive relation constraint $ChainResponse(\rho, \sigma)$: It implies $NotChainSuccession(\rho, \sigma')$ for every $\sigma' \in \mathfrak{A} \setminus \{\rho, \sigma\}$. Thus, $ChainResponse(\rho, \sigma)$ has the potential of triggering the removal of $|\mathfrak{A}| - 2$ negative constraints due to redundancy. The last criteria adopted pertain confidence and interest factor, in order to prefer those constraints whose parameters occur in most traces. In Algorithm 1, the computation of this ranking is encapsulated inside function "sortBySupportCategoryConfidenceIF", which returns a list of constraints ordered according to the aforementioned criteria. In Table 2b, the result of the sorting is reported.

After the sorting, constraints are iteratively considered for inclusion in the refined model, by iterating through the corresponding ranked lists. Constraints in the list of M^{S}, i.e., $C_i^{\mathrm{M}^{\mathrm{S}}} \in M^{\mathrm{S}}_{\mathrm{list}}$, are only checked for redundancy, whereas constraints in M^{U}, $C_i^{\mathrm{M}^{\mathrm{U}}} \in M^{\mathrm{U}}_{\mathrm{list}}$, are checked for both redundancy and consistency. For every constraint $C_i^{\mathrm{M}^{\mathrm{S}}} \in M^{\mathrm{S}}_{\mathrm{list}}$, redundancy is checked by leveraging language-inclusion. In particular, this is done by computing the FSA $A^{C_i^{\mathrm{M}^{\mathrm{S}}}}$ for $C_i^{\mathrm{M}^{\mathrm{S}}}$, and then checking whether its generated language $\mathscr{L}\left(A^{C_i^{\mathrm{M}^{\mathrm{S}}}}\right)$ is included inside $\mathscr{L}(A)$, which considers the contribution of all constraints maintained so far. If this is the case, then the constraint is dropped. Otherwise, A is extended with the contribution of this new constraint (by computing the product $A \times A^{C_i^{\mathrm{M}^{\mathrm{S}}}}$), and $C_i^{\mathrm{M}^{\mathrm{S}}}$ is added to the set M^{R} of constraints to be returned. In the example of Table 2b, $CoExistence(\mathtt{a}, \mathtt{d})$ is analysed after the existence constraints $Init(\mathtt{a})$ and $End(\mathtt{d})$, based on the preliminary sorting operation. It thus turns out to be redundant, because $Init(\mathtt{a})$ and $End(\mathtt{d})$ already specify that both `a` and `d` will occur in every trace. Therefore, they will necessarily always co-occur.

Redundancy and consistency checking of the constraints $C_i^{\mathrm{M}^{\mathrm{U}}} \in M^{\mathrm{U}}_{\mathrm{list}}$ is performed by the "resolveConflictAndRedundancy" procedure (Algorithm 2). The procedure checks the consistency of those constraints that are not redundant. The redundancy is, again, checked based on the language inclusion of the language generated by the currently analyzed constraint $\mathscr{L}\left(A^{C_i^{\mathrm{M}^{\mathrm{U}}}}\right)$ in $\mathscr{L}(A)$, where A is the automaton that accumulates the contribution of all constraints that have been kept so far. The consistency is checked through a language emptiness test, performed over the intersection of $\mathscr{L}\left(A^{C_i^{\mathrm{M}^{\mathrm{U}}}}\right)$ and $\mathscr{L}(A)$. This is done by checking that $\mathscr{L}\left(A \times A^{C_i^{\mathrm{M}^{\mathrm{U}}}}\right) \neq \emptyset$. In case a conflict is detected, we do not immediately drop the conflicting constraint, but we try, instead, to find a more relaxed constraint that retains its intended semantics as much as possible, but does not incur in a conflict. To do so, we employ the constraint subsumption hierarchy (cf. Section 2.2). In particular, we employ the relaxation operator to retrieve the parent constraint of the conflict-

ing one, and we recursively invoke the "resolveConflictAndRedundancy" procedure over the parent. The recursion terminates when the first non-conflicting ancestor of the conflicting constraint is found, or when the top of the hierarchy is reached. The two cases are resp. covered in the example of Table 2b by *ChainResponse*(b, a), replaced by *AlternateResponse*(b, a), and by *NotChainSuccession*(a, d), which is removed because a non-conflicting ancestor does not exists. Note that *NotChainSuccession*(a, d) is to be eliminated because of the interplay of the other two *NotChainSuccession* constraints, *Init*(a) and *End*(d). *ChainResponse*(b, a) is in conflict with *ChainResponse*(b, c).

If the constraint under analysis is a coupling constraint, then we know that it is constituted by the conjunction of a corresponding pair of forward and backward constraints. In this situation, it could be the case that all the relaxations of the coupling constraint along the subsumption hierarchy continue to be conflicting, but the conflict would be removed by just considering either its forward or backward component (or a relaxation thereof). Consequently, we also recursively invoke the "resolveConflictAndRedundancy" procedure on these two components.

Finally, a last complete pass over constraints in M^{R} is done, to check again whether there are subsumption-hierarchy redundancies. If so, M^{R} is pruned accordingly.

5 Experiments and results

Our experimentation is based on the application of the proposed approach to the event log provided for the BPI challenge 2012. In the first set of experiments, we use MINERful to mine the log. We discover the set of constraints with a support higher than 75%, a confidence higher than 12.5%, and an interest factor higher than 12.5%. The discovered constraints are 306. The total execution time is of 9,171 milliseconds. By applying the proposed algorithm, we obtain 130 constraints in total. In the original set of 306 there are 2 conflicting constraints that make the entire model inconsistent. These constraints are *NotChainSuccession*(A_PREACCEPTED, W_Completeren aanvrag), conflicting with *ChainResponse*(A_PREACCEPTED, W_Completeren aanvrag), and *NotChainSuccession*(W_Completeren aanvraag, A_ACCEPTED), conflicting with *ChainResponse*(W_Completeren aanvraag, A_ACCEPTED) for similar reasons. Note that the percentage of reduction over the set of discovered constraints (that was already pruned based on the subsumption hierarchy) is of 58%.

In the second set of experiments, we have applied the Declare Maps Miner to mine the log. We discovered the set of constraints with a support higher than 75% confidence higher than 12.5% and interest factor higher than 12.5%. The set of discovered constraints pruned based on the diverse pruning techniques provided by the tool contains 69 constraints. By applying the proposed algorithm starting from this set, we obtain 41 constraints (with an execution time of 2,764 milliseconds). The percentage of reduction is still of around 40%.

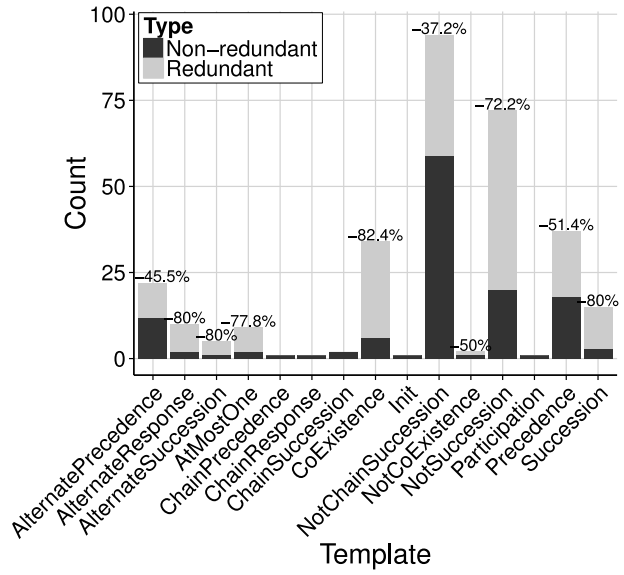


Fig. 2: Redundancy reduction w.r.t. templates

Figure 2 shows the number of discovered constraints using MINERful. In particular, the plot shows the percentage of templates that are redundant and then pruned by the proposed algorithm and the ones that are not redundant and, therefore, discovered. For some templates, it is easy to explain why a high percentage of constraints become redundant. For example, *CoExistence* constraints are more often pruned because they are weaker than others and are transitive so that very often their transitive closures become redundant [**?**]. For example, if *CoExistence*(a, b), *CoExistence*(b, c), and *CoExistence*(a, c) are valid, one of them is always redundant. On the other hand, other constraints, like the ones based on "chain" templates are stronger and not transitive and then pruned less often.

In general, redundant constraints can be pruned based on very complex reduction rules. For example, from our experiments, we derived that *AtMostOne*(A_FINALIZED) becomes redundant due to the presence in combination of *AtMostOne*(A_PARTLYSUBMITTED), *Participation*(A_PARTLYSUBMITTED), and *AlternatePrecedence*(A_PARTLYSUBMITTED, A_FINALIZED). Indeed, *Participation*(A_PARTLYSUBMITTED) and *AtMostOne*(A_PARTLYSUBMITTED) combined ensure that A_PARTLYSUBMITTED occurs exactly once. Then *AlternatePrecedence*(A_PARTLYSUBMITTED, A_FINALIZED) ensures that either A_FINALIZED does not occur or if it occurs it is preceded by the unique occurrence of A_PARTLYSUBMITTED without the possibilities of other occurrences of A_FINALIZED in between. Another example is *NotSuccession*(W_Nabellen offertes, A_SUBMITTED), which is redundant with the combination of *Init*(A_SUBMITTED),

AtMostOne(A_PARTLYSUBMITTED), *Participation*(A_PARTLYSUBMITTED), and *ChainSuccession*(A_SUBMITTED, A_PARTLYSUBMITTED). Indeed, *AtMostOne*(A_PARTLYSUBMITTED) and *Participation*(A_PARTLYSUBMITTED) combined ensure that A_PARTLYSUBMITTED occurs exactly once. This constraint in combination with *ChainSuccession*(A_SUBMITTED, A_PARTLYSUBMITTED) and *Init*(A_SUBMITTED) ensures that A_SUBMITTED occurs only once at the beginning of every trace and, therefore, it can never occur after any other activity.

All experiments were run on a machine equipped with an Intel Core i5-3320M, CPU at 2.60GHz, quad-core, Ubuntu Linux 12.04 operating system. The tool has been implemented in Java SE 7 and integrated with the MINERful declarative process miner. It can be downloaded at: www.github.com/cdc08x/MINERful.

6 Related work

Our research relates to three streams of research: Consistency checking for knowledge bases, research on process mining, and specifically research on DECLARE. Research in the area of knowledge representation has considered the issue of consistency checking. In particular, in the context of Knowledge-based configuration systems, Felfernig et al. [**?**] have challenged the problem of finding the core cause of inconsistencies within the knowledge base during its update test, in terms of minimal conflict sets (the so-called diagnosis). The proposed solution relies on the recursive partitioning of the (extended) CSP problem into subproblems, skipping those that do not contain an element of the propagation-specific conflict [**?**]. In the same research context, the work described in [**?**] focuses on the detection of non-redundant constraint sets. The approach is again based on a divide-and-conquer approach, that favours however those constraints that are ranked higher in a lexicographical order. Differently from such works, we tend to exploit the characteristics of DECLARE templates in a sequential exploration of possible solutions. As in their proposed solutions, though, we base upon a preference-oriented ranking when deciding which constraints to keep in the returned set.

The problem of consistency arises in process mining when working with behavioural constraints. Constraint sets as those of the α algorithm [**?**] and its extension [**?**] or behavioural profiles [**?**,**?**] are per construction consistent. DCR graphs are not directly discussed from the perspective of consistency [**?**], but benefit from our work due to their grounding in Büchi automata.

More specifically, our work is related to research on DECLARE and strategies to keep sets small and consistent. In [**?**], the authors present an approach based on the instantiation of a set of candidate DECLARE constraints that are checked with respect to the log to identify the ones that are satisfied in a higher percentage of traces. This approach has been improved in [**?**] by reducing the number of candidates to be checked through an apriori algorithm. In [**?**], the same approach has been applied for the repair of DECLARE models based on log and for guiding the discovery task based on apriori knowledge provided in different forms. In this work, some simple reduction rules are presented. These reduction rules are,

however, not sufficient to detect redundancies due to complex interactions among constraints in a discovered model as demonstrated in our experimentation.

In [**?**,**?**], the authors present an approach for the mining of declarative process models expressed through a probabilistic logic. The approach first extract a set of integrity constraints from a log. Then, the learned constraints are translated into Markov Logic formulas that allow for a probabilistic classification of the traces. In [**?**,**?**], the authors present an approach based on Inductive Logic Programming techniques to discover DECLARE process models. These approaches are not equipped with techniques for the analysis of the discovered models like the one presented in this paper.

In [**?**,**?**], the authors introduce a two-step algorithm for the discovery of DECLARE constraints. As a first step, a knowledge base is built, with information about temporal statistics gathered from logs. Then, the statistical support of constraints is computed, by querying that knowledge base. Also these works introduce a basic way to deal with redundancy based on the subsumption hierarchy of DECLARE templates that is non capable to deal with redundancies due to complex interactions of constraints.

In [**?**], the authors propose an extension of the approach presented in [**?**,**?**] to discover target-branched DECLARE constraints, i.e., constraints in which the target parameter is replaced by a disjunction of actual tasks. Here, as well as redundancy reductions based on the subsumption hierarchy of DECLARE constraints, also different aspects of redundancy are taken into consideration that are characteristic of target-branched DECLARE, such as set-dominance.

7 Conclusion

In this paper, we addressed the problems of redundant and inconsistent constraint sets that are potentially generated by declarative process mining tools. We formalised the problem based on the notion of automata-product monoid and devised the corresponding analysis algorithms. The evaluation based on our prototypical implementation shows that typical constraint sets can be further pruned such that the result is consistent and locally minimal. Our contribution complements research on declarative process execution and simulation and provides the basis for a fair comparison of procedural and declarative representations.

In future research, we aim at extending our work towards other perspectives of processes. When mining declarative constraints with references to data and resources, one of the challenges will be to identify comparable notions of subsumption and causes of inconsistency. We also plan to follow up on experimental research comparing Petri nets and DECLARE. The notions defined in this paper help design declarative and procedural process models that are equally consistent and minimal, such that an unbiased comparison would be feasible.