Dear Author,

Please check your proof carefully and mark all corrections at the appropriate place in the proof (e.g., by using on-screen annotation in the PDF file) or compile them in a separate list. Note: if you opt to annotate the file with software other than Adobe Reader then please also highlight the appropriate place in the PDF file. To ensure fast publication of your paper please return your corrections within 48 hours.

For correction or revision of any artwork, please consult http://www.elsevier.com/artworkinstructions.

Any queries or remarks that have arisen during the processing of your manuscript are listed below and highlighted by flags in the proof. Click on the Q link to go to the location in the proof.

<table>
<thead>
<tr>
<th>Location in article</th>
<th>Query / Remark: click on the Q link to go</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Please confirm that given names and surnames have been identified correctly and are presented in the desired order.</td>
</tr>
<tr>
<td>Q2</td>
<td>Your article is registered as belonging to the Special Issue/Collection entitled &quot;SI: BPM 2015&quot;. If this is NOT correct and your article is a regular item or belongs to a different Special Issue please contact <a href="mailto:j.rm@elsevier.com">j.rm@elsevier.com</a> immediately prior to returning your corrections.</td>
</tr>
<tr>
<td>Q3</td>
<td>Please complete and update the reference given here (preferably with a DOI if the publication data are not known): [47]. For references to articles that are to be included in the same (special) issue, please add the words ‘this issue’ wherever this occurs in the list and, if appropriate, in the text.</td>
</tr>
<tr>
<td>Q4</td>
<td>Acronyms have been included from the source file, kindly validate.</td>
</tr>
<tr>
<td>Q5</td>
<td>Please check the citation of Ref. [76] here and elsewhere.</td>
</tr>
<tr>
<td>Q6</td>
<td>Please supply the place of publication for Refs. [1,2,4,7,9,11-14,19,21-24,26,28-32,41,49,56,59-63,65,68,70-73,76] if available.</td>
</tr>
</tbody>
</table>

Thank you for your assistance.
Resolving inconsistencies and redundancies in declarative process models

Claudio Di Ciccio, Fabrizio Maria Maggi, Marco Montali, Jan Mendling

- An approach for the automated detection of inconsistencies and redundancies in declarative process models is proposed.
- The solution to the problem is based upon the notion of product-automata monoid.
- The detection is made by checking the language emptiness and language inclusion of finite state automata.
- The implemented tool is integrated with a process discovery algorithm and available online.
1. Introduction

The automated discovery of processes is the branch of the process mining discipline that aims at constructing a process model on the basis of the information reported in event data. The underlying assumption is that the recorded events indicate the sequential execution of the to-be-discovered process activities. The compact and correct representation of the behaviour observed in event data is one of the major concerns of process mining. Process discovery algorithms are classified according to the type of process model that they return, i.e., either procedural or declarative. Procedural process discovery techniques return models that explicitly describe all the possible executions allowed by the process from the beginning to the end. The output of declarative process discovery algorithms consists of a set of constraints, which exert conditions on the enactment of the process activities. The possible executions are implicitly established as all those ones that respect the given constraints. Mutual strengths and weaknesses of declarative and procedural models are discussed in [1,2].

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 [3]. In turn, similar facilities are not provided for novel declarative modelling languages like DECLARE. 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 include in the output those constraints that are individually satisfied in the log in more than a given number of cases. The interaction of returned constraints is thereby neglected, with the consequence that subsets of constraints can end up contradicting one another. Second, it is currently unclear whether a given constraint set is free of redundancies. Since there are constraint types that imply one another, it is possible that the generated constraint sets are partially redundant. The lack of formal techniques for handling these two issues is unsatisfactory from both a research and a practical angle. This 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 redundancy-freedom.

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
that we use for our evaluation. By analysing event log benchmarks,
we are able to show that inconsistencies and redundancies occur
in process models automatically discovered by state-of-the-art
tools. First, our technique can take such process models as input
and return constraints sets that are consistent. To this end, con-
tradictory subsets are identified and resolved by removing the
constraints generating the conflict. Second, our technique elim-
(1) Precedence(a, b) and (2) Precedence(a, c), as well as (3)
Response(a, c) and (4) Response(b, c): those four constraints hold
true in the event log as well. Nevertheless, if a is already bound to
be the first task to be carried out in every process instance (Init(a)),
clearly no other task can be executed if a is not done before.
Therefore, the first two constraints can be trivially deduced by
Init(a). They add no information, yet they contribute to useless-
ly enlarge the set of constraints returned to the user as the outcome
of the discovery. By the same line of reasoning, the third and
fourth constraints are superfluous with respect to End(b).

Intuitively, this example outlines the problem of redundancy, which
is one of the two challenges that we tackle with this research
work: the objective is to remove from the set of constraints in the
discovered process model those ones that do not add information,
i.e., that are not restricting the process behaviour any further given
the remaining ones.

In the context of declarative process discovery, event logs can be
affected by recording errors or report exceptional deviations from
the usual enactments [7,66]. In such cases, constraints that
were originally part of the process may be violated in some of the
recorded executions. If discovery algorithms take into account only
those constraints that always hold true in the event log, a mini-
 mum amount of noise might already cause several constraints to
be discarded from the returned set [8–10]. To circumvent this is-
 sue, declarative discovery algorithms offer the possibility to tune a
so-called support threshold: it specifies the minimum fraction of
cases in which a constraint is fulfilled within the event log to let
such constraint be included in the discovered model. However,
this comes at the price of possibly having conflicts in the model
though: constraints that hold true in a fraction of the event log
above the set threshold can contradict other constraints. In such a
case, the model becomes unsatisfiable, i.e., it exerts conditions that
cannot be met by any possible execution. Such a model would
clearly be too no avail to the discovery intents. This issue outlines
the problem of inconsistencies in the discovered model, which we
challenge in this research paper.

The aim of the presented approach is therefore twofold: Given a
discovered declarative process model, we want to (1) remove its
inconsistencies and (2) remove its redundancies. To pursue these
objectives, we aim at keeping the process behaviour as similar as
possible to the original one when removing inconsistencies, and
retaining the minimum number of constraints that still represent
the same original behaviour while getting rid of the redundancies.
The number of combinations of constraints to test for the optimum
of both problems is not tractable in practice, because every subset of
the original constraints set should be confronted with the others.
Our solution instead requires a polynomial number of checks over
constraints to provide a sub-optimal yet effective solution. Fur-
thermore, different criteria can be adopted to express (1) the de-
sired behavioural closeness and (2) the preferability of constraints
to be retained. To this extent, our solution envisages (1) the re-
laxation of conditions exerted by the contradicting constraints and
(2) different ranking criteria for constraints, respectively.

2. Motivation

Declarative process models consist of sets of constraints ex-
erted on tasks, which define the rules to be respected during the
process execution. A well-established language for modelling de-
clarative processses is DECLARE [5,6]. DECLARE defines a set of default
templates, which are behavioural rules that refer to parameters in
order to abstract from tasks. In DECLARE, e.g., Init(x) is a template
imposing that a given parametric task x must be the one with
which every process instance starts. End(x) specifies that every
process instance must terminate with the given task x. Response(x,y)
states that if task x is carried out, then task y must be
eventually executed afterwards. Precedence(x,y) imposes that y
can only be performed if x has been previously executed.

Let us consider a simple example process having three tasks, a,
b, and c. By indicating the execution sequence of tasks with their
name, possible enactments that fulfill a process model consisting of
Init(a) and End(c) are: (i) ababac and (ii) ababac. If we consider an
event log made of the aforementioned execution sequences and
use any declarative discovery algorithm to reveal a declarative
process model that could have generated them, it would correctly
return a set of constraints including Init(a) and End(c) because
they are always satisfied. However, the set of constraints would include
also (1) Precedence(a, b) and (2) Precedence(a, c), as well as (3)
Response(a, c) and (4) Response(b, c): those four constraints hold
true in the event log as well. Nevertheless, if a is already bound to
be the first task to be carried out in every process instance (Init(a)),
clearly no other task can be executed if a is not done before.
Therefore, the first two constraints can be trivially deduced by
Init(a). They add no information, yet they contribute to useless-
ly enlarge the set of constraints returned to the user as the outcome
of the discovery. By the same line of reasoning, the third and
fourth constraints are superfluous with respect to End(b).

Intuitively, this example outlines the problem of redundancy, which
is one of the two challenges that we tackle with this research
work: the objective is to remove from the set of constraints in the
discovered process model those ones that do not add information,
i.e., that are not restricting the process behaviour any further given
the remaining ones.

In the context of declarative process discovery, event logs can be
affected by recording errors or report exceptional deviations from
the usual enactments [7,66]. In such cases, constraints that
were originally part of the process may be violated in some of the
recorded executions. If discovery algorithms take into account only
those constraints that always hold true in the event log, a mini-
mum amount of noise might already cause several constraints to
be discarded from the returned set [8–10]. To circumvent this is-
sue, declarative discovery algorithms offer the possibility to tune a
so-called support threshold: it specifies the minimum fraction of
cases in which a constraint is fulfilled within the event log to let
such constraint be included in the discovered model. However,
this comes at the price of possibly having conflicts in the model
though: constraints that hold true in a fraction of the event log
above the set threshold can contradict other constraints. In such a
case, the model becomes unsatisfiable, i.e., it exerts conditions that
cannot be met by any possible execution. Such a model would
clearly be too no avail to the discovery intents. This issue outlines
the problem of inconsistencies in the discovered model, which we
challenge in this research paper.

The aim of the presented approach is therefore twofold: Given a
discovered declarative process model, we want to (1) remove its
inconsistencies and (2) remove its redundancies. To pursue these
objectives, we aim at keeping the process behaviour as similar as
possible to the original one when removing inconsistencies, and
retaining the minimum number of constraints that still represent
the same original behaviour while getting rid of the redundancies.
The number of combinations of constraints to test for the optimum
of both problems is not tractable in practice, because every subset of
the original constraints set should be confronted with the others.
Our solution instead requires a polynomial number of checks over
constraints to provide a sub-optimal yet effective solution. Fur-
thermore, different criteria can be adopted to express (1) the de-
sired behavioural closeness and (2) the preferability of constraints
to be retained. To this extent, our solution envisages (1) the re-
laxation of conditions exerted by the contradicting constraints and
(2) different ranking criteria for constraints, respectively.

3. Declarative process modelling and mining

This section defines the formal background for our research
problem. In particular, we introduce and revisit the concepts of
event logs and of declarative process modelling and mining.

Please cite this article as: C. Di Cicco, et al., Resolving inconsistencies and redundancies in declarative process models, Information
Notational conventions: We adopt the following notations. Given a set \( X \), (i) the multi-set of \( X \) is denoted as \( M(X) \); (ii) the power-set of \( X \) is denoted as \( P(X) \); (iii) a sequence of elements \( x_i \in X \) is denoted by the juxtaposition of its elements \( x_1, x_2, \ldots, x_n \); (iv) the cardinality of \( X \) is denoted as \(|X|\); the same notation applies both to the length of sequences and the cardinality of multi-sets.

Identifiers in cursive sans-serif format will be written in sans-serif letters when assigned to actual parameters. Generic identifiers of tasks, e.g., will be indicated as \( a, b, c \), whereas concrete assignments of task identifiers will be written as \( a, b, \) and \( c \).

3.1. Event logs

An event is a system-recorded information reporting on the execution of a task during the execution of a process. Events are labelled with so-called event classes, i.e., task names [11]. We assume that each event uniquely corresponds to the execution of a single task. This assumption builds upon the work on event class reconciliation of Baier et al. [12, 13]. We thus abstract every event with its event class, in turn related to a task. A finite sequence of events is named trace. A complete trace represents the execution of a process instance from the beginning to the end. An event log is a collection of traces. If multiple process instances have been executed by conducting the same sequence of tasks, multiple traces in the event log consist of the same sequence of events, accordingly.

Formally, an event log \( L \) is a multi-set of traces \( t_i \) with \( i \in [1, |L|] \), which in turn are finite sequences of events \( e_{ij} \) with \( i \in [1, |L|] \) and \( j \in [1, |L|] \) [14–16]. The log alphabet \( A \) is the set of symbols identifying all possible tasks and event classes. We write \( a, b, c \) to refer to them. Without loss of generality, we also refer to events as occurrences of symbols in \( A \). By denoting the set of sequences of tasks as \( A^* \) we have that \( L \in \mathcal{M}(A^*) \). An example of log is: \( L = \{ t_1, t_2 \} \) where \( t_1 = \text{abcacbac} \) and \( t_2 = \text{ababc} \). Event logs are the

| Table 1
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>Cardinality templates</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Existence templates</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Position templates</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Forward-unidirectional relation templates</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Backward-unidirectional relation templates</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Relation templates</td>
</tr>
<tr>
<td>Negative templates</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Please cite this article as: C. Di Ciccio, et al., Resolving inconsistencies and redundancies in declarative process models, Information Systems (2016), http://dx.doi.org/10.1016/j.is.2016.09.005
fundamental input of automated process discovery algorithms
[14,17].

3.2. Declarative process modelling languages

A declarative process modelling language represents the behaviour of processes by means of constraints, i.e., rules that must not be violated during the execution of process instances. Such rules are meant to be exerted over tasks in the context of temporal structures like imperative process models or logs. To date, \textsc{DECLARE} is one of the best well-established declarative process modelling languages. It provides a standard library of templates (repertoire), i.e., behavioural constraints parameterised over activities. Table 1 lists the constraints that will be considered in this paper.

Typical examples of \textsc{DECLARE} constraints are \textit{Participation}(a) and \textit{Response}(b, c). The former specifies that a must be executed in every process instance. The latter declares that if b is executed, then c must eventually follow. The constrained task of \textit{Participation}(a) is a, whereas the constrained tasks of \textit{Response}(b, c) are b and c. The template of \textit{Participation}(a) is Participation, whilst the template of \textit{Response}(b, c) is Response. Participation is an example of existence template, because it asserts conditions on the execution of a single activity. Response is an example of a relation template as it specifies conditions over pairs of activities. For relation templates, activations and targets are defined: the former is a task whose execution imposes obligations on the enactment of another task, i.e., the target. E.g., b is the activation and c is the target of \textit{Response}(b, c), because the execution of b requires c to be executed eventually.

Formally, a template is a predicate $C_{\ell} \in \mathcal{E}$, where $\mathcal{E}$ is the \textsc{DECLARE} repertoire and $n$ denotes its arity, i.e., the number of parameters [18]. In this article, we consider constraints of arity not higher than 2, because they constitute the subset of constraints discovered by the majority of declarative process miners [19–21]. Nevertheless, the presented approach can be seamlessly extended to the case of constraints of higher arity. Existence templates are unary, whereas relation templates are binary. Formal parameters of constraints are denoted by $x$ and $y$ for binary constraints or as $x_1, ..., x_n$ for constraints of a generic arity $n$. We will interchangeably use, e.g., $C_{\ell}$ or $C(x, y)$ to denote a template of arity 2. Templates will be denoted as $C$ when their arity is unspecified or clear from the context.

A constraint is the application of a template over tasks by means of the assignment of its formal parameters to elements in $\mathcal{A}$. Formally, given a template $C_{\ell} \in \mathcal{E}$, a parameter assignment $\gamma_{\ell}$ is a function from $\mathcal{A}$ to $\mathcal{E}$, where $\mathcal{E}$ is the set of integers ranging from 1 to $n$, $\gamma_{\ell}(i)$ means to assign the $i$-th parameter of $C_{\ell}$ to a task in $\mathcal{A}$, in compliance with the positional notation of the parameters of predicates. The $C_{\ell}$-constraint resulting from $\gamma_{\ell}$-written $C_{\ell}$, is also represented as $C = C(\gamma_{\ell}(1), ..., \gamma_{\ell}(n))$. For example, $\textit{Response}(b, c)$ denotes the constraint resulting from the application of $\gamma_{\ell}(1) = b$ and $\gamma_{\ell}(2) = c$.

In light of the above, and taking inspiration from the tabular representation of behavioural relations in [22], we can define a declarative process model as follows.

**Definition 3.1 (Declarative process model).** A declarative process model is a tuple $M = \langle \mathcal{A}, \mathcal{E}, \Gamma \rangle$ where:

- $\mathcal{A}$ is a finite non-empty set of tasks;
- $\mathcal{E}$ is a finite non-empty repertoire of templates;
- $\Gamma$ is a subset of the constraints universe $\mathcal{C}^\mathcal{A}$, which corresponds to the set of all constraints $\mathcal{C}$ that derive from the instantiation of every template $C_{\ell} \in \mathcal{E}$ with every possible assignment to tasks in $\mathcal{A}$. Note that by definition two different assignments $\gamma_a$ and $\gamma_b$ can be applied to the same constraint $C_{\ell}$. It is the case, e.g., when $\textit{Response}(b, a)$ and $\textit{Response}(a, b)$ both belong to the declarative process model: $\gamma_a(1) = a$ and $\gamma_a(2) = b$, $\gamma_b(1) = b$ and $\gamma_b(2) = a$, hence $\textit{Response}(a, b) \neq \textit{Response}(b, a)$ and $\textit{Response}(a, b) \neq \textit{Response}(b, a)$.

$\textsc{DECLARE}$ is a declarative process modelling language providing the repertoire of templates listed in Table 1. Having, e.g., $A = \{a, b, c\}$, in $\textsc{DECLARE}$ $\mathcal{E}$ would contain $\textit{Init}(a)$, $\textit{Init}(b)$, $\textit{Init}(c)$, $\textit{Response}(a, b)$, $\textit{Response}(b, a)$, $\textit{Response}(b, c)$, $\textit{Response}(c, b)$, $\textit{Response}(c, a)$, etc.

We respectively indicate activation and target of a template $C$ as $C_{a}$ and $C_{t}$. Hence, $\textit{Response}(x, y)_{a} \neq x$ and $\textit{Response}(x, y)_{t} \neq y$. With a slight abuse of notation, we use the same notation also for constraints: $\textit{Response}(a, b)_{a} \neq a$, $\textit{Response}(a, b)_{t} \neq b$. Moreover, we assume that for a template $C(x)$ of arity 1, activation and target coincide: $C(x)_{a} \neq C(x)_{t} = x$.

As said, events are meant to be recordings of the tasks carried out during the process enactment. Therefore, we will interchangeably interpret \textsc{DECLARE} rules as (i) behavioural relations between tasks in a process model or (ii) conditions exerted on the occurrence of events in traces. We will henceforth consider that, e.g., $\textit{Participation}(a)$ imposes that every trace contains at least an occurrence of $a$. Likewise, $\textit{Response}(b, c)$ indicates that after the occurrence of $b$, $c$ occurs afterwards in the trace. Both $t_1$ and $t_2$ in the example log $L$ are compliant with $\textit{Participation}(a)$ and $\textit{Response}(b, c)$. Such conceptual matching is typical of \textsc{DECLARE} mining approaches [9,23], as event logs are used to analyse to what extent constraints are respected by counting the number of fulfilments within traces.

3.2.1. Evaluation and satisfiability of a declarative process model

Since constraints are predicates, they can be evaluated and checked for satisfiability. In particular, as exposed by Räim et al. [24], every constraint of a declarative process model can be evaluated over traces by adopting a semantics based on linear temporal structures. We thus introduce the notion of evaluation of a constraint over a trace as a function $\eta: \mathcal{E} \times \mathcal{A}^* \rightarrow \{\top, \bot\}$, such that:

$$\eta(C, t) = \begin{cases} \top & \text{if } C \in \mathcal{E} \text{ evaluates to true over } t \in \mathcal{A}^* \\ \bot & \text{otherwise.} \end{cases}$$

(1)

If the conditions imposed by the constraint are satisfied by every event in the trace, then the trace fulfils the constraint, i.e., the constraint evaluates to true ($\top$) over the trace.

With a slight abuse of notation, we denote the evaluation of a declarative process model $M = \langle \mathcal{A}, \mathcal{E}, \Gamma \rangle$ over a trace by means of the same symbol $\eta$, and define it as follows:

$$\eta(M, t) = \begin{cases} \top & \text{if for all } C \in \Gamma \subseteq \mathcal{E}^\mathcal{A}, \quad \text{\eta evaluates to true over } t \in \mathcal{A}^* \\ \bot & \text{otherwise.} \end{cases}$$

(2)

The notion of evaluation of a declarative process model leads to the satisfiability problem, i.e., checking whether there exists a trace over which the model evaluates to true. Hereinafter, we denote the set of traces that satisfy a declarative process model $M$ as its language:

$$\mathcal{L}(M) = \{ t \in \mathcal{P}(\mathcal{A}^*) : \eta(M, t) \}.$$  

(3)

Since a declarative process model is evaluated to $\top$ on a trace $t$ only if all its constraints are evaluated to $\top$ on $t$ (Eq. (2)), given two models $M = \langle \mathcal{A}, \mathcal{E}, \Gamma \rangle$ and $M' = \langle \mathcal{A}, \mathcal{E}, \Gamma' \rangle$ where $\Gamma' \subseteq \Gamma$, it follows...
that $\mathcal{L}(M) \subseteq \mathcal{L}(M')$. The declarative process model $M = (\mathcal{A}, \mathcal{E}, \emptyset)$ trivially accepts any trace consisting of elements of $\mathcal{A}$, because it imposes no constraints. Hence, $\mathcal{L}(M) = \mathcal{P}(\mathcal{A})$ is the most under-fitting declarative process model, but not interesting from a business process perspective, since it accepts any possible behaviour. Likewise, empty process models, i.e., not satisfiable by any trace, are equivalently ineffective. As an example, we can consider the DECLARE constraints $\text{CoExistence}(a, b)$ and $\text{NotCoExistence}(a, b)$. The first one states that if a occurs in a trace, then b cannot occur, and vice versa. It follows that a model $M_a = \{a, b\}, \mathcal{E}, \{\text{CoExistence}(a, b), \text{NotCoExistence}(a, b)\}$ is unsatisfiable, because the two constraints are contradicting, which implies $\mathcal{L}(M_a) = \emptyset$. Notice that also the declarative process model $M_a = \{a, b\}, \mathcal{E}, \{\text{CoExistence}(a, b), \text{NotCoExistence}(a, b), \text{Init}(a)\}$ is still unsatisfiable, although $\text{Init}(a)$ is not in contradiction with $\text{CoExistence}(a, b)$ nor with $\text{NotCoExistence}(a, b)$. This observation leads to the problem that we want to address: finding and removing contradicting constraints that make declarative process models unsatisfiable.

### 3.2.2. Discovery of a declarative process model

A declarative process model can be discovered by evaluating all constraints in the constraints universe over the event log and returning all and only those constraints that evaluate to $\top$ over the event log. However, this would make the discovered model over-fitting, with the consequence that if the event log contained errors, then the discovered model would be affected by erroneously discarded or added constraints [25]. To overcome this issue, metrics have been introduced that make the discovered model less prone to faulty log entries.

Taking inspiration from the area of data mining [26], we adopt the support metric [19,21]. Support assesses the degree of fulfilment of constraints in the event log by scaling the number of traces fulfilling the constraint by the number of traces in the log. Support is defined as a function $\sigma: \mathcal{E}^* \times \mathcal{M}(\mathcal{A}^*) \rightarrow [0, 1] \subseteq \mathbb{R}_{\geq 0}$, with $\mathbb{R}_{\geq 0}$ being the set of positive real numbers, computed as follows:

$$\sigma(C, L) = \frac{|\{t \in L : C \in t\}|}{|L|}.$$  

(4)

Given the example event log from above, $L = \{t_1, t_2\}$ where $t_1 = \text{abacbab}$ and $t_2 = \text{ababc}$. We have that: $\sigma(\text{Participation}(a, L)) = 1.0$, $\sigma(\text{Response}(b, c, L)) = 1.0$, and $\sigma(\text{Response}(a, b, L)) = 0.5$, because no b follows the last a occurring in $t_1$.

Typically, a discovered declarative process model consists of those constraints having a support higher than a user-specified threshold: Those that are fulfilled in a significant number of cases belong to the discovered model. However, the amount of constraints that the discovered model consists of is usually overwhelming, when only relying on such criterion. Therefore, metrics for assessing the relevance of constraints have been established, i.e., confidence $\kappa$ and interest factor $\iota$, which scale the support by the ratio of traces in which the activation occurs, resp. both the constrained tasks occur. Confidence and interest factor are defined as functions $\kappa: \mathcal{E}^* \times \mathcal{M}(\mathcal{A}^*) \rightarrow [0, 1] \subseteq \mathbb{R}_{\geq 0}$ and $\iota: \mathcal{E}^* \times \mathcal{M}(\mathcal{A}^*) \rightarrow [0, 1] \subseteq \mathbb{R}_{\geq 0}$, respectively, which are computed as follows:

$$\kappa(C, L) = \frac{|\{t \in L : C \in t\}|}{|L|}.$$  

(5)

$$\iota(C, L) = \frac{|\{t \in L : C_1 \in t \text{ and } C_2 \in t\}|}{|L|}.$$  

(6)

Different variants of calculating these metrics have been proposed [19,27,28]. Notice that both $\kappa$ and $\iota$ scale the value of $\sigma$ by a number included in the range $[0, 1]$. By their definition, it always holds true that given a constraint $C$ and an event log $L$, $0 \leq \kappa(C, L) \leq \sigma(C, L) \leq \iota(C, L) \leq 1$.

### 3.3. DECLARE template types and subsumption

DECLARE is a declarative process modelling language that provides a repertoire of templates for the specification of constraints over tasks. The list of templates considered in this paper is provided in Table 1. Here, we describe how the templates are divided into types and constitute a subsumption hierarchy [29,27,25], as illustrated in Fig. 1.

Examples of constraints in DECLARE are: (i) $\text{Participation}(a)$, $\text{Succession}(b, c)$, and $\text{Response}(a, b, L)$.
specifying that task a must occur in every trace; (ii) AtMostOne(a), declaring that a must occur not more than once in a trace; (iii) RespondedExistence(a, b), imposing that if a occurs in a trace, then also b must occur in the same trace. Participation and AtMostOne are existence templates. Because they exert restrictions on the number of occurrences of a task in a trace, they belong to the type of cardinality constraint templates. Init(a) and End(a) are existence constraints stating that a must be the first, resp. the last, event occurring in all traces. Therefore, they belong to the type of position constraints. Both their templates are subsumed by Participation, because they imply that the constrained task occurs in every trace in order to be the first or the last one. Fig. 1 (a) illustrates the subsumption hierarchy of existence constraint templates. Templates are indicated in solid boxes. The subsumption between templates is drawn with a line starting from the subsumed template and ending in the subsuming one, with an empty triangular arrow recalling the UML IS-A graphical notation.

As Fig. 1 (b) illustrates, RespondedExistence(x, y) generates an offspring of related relation templates. Its directly subsumed templates (children) are Response(x, y) and Precedence(x, y). Response(a, b) imposes that eventually after an occurrence of a (the activation), b (the target) must occur. Dually, Precedence(a, b) requires that before an occurrence of the activation task a, target task b occurs. Both constraints strengthen the conditions exerted by RespondedExistence by specifying that not only must the target occur, but also in which relative position in the trace (after or before the activation). However, the role of activation and target are swapped in Precedence, w.r.t. RespondedExistence. Owing to this, RespondedExistence and Response belong to the type of forward-unidirectional relation templates, whereas Precedence is a backward-unidirectional relation template.

Technically, given two n-ary templates C, C′ ∈ C, we say that C is subsumed by C′ written C ⊑ C′, if for every trace t ∈ T and every parameter assignment x′ from the parameters of C to tasks in X, whenever t complies with the instantiation of C with x′, then t also complies with the instantiation of C′ with x′. For binary templates, we write C ⊑ C′ if the subsumption holds by inverting the parameters of C w.r.t. those in C, i.e., by considering templates C(x, y) and C(y, x). We thus have that Response ⊑ RespondedExistence. By the same line of reasoning, we have that Precedence ⊑ RespondedExistence. C is transitive (if C ⊑ C′ and C ⊑ C″ then C ⊑ C″) and reflexive (C ⊑ C). We also introduce the inverse relation of C, i.e., C′, which we use to indicate that C subsumes C′, i.e., C′ ⊑ C; e.g., RespondedExistence ⊑ Response, and RespondedExistence ⊑ Precedence. In the following, we extend the usage of ⊑ and its inverse relation to constraints too: C ⊑ C′ means that constraint C is subsumed by constraint C′, e.g., Response(a, b) ⊑ RespondedExistence(a, b).

In the subsumption hierarchy of both Response and Precedence, the direct child templates are AlternateResponse and AlternatePrecedence. The concept of alternation strengthens the parent template by adding the condition that between pairs of activation and target, not any other activation occurs. The subsumption hierarchy concludes with ChainResponse and ChainPrecedence: they impose that occurrences of activation and target are immediately adjacent.

The conjunction of a forward-unidirectional relation template and a backward-unidirectional relation template belonging to the same level of the subsumption hierarchy generates the so-called coupling templates: Succession(x, y), e.g., holds when both Response(x, y) and Precedence(x, y) hold true. In addition, the coupling template CoExistence is equal to the conjunction of RespondedExistence and RespondedPrecedence. For every coupling template C, a function fw(C) and bw(C) are defined that resp. return the related forward-unidirectional relation and the backward-unidirectional relation templates. Hence, fw(Succession(x, y)) = Response(x, y) and bw(Succession(x, y)) = Precedence(x, y). In Fig. 1 (b), the functions fw and bw are indicated by grey arcs labelled as forward and backward, respectively. With a slight abuse of notation, we will adopt function symbols fw and bw not only for templates but also for constraints.

Finally, coupling templates CoExistence, Succession and ChainSuccession correspond to other templates that share the same activations and exert opposite conditions on the targets: resp., NotCoExistence, NotSuccession and NotChainSuccession. For instance, CoExistence(a, b) states that a and b always co-occur in a trace. NotCoExistence(a, b) states instead that if either a (resp. b) occurs in the trace, then b (resp. a) cannot. Owing to this, NotCoExistence, NotSuccession and NotChainSuccession are named negative templates. Given a negative template, e.g., NotCoExistence, we say that it negates the corresponding coupling template, e.g., CoExistence. Due to the opposite conditions exerted on the targets, the subsumption hierarchy gets also reverted w.r.t. the corresponding negated templates: NotCoExistence ⊑ NotSuccession ⊑ NotChainSuccession. In Fig. 1 (b), negative templates are graphically linked to their corresponding coupling templates by means of wavy grey arcs labelled as negates.

Based on the concept of subsumption, we can define the notion of relaxation R. 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 R returns a predicate that would hold true for any possible trace, i.e., T. Formally, given a template C ∈ C, we have:

\[
R(C) = \begin{cases} 
\emptyset & \text{if } (i) \ C \subseteq \emptyset, \ (ii) \ C \subseteq C', \text{ and } (iii) \emptyset \subseteq C \text{ s.t. } C \subseteq C' \text{ and } C' \subseteq C'' \text{,} \\
T & \text{otherwise} 
\end{cases}
\]

We extend the relaxation operator and the subsumption relation to the domain of constraints, such that, e.g., \( R(\text{Response}(a, b)) = \text{RespondedExistence}(a, b) \).

### 3.4. Semantics of DECLARE as regular expressions

A plethora of semantics for DECLARE templates have been proposed in the literature by relying on different logic-based approaches, including Linear Temporal Logic (LTL) [30,31], Linear Temporal Logic on Finite Traces (LTL F) [32,33], First Order Logic (FOL) formulae over finite ordered traces [27,33] and abductive logic programming [34]. We adopt regular expressions (REs) because they allow us to take advantage of well-established techniques for calculating automata products, which are at the base of our approach. Table 1 lists the translation of templates into REs.

Regular expressions are a formal notation to compactly express finite sequences of characters, a.k.a. matching strings. The syntax of REs consists of any juxtaposition of characters of a given alphabet, optionally grouped by enclosing parentheses ( and ), to which the following operators can be applied: binary alternation | and concatenation, and the unary Kleene star *.

A regular expression a(bc)* denotes any string starting with a, followed by any number of repetitions of the pattern (sub-string) bc (optionally, none) and closed by either d or e, such as ad, abc, abcbce and ae. Table 1 adopts the POSIX standard for the following additional shortcut notations: (i) . respectively denote any character or any character but x, (ii) + operators respectively match from one to any and from none to one occurrences of the preceding pattern. We also make use of (iii) the parametric quantifier ( .m), with m being integer higher than 0, which specifies the maximum number of repetitions of the preceding pattern, and (iv) the parametric quantifier ( .n), with n being integer higher than or equal to 0, which specifies the minimum number of repetitions of the preceding pattern. We recall here that (i) REs are closed under the conjunction operator & [35], and (ii) the expressive power of REs completely covers regular languages [36], thus (iii) since regular grammars are...
recognisable through RES [37,36], for every RE, a corresponding
deterministic finite state automaton (FSA) exists, accepting all and
only the matching strings [38]. The conjunction operator & satis-
fies commutativity and associativity. Its identity element is *.

3.5. Finite state automata

A (deterministic) FSA is a finite-state labelled transition system

\[ A = (\Sigma, S, s_0, \delta, S_f) \]

where: \( \Sigma \) is an alphabet; \( S \) is the finite non-
empty set of states; \( s_0 \in S \) is the initial state; \( \delta : S \times \Sigma \rightarrow S \) is the
transition function, i.e., a function that, given a starting state and a
character of the alphabet, returns the target state (if defined);
\( S_f \subseteq S \) is the set of final (accepting) states [37]. For the sake of
simplicity, we will omit the qualification “deterministic”. A finite
path \( \pi \) of length \( n \) over \( A \) is a sequence \( \pi = \left< \pi_1, \ldots, \pi_n \right> \) of triples

\[ \pi_i = \left< \pi_i^1, \pi_i^2, \pi_i^3 \right> \in \delta, \]

for which the following conditions hold true:
(i) \( \pi_1^1 \), the first tuple, is such that \( \pi_1^3 = s_0 \) (i.e., \( \pi \) starts from
the initial state of \( A \)) and (ii) the starting state of \( A \) is the target state
of \( \pi_i-1 \), i.e., \( \pi_i^1 \) is such that \( \pi_i^1 = \pi_i^3 \). A diagram of
length \( n \geq 0 \), i.e., a concatenation \( \sigma = \sigma_1 \ldots \sigma_n \) of characters \( \sigma_i \in \Sigma \),
is accepted by \( A \) if a path \( \pi \) of length \( n \) is defined over \( A \) and is such that
(i) for every \( i \in [1, n] \), \( \pi_i = \left< \pi_i^1, \pi_i^2, \pi_i^3 \right> \), and (ii)
\( \pi^n = \left< \pi_1^1, \pi_n^2, \pi_n^3 \right> \) is s.t. \( \pi_n^3 = s_f \). We overload the \( \mathcal{L} \) notation by
denoting as \( \mathcal{L}(A) \subseteq \Sigma^* \) the (possibly infinite) set of strings ac-
cpected by \( A \).

A product of two FSAs \( A \) and \( A' \) accepts the intersection of languages (sets of
accepted strings) of each operand:

\[ \mathcal{L}(A \times A') = \mathcal{L}(A) \cap \mathcal{L}(A') \]

The product of FSAs is an isomorphism for the conjunction of RES, i.e.,
the product of two FSAs respectively corresponding to two RES is
equivalent to the FSA that derives from the conjunction of the two
FSAs [40]; given the RES \( r, r' \) and naming as \( \mathcal{A} \) the operation
leading from an RE to the corresponding FSA, we have that
\( \mathcal{A}(r \land r') = \mathcal{A}(r) \times \mathcal{A}(r') \). The product operator \( \times \) is commutative
and associative. The identity element for \( \times \) over the alphabet \( \Sigma \) is
\( \mathcal{A}^0 = (\Sigma, \{s_0\}, s_0, \{s_0\} \times \{s_0\}, \{s_0\}) \) (Fig. 2(a)). It accepts all
strings over \( \Sigma \); \( \mathcal{L}(\mathcal{A}^0) = \Sigma^* \). The absorbing element:
\( \mathcal{A}^0 = (\Sigma, \{s_0\}, s_0, \{s_0\} \times \{s_0\}, \{s_0\}) \) (Fig. 2(b)). It does not accept any string:
\[ \mathcal{L}(\mathcal{A}^0) = \emptyset. \]

4. Formalisation of the problem

In this section, we present the twofold problem tackled in this
work. First, we want to avoid that the discovered declarative
process models contain inconsistencies, i.e., contradictions among
constraints that make the overall model unsatisfiable. Second, we
want to minimise the number of constraints in the discovered
declarative process models, in particular by eliminating those that
are redundant.

The main idea of declarative process discovery is that the
overfitting of the discovered models can be avoided by defining
thresholds for parameters such as support, confidence and interest
factor. By choosing a support threshold smaller than 100%, we can
easily obtain constraint sets that are supported by different parts
of the log and that contradicts each other. E.g., when using MI-
NERful on the BPI challenge 2012 event log with a support
threshold of 75%, it returns the constraints Participation(A_Pre-
accepted), NotChainSuccession(A_Preaccepted,W_Completeren
aanvrag), and ChainResponse(A_Preaccepted,W_Completeren
aanvrag), which have an empty set of traces that fulfil all of them.
In fact, the first constraint imposes that A_Preaccepted must be
executed at least once, the second constraint imposes that
A_Preaccepted is never directly followed by W_Completeren
aanvrag, whereas the third one requires that if A_Preaccepted is
executed, W_Completeren aanvrag must immediately follow.
Clearly, such inconsistent constraint sets should not be returned
by the discovery algorithm.

Please cite this article as: C. Di Ciccio, et al., Resolving inconsistencies and redundancies in declarative process models, Information
Systems (2016), http://dx.doi.org/10.1016/j.is.2016.09.005
4.2. The minimality problem

The second problem we tackle in this work is minimality. This problem is concerned with the informative content of the discovered declarative process model. The goal is to understand whether all its constraints effectively contribute to the separation between compliant and non-compliant traces, or are instead redundant. Let \( \mathcal{M} = (\mathcal{A}, \mathcal{C}, \mathcal{E}) \) and let \( \mathcal{C} \subseteq \mathcal{C} \) be a constraint of \( \mathcal{M} \). Intuitively, we say that \( \mathcal{C} \) is redundant in \( \mathcal{M} \) if the set of compliant traces (i.e., the language defined by \( \mathcal{M} \)) is not affected by the presence of \( \mathcal{C} \). Formally, let \( \mathcal{M}' = (\mathcal{A}, \mathcal{C}, \mathcal{E} \setminus \{\mathcal{C}\}) \) be the declarative process model obtained from \( \mathcal{M} \) by removing constraint \( \mathcal{C} \). We then have that \( \mathcal{C} \) is redundant in \( \mathcal{M} \) if \( \mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}') \).

In this light, addressing the minimality problem means transforming the discovered declarative process model into one that is language-wise equivalent, but does not contain redundant constraints. Models that contain redundancies are pointless difficult to understand for the process analysts, since redundant constraints do not provide any additional information about the permitted and forbidden behaviours.

Like for consistency, this problem is inherently difficult and calls for the application of suitable strategies that find a reasonable trade-off between optimality and computational efficiency. In fact there are, in general, exponentially many ways of making a model redundancy-free. This is, again, particularly critical in the context of declarative process discovery. Every newly discovered constraint could in fact introduce redundancy, which could be removed either by ignoring the newly discovered constraint or by dropping a set of already discovered redundant constraints with the new one.

To show to which extent this problem is present in concrete declarative process modelling languages such as DECLARE, we recall the fact that DECLARE templates can be organised in a hierarchy of constraints, depending on a notion of subsumption, as discussed in Section 3.3. This notion of subsumption is tightly related to that of redundancy, since a constraint can be immediately recognised as redundant if it is subsumed by another constraint present in the partial declarative process model discovered so far. However, we stress the fact that redundancy could be detected, in some cases, only by analysing the model as a whole, and not just considering pairs of constraints.

When using MINERful on the BPI challenge 2012 event log with a support threshold of 75%, it returns the constraints ChainResponse(A Submitted A PartySubmitted) and NotChainSuccession(A Submitted A Accepted). The latter constraint is clearly redundant, because the former requires the first task following A Submitted to be A PartySubmitted. Therefore, no other task but A PartySubmitted can directly follow. A fortiori, A Submitted and A Accepted cannot be in direct succession. Clearly, such redundant constraint pairs should not be returned.

4.3. Framing the problem

In Sections 4.1 and 4.2, we have introduced the issues of consistency and redundancy in declarative process models. We now frame these problems in the context of declarative process discovery.

Our goal is to define effective post-processing techniques that, given a previously discovered DECLARE model \( \mathcal{M} \) 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 \( \mathcal{M} \) 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 prioritised, and ultimately decide whether a variant \( \mathcal{M}' \) of \( \mathcal{M} \) has to be preferred over another variant \( \mathcal{M}^* \).

In principle, we could obtain an optimal solution by exhaustive enumeration, executing the following steps:

1. The vocabulary \( \mathcal{A} \) of \( \mathcal{M} \) is extracted.
2. The set \( \mathcal{C}^e \) of all possible candidate constraints is built.
3. The power-set \( \mathcal{P}(\mathcal{C}^e) \) of all possible subsets of \( \mathcal{C}^e \), i.e., of all possible DECLARE models using constraints in \( \mathcal{C}^e \), is computed.
4. A set of candidate models \( \mathcal{M} \) over \( \mathcal{A} \) and \( \mathcal{C} \) is obtained from \( \mathcal{P}(\mathcal{C}^e) \), by filtering away those models that are inconsistent or contain redundant constraints.
5. A ranking of the models in \( \mathcal{M} \) is established, considering their similarity to the original discovered model \( \mathcal{M} \).

However, this exhaustive enumeration is unfeasible in the general case, given the fact that it requires to iterate over the exponentially many models in \( \mathcal{P}(\mathcal{C}^e) \), an intractably huge state space. Consequently, we devise a heuristic algorithm that mediates between optimality of the solution and computational efficiency. In summary, its main features are the following:

- It produces as output a consistent variant of the initial model \( \mathcal{M} \). This is a strict, necessary requirement.
- The algorithm works in an incremental fashion, i.e., it constructs the variant of \( \mathcal{M} \) by iteratively selecting constraints. Once a constraint is added, it is not retracted from the model. This is done by iterating through the candidate constraints in descending order of suitability. The degree of suitability is dictated by an ordering relation that sorts the constraints before the algorithm starts the checking phase. An example of such an ordering relation is the ranking of constraints on the basis of their support, confidence, and interest factor: it makes the algorithm retain the constraints that better fit the log from which they were discovered. On the one hand, this drives our algorithm to favour more suitable constraints and remove less suitable constraints in the case of an inconsistency or a redundancy. 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 redundancy-free and minimal in the number of constraints, but we still achieve a local minimum. Our experimental findings show that this local minimum is satisfactory, since the algorithm is able to significantly reduce the number of redundant constraints w.r.t. the state-of-the-art discovery algorithms.

5. 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 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, which 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. Notice that the commutativity of the automata product guarantees that conflicting constraints are found, regardless of the order with which they are checked. The algorithm proceeds without visiting the same node in the search space twice.

5.1. Constraints as automata

As already shown in [21], 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. A constraint is thus evaluated to true over a trace if and only if the corresponding string is matched by the constraint’s regular expression.

Using the POSIX wildcards, we can express, e.g., \textit{Init}(a) as a .* and \textit{Response}(a, b) as \textit{.*a}.*[\textit{~a}].* The comprehensive list of transpositions for DECLARE templates is listed in Table 1 and explained in [40]. Henceforth, we will refer to such a mapping as \( \mathcal{E}_{\text{Reg}}(C) \), which takes as input a constraint \( C \) and returns the corresponding RE: E.g., \( \mathcal{E}_{\text{Reg}}(\text{Response}(a, b)) = [\textit{~a}].*[\textit{~a}].*[\textit{a}].\). We consider the operations of conjunction between DECLARE constraints \( \land \) and intersection between RES \( \& \), \( \mathcal{E}_{\text{Reg}}(C \land C') = \mathcal{E}_{\text{Reg}}(C) \& \mathcal{E}_{\text{Reg}}(C') \).

As mentioned in Section 3.4, an RE can always be associated to a deterministic labelled FSA, which accepts all and only those finite strings that match the RE. We name as \( \mathcal{A} \) the operation leading from an RE to an FSA, thus we have that a DECLARE constraint can be associated with its corresponding FSA, \( \mathcal{A} : \mathcal{E}_{\text{Reg}} \rightarrow \mathcal{A} \). Henceforth, we also call \( \mathcal{A} \) the C-automaton.

Under this interpretation, a constraint \( C \) is evaluated to true over a trace if and only if its events can be replayed as a finite path on the C-automaton that terminates in an accepting state, i.e., when its corresponding string is accepted by the C-automaton.

Considering an example declarative process model \( M = (\mathcal{A}, \{ \text{Init}, \text{AlternateResponse}, \text{ChainPrecedence}, \ldots \}, \Gamma) \) with \( \mathcal{A} = \{a, b, c\} \) and

\[
\Gamma = \{ \text{Init}(a), \text{AlternateResponse}(a, b), \text{ChainPrecedence}(b, c) \};
\]

the C-automata of some set of constraints \( \Gamma \) are drawn in Fig. 3: \( \mathcal{A}(\mathcal{E}_{\text{Reg}}(\text{Init}(a))) \), \( \mathcal{A}(\mathcal{E}_{\text{Reg}}(\text{AlternateResponse}(a, b))) \), and \( \mathcal{A}(\mathcal{E}_{\text{Reg}}(\text{ChainPrecedence}(b, c))) \) are depicted in Fig. 3(a), (b), and (c), respectively. Fig. 4 shows the product automata that are derived from the intersection of such constraints: \( \mathcal{A}(\mathcal{E}_{\text{Reg}}(\text{Init}(a))) \times \mathcal{A}(\mathcal{E}_{\text{Reg}}(\text{AlternateResponse}(a, b))) \) is illustrated in Fig. 4(a), and \( \mathcal{A}(\mathcal{E}_{\text{Reg}}(\text{Init}(a))) \times \mathcal{A}(\mathcal{E}_{\text{Reg}}(\text{AlternateResponse}(a, b))) \times \mathcal{A}(\mathcal{E}_{\text{Reg}}(\text{ChainPrecedence}(b, c))) \) illustrated in Fig. 4(b).

We remark that by applying \( \mathcal{A} \) to the RE of a conjunction of constraints, we obtain an FSA that corresponds to the product \( \times \) of the FSAs for the individual constraints [35]:

\[
\mathcal{A}(\mathcal{E}_{\text{Reg}}(C \land C')) = \mathcal{A}(\mathcal{E}_{\text{Reg}}(C)) \times \mathcal{A}(\mathcal{E}_{\text{Reg}}(C')).
\]

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.

5.2. Sorting constraints

The objective of the algorithm is to visit the constraints in the declarative process model only once. At every visit, the analysed constraint is checked whether it is conflicting or redundant. In the first case, it is relaxed (replaced by the subsuming constraint) and checked again. If the constraint is not subsumed by any another, it is removed from the set of constraints. In case of redundancy, the constraint under analysis is removed from the set of discovered constraints. The order in which the constraints are checked is thus of utmost importance, as it determines their priority. 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 identify four notions of ordering relations for declarative process models. Three of them are suitable for all models, i.e., (i) order on the degree of activation linkage \( \preceq \), (ii) partial order on the type \( \preceq_s \), and (iii) partial order on the subsumption \( \preceq_r \); the last one is specific for discovered models, i.e., (iv) order on support, confidence, and interest factor \( \preceq_m \).

The first ordering relation is based on the notion of degree of activation linkage. Given a constraint \( C \) in the template instantiation relation \( \Gamma \) of a model \( M \) and its activation \( C_\mathcal{A} \), it counts the number of tasks that play the role of target in constraints that share the same activation of \( C \). Formally

\[
\rightarrow(\Gamma, C) = \{ a \in \mathcal{A} : \exists C' \in \Gamma \text{ s.t. } a = C_\mathcal{A} \text{ and } C' \preceq_C C \}
\]

Recalling that we consider for unary constraints the activation to coincide with the target, for example the model

\[
M = (\mathcal{A}, \{ \text{InitResponse}, \text{ChainPrecedence}_1, \ldots \}, \Gamma)
\]

where \( \mathcal{A} = \{a, b, c\} \) and

\[
\Gamma = \{ \text{InitResponse}(a, b), \text{ChainPrecedence}_1(b, c) \}
\]

we have that (i) \( \rightarrow(\Gamma, \text{InitResponse}(a, b)) = 2 \), (ii) \( \rightarrow(\text{AlternateResponse}(a, b), \Gamma) = 2 \), (iii) \( \rightarrow(\text{ChainPrecedence}(b, c), \Gamma) = 1 \). The (total) order

\[
\rightarrow_{\mathcal{M}} = \langle \rightarrow(\Gamma, C) : C \in C \rangle
\]

is defined as the lexicographic sorting of the set of non-redundant consistent DECLARE models.

Please cite this article as: C. Di Ciccio, et al., Resolving inconsistencies and redundancies in declarative process models, Information Systems (2016), http://dx.doi.org/10.1016/j.is.2016.09.005
on the degree of activation linkage is thus defined as follows:

\[ C \leq C \iff \sim (C, \Gamma) \leq \sim (C, \Gamma), \quad \text{with } C, C \in \Gamma. \]  

(9)

This relation is meant to sort constraints by the number of tasks that are subject to conditions over the execution of their activation.

The partial order on the type of constraints is driven by the expertise acquired in the last years in the context of DECLARE discovery [29,19]. 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 redundancy. As an example, consider the case of the Init template: given \( a \in A \), if \( \text{Init}(a) \) holds true, then also the relation constraint \( \text{Precedence}(a, b) \) is guaranteed to hold true for every \( b \in A \setminus \{a\} \). This means that, in the best case, \( |\Gamma| - 1 \) constraints will be removed because they are all redundant with \( \text{Init}(a) \). Similarly, consider the positive relation constraint \( \text{ChainResponse}(a, b) \): it implies \( \text{NotChainSuccession}(a, c) \) for every \( c \in A \setminus \{a, b\} \). Thus, \( \text{ChainResponse}(a, b) \) has the potential of triggering the removal of \( |\Gamma| - 2 \) negative constraints due to redundancy. Therefore, the ordering by type sorts constraints according to the following ranking from the highest to the lowest:

1. negative constraints,
2. cardinality constraints,
3. coupling constraints,
4. forward- and backward-unidirectional relation constraints,
5. position constraints,
6. constraints.

We define the partial order on the subsumption as follows:

\[ C \leq_{C} C \iff C \subseteq C. \]  

(10)

Those constraints that have the highest likelihood to induce other constraints are ranked the highest by both the last two orderings. Therefore, their application seems to be suitable to prune out the highest number of constraints, especially during the redundancy check. Therefore, we introduce the hybrid ordering relation \( \leq_{\Gamma, C} \) defined as follows:

\[ C \leq_{\Gamma, C} C' \iff (C \subseteq C' \land \sim (C, C') \land \sim (C, C')) \vee C \subseteq C. \]  

(11)

In essence, it compares the type of constraints \( C \) and \( C' \). If they are the same, i.e., \( C \subseteq C' \land \sim (C, C') \) holds true, then the comparison is made on the basis of \( \leq_{C} \).

Finally, for discovered declarative process models, functions \( \sigma, \kappa \) and \( \iota \) are defined for constraints. Therefore, the last ordering relation based on these functions can be applied:

\[ C \leq_{\sigma, \kappa, \iota} C \iff (C, L) \leq (\sigma(C, L) \lor \kappa(C, L) \lor \iota(C, L)) \]  

(12)

Such an ordering is meant to give priority to those constraints that are violated the least within the event log or whose constrained activities occur most frequently. The idea is to remove those constraints that are redundant or conflicting starting from those that are less fitting with the event log.

The aforementioned ordering relations are not strict, because they are not asymmetric: For instance, in the example given for the order on the degree of activation linkage, both \( \text{Init}(a) \leq_{C} \text{AlternateResponse}(a, b) \) and \( \text{AlternateResponse}(a, b) \leq_{C} \text{Init}(a) \) hold true. Since the number of constraints in a model is finite, we can assume the existence of a strict total order over constraints \( \leq_{s} \) s.t. only one of the following three statements holds for every pair of constraints \( C, C' \): Either (i) \( C \leq_{s} C' \), or (ii) \( C' \leq_{s} C \), or (iii) \( C = C' \). Relation \( \leq_{s} \) can be based, e.g., on a perfect hash relation, or on an enumeration-based ordering of constraints. This notion allows us to postulate the creation of a strict total order relation \( \leq_{s} \) based on a sequential application of any combination of \( \{s_{1}, \ldots, s_{n}\} \) of the aforementioned ordering relations \( \leq_{\Gamma}, \leq_{\Gamma, C} \), and \( \leq_{C} \). This is inductively defined as follows:

\[ C \leq_{s} C \equiv \bigvee_{i \in \{1, \ldots, n\}} (C \subseteq C' \land \leq_{s} C' \lor C \leq_{s} C). \]  

(13)

This relation orders constraints by applying the first ordering relation \( \leq_{\Gamma} \) to the pair of constraints \( (C, C') \). If they are such that \( C \leq_{s} C' \) and \( C' \leq_{s} C \), then the comparison by means of \( \leq_{s} \) is applied, and so forth till the \( n \)-th stage is reached. Then, \( \leq_{s} \) is applied. By definition, \( \leq_{s} \) is a strict total order, therefore it holds that \( C \leq_{s} C' \) and \( C' \leq_{s} C \) if and only if \( C = C' \).

We introduce an algorithm herein to refer to as \( \text{sortBy}(\Gamma, \langle s_{1}, \ldots, s_{n}\rangle, s^{2}) \). Its input consists of (i) a set of constraints \( \Gamma \), (ii) a set of ordering relation symbols \( \langle s_{1}, \ldots, s_{n}\rangle \) over constraints, and (iii) a constant \( s^{2} \in \{\text{ASC}, \text{DESC}\} \), specifying whether the sort has to be performed in an ascending (ASC) or descending (DESC) order. It returns a list of constraints ordered on the basis of the strict total order relation \( \leq_{s} \).

Fig. 5 depicts the following example DECLARE model:
with the ordering relation 
(specifying whether a second
), bearing a set of constraints
) is the output of mining al-
) is recognised as redundant resides in the fact
that it imposes that e occurs after d. However, the already visited constraint Participation(f) and NotCoExistence, d respectively state that f always occur and that its occurrence implies d to not occur at all. Therefore, specifying conditions based on the occurrence of task (d) is redundant because d cannot occur. For what Participation(a) is concerned, it is classified as redundant because of the already visited constraints Precedence(a, b), Participation(c), and Precedence(b, c). They specify that c must always occur (Participation(c)), and that if c occurs, then b must precede it (Precedence(b, c)). Hence, b always occurs too. Precedence(a, b) specifies that b cannot occur if it is not preceded by a. Thus, a must occur as well and Participation(a) can be classified as redundant.

Notice that Response(f, h) is redundant too. In fact, Response(f, g) specifies that if f occurs, then g must occur too. Response(g, h) imposes that, after g, h must occur. As a consequence, after f, h must also occur. However, Response(g, h) is checked only as the very last constraint in the list, hence after Response(f, h). Therefore, Response(f, h) cannot be recognised as redundant. The redundancy would be detected if a second iteration was conducted over the list, by considering whether the current constraint is already implied by all the others. This is the reason why our approach provides for such a second check, which comes at the price of a slower computation though. Notice that we execute the second iteration from the last element to the first one so that constraints with lower priority are processed (and in case eliminated) first.

Finally, we remark here that there is no risk of overlooking contradicting constraints. This can be intuitively explained by the fact that a contradicting constraint always leads to an empty constraint among the ones in the list, regardless of whether it is evaluated as the first or last. The different order can only affect which constraint among the ones in conflict is checked last and hence classified as contradicting.

5.3. The algorithm

Algorithm 1 outlines the pseudocode of our technique. Its input consists of: (i) A DECLARE model \( M = (\mathcal{A}, \mathcal{E}, \Gamma) \) discovered from an event log \( L \in \mathcal{M}(\mathcal{A}) \), bearing a set of constraints \( \Gamma \) defined over log alphabet \( \mathcal{A} \) and repertoire \( \mathcal{E} \), (ii) a list of ordering relation symbols \( \leq \), and (iii) a Boolean flag \( \mathcal{F} \) indicating whether a second redundancy check has to be performed or not. For every \( C \in \Gamma \), 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 Maps Miner or MINERful. Table 2(a) shows an example of \( \Gamma \), i.e. \( \Gamma = \{ \leq \} \).
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 versions. For the sake of readability, these additional constraints are not reported in Table 2. Table 2(b) shows the output that corresponds to the post-processing of Table 2(a), provided that the ordering relation symbols provided are \(\leq\) and \(\sigma\). 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.

Algorithm 1. Algorithm `makeConsistent`\(\left( {M, \{ \leq_1, \ldots, \leq_5 \}, r^R} \right)\), returning the suboptimal solution to the problem of finding a minimal set of non-conflicting constraints in a discovered DECLARE model. Its input consists of a declarative process model \(M = \langle A, \mathcal{E}, \Gamma \rangle\), a list of ordering relation symbols \(\{\leq_1, \ldots, \leq_5\}\), and a Boolean flag \(r^R\), enabling a second check for redundancies.

Given \(M\) and its constraints set \(\Gamma\), the first operation `removeSubsumptionHierarchyRedundancies` prunes out redundant constraints from \(\Gamma\) based on the subsumption hierarchy. The procedure removes the subsuming constraints if their support is less than or equal to the subsumed ones. Forward and backward constraints are also eliminated if the corresponding coupling constraint has an equivalent support. The result is stored in \(\Gamma^S\). The details of this operation have already been described in [27]. 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 2(b), this operation is responsible for the elimination of Participation\(\langle a \rangle\), due to the fact that \(Init\) is known to hold true.

Thereafter, we partition \(\Gamma\) into two subsets, i.e.: (i) \(\Gamma^S\) consisting of those constraints that are verified over the entire event log (i.e., having a support of 1.0), and (ii) \(\Gamma^U\) containing the remaining constraints. The reason for doing this is that the former is guaranteed to have no conflict: given the fact that constraints are discovered using the alphabet of the event log, those that have a support of 1.0 can be joined, giving rise to a consistent constraint model.

Even though constraints in \(\Gamma^S\) 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 \(\Gamma^U\) may contain both redundancies and inconsistencies. Table 2(b) presents the partition of \(\Gamma\) into \(\Gamma^S\) and \(\Gamma^U\).

First, we initialise an FSA \(A\) to be the identity element w.r.t. the automata product. In other words, \(A\) is initialised to accept any sequence of events that map to a task in the log alphabet. This
automaton incrementally incorporates the constraints of the input model based on their priority. To set up redundancy elimination in \( r^s \) as well as redundancy and inconsistency elimination in \( r^u \), we then order their constitutive constraints according to the criteria specified by the user, \( \langle S_1, \ldots, S_n \rangle \). The ranking determines the priority with which constraints are analysed.

After the sorting, constraints are stepwise considered for inclusion in the refined model by iterating over the corresponding ranked lists. Constraints in \( r^s \), i.e., \( C_i^s \in r^s_{\text{list}} \), are only checked for redundancy, whereas constraints in \( r^u, C_i^u \in r^u_{\text{list}} \), are checked for both redundancy and consistency. For every constraint \( C_i^u \in r^u_{\text{list}} \), redundancy is checked by leveraging language inclusion. In particular, this is done by computing the FSA \( A_i^{c_i^u} \) for \( C_i^u \) and then checking whether its generated language \( L(A_i^{c_i^u}) \) is included inside \( L(A) \), which considers the contribution of all constraints processed 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 \( A \times A_i^{c_i^u} \) and \( C_i^u \) is added to the set \( r^s \) of constraints to be returned. In the example of Table 2(b), \( \text{Existence}(a, d) \) is analysed after the existence constraints \( \text{Init}(a) \) and \( \text{End}(d) \) based on the preliminary sorting operation. It thus turns out to be redundant, because \( \text{Init}(a) \) and \( \text{End}(d) \) already specify that both \( a \) and \( d \) will occur in every trace. Therefore, they will necessarily always co-occur.

Redundancy and consistency checks of constraints \( C_i^u \in r^u_{\text{list}} \) is performed by the \text{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 analysed constraint \( L(A_i^{c_i^u}) \) in \( L(A) \), 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 \( L(A_i^{c_i^u}) \) and \( L(A) \). This is done by checking that \( L(A_i^{c_i^u}) \cap L(A) = \emptyset \). 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 use the constraint subsumption hierarchy. In particular, we use the relaxation operator to retrieve the parent constraint of the conflicting one, and we recursively invoke the \text{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 2(b) by \( \text{ChainResponse}(b, a) \), replaced by \( \text{AlternateResponse}(b, a) \), and by \( \text{NotChainSuccession}(a, d) \), which is removed because a non-conflicting ancestor does not exists. Note that \( \text{NotChainSuccession}(a, d) \) is to be eliminated because of the interplay of the other two \( \text{NotChainSuccession} \) constraints, \( \text{Init}(a) \) and \( \text{End}(d) \). \( \text{ChainResponse}(b, a) \) is in conflict with \( \text{ChainResponse}(b, c) \).

**Algorithm 2.** Algorithm \text{resolveConflictAndRedundancy} \( \left( A, r^s, C, r^u \right) \), adding a constraint \( C \) to the set of constraint \( r^s \), if it has not already been checked (and thus included in \( r^u \)), and is neither conflicting nor redundant with the already added constraints.

![Fig. 6. Product automaton of Init(a), AlternateResponse(a,b), ChainPrecedence(b,c), and AtMostOne(c).](image-url)

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 \text{resolveConflictAndRedundancy} procedure on these two components. To limit the issue of missing some redundancies, due to the single iterative check over the elements in the constraint lists, a second check can be performed. The technique is slightly different from the one applied before and requires more computational time. For this reason, (i) it is performed after the first iteration has taken place, so as to diminish the amount of constraints to be verified and (ii) it is an optional functionality, activated by the user-specified flag \( r^s \). Its pseudocode is listed in Algorithm 3. It takes as input a set of non-conflicting constraints \( r^s \) and a list of order relation symbols \( \langle S_1, \ldots, S_n \rangle \). First, the product-automaton \( A^p \) is built that consists of the product of all constraint-automata of elements in \( r^s \). Thereafter, constraints in \( r^s \) are sorted according to \( \langle S_1, \ldots, S_n \rangle \) in ascending order, i.e., the reverse order w.r.t. the

Please cite this article as: C. Di Ciccio, et al., Resolving inconsistencies and redundancies in declarative process models, Information Systems (2016). http://dx.doi.org/10.1016/j.is.2016.09.005
first pass (this is because constraints with a lower priority are processed first and eliminated if redundant). The ordered list that comes out of the sorting is \( R_{\text{list}} \). For each constraint \( C \) in \( R_{\text{list}} \), a new product-automaton is built that considers all constraints in \( R_k \) except \( C \). It is hereinafter indicated as \( R'_C \). The redundancy check is thus performed: if the language accepted by \( R'_C \) is a subset of the language accepted by \( R_k \), then the declarative process model excluding \( C \) is as restrictive as the declarative process model including \( C \). This means that \( C \) can be classified as redundant.

**Algorithm 3.** Algorithm resolveRedundancies\(^\text{II} \)(\( R_k, \{ \leq, \ldots, \leq_n \} \)), removing from \( R_k \) those constraints that are redundant. The check is applied once for every constraint in \( R_k \), in ascending order w.r.t. the criteria specified by \( \{ \leq_1, \ldots, \leq_n \} \).

**Input:** A set of non-conflicting constraints \( R^\Gamma \) and a list of ordering relation symbols \( \{ \leq_1, \ldots, \leq_n \} \)

```plaintext
1 /* Sort constraints in \( R^\Gamma \) in ascending order */
2 \( R_{\text{list}} \leftarrow \text{sortBy}(R^\Gamma, \{ \leq_1, \ldots, \leq_n \}, \text{Asc}) 
3 /* Build the product-automaton of constraints in \( R^\Gamma \) */
4 \( R^\Gamma \leftarrow \text{Asc}(R^\Gamma_1 \times \ldots \times R^\Gamma_l) \quad \text{with } C^\prime_1, \ldots, C^\prime_l \in R^\Gamma \) and \( l = |R^\Gamma| 
5 /* Resolve redundancies */
6 \text{for } C \in R_{\text{list}} \text{ do}  
7 \( R'_C \leftarrow \bigcup_{C' \in R^\Gamma \setminus \{C\}} \{C'\} \quad \text{// All constraints in } R^\Gamma \text{ except } C 
8 /* Build the product-automaton of all constraints in } R^\Gamma \text{ except } C */
9 \( R^\Gamma_C \leftarrow \text{Asc}(R^\Gamma_{C_1} \times \ldots \times R^\Gamma_{C_m}) \quad \text{with } C^\prime_1, \ldots, C^\prime_m \in R^\Gamma_C \) and \( m = |R^\Gamma_C| 
10 \text{if } \text{Asc}(R^\Gamma_C) \subseteq \text{Asc}(R^\Gamma) \quad \text{then} \quad \text{// If } C \text{ is redundant}  
11 \( R^\Gamma \leftarrow R^\Gamma \setminus \{C\} \quad \text{// Remove } C \text{ from the set of returned constraints} 
```

The algorithm proceeds iteratively for all remaining constraints in \( R_{\text{list}} \). Although every element in the list is visited once, the overall procedure is expensive in terms of computation time because a new product-automaton must be built at every step by the cross-product of \( l - 1 \) constraint-automata, where \( l \) is the cardinality of the set of input constraints \( R^\Gamma \). The procedure cannot take advantage of the associativity of the cross-product operation, because temporary automata are built at every step without storing the intermediate result of \( l - 2 \) cross-products. However, the additional computational effort is compensated by a higher accuracy in the redundancy-check: in the example of Section 5.2, the redundant constraint \( \text{Response}(a, b) \) would be detected by the second-pass algorithm, whilst it was not captured in the first check.

Finally, a last complete pass over constraints in \( R_k \) is done, to check again whether there are subsumption-hierarchy redundancies. If so, \( R_k \) is pruned accordingly.

**Complexity of the algorithm:** We close this section by elaborating on the complexity of the algorithm, considering as input the number of constraints contained in the discovered model. To better highlight the different sources of complexity, we consider the overall complexity as well as the complexity obtained from the crude algorithm, without considering the contribution of the automata-manipulating operations. This is particularly important because, even though in the worst case the automata-manipulating operations are exponential in the number of constraints, in practice, they have a nonmonotonic behavior. Consider, for example, the cross-product operation between an automaton \( A \) and the automaton \( A' = \text{Asc}(E_{\text{reg}}(C)) \) of constraint \( C \): it is not guaranteed that \( A \times A' \) has a number of states bigger than that of \( A \). This is witnessed, e.g., by the automaton in Fig. 6: The size of the automaton of Fig. 3(b), generated by \( \text{Init}(a), \text{AlternateResponse}(a, b) \), and \( \text{ChainPrecedence}(b, c) \), is bigger than the size of its cross-product with the automaton of AtMostOne(c).

**Theorem 5.1.** Given a \( \text{Declare} \) model \( M \) containing \( n \) constraints, a list \( \{ \leq_1, \ldots, \leq_n \} \) of ordering relation symbols, and a Boolean flag \( r^\Gamma \), algorithm makeConsistent\((M, \{ \leq_1, \ldots, \leq_n \}, r^\Gamma) \) runs in time

- \( O(2^n) \), if the automata-manipulating operations are considered part of the algorithm;
- \( O(n^3) \), if \( r^\Gamma \) is true (i.e., the algorithm includes redundancy double check) and the automata-manipulating operations are considered not part of the algorithm;
- \( O(n \cdot \log(n)) \), if \( r^\Gamma \) is false (i.e., the algorithm skips redundancy double check) and the automata-manipulating operations are considered not part of the algorithm.

**Proof.** As shown in [27], operation removeSubsumptionHierarchyRedundancies (lines 1 and 18 in Algorithm 1) requires a check based on a depth-first visit on a subsumption hierarchy’s direct acyclic graph (see Fig. 1) for every constraint in the input model. For the coupling constraints, also their related forward- and backward-unidirectional relation constraints are considered. The hierarchy structure is however fixed and the number of steps for the visit are thus limited. In particular, the worst case is represented by \( \text{ChainSuccession}(a, b) \), as it can be seen in Fig. 1. It requires at most 6 comparisons: 4 for the subsuming constraints, i.e., \( \text{AlternateSuccession}(a, b) \), \( \text{Succession}(a, b) \), \( \text{CoExistence}(a, b) \), \( \text{RespondedExistence}(a, b) \), and 2 for the forward- and backward-unidirectional relation constraint, resp. \( \text{ChainResponse}(a, b) \) and \( \text{ChainPrecedence}(a, b) \). Both invocations thus cost \( O(n) \). By construction, operation removeSubsumptionHierarchyRedundancies returns a set of constraints \( r^\Gamma \) such that \( |r^\Gamma| \leq n \).

Lines 7 and 14 of Algorithm 1 both apply a sorting algorithm to sets of constraints \( r^\Gamma \) and \( r^\Gamma \), respectively. Notice that \( |r^\Gamma| \leq n \) and \( |r^\Gamma| \leq n \). An efficient algorithm such as merge-sort consequently requires \( O(n \cdot \log(n)) \) for this step.

The instructions from line 8 to line 13 of Algorithm 1 are repeated for all constraints in \( R^\Gamma \). Within the loop, \( \text{Asc} \) and \( E_{\text{reg}} \) operations require \( O(1) \), since they are applied to a single constraint. To separate the sources of complexity inherent to the algorithm, and those coming from the manipulation of automata, we explicitly denote the complexity of the language-inclusion check of line 11 and the automata-product of line 12 as \( O_{\text{Asc}} \) and \( O_{\text{Asc}} \), respectively. With this notation at hand, we get that the loop from line 8 to line 13 of Algorithm 1 can be executed in \( O(n \cdot (O_{\text{Asc}} + O_{\text{Asc}})) \).

At line 15 of Algorithm 1, a loop over the constraints in \( r^\Gamma \)
starts. For every constraint in $r_3$, procedure resolveConflictAndRedundancy (Algorithm 2) is invoked. The time complexity of the procedure is affected again by the cross-product operation and the language-check. The recursive calls of lines 10, 12 and 13 are limited and independent of the number of constraints in $r$: the worst case is represented by a ChainSuccession$(a, b)$ constraint passed in input as $C$, because it presents a chain of 4 subsuming constraints (hence, in 4 cases a $\mathcal{R}(C)$ exists) and has both a forward- and a backward-unidirectional relation constraint, namely ChainResponse$(a, b)$ and ChainPrecedence$(a, b)$. For each of them, 4 further subsuming constraints exist as well (cf. Fig. 1(b)). The total amount of recursive calls is thus at most 12. However, every invocation of procedure resolveConflictAndRedundancy for each constraint can include in $r$ up to 3 new constraints in place of the passed one, because the subsuming, forward- and backward-unidirectional relation constraints may be added in line 7, executed within the respective recursive invocations at lines 10, 12 and 13. Therefore, $|r| \leq 3n$, and, in turn, the time required by the loop starting at line 15 of Algorithm 1 is $O(n(\mathcal{O}_{A}^{T} + \mathcal{O}_{I}^{T}))$.

Finally, procedure resolveRedundancies (Algorithm 3) is invoked if and only if parameter $\rho_1$ is set to $\tau$ (line 17 of Algorithm 1). The constraints in $r$ are preliminarily sorted at line 1 of Algorithm 3, with cost $O(n \log(n))$. Then, automaton $A^k$ is built as the cross-product of all constraints in $r$. The computational complexity of this step is $O(n(\mathcal{O}_{A}^{I}))$. Within the loop starting at line 3, for every constraint $C \in r$, the new automaton $A^k$ is built as the cross-product of all constraints in $r$ except $C$ (i.e., $A^k \setminus C$). Each execution of the cross-product brings, again, a cost of $O(n(\mathcal{O}_{A}^{I}))$. At line 6, the languages of $A^k$ and $A^k \setminus C$ are compared, with cost $\mathcal{O}_{A}^{I}$. All in all, the computational cost of resolveRedundancies is thus $O(n^2 (\mathcal{O}_{A}^{I} + \mathcal{O}_{A}^{T}))$.

To conclude, we notice that the two costs $\mathcal{O}_{A}^{I}$ and $\mathcal{O}_{A}^{T}$ can be uniformly represented by a single source of complexity $\mathcal{O}_{A}$, since the inclusion checks all depend on previous cross-product constructions.

Consequently, the overall algorithm runs in time:

$$0\left(n \log(n) + \mathcal{O}_{A}^{I} + \mathcal{O}_{A}^{T} + n^2 \mathcal{O}_{A}^{T}\right)$$

The three statements of the theorem directly follows, by noticing that computing the cross-product automaton of (at most) $n$ regular expressions is, in the worst case, exponential in $n$ [43,44], and that the third element of the sum above is only present when $r^3$ is true. □

**Table 3**

<table>
<thead>
<tr>
<th>(a) MINERful</th>
<th>(b) Declare Maps Miner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original model</td>
<td>306 cns.</td>
</tr>
<tr>
<td>Detected conflicts</td>
<td>2 cnss.</td>
</tr>
<tr>
<td>Detected redundancies</td>
<td>174 cnss.</td>
</tr>
<tr>
<td>Gain</td>
<td>57.52%</td>
</tr>
<tr>
<td>Time</td>
<td>9171 ms</td>
</tr>
</tbody>
</table>

6. Experiments and results

In this section, we illustrate the evaluation of our implemented approach. The approach has been validated in terms of (i) efficacy, measured by the number of pruned redundant constraints and detected inconsistencies, and (ii) efficiency, measured as computation time. Experiments have been conducted on process models discovered from real-world collections of event logs, provided by the IEEE Task Force on Process Mining on the 3TU Datacentrum platform, i.e.,

- the event log of a loan application process of a Dutch financial institute, published in the context of the BPI challenge 2012 [42],
- the collection of three event logs of the Volvo IT incident and problem management, resp. describing closed issues, incidents, and open problems, from the BPI challenge 2013 [43],
- the event log of ITIL processes of Rabobank Group ICT, from the BPI challenge 2014 [46], and
- the event log of a road traffic fines management process [47].

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

In the following subsections, we show (i) an in-depth analysis of the results obtained by checking consistency and redundancy of the models discovered with MINERful and Declare Maps Miner from the loan application process log and (ii) a summary of the results obtained from the redundancy check conducted over the models discovered from all aforementioned logs. We recall here that both MINERful and Declare Maps Miner already provide ad hoc techniques to reduce the size of the returned models [27,19]. Nevertheless, no mechanism allows them to remove inconsistencies or those non-trivial redundancies that our approach is able to find out.

6.1. Consistency and redundancy checking (BPI challenge 2012)

Here, we describe the outcome of the application of the proposed approach to detect inconsistencies over a declarative process model discovered from the event log provided for the BPI challenge 2012. The BPI challenge 2012 log was chosen for such an analysis because it is the one that presents inconsistencies in the discovered model already at relatively elevated thresholds of support. The event log pertains to an application process for personal loans or overdrafts. It contains 262,200 events distributed across 24 event classes and includes 13,087 traces. With this experiment, we show that our approach is capable of pruning the discovered models by detecting inconsistencies within the constraints discovered by two state-of-the-art declarative process discovery algorithms: MINERful and Declare Maps Miner. We set up both miners to return constraints with a support higher than 75%, a confidence higher than 12.5%, and an interest factor higher than 12.5%. In a realistic scenario, indeed, event logs could contain errors due to recording mistakes or exceptional deviations from the usual execution [7]. Therefore, it makes sense to include those rules that are not fulfilled in the totality of the cases. Since we do not know how many errors or exceptional process enactments affected the log under analysis, we based our choice upon previous studies on the sensitivity of the discovered DECLARE constraints to the presence of noise in event logs [25]. The levels of confidence

---

1 http://data.3tu.nl/repository/collection:event_logs_real
Table 4: Results of the experiments over the model discovered from the BPIC 2014 [46] log.

(a) Single pass

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>26</td>
<td>1451</td>
<td>92.273</td>
<td>46.493</td>
<td>36.927</td>
</tr>
<tr>
<td>a</td>
<td>T3 ⊑ σκi 28</td>
<td>1154</td>
<td>92.683</td>
<td>46.402</td>
<td>37.494</td>
</tr>
<tr>
<td>b</td>
<td>T3 ⊑ σκi 27</td>
<td>1027</td>
<td>92.774</td>
<td>46.218</td>
<td>37.420</td>
</tr>
<tr>
<td>T2 ⊑ a</td>
<td>29</td>
<td>1911</td>
<td>92.583</td>
<td>45.919</td>
<td>37.112</td>
</tr>
<tr>
<td>T2 ⊑ a</td>
<td>28</td>
<td>1677</td>
<td>92.488</td>
<td>45.831</td>
<td>36.965</td>
</tr>
<tr>
<td>σκi ⊑ T2 15</td>
<td>1584</td>
<td>93.223</td>
<td>47.315</td>
<td>36.936</td>
<td></td>
</tr>
<tr>
<td>σκi ⊑ T2 15</td>
<td>1737</td>
<td>93.223</td>
<td>47.315</td>
<td>36.936</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>T2 ⊑ σκi 28</td>
<td>1824</td>
<td>93.223</td>
<td>47.315</td>
<td>36.936</td>
</tr>
<tr>
<td>a</td>
<td>σκi σκi 28</td>
<td>1078</td>
<td>92.683</td>
<td>46.402</td>
<td>37.494</td>
</tr>
<tr>
<td>a</td>
<td>σκi σκi 30</td>
<td>1565</td>
<td>92.488</td>
<td>45.831</td>
<td>36.965</td>
</tr>
<tr>
<td>σκi ⊑ T2 15</td>
<td>1571</td>
<td>93.223</td>
<td>47.315</td>
<td>36.936</td>
<td></td>
</tr>
<tr>
<td>σκi ⊑ T2 15</td>
<td>1608</td>
<td>93.223</td>
<td>47.315</td>
<td>36.936</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>σκi σκi 28</td>
<td>1231</td>
<td>92.683</td>
<td>46.402</td>
<td>37.494</td>
</tr>
<tr>
<td>σκi ⊑ T2 30</td>
<td>1565</td>
<td>92.488</td>
<td>45.444</td>
<td>36.525</td>
<td></td>
</tr>
<tr>
<td>σκi ⊑ T2 15</td>
<td>1824</td>
<td>93.223</td>
<td>47.315</td>
<td>36.936</td>
<td></td>
</tr>
<tr>
<td>Random</td>
<td>26</td>
<td>1451</td>
<td>92.273</td>
<td>46.493</td>
<td>36.927</td>
</tr>
</tbody>
</table>

(b) Double pass

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>T2 ⊑ σκi 186</td>
<td>24 767</td>
<td>91.639</td>
<td>43.108</td>
<td>34.395</td>
</tr>
<tr>
<td>b</td>
<td>T3 ⊑ a σκi 187</td>
<td>25 937</td>
<td>91.639</td>
<td>43.108</td>
<td>34.395</td>
</tr>
<tr>
<td>T2 ⊑ a σκi 185</td>
<td>25 684</td>
<td>91.383</td>
<td>42.909</td>
<td>35.295</td>
<td></td>
</tr>
<tr>
<td>T2 ⊑ a σκi 186</td>
<td>28 954</td>
<td>91.383</td>
<td>42.909</td>
<td>35.295</td>
<td></td>
</tr>
<tr>
<td>σκi ⊑ T2 199</td>
<td>35 621</td>
<td>91.401</td>
<td>43.122</td>
<td>35.472</td>
<td></td>
</tr>
<tr>
<td>σκi ⊑ T2 190</td>
<td>36 020</td>
<td>91.401</td>
<td>43.122</td>
<td>35.472</td>
<td></td>
</tr>
<tr>
<td>σκi ⊑ T2 187</td>
<td>27 071</td>
<td>91.639</td>
<td>43.108</td>
<td>34.395</td>
<td></td>
</tr>
<tr>
<td>σκi ⊑ T2 186</td>
<td>24 596</td>
<td>91.639</td>
<td>43.108</td>
<td>34.395</td>
<td></td>
</tr>
<tr>
<td>τ</td>
<td>σκi σκi 184</td>
<td>27 362</td>
<td>91.383</td>
<td>42.909</td>
<td>35.295</td>
</tr>
<tr>
<td>τ</td>
<td>σκi σκi 184</td>
<td>27 257</td>
<td>91.383</td>
<td>42.909</td>
<td>35.295</td>
</tr>
<tr>
<td>τ</td>
<td>σκi σκi 199</td>
<td>37 793</td>
<td>91.401</td>
<td>43.122</td>
<td>35.472</td>
</tr>
<tr>
<td>τ</td>
<td>σκi σκi 199</td>
<td>37 316</td>
<td>91.401</td>
<td>43.122</td>
<td>35.472</td>
</tr>
<tr>
<td>τ</td>
<td>σκi σκi 186</td>
<td>25 771</td>
<td>91.639</td>
<td>43.108</td>
<td>34.395</td>
</tr>
<tr>
<td>τ</td>
<td>σκi σκi 184</td>
<td>27 259</td>
<td>91.383</td>
<td>42.909</td>
<td>35.295</td>
</tr>
<tr>
<td>σκi ⊑ T2 199</td>
<td>36 956</td>
<td>91.401</td>
<td>43.122</td>
<td>35.472</td>
<td></td>
</tr>
<tr>
<td>σκi ⊑ T2 199</td>
<td>35 188</td>
<td>91.401</td>
<td>43.122</td>
<td>35.472</td>
<td></td>
</tr>
<tr>
<td>Random</td>
<td>188</td>
<td>25 041</td>
<td>91.394</td>
<td>42.891</td>
<td>35.244</td>
</tr>
</tbody>
</table>

and interest factor are motivated by the need to limit the unavoidable increased number of constraints that are included in the result as the support threshold is lowered.

Table 3 summarises the results of the experiment. In the first set of experiments (Table 3(a)), we used MINERful. The number of discovered constraints was 306. On top of that, we applied the proposed algorithm, setting (\(\leq_{\text{min}}, 5\text{min}, \leq_{T2}\)) as the ordering relation symbols and \(P\) to false. We obtained 130 constraints in total, with an execution time of 9171 ms. In the original set of 306, there were two sets of conflicting constraints that made the entire model inconsistent. The first set was \(\text{NotChainSuccession}(A\text{-Pre-accepted}, W\text{-Completeren aanvraag}), \text{ChainResponse}(A\text{-Pre-accepted,W\text{-Completeren aanvraag}), and \text{Participation}(A\text{-Pre-accepted})\). The second set was \(\text{NotChainSuccession}(W\text{-Completeren aanvraag}, A\text{-Accepted}), \text{ChainResponse}(W\text{-Completeren aanvraag, A\text{-Accepted}), and \text{Participation}(W\text{-Completeren aanvraag)\). Both inconsistencies were detected by our algorithm. Note that the percentage of reduction over the set of discovered constraints (that was already pruned based on the subsumption hierarchy) was of 58%.

In the second set of experiments (Table 3(b)), we used the Declare Maps Miner. We discovered a set of constraints using the same thresholds for support, confidence and interest factor adopted for the previous experiment. The tool (that provides an ad hoc technique for pruning) discovered 69 constraints. By applying the proposed algorithm starting from this set, we obtained 41 constraints (with an execution time of 2764 ms). The percentage of reduction was still around 40%.

Redundant constraints can be pruned based on complex reduction rules that are not supported by the state-of-the-art declarative process discovery algorithms. For example, from our experiments, we derived that \(\text{AtMostOne}(A\text{-Finalized})\) becomes redundant due to the presence in combination of \(\text{AtMostOne}(A\text{-PartlySubmitted}), \text{Participation}(A\text{-PartlySubmitted}), and \text{AlternatePrecedence}(A\text{-PartlySubmitted})\). Indeed, \(\text{Participation}(A\text{-PartlySubmitted})\) and \(\text{AtMostOne}(A\text{-PartlySubmitted})\) combined ensure that \(A\text{-PartlySubmitted}\) occurs exactly once. Then \(\text{AlternatePrecedence}(A\text{-PartlySubmitted})\) ensures that either \(A\text{-Finalized}\) does not occur or if it occurs it is preceded by the unique occurrence of \(A\text{-PartlySubmitted}\) without the possibilities of other occurrences of \(A\text{-Finalized}\) in between. Another

Table 5: Highest amounts of detected redundancies in the process models.

<table>
<thead>
<tr>
<th>Process</th>
<th>Input cns.</th>
<th>Redundancies</th>
<th>1st pass</th>
<th>2nd pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPIC 2012</td>
<td>226</td>
<td>204</td>
<td>(90.265%)</td>
<td>(68.142%)</td>
</tr>
<tr>
<td>BPIC 2013/1</td>
<td>30</td>
<td>12</td>
<td>(40%)</td>
<td>(30%)</td>
</tr>
<tr>
<td>BPIC 2013/2</td>
<td>36</td>
<td>36</td>
<td>(47.368%)</td>
<td>(27.652%)</td>
</tr>
<tr>
<td>BPIC 2013/3</td>
<td>20</td>
<td>5</td>
<td>(25%)</td>
<td>(25%)</td>
</tr>
<tr>
<td>BPIC 2014</td>
<td>108</td>
<td>38</td>
<td>(35.185%)</td>
<td>(25.926%)</td>
</tr>
<tr>
<td>Fines</td>
<td>46</td>
<td>30</td>
<td>(65.217%)</td>
<td>(54.348%)</td>
</tr>
</tbody>
</table>
example is \textsc{NotSuccession}(W, Nabellen offertes, A, Submitted), which is redundant with the combination of \textsc{Init}(A, Submitted), \textsc{AtMostOne}(A, PartlySubmitted), \textsc{Participation}(A, PartlySubmitted), and \textsc{ChainSuccession}(A, Submitted, A, PartlySubmitted). Indeed, \textsc{AtMostOne}(A, PartlySubmitted) and \textsc{Participation}(A, PartlySubmitted) combined ensure that A, PartlySubmitted occurs exactly once. This constraint in combination with \textsc{ChainSuccession}(A, Submitted, A, PartlySubmitted) and \textsc{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.

We want to finally remark that the pruning of redundancies of our approach does not alter the behaviour of the returned model. Those constraints that are eliminated are indeed those that can be removed without affecting the way in which the process can be enacted, because they do not restrict the possible executions any further. This is substantially different from the pruning based on thresholds like support, confidence and interest factor that modifies the set of the allowed behaviours. The next subsection...

\section*{Table 6}

Best set-ups for tests over process models discovered from real-world logs.

|---------|---------|--------------|-----------|----------------|---------------|---------------|

\begin{tabular}{llllll}
(a) Maximising pruned redundancies & BPI & 2012 & T & 172 & 2265 & 98.567 & 53.430 & 27.920 \\
 & BPI & 2013/1 & T & 21 & 262 & 92.659 & 37.463 & 25.696 \\
 & BPI & 2013/2 & T & 30 & 1565 & 92.488 & 45.444 & 36.525 \\
 & BPI & 2014 & T & 25 & 328 & 96.744 & 54.648 & 33.710 \\
 & Fines & T & 25 & 328 & 96.744 & 54.648 & 33.710 \\

(b) Maximising support, confidence, and interest factor & BPI & 2012 &-decoration & 138 & 3283 & 98.659 & 53.304 & 29.626 \\
 & BPI & 2013/1 & -- & 6 & 287 & 93.384 & 42.105 & 35.299 \\
 & BPI & 2013/2 & T & 6 & 1376 & 98.546 & 39.187 & 28.764 \\
 & BPI & 2013/3 & T & 3 & 211 & 93.489 & 33.155 & 20.270 \\
 & BPI & 2014 & T & 3 & 210 & 93.489 & 33.155 & 20.270 \\
 & Fines & T & 28 & 1154 & 92.683 & 46.402 & 37.494 \\

(c) Minimising computation time & BPI & 2012 & -- & 149 & 1643 & 99.163 & 52.332 & 29.551 \\
 & BPI & 2013/1 & -- & 6 & 259 & 93.403 & 39.496 & 28.764 \\
 & BPI & 2013/2 & -- & 21 & 1165 & 96.129 & 36.108 & 25.423 \\
 & BPI & 2013/3 & -- & 3 & 210 & 93.489 & 33.155 & 20.270 \\
 & BPI & 2014 & -- & 27 & 1027 & 92.774 & 46.218 & 37.420 \\
 & Fines & -- & 23 & 315 & 97.027 & 63.272 & 37.685 \\

(d) Random & BPI & 2012 & Random & 129 & 6041 & 97.475 & 60.052 & 29.940 \\
 & BPI & 2013/1 & Random & 3 & 339 & 91.437 & 36.635 & 29.517 \\
 & BPI & 2013/2 & Random & 11 & 1511 & 95.054 & 37.062 & 25.423 \\
 & BPI & 2013/3 & Random & 0 & 265 & 92.936 & 28.043 & 19.053 \\
 & BPI & 2014 & Random & 26 & 1451 & 91.394 & 42.891 & 35.244 \\
 & Fines & Random & 12 & 470 & 92.273 & 46.493 & 36.927 \\

Please cite this article as: C. Di Cicco, et al., Resolving inconsistencies and redundancies in declarative process models, Information Systems (2016), http://dx.doi.org/10.1016/j.is.2016.09.005
illustrates a comprehensive view on the results of redundancy checking performed on a large set of real-world event logs.

6.2. Redundancy checking analysis

To assess the capability of the proposed approach to identify and prune the redundant constraints from a model, we ran the implemented algorithm on process models discovered from real-world logs. We utilised MINERful to discover the declarative process models from every log provided by the past editions of the BPI challenges in (i) 2012, (ii) 2013, (iii) 2014, and from (iv) the real-world event log of road traffic fines management process. Hereinafter, we will refer to the respective discovered process models as (i) “BPIC 2012”, (ii) “BPIC 2013/1”, “BPIC 2013/2”, and “BPIC 2013/3”, (iii) “BPIC 2014”, and (iv) “Fines”.

We have set the support threshold to 75% for the discovery phase. For the first log in the list, we have set the thresholds for confidence and interest factor to 25% and 12.5%, respectively, whereas for the remaining ones we have used the thresholds 12.5% and 6.25%. These values were chosen so as to keep the returned constraints in a range that allowed several computationally intensive routines: (i) 226 for BPIC 2012, (ii) 30 for BPIC 2013/1, 76 for BPIC 2013/2, and 20 for BPIC 2013/3, (iii) 108 for BPIC 2014, and (iv) 46 for Fines. We have applied our technique on every discovered process model using every combination of defined ordering relations $\leq, \leq_1, \leq_2$, and $\leq_3$, plus a random sort used as a baseline. In all the aforementioned cases, the proposed algorithm was run twice: once having the Boolean flag $\beta$ set to true, thus enabling the second pass over the pruned constraints and once having $\beta$ set to false. For every run, we have measured the number of redundancies pruned, the computation time needed, and the average support $\sigma$, confidence $\kappa$, and interest factor $\iota$ of the returned constraints. We use the first metric to assess the efficiency of our approach, the second one to evaluate its efficiency, and the last three to estimate the fitness of the pruned model w.r.t. the original log.

Table 4 shows the obtained results on BPIC 2014 with (Table 4(b)) and without (Table 4(a)) the second-pass procedure enabled, respectively. The ordering relations are listed in the tables in their order of application. For the sake of readability, only the subscript under the $\leq$ symbol is shown: Hence, e.g., $T_2 \leq_1$ stands for the consecutive application of ordering relations $T_2$ and $\leq_1$, to sort the constraints. In Table 4(a), the highlighted lines show the best sorting policies when the second pass is not enabled in terms of number of computation time, average support/confidence/interest factor of the returned constraints, and detected redundancies. The results confirm the influence of the adopted sequences of ordering relations on the examined metrics: The lowest computation time is achieved when $\leq_3$ is applied first (1027 ms), the highest combination of average $\sigma$, $\kappa$, $\iota$ is obtained when $\leq_3$ is applied first (resp., 0.932, 0.473, 0.369), and the highest number of detected redundancies (30 over 108, i.e., 27.778%) is obtained when $\leq_1$ is applied first. Table 4(b) shows the effect of the application of the second-pass check: the number of detected inconsistencies considerably raises in the range of 38–40 with a tangible gain of 26–166.67% over the first pass. This, however, comes at the price of a far slower computation time, about 10–20 times slower, and of a general decrease in terms of average support/confidence/interest factor. We remark here that the number of performed checks remains acceptable despite the second pass: in no case they amount to more than 199, hence not more than twice the number of the constraints in the original model (108). This helps the computation time to remain under 40 s in all cases. We recall here that an exhaustive search would have required up to approximately $3 \times 10^{32}$ checks over an equivalent number of cross-products between automata, thus being computationally infeasible.

Table 7 shows the best boost effect in terms of detected redundancies given by the second-pass strategy. The highest numbers in terms of pruned redundant constraints are depicted there. Noticeably, 90.265%, 65.217%, and 47.368% of constraints are classified as redundant for BPIC 2012, Fines, and BPIC 2013/2, respectively. This entails that a significant number of constraints could have been omitted from the returned models without

Please cite this article as: C. Di Cicco, et al., Resolving inconsistencies and redundancies in declarative process models, Information Systems (2016), http://dx.doi.org/10.1016/j.is.2016.09.005
altering the set allowed behaviours.

Fig. 7 shows the proportion of pruned constraints over all analysed logs using the sorting that worked best in terms of redundancy detection, i.e., \( \{ \leq \rightarrow \} \). The abscissa indicates the types of templates. Triples of bars respectively represent the cumulative number of constraints that sum up (i) in all input models, (ii) after the first redundancy check, and (iii) after the second-pass redundancy check. Horizontal lines describe the average percentage of pruned constraints after the two redundancy check phases, resp. 50.4% and 64.03%. Relation constraints in particular tend to be more subject to redundancy, because more than half of them are pruned by the application of the proposed algorithm. This can be due to several factors. First, existence constraints can imply many relation constraints, as in the case, e.g., of Participation\( (a) \), implying that also RespondedExistence\( (b, a) \) holds true for all \( b \in \mathfrak{S} \setminus \{ a \} \). Furthermore, non-chain relation constraints are also transitive: Therefore, if, e.g., Response\( (a, b) \) and Response\( (b, c) \) hold true, also Response\( (a, c) \) must hold true. Finally, relation templates create a hierarchical structure of subsumptions, thus parents along the hierarchy branches tend to be often pruned out: this is the case, e.g., when ChainPrecedence\( (a, b) \) holds true, thus making AlternatePrecedence\( (a, b) \), Precedence\( (a, b) \) and RespondedExistence\( (b, a) \) redundant.

Table 6 shows how the sorting order influences the resulting model in all the examined cases. All these results refer to the application of the proposed algorithm only in its first phase, because the second pass tends to alter and level off the metrics of interest. Table 6(a) presents the sequences of ordering relations that maximise the number of pruned constraints: for all models, \( \{ \leq \rightarrow \} \) is the only ordering relation that occurs in every combination. Table 6(b) lists the sequences of ordering relations that allow for the highest support, confidence and interest factor. As expected, \( \leq \) is always involved. Table 6(c) presents the best achieved computation times. In this case, \( \leq \) is always in the list of ordering relations. Finally, Table 6(d) shows the application of a random sort as a baseline: As expected, the random sort leads to less efficacy in terms of detected redundancies, to a higher computation time, and to an average lower support, with respect to all the values achieved by the other sorting criteria.

The results listed in Table 7 are aggregated on the basis of applied ordering relations. In particular, each of them shows mean and standard deviation values for the metrics under analysis, resp. (i) Table 7(a) for the number of pruned constraints, (ii) Table 7(c) for the support, confidence and interest factor of the returned constraints, and (iii) Table 7(b) for the computation time. Highlighted rows evidence the set-ups performing best, i.e., (i) \( \{ \leq \rightarrow \} \), allowing for pruning 42.5 constraints on average, (ii) \( \{ \leq \} \), allowing for computation times of 854.33 ms on average, (iii) \( \{ \leq, \rightarrow \} \), producing constraints that have an average support, confidence, and interest factor of resp. 95.331, 45.618, and 30.485.

The outcomes illustrated both in Tables 6 and 7 show that the user’s choice on the sorting criteria influences the quality of the result in terms of (1) computation time, (2) pruned redundancies, or (3) fitness w.r.t. the event log. (1) When a faster computation is required, the order on the degree of activation linkage should be chosen. This is due to the fact that such a criterion speeds up the building of the product automaton, which is the most expensive operation in terms of computation time. The activations of the constraints with a higher degree of activation linkage are indeed involved as activations of several constraints. Consequently, the higher number of restrictions exerted on the same activation tends to be reflected in a limited number of states and transitions of the associated product-automaton. (2) To maximise the amount of pruned constraints, the partial order on the type and subsumption turns out to be the best choice. Such a criterion was indeed introduced for this purpose. The partial order on the type of constraints tends to rank as first those constraints that entail several other constraints, i.e., those that induce more redundancies. (3) Finally, to keep the constraints with a higher fitness w.r.t. the event log, the order on support, confidence, and interest factor of constraints should be taken into account, because it ranks first those constraints that were fulfilled and activated more often in the event log. As they are ranked first, they are assigned a higher priority when it comes to pruning redundancies out. Because the sorting criteria can be sequentially composed to build a strict total order over the constraints based on the hierarchical application of (partial) orders, different combinations of the aforementioned criteria can be used. The relatively small amount of required running time benefits an interactive selection of the criteria by the users. To further refine the results achieved, the second pass can be enabled at the price of a higher computational effort.

Throughout this section, the results of the application of our approach to real-world benchmark data have been reported and analysed in depth. Experimental evidence has shown that the proposed algorithm can find inconsistencies in the discovered declarative models, which is an unprecedented result in the literature. Furthermore, redundancies are found within the discovered constraint sets, which amounted to approximately one-third on average with a single-pass checking, and increased by a supplementary 15% with a second pass. Both achievements were yielded in reasonable computation times: up to 3 s for the single-pass and up to 40 s for the double-pass. As per the experimental results, the proposed approach enhances the output of both MI-NERful and Declare Maps Miner, i.e., the two state-of-the-art declarative process miners. The following section examines the works published so far in the literature that deal with topics related to our research endeavour.

7. 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 investigated the issue of consistency checking. In particular, in the context of knowledge-based configuration systems, Felfernig et al. [48] have challenged the problem of finding the core cause of inconsistencies within the knowledge base during its update test in terms of minimal conflicting sets (the so-called diagnosis). The proposed solution relies on the recursive partitioning of the (extended) constraint satisfaction problem into subproblems, skipping those that do not contain an element of the propagation-specific conflict [49]. In the same research context, the work described in [50] focuses on the detection of non-redundant constraint sets. The approach is again based on a divide-and-conquer approach, which 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 idea to apply process mining in the context of workflow management systems has been introduced in [51]. Processes are modelled as directed graphs in which vertices represent the activities and edges stand for the dependencies between them. Cook and Wolf [52], at the same time, investigate similar issues in the context of software engineering processes. They describe three methods for process discovery: (i) neural network-based, (ii) purely algorithmic, and (iii) adopting a Markovian approach. The purely algorithmic approach builds a finite state machine

Please cite this article as: C. Di Cicco, et al., Resolving inconsistencies and redundancies in declarative process models, Information Systems (2016), http://dx.doi.org/10.1016/j.is.2016.09.005
where states are fused if their futures (in terms of possible behaviours for the next k steps) are identical. The Markovian approach uses a mixture of algorithmic and statistical methods, so as to cope with noise. However, the results presented in [52] are limited to sequential behaviour only. From [51] onwards, many techniques have been proposed. The \( \alpha \)-algorithm [53] and its extensions \( \alpha^+ \) [54], \( \alpha^* \) [55], and \( \alpha_i^k \) [56] are algorithmic solutions that exploit exclusive behavioural relations between pairs of activities to discover a procedural process model from an event log. Differently from the declarative constraints of DECLARE, such relations are mutually exclusive: since they do not overlap, they are by construction non-redundant. Behavioural profiles have been introduced by Weidlich et al. [57] as metrics to compare the similarity of procedural process models, as well as to measure the compliance of reported process executions w.r.t. a normative process model [58]. They also partition the product space of activities without semantic overlaps.

The work of Polyvyanyy et al. [22] proposes a repertoire of exclusive behavioural relations between pairs of activities, along with a thorough study of their logical and mathematical properties. These behavioural relations are used as a means to analyse or discover procedural models, and cannot be applied to declarative languages.

Our work is related to research on declarative process discovery and modelling. In [9], the authors introduce the first version of Declare Maps Miner, an approach based on the instantiation of a set of candidate DECLARE constraints that are checked against an event log to identify the ones that are satisfied in a higher percentage of traces. This approach has been improved in [19] by reducing the number of candidates to be checked through an Apriori algorithm, originally developed by Agrawal and Srikant for mining association rules [59]. In [60], 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 [10,28], the authors present an approach for the mining of declarative process models expressed through a probabilistic logic. The approach first extracts a set of integrity constraints from a log. Then, the learned constraints are translated into Markov Logic formulae that allow for a probabilistic classification of the traces. In [61,20,25], 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 [21,27], the authors introduce MINERful, 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 not capable to deal with redundancies due to complex interactions of constraints. In [62], the authors propose an extension of the approach presented in [21,27] 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.

Different logic-based approaches have been used to define the semantics of the DECLARE templates. In principle, they have been expressed by means of LTL formulae [63], as in [30,31]. Their interpretation on finite traces with LTL has been later clarified by [32,33]. In [20], SCiFF integrity constraints have been used, based on abductive logic programming [34]. Building on the fact that LTL has the same expressive power as FOL over finite traces [64,65], the works in [27,33] describe DECLARE templates in such formal representation. In [66], DECLARE constraints are translated into equivalent Petri nets with weighted, reset and inhibitor arcs. In [40,21], REs are used to define the semantics of the DECLARE repertoire. Since REs and Monadic Second Order Logic (MSO) over finite traces [33,67] have equivalent expressiveness, REs have a higher expressive power than LTL, and, as such, are a suitable language to include the formulation of DECLARE. This additional expressiveness is exploited in [68] to model DECLARE meta-constraints that account for compensations and contextual constraints. Interestingly, the techniques presented in this article can be seamlessly applied to such enriched models.

Dynamic Condition Response Graphs (DCR Graphs) [69] are a well-known declarative process modelling language alternative to DECLARE. They are not directly discussed from the perspective of consistency and redundancy [70], but can benefit from our work due to their grounding in Büchi automata [71].

Recently, De Smedt et al. [72] have conducted extensive studies on the so-called "hidden dependencies" [73], i.e., on the generation of implicit constraints tying activities due to the interaction of other constraints explicitly defined in a process model. The work of De Smedt et al. has lead to an approach that automatically uncovers the hidden dependencies in order to improve the understandability of declarative models.

8. Conclusion

In this paper, we addressed the problems of eliminating redundant and inconsistent constraint sets that are potentially generated by declarative process mining tools. After providing a formal definition of declarative models, we have formalised the problem and discussed its intractability due to its inherent exponential complexity. Thereupon, we have described our solution based on the notion of automata-product monoid and devised the corresponding analysis algorithms. The evaluation based on our prototypical implementation demonstrates that constraint sets discovered with state-of-the-art declarative process miners can be further pruned such that the result is consistent and locally minimal.

Our approach always finds all the conflicting constraints in a declarative process model. Furthermore, it substantially reduces the size of the discovered models by removing those constraints that do not alter by any means the allowed behaviours. It detects an elevated number of redundancies in process models returned by state-of-the-art discovery algorithms, notwithstanding the fact that they have built-in ad hoc procedures to circumvent the problem of redundancy.

The sorting criteria adopted to sequentially check the constraints play a crucial role in determining the quality to prioritise in the result. The partial order on the type and subsumption yields a higher number of detected redundancies. The order on support, confidence, and interest factor promotes those constraints that are satisfied the most and involving the most frequent tasks, hence raising the average support, confidence and interest factor of the constraints in the returned model. The order on the degree of activation linkage favours a more efficient computation in terms of time. By choosing which criteria to adopt and in which sequence, the user can thus influence the outcome. Nevertheless, it is in our plans to relieve the human actors from this choice, so as to let them specify the preferred target quality to strive for and make the algorithm automatically find a local optimum by trying different
combinations of the aforementioned sorting criteria. Such an extension would be backed by the relatively small computation time required to return the intermediate results to be compared. As shown in the paper, the approach is capable of processing real-world process models in relatively short time. Furthermore, we are planning to integrate our approach with the one of De Smedt et al. [72], which unveils the hidden dependencies among DECLARE constraints. The latter would drive the search for redundancies seen as dependencies from a core set of constraints, interactively decided by the user. It is also in our plans to involve the users to receive feedback on the outcome of the pruning technique in order to gain a better understanding on the perceived quality of the results, in a similar way to the studies reported in [74,75].

In future research, we also aim at extending our work towards redundancy freedom, so as to ensure that, although not necessarily minimal, the discovered model is provably free of redundancies. Furthermore, we want to go beyond the pure control-flow perspective and extend our technique to the case of data- and resource-aware declarative process mining. When mining declarative constraints with references to data and resources, one of the challenges is to identify comparable notions of subsumption and causes of inconsistency. We also plan to follow up on experimental research comparing Petri nets and DECLARE [1,2]. Prior experiments in that regard shed light on the fact that the declarative modelling approach suffered from the lack of consistency and redundancy checks [75,70]. 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.

References