Resolving Inconsistencies in Declarative Process Models based on Culpability Measurement

Carl Corea, Matthias Deisen, and Patrick Delfmann

Institute for Information System Research, University of Koblenz-Landau, Germany
{ccorea,mdeisen,delfmann}@uni-koblenz.de

Abstract. Contrary to traditional process models, declarative process models define a set of declarative constraints to specify the behavior which a process should adhere to. In the scope of process mining, declarative process discovery aims to derive such constraint sets from event logs. Here, a problem for current discovery techniques is that of inconsistency. That is, dependent of certain event log characteristics, the derived constraint set may contain contradictory constraints. This in turn however makes the discovered model unusable, as contradictory constraints make it impossible to execute declarative process models, thus hampering previous process discovery efforts. In this work, we present an approach for resolving inconsistencies in declarative process models, based on methods from the scientific field of inconsistency measurement. We introduce our approach algorithm and evaluate its feasibility with data sets of the BPI Challenge 2017.

Keywords: Declarative Process Models, Inconsistency Resolution, Declare

1 Introduction

Process Discovery is a key part of Process Mining and comprises techniques for the automated derivation of process models [18, 20]. The creation of processes models is performed by the means of discovery algorithms and techniques, which are applied to entail process models based on observed behaviour, for example that in event logs [6, 17]. Recent efforts have been directed towards so-called declarative process models [17]. Where procedural process models provide an imperative description of how exactly company processes should be performed, declarative process models consist of a set of constraints, that must not be violated. Hence, contrary to the explicitly confined process execution in procedural process models, process execution in declarative process models is all allowed behaviour within the set of constraints.

As the declarative constraint set defines how processes can be executed, the quality of this set is of central importance to companies, in order to ensure a correct and compliant process execution. A potential problem here is that of inconsistency [6]. That is, the set of constraints must not contain any contradictory constraints, as this can make it impossible to perform the process such that it satisfies the set of declarative constraints, i.e. the declarative process model cannot be executed. Ensuring the

14th International Conference on Wirtschaftsinformatik,
February 24-27, 2019, Siegen, Germany

consistency of declarative process models in the scope of process discovery is widely recognized as a challenging task [6, 8, 15]. In cases of inconsistency, methods are needed to provide an analysis and (semi-) automated resolution capabilities, as a manual analysis and resolution is not feasible in practice due to the potential size of declarative process models.

In this paper, we describe an approach that can automatically resolve inconsistencies in declarative process models. The approach applies quantitative measures from the scientific field of inconsistency measurement [10]. Such measures allow to assign a numerical value to individual constraints, with the informal meaning that a higher value reflects a higher degree of inconsistency. In result, this quantitative assessment is used to pin-point and delete causes of inconsistency in the declarative process model while ensuring a low amount of information loss. We implemented our approach and conducted an evaluation of data sets of the Business Process Intelligence Challenge 2017.

The remainder of this work is structured as follows. In Section 2, we provide a motivational example and introduce preliminaries. Next, Section 3 introduces our approach algorithm to resolve inconsistencies in declarative process models. The evaluation of our implemented algorithm is presented in Section 4. Last, we discuss our approach in the context of related work in Section 5 and conclude in Section 6. As a design choice, we base our approach on the Declare formalisms, which is a widely acknowledged modelling language for declarative process models [6, 12, 17]. Our approach can however be extended to arbitrary declarative process modelling languages as defined subsequently.

2 Background

This section provides an example for the necessity of resolving inconsistencies in declarative process models. Also, prerequisites for the Declare modelling language and culpability measurement are introduced.

2.1 Declare Modelling Language and Motivational Example

Declarative process models can be defined as a set of constraints, modelling the allowed behaviour that processes should adhere to.

Definition 1 (Declarative Process Model). A declarative process model is a tuple $\boldsymbol{M} = (\boldsymbol{A}, \boldsymbol{T}, \boldsymbol{C})$, where $\boldsymbol{A}$ is a set of *tasks*, $\boldsymbol{T}$ is a set of pre-defined constraint *templates*, and $\boldsymbol{C}$ is the set of actual *constraints*, which instantiate the template elements in $\boldsymbol{T}$ with activities in $\boldsymbol{A}$.

In this paper, we consider Declare, which is a popular declarative modelling language based on a set of constraint templates. Such templates can be used to define declarative constraints by passing tasks as parameters. For instance, the template *init(x)* indicates that every process instance must start with a task *x*. Declare can be used to define the

relationships between two individual tasks, by the means of so-called relationship templates, each taking two input parameters. For example, *Response(x,y)* indicates that for any process instance, if *x* occurs, then later *y* must occur. The template *ChainResponse(x,y)* denotes that for any process instance, if *x* occurs, then *y* must occur *immediately* after *x*, i.e., a directly follows relation. Due to space limitations, we ommit a further detailed discussion of all relationship templates. An overview and a short description is provided in Figure 1. Please see [6, 17] for a further discussion.

The template relationships can be hierarchically ordered. Figure 2 shows the subsumption hierarchy of relational Declare templates considered in this work. For example, a *ChainResponse* between tasks *a* and *b* is also a *Response* between *a* and *b*.

Three important templates for the remainder of our discussion are the so-called *negative* relationship templates *NotCoExistence(x,y)*, *NotSuccession(x,y)* and *NotChainSuccession(x,y)*. *NotCoExistence(x,y)* states that tasks *x* and *y* must never appear in the same process instance. *NotSuccession(x,y)* states that *y* must never occur after *x* in a process instance (regardless of how many tasks are in-between *x* and *y*). *NotChainSuccession(x,y)* states that *y* must not *directly* follow *x* for any process instance. These three negative templates can cause inconsistency of the overall declarative process model in combination with other templates. For example, *ChainSuccession(a,b)* and *NotChainSuccession(a,b)* must not appear simultaneously in the same constraint set, as this is a logical contradiction. As the set of Declare templates is pre-defined, it is possible investigate which combinations of templates can cause inconsistencies. Here we distinguish between three types of Declare inconsistencies, namely *trivial inconsistencies*, *generalization-based inconsistencies* and *path-based inconsistencies*.

- **Trivial Inconsistencies.** We define trivial inconsistencies as the co-existence of any negative constraint and its direct complement, i.e. with the same parameters, in the same constraint set. That is, any constraint-set with the templates *CoExistence(x,y)* and *NotCoExistence(x,y)*, OR, *Succession(x,y)* and *NotSuccession(x,y)*, OR, *ChainSuccession(x,y)* and *NotChainSuccession(x,y)* is trivially inconsistent.

- **Generalization-based Inconsistencies.** The subsumption hierarchy entails that generalization relations can also impose inconsistencies in combination with negative templates. For instance, every *ChainSuccession(x,y)* is also a *Succession* between *x* and *y*. Therefore, the two constraints *ChainSuccession(x,y)* and *NotSuccession(x,y)* are contradictory to each other. Based on the subsumption hierarchy shown in Figure 2, all possible combinations of generalization-based inconsistencies can be defined. Intuitively, *NotCoExistence(x,y)* contradicts all other (non-negative) templates with the parameters *x* and *y*, as any possible occurrence of both *x,y* contradicts the *NotCoExistence* of *x* and *y*. *NotSuccession(x,y)* contradicts *Precedence(x,y), Succession(x,y), Response(x,y)* and all inheriting template types. Last, *NotChainSuccession(x,y)* contradicts *ChainPrecedence(x,y), ChainSuccession(x,y)* and *ChainResponse(x,y)*.

Type	Notation	Template and description
Existence templates		**Cardinality templates**
	1..* x	*Participation*(x) x occurs at least *once*
	0..1 x	*AtMostOne*(x) x occurs at most *once*
		Position templates
	INIT x	*Init*(x) x is the *first* to occur
	END x	*End*(x) x is the *last* to occur
Relation templates		**Forward-unidirectional relation templates**
	x y	*RespondedExistence*(x, y) If x occurs, then y occurs too
	x y	*Response*(x, y) If x occurs, then y occurs after x
	x y	*AlternateResponse*(x, y) If x occurs, y occurs afterwards before x recurs
	x y	*ChainResponse*(x, y) If x occurs, y occurs immediately after it
		Backward-unidirectional relation templates
	x y	*Precedence*(x, y) y occurs only if preceded by x
	x y	*AlternatePrecedence*(x, y) y occurs only if preceded by x with no other y in between
	x y	*ChainPrecedence*(x, y) y occurs only if x occurs immediately before it
		Coupling templates
	x y	*CoExistence*(x, y) x occurs iff. y occurs
	x y	*Succession*(x, y) x occurs iff. it is followed by y
	x y	*AlternateSuccession*(x, y) x and y occur iff. they follow one another, alternating
	x y	*ChainSuccession*(x, y) x and y occur iff. y immediately follows x
		Negative templates
	x y	*NotChainSuccession*(x, y) x and y occur iff. y does not immediately follow x
	x y	*NotSuccession*(x, y) x can never occur before y
	x y	*NotCoExistence*(x, y) x and y never co-occur

Figure 1. Overview of Declare templates (Taken from [6])

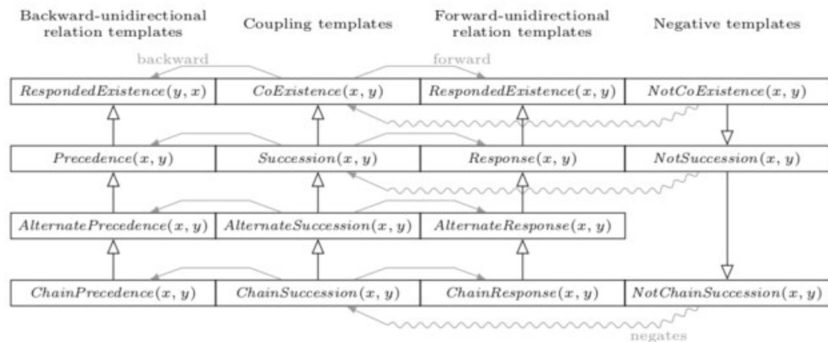


Figure 2. Declare Subsumption Hierarchy (Taken from [6])

- **Path-based Inconsistencies.** So far, we have only discussed the relation of templates with identical parameter signature, e.g. *(x,y)* for both templates. However, *transitive* relations between individual constraints must also be considered. Consider a declarative constraint set consisting of *ChainSuccession(a,b), ChainSuccession(b,c)* and *NotSuccession(a,c)*. The first two constraints state that for every process instance, *a* must directly be followed by *b*. Accordingly, *b* must be directly followed by *c*. Yet, the constraint *NotSuccession(a,c)* demands that *a* must never occur before *c* in the same process instance. Thus, the demands of the first two constraints are contradictory with *NotSuccession(a,c)*. Therefore, inconsistency can occur between groups of templates even if they don't have the same parameters. The definition of path-based inconsistencies will be further discussed in Section 3 based on the definition of so-called *task entailment graphs*.

To clarify, consider the following declarative process model D_1 based on Declare.

ChainSuccession(a,b)	*(i)*	*Response(b,d)*	*(vi)*
NotChainSuccession(a,b)	*(ii)*	*ChainResponse(d,e)*	*(vii)*
NotSuccession(a,b)	*(iii)*	*ChainResponse(e,c)*	*(viii)*
ChainSuccession(b,c)	*(iv)*	*ChainResponse(a,b)*	*(ix)*
NotSuccession(a,c)	*(v)*		

Figure 3. Exemplary Declare model D_1

First, the constraint set in Figure 3 has a trivial inconsistency, as *(i) ChainSuccession(a,b)* and *(ii) NotChainSuccession(a,b)* is contradictory. Also, there are generalization-based inconsistencies. *NotSuccession(a,b)* also entails *NotChainSuccession(a,b)*, therefore, line *(iii)* contradicts line *(i)*. This also yields a conflict between line (iii) and line (ix). Next, the constraints in lines *(i)* and *(iv)* constrain the tasks *abc* as a valid process instance. This path is in contradiction to *(v) NotSuccession(a,c)*, thus a path-based inconsistency. Also, the constraints in line *(i), (vi), (vii)* and *(viii)* define the sequence *abdec* as a valid process instance, which constitutes a path-based inconsistency to *(v) NotSuccession(a,c)*. Path-based inconsistencies also arise for *(ix)*, *(vi), (vii), (viii)* with *(v)*, and (ix), (iv) with (v).

The question may arise how a constraint set such as in Figure 3 can even exist in the first place. Following Di Ciccio et al. (2017), Declare models are mostly automatedly generated in the scope of declarative process discovery. A central problem here are certain completeness charactericstics in the underlying event logs. Recording errors or irregularities in process execution can lead to incomplete or distorted logs. To cope with such problems, most declarative process discovery algorithms introduce the notion of a *support factor*, that is, a parameter defining the fraction of traces in which a discovered template must occur in, in order to accept it in the resulting declarative constraint set [6]. Due to the mentioned problems of log completeness, it can make sense to lower this support factor. However, while this increases the amount of constraints, this does not ensure that these constraints are consistent in relation to each

other. Thus, methods for post-evaluation in process discovery are needed to ensure that the returned constraint set is consistent.

2.2 Culpability Measurement

A scientific field concerned with the analysis of inconsistent information is the field of Inconsistency Measurement, cf. Grant and Martinez (2018). Here, a central object of study are quantitative measures, which allow to assign a numerical value to (elements of) a constraint set, with the informal meaning that a higher value reflects a higher degree of inconsistency, cf. Thimm (2016) for a survey.

A core notion of our approach are *culpability measures*, which are a type of mentioned quantitative measures [5]. Informally, culpability assesses the degree of blame that an individual constraint carries in the context of the overall inconsistency of the constraint set [11]. Let $\mathfrak{D}$ be the set of all declare constraint sets, and $\mathfrak{C}$ the set of individual constraints in a set $\in \mathfrak{D}$. Then, a culpability measure $\mathcal{C}$ is a function

$$\mathcal{C}: \mathfrak{D} \times \mathfrak{C} \rightarrow [0, \infty) \quad (1)$$

which assigns a non-negative real value to a mapping of a declare constraint set and an individual constraint. An example is the so-called $C_{\#}$ measure [11] which assesses the culpability of a constraint γ for a constraint set $\boldsymbol{D}$ via

$$C_{\#}(\boldsymbol{D}, \gamma) = | m \in \boldsymbol{MIS}(\boldsymbol{D}) \mid \gamma \in m | \quad (2)$$

This measure is based on so-called minimal inconsistent subsets of the constraint set $\boldsymbol{D}$. Given a constraint set $\boldsymbol{D}$, the minimal inconsistent subsets (***MIS***) of $\boldsymbol{D}$ are defined via

$$\boldsymbol{MIS}(\boldsymbol{D}) = \{\boldsymbol{D'} \subseteq \boldsymbol{D} \mid \boldsymbol{D'} \textit{ is inconsistent and minimal in terms of set inclusion}\} \quad (3)$$

This definition of minimal inconsistent subsets can be used to find inconsistencies in declarative constraint sets. We revisit the exemplary constraint set D_1 from Figure 3. Figure 4 shows the minimal inconsistent subsets of D_1.

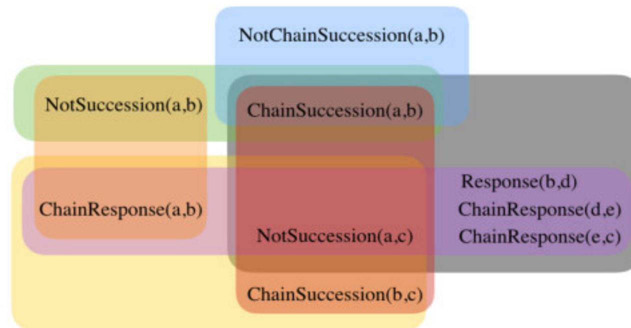


Figure 4. Minimal Inconsistent Subsets for Example D_1 from Figure 3

Based on these ***MIS***, the $C_{\#}$ measure counts the number of minimal inconsistent subsets that a constraint γ belongs to. Applying the $C_{\#}$ measure for the ***MIS*** shown in Figure 4 results in the quantification shown in Figure 5.

C#(D1, ChainSuccesion(a,b))	*= 4*	*C#(D1, Response(b,d))*	*= 2*
C#(D1, NotChainSuccession(a,b))	*= 1*	*C#(D1, ChainResponse(d,e))*	*= 2*
C#(D1, NotSuccession(a,b))	*= 2*	*C#(D1, ChainResponse(e,c))*	*= 2*
C#(D1, ChainSuccession(b,c))	*= 2*	*C#(D1, ChainResponse(a,b))*	*= 3*
C#(D1, NotSuccession(a,c))	*= 4*		

Figure 5. Constraint Culpability for Example D_1 from Figure 3

As a driver for inconsistency resolution, the authors in [5] introduced the notion of a *culpability ranking*, which orders the constraints by their degree of culpability. For the quantification shown in Figure 5, this leads to the following ranking

$$\langle i, v, ix, iii, iv, vi, vii, viii, ii \rangle \quad (4)$$

indicating a prioritization of which constraints should be attended to.

In the following Section, we discuss how this culpability ranking can be exploited for inconsistency resolution and show our algorithm, which includes the computation of minimal inconsistent subsets, culpability assessment and ranking.

3 Inconsistency Resolution Algorithm

3.1 Approach

To resolve inconsistency in a declarative constraint set, there are generally two possibilities [6]: One, a *new* constraint set is constructed, by iteratively moving elements of the old set to the new set, if their introduction does not cause an inconsistency. This technique can be compared to an optimisation problem. A potential disadvantage here is an information loss due to local optima [6]. A different possibility to resolve inconsistency is to use the original constraint set, and iteratively *delete* elements until the constraint set is consistent again. In this context, Grant and Hunter (2011) denote this approach as stepwise inconsistency resolution, and discuss its advantages related to less information loss. In order to resolve inconsistencies with the goal of mitigating information loss, we therefore base our approach on the latter approach of stepwise resolution. Let a constraint set $\boldsymbol{D}$, the deletion of a constraint γ from $\boldsymbol{D}$ is defined via $deletion(\gamma) = \boldsymbol{D} \setminus \{\gamma\}$. Stepwise inconsistency resolution by *deletion* is thus a sequence of deletions $S = <d_1,\ldots,d_n>$. We denote $d_\gamma(\mathbf{D})$ as the constraint set obtained from deleting the element γ from $\boldsymbol{D}$. We denote $\mathbf{\Gamma}_S = <\gamma_1,\ldots,\gamma_n>$ as the individual constraints deleted in the Sequence S via the deletions $d_1,\ldots,d_n$.

The challenge is to find suitable elements which to delete, such that the sequence S results in a low amount of information loss. The authors in [11] do not provide means

to determine which elements should be deleted. Here, our algorithm extends the approach by [11] and utilizes the introduced culpability measure $C_{\#}$ to automatedly determine which elements to delete. Let a constraint γ with a $C_{\#}$ value of n_{γ}, then, deleting γ results in the resolution of n_{γ} minimal inconsistent subsets [9]. Thus, we exploit the culpability ranking to find the constraint with the highest culpability, which maximises the number of resolved minimal inconsistent subsets with loosing only one constraint. For example, as can be seen from Figure 4, deleting the constraint *ChainSuccession(a,b)* with the highest culpability value of 4 would result in the deletion of 4 minimal inconsistent subsets.

We iteratively perform the steps of *analysis*, *ranking* and *deletion* until the constraint set is consistent. In the following, we discuss the these steps of our algorithm in detail.

3.2 Algorithm

Our algorithm is shown in Figure 6.

Algorithm 1 Inconsistency Resolution

```
1: procedure RESOLVEINCONSISTENCIES(constraintSet)
2:     mis ← computeMIS(constraintSet)
3:     tempMaxConstraint ← ∅
4:     tempValue ← 0
5:     while mis.size > 0 do
6:         for each c : constraintSet do
7:             culpability ← | m ∈ mis | c ∈ m |
8:             if culpability > tempValue then
9:                 tempMaxConstraint ← c
10:                tempValue ← culpability
11:        constraintSet.delete(tempMaxConstraint)
12:        mis ← computeMIS(ConstraintSet)
```

Figure 6. Approach Algorithm

The algorithm takes as input parameter a set of constraints. A first step is the computation of minimal inconsistent subsets in line 2. (cf. the below discussion). The algorithm then performs the steps of a) analyzing the constraint with the highest culpability (lines 6-10), and b) deleting the constraint with the highest culpability (line 11), in an iterative

Algorithm 2 Computation of Minimal Inconsistent Subsets

```
1:  function COMPUTEMIS(constraintSet)
2:      mis ← {∅}
3:      negativeTemplates ← {NotCoExistence, NotSuccession, NotChainSuccession}
4:      negativeConstraints ← getConstraintsInType(negativeTemplates)
5:
6:      for each n : negativeConstraints do
7:          contradictions ← getRulesByTypeAndParams(n.type.complement, n.params)
8:          for each c : contradictions do
9:              mis ← mis ∪ {n, c}
10:
11:     for each n : negativeConstraints do
12:         for each compType : n.type.complementSet do
13:             contradictions ← getRulesByTypeAndParams(compType, n.params)
14:             for each c : contradictions do
15:                 mis ← mis ∪ {n, c}
16:
17:     graph = new ConstraintGraph(constraintSet)
18:     for each ns : getConstraintsByType(NotSuccession) do
19:         pathSet ← computeAllPaths(graph, ns.params)
20:         for each p : pathSet do
21:             mis ← mis ∪ {ns, p.getConstraints()}
        return mis
```

Figure 7. Computation of Minimal Inconsistent Subsets in Declare

manner, while there are still inconsistencies (line 5). To clarify, the deletion of constraints intuitively bears the danger of information loss. Yet, deleting constraints might be necessary to restore consistency, i.e. if there exist inherently contradictory constraints. In this context, our approach offers a recommendation as to which element should be deleted, with the goal of maximizing consistency while deleting the lowest amount of constraints possible. We also store the deleted elements and present them to the user for further inspection after the automated resolution process.

A substantial part of our algorithm is the computation of minimal inconsistent subsets, shown in Figure 7. The function depicted in Figure 7 takes as input parameter the declarative constraint set, and returns the set of minimal inconsistent subsets. We initialize an empty set *mis* for storing minimal inconsistent subsets (line 2). Then, the negative template *types*, as well as the individual *constraints* in the *constraintSet* are defined (lines 3-4). In the following, we denote constraints of the types defined in line 3 as *negative constraints*. We then proceed to compute inconsistencies.

Trivial Inconsistencies (lines 6-9). We iterate over all negative constraints to verify if there is a corresponding complement template with the identical parameter signature. This is performed with the method *findRulesByTypeAndParams(type, params)*, which returns a set of sets. This function takes as input a specific template type and parameters (i.e. an ordered pair of tasks, e.g. *a,b*) and returns a set of sets, where the inner sets contain all corresponding constraints. As depicted in line 7, we pass the complement of the respective negative constraints as a parameter. Consequently, the function used in line 7 returns all trivial inconsistencies to the respective constraint *c*. A new minimal

inconsistent subset containing a pair with the found constraint and constraint *c* is added to the set of minimal inconsistent subsets.

Generalization-Based Inconsistencies (lines 11-15). Section 2 defines the complements to the negative template types *NotCoExistence*, *NotSuccession* and *NotChainSuccession*. We denote these complements as the *complementSet* of the respective negative template. For all negative constraints γ, we iterate over all constraints of a template type contained within the *complementSet* of γ, with identical parameter signature (line 12). To compute inconsistencies, we again use the function *findRulesByTypeAndParams*. Here, we pass as parameters the current complement type to the current negative constraint, and the parameters of the negative constraint (line 13). This yields all generalization-based inconsistencies to the current negative constraint. We add a set containing the negative constraint and the computed contradicting constraint to the set of minimal inconsistent subsets (line 15).

Path-Based Inconsistencies (lines 17-21). After the previous two steps, all constraints in contradiction to *NotCoExistence*- and *NotChainSuccession* constraints have been identified. The detection of all contradictions to templates of type *NotSuccession* requires a path-based perspective, due to inconsistencies arising through transitivity. To find path-based inconsistencies, we construct a graph-like structure consisting of the relations between individual Declare constraints (line 17). We denote this as a task entailment graph.

Definition 2 (Task Entailment Graph). Let the declarative process model $\boldsymbol{M}$, the task entailment graph of $\boldsymbol{M}$ is a tuple $\boldsymbol{G} = (\boldsymbol{A}, \boldsymbol{E}, \boldsymbol{t})$, where $\boldsymbol{A}$ is the set of tasks in $\boldsymbol{M}$, $\boldsymbol{E} \subseteq \boldsymbol{A} \times \boldsymbol{A}$ is the set of directed edges between activities, and $\boldsymbol{t}$ is a function $\boldsymbol{t}: \boldsymbol{E} \rightarrow \boldsymbol{T}$, which maps an individual edge to a template type from $\boldsymbol{T}$ in $\boldsymbol{M}$.

An edge $\boldsymbol{e} \in \boldsymbol{E} = (a,b)$ indicates that the task *a* entails the task *b* via the constraint $\boldsymbol{t(e)}$. To compute path-based inconsistencies, we iterate over all negative constraints of type *NotSuccession* (line 18). Let the parameters of the respective negative constraint be *a* and *b*, then, we compute all paths from *a* to *b* in the task entailment graph (line 19). Each path from *a* to *b* constitutes a contradiction to *NotSuccesion(a,b)*, thus the path is added to the set of minimal inconsistent subsets (line 21).

We revisit the example constraint set from the example in Figure 3. All non-negative constraints are used to construct the task entailment graph, shown in Figure 8. The graph can be understood, such that task *b* is entailed by task *a* via the template type *ChainSuccession*. The rest of the graph is constructed accordingly.

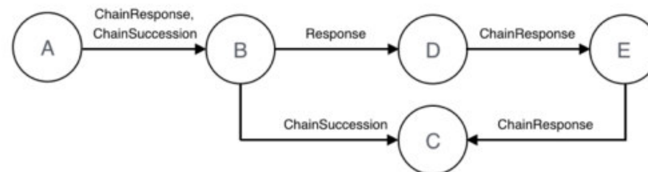
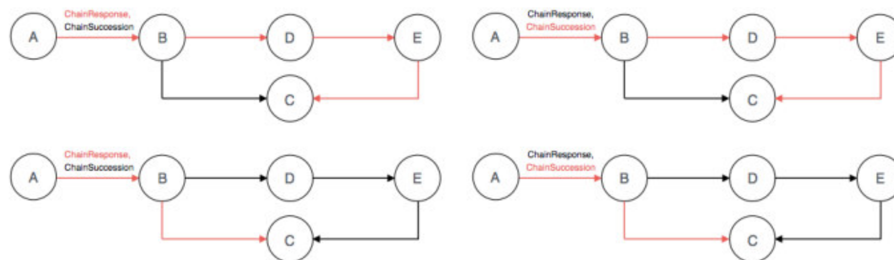


Figure 8. Task Entailment Graph for Example D_1 from Figure 3

The constraint set in this example also contained *NotSuccession(a,c)*. To search for a contradiction to this negative constraint, we verify if there are paths from *a* to *c*. In total, there are four possible paths from *a* to *c*, shown in Figure 9 (Note that the edge from task a to b in the graph has two types, due to the constraints *ChainResponse(a,b)* and *ChainSuccession(a,b)*). Via these four paths, *c* can transitively be entailed from *a.* Hence, all these paths contradict *NotSuccession(a,c)*. Accordingly, the constraint *NotSuccession(a,c)* and all constraints on the identified paths in the task entailment graph
respectively constitute minimal inconsistent subset.



1. **Figure 9.** Paths found from task *a* to task *c*, for Example D_1 from Figure 3

In our algorithm, following the computed minimal inconsistent subset, the constraint with the highest $C_\#$ culpability is deleted. In the example, this is *ChainSuccession(a,b)* with a $C_\#$ value of 4. The proposed algorithm would therefore delete this constraint to generate $d_{ChainSuccession(a,b)}(D_1)$. Continuing the example, the element with the subsequent highest culpability would be *ChainResponse(a,b)* = 3. A deletion would consequently eliminate all remaining ***MIS***, with a total information loss of two constraints. The sequence $\mathbf{\Gamma}_S$ traces the constraints which were deleted for further inspection, i.e. here $\mathbf{\Gamma}_S$ = <*ChainSuccession(a,b) , ChainResponse(a,b)*>.

To conclude, our approach computes all minimal inconsistent subsets with the above algorithm. Then, we delete the constraint with the highest culpability value w.r.t. the $C_\#$ measure. This process is iteratively repeated until the set of constraint is consistent.

4 Evaluation

We implemented our approach to resolve inconsistencies in Declare models in Java. Our evaluation is based on data from the Business Processing Intelligence Challenge 2017 (BPI). In the scope of the BPI, real life event logs are provided.

The analyzed log[1] of the 2017 challenge is an event log of a Dutch financial institute and comprises 262.200 events in 13.087 cases. From this log, we mined a declarative process model in the Declare language using Minerful, which is a state-of-the-art tool for declarative process discovery [6]. We mined three different process

[1] https://www.win.tue.nl/bpi/doku.php?id=2017:challenge

models, with the respective support factors of 75%, 85% and 95%. We then applied our algorithm to all three models. Table 1 summarizes our evaluation results.

Table 1. Results from the application of our algorithm to the BPI challenge log

Support Factor	75%	85%	95%
Discovered Constraints	305	232	207
Initial number of MIS	28954	731	639
Deleted Elements needed	5	1	1
Information Loss	1,63 %	0,43%	0,48%
Runtime	101099ms	9148ms	4695ms

The declarative model mined with a support factor of 75% (*M1*) consisted of 305 constraints. We computed over 28000 inconsistencies in this constraint set, which shows that the lowering the support factor can result in inconsistent models, even for state-of-the-art discovery algorithms. Our algorithm was able to resolve all these subsets in *M1* by deleting a total of only 5 constraints, which equals a total information loss of 1.6% of all initial constraints. This was possible by iteratively determining the constraint with the highest culpability, as deleting this constraint warrants the highest number of resolved ***MIS*** for each iteration. Figure 10 shows an overview of the number of ***MIS*** remaining after each iteration of our algorithm for *M1*. As can be seen, in each of the first three iterations, the number of MIS is reduced roughly by half. Then, a significant reduction from roughly 4000 to 700 MIS can be achieved in iteration 4. Figure 10 also shows the $C_{\#}$ culpability value of the respective constraint with highest culpability in each iteration. As can be seen, the first iteration resolves around 16000 of all 28954 MIS.

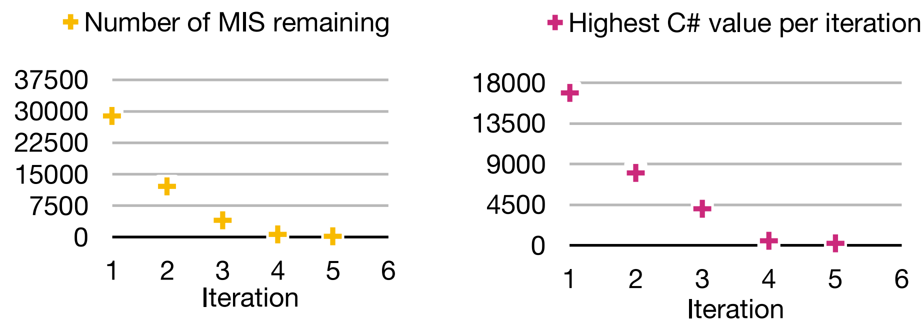


Figure 10. Number of MIS remaining (left) and highest C# culpability for a constraint (right) during the 5 algorithm iterations of resolving *M1*.

Specifically, for *M1*, the five constraints which needed to be deleted to completely resolve inconsistency are shown in Figure 11. The deletions as in Figure 11 can be presented to users. Should they deem this constraint must not be deleted under any circumstances, our approach can be extended with a „whitelist“.

Response(A_Incomplete, O_Accepted), $C_{\#} = 16890$
Precedence(W_Validateapplication, W_PersonalLoancollection), $C_{\#} = 8020$
Response(A_Incomplete, A_Pending), $C_{\#} = 3348$
Precedence(O_CreateOffer, W_PersonalLoancollection), $C_{\#} = 496$
Precedence(O_Created, W_PersonalLoancollection), $C_{\#} = 200$

Figure 11. Constraints deleted corresponding to Figure 10.

For the other two process models, mined with 85%, resp. 95%, support factor, we observed an interesting case. The algorithm was able to make the set of constraints consistent by deleting only one element. In both cases, this was the constraint *NotCoExistence(W_PersonalLoancollection, O_Sent)*. Thus, this constraint was part of all ***MIS*** in both models. Apparently, the support factor configuration yielded exactly this one constraint which makes the entire model unusable. This is a good case for our approach idea of determining the constraint that has the highest blame in the overall inconsistency. Based on culpability measurement, our algorithm could effectively resolve all inconsistencies in a low runtime, deleting only one constraint.

Our results show, that even for a low number of constraints, the number of inconsistencies can be rather large. This underlines the need for post-processing techniques in process discovery such as our approach. Our algorithm was able to resolve these inconsistencies.

5 Related Work

This work is related to consistency checking in Business Process Management (BPM) and declarative process discovery.

Consistency checking in BPM is widely recognized as a challenging task [2-6, 13, 15]. A core task here is to ensure the consistency and correctness of BPM related artifacts. We have focused on the area of verifying the consistency of declarative process model artifacts, respectively a resolution of inconsistency. There have been many proposals for the *qualitative* analysis of the reasons of inconsistencies [1, 7, 14]. However, works such as [13, 15, 19] point out the benefits of a *quantitative* analysis. The intuition here is that individual problems can have a different severity. Hence, using such a quantitative assessment as proposed offers a more sophisticated insight on how to resolve inconsistencies in declarative constraint sets [5, 13].

As a directly related work, we identify the report by Di Ciccio et al (2017). Those authors also proposed an approach to resolve inconsistencies in Declare constraint sets. Those authors however do not consider all ***MIS***, but rather build a maximal consistent *new* constraint set. As discussed, we adapt the approach by [9] of resolution by *deletion*. The authors in [6] point out, that the computation of inconsistencies by comparing all

possible subsets of constraints is intractable. We agree, that this would be indeed unfeasible. To solve this problem, we therefore defined different inconsistency types in Declare, based on the set of pre-defined template types, which allows for informed comparisons and feasible computation. Our evaluation shows, that the computation of ***MIS*** can be performed in a feasible run-time.

Our algorithm promotes declarative process discovery. Due to the introduced notion of a support factor, existing process discovery techniques can yield inconsistent declarative process models [6, 17]. Our work contributes with an approach for automated resolution, allowing for effective post-processing techniques in process discovery.

6 Conclusion

In this work, we presented an approach for resolving inconsistencies in declarative process models. Our approach was implemented for process models in the Declare language. Our evaluation based on a real-life event log showed that it was possible to compute and resolve around 28.000 minimal inconsistent subsets in a feasible runtime. In the process model mined with 75% support factor in our evaluation, there were 305 constraints. However, only 87 constraints were part of any ***MIS*** (i.e. all other constraints had a $C_{\#}$ value of 0). We therefore argue it makes sense to only consider those constraints that contribute towards inconsistency, as in our approach. Deleting constraints with a $C_{\#}$ value of 0 cannot decrease inconsistency in Declare [9]. Deleting the element with the highest value resolves the most ***MIS***. To the best of our knowledge, this work is the first to incorporate culpability measurement.

In our evaluation, all models could be resolved with a deletion of max. 5 constraints, which equaled an information loss of under 2% in all models. This shows that our algorithm is aligned with our goal of ensuring a low amount of information loss. As a direct limitation, elements to delete are automatedly selected. Hence, it would be possible that knowledge which domain experts would not delete is automatedly deleted. To solve this problem, we store the sequence of deleted constraints for further inspection by domain experts. The aim of this paper was to present an approach for fully automated resolution approach. In future work, we will include the possibility to define constraints which must not be deleted. In case such „marked“ constraint would have the highest culpability value in an iteration run, it would simply be possible to delete the constraint with the next highest $C_{\#}$ value based on the computed culpability ranking.

A limitation of this work is, that our algorithm considers only the predefined set of Declare templates. While the majority of declarative discovery tools only focus on these templates, future work should be directed to find ***MIS*** in arbitrary constraints.

As a key learning, we observed that the support factor used during declarative process mining strongly impacts the resulting model. This should be considered by companies seeking to implement declarative process discovery. Automated approaches as presented in this work should be utilized to verify the consistency of the resulting models to allow for correct and compliant process execution.

References

1. Awad, A., Smirnov, S., and Weske, M. (2009). Towards resolving compliance violations in business process models. GRCIS. CEUR-WS. org.
2. Batoulis, K., Nesterenko, A., Repitsch, G., and Weske, M. (2017). Decision management in the insurance industry: Standards and tools. In Proceedings of the BPM 2017 Industry Track (BPM 2017), Barcelona, Spain, September 10-15, 2017., pages 52–63.
3. Batoulis, K. and Weske, M. (2017a). Soundness of decision-aware business processes. In International Conference on Business Process Management, pages 106–124. Springer.
4. Calvanese, D., Dumas, M., Laurson, U., Maggi, F. M., Montali, M., and Teinemaa, I. (2016). Semantics and analysis of dmn decision tables. In International Conference on Business Process Management, pages 217–233. Springer.
5. Corea, C. and Delfmann, P. (2018). Supporting business rule management with inconsistency analysis. In Proceedings of the BPM 2018 Industry Track (BPM 2018), Sydney, Australia, September 09-14, 2018.
6. Di Ciccio, C., Maggi, F. M., Montali, M., and Mendling, J. (2017). Resolving inconsistencies and redundancies in declarative process models. *Information Systems*, 64:425–446.
7. Ghose, A. and Koliadis, G. (2007). Auditing business process compliance. In International Conference on Service-Oriented Computing, pages 169–180. Springer.
8. Graham, I. (2007). Business rules management and service oriented architecture: a pattern language. John wiley & sons.
9. Grant, J., & Hunter, A. (2011). Measuring consistency gain and information loss in stepwise inconsistency resolution. In European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty (pp. 362-373). Springer, Berlin, Heidelberg.
10. Grant, J. and Martinez, M. V. (2018). Measuring Inconsistency in Information. College Publications.
11. Hunter, A., Konieczny, S., et al. (2008). Measuring inconsistency through minimal inconsistent sets. *KR*, 8:358–366.
12. Imgrund, F., Malorny, M., and Janiesch, C. (2017). Eine Literaturanalyse zur integration von business rules und business process management. 13. Internationale Tagung Wirtschaftsinformatik, WI 2017, St.Gallen, 2017.
13. Lu, R., Sadiq, S., and Governatori, G. (2008). Measurement of compliance distance in business processes. Information Systems Management, 25(4):344– 355.
14. Polyvyanyy, A., Ouyang, C., Barros, A., and van der Aalst, W. M. (2017). Process querying: Enabling business intelligence through query-based process analytics. *Decision Support Systems* 100: 41-56.
15. Sadiq, S. and Governatori, G. (2015). Managing regulatory compliance in business processes. In Handbook on Business Process Management 2, pages 265–288. Springer.
16. Thimm, M. (2016). On the compliance of rationality postulates for inconsistency measures: A more or less complete picture. *Künstliche Intelligenz*.
17. van Der Aalst, W. M., Pesic, M., and Schonenberg, H. (2009). Declarative workflows: Balancing between flexibility and support. Computer Science-R&D, 23(2):99–113.
18. van der Aalst, W. M. P., La Rosa, M., and Santoro, F. M. (2016). Business process management. Business & Information Systems Engineering, 58(1):1–6.
19. Weidlich, M., Mendling, J., and Weske, M. (2011). Efficient consistency measurement based on behavioral profiles of process models. IEEE Transactions, 37(3):410–429.
20. Weske, M. (2010). Business process management: concepts, languages, architectures. Springer Publishing Company, Incorporated.