Quasi-Inconsistency in Declarative Process Models

Carl Corea[✉] and Patrick Delfmann

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

Abstract. The field of declarative process discovery comprises techniques for mining declarative constraint sets from event logs. While current techniques verify the relation of individual constraints to the log, they do not consider the interrelation between constraints. This can lead to logical contradictions between the discovered constraints. In this work, we introduce a new form of such contradictions entitled implicit inhibitors. In short, these are sets of constraints which will always be activated together, but demand contradicting reactions. In turn, such constraint sets can be denoted as quasi-inconsistent, as the contained constraints are unsatisfiable should they be activated together. We introduce a structured approach to detect and analyze quasi-inconsistencies in declarative process models and evaluate our approach through formal analysis and run-time experiments on real-life data-sets.

Keywords: Declarative constraints · Implicit inhibition · Declare

1 Introduction

Declarative process models consist of constraints which specify the behavior which company processes should adhere to. Process execution in declarative process models is thus all allowed behaviour within the set of constraints. The semantics of declarative constraints is mostly formalized with temporal logic, e.g. with modelling languages such as DECLARE [6,14]. For example, the DECLARE constraint CHAINRESPONSE(a, b) imposes that if a task a occurs, it must be directly followed by a task b. Likewise, RESPONSE(a, b) states, that if a task a occurs, it must be eventually followed by a task b. When utilizing declarative models, companies face numerous current challenges: In the scope of process discovery, current discovery techniques can yield sets of constraints which are unusable or confusing to modelers [7]. Also, human modelling errors or merging models in the scope of company mergers can yield erroneous models [3].

As an example, consider the constraint sets $\mathcal{C}_1$ and $\mathcal{C}_2$, defined via

$$\mathcal{C}_1 = \{\textsc{ChainResponse}(a, b), \textsc{NotResponse}(a, b)\} \qquad \mathcal{C}_2 = \{\textsc{ChainResponse}(a, b), \textsc{ChainResponse}(b, c), \textsc{NotResponse}(a, c)\}.$$

In both cases, the task a is inhibited by (multiple) constraints which define which tasks must or must not follow. However, these constraints demand logically contradicting reactions to the occurence of task a. In turn, should the task a occur, the declarative process model cannot further be executed.

Motivation: Can't this be already solved with finite state automata? The observant reader might ask whether the above examples are not *inconsistencies* as defined in Di Ciccio et al. [7], and thus could already be detected by existing approaches such as automata products. In short, the above examples are not inconsistencies, but rather *quasi*-inconsistencies, explained as follows.

Di Ciccio et al. [7] have discussed the problem of inconsistent constraints that can be returned during process discovery. Those authors define inconsistency as a declarative process model which does not accept any execution trace, i.e. it is unsatisfiable. An example would be the constraint set $\mathcal{C}_3$, defined via

$$\begin{aligned}\mathcal{C}_3 = \{&\textsc{Participation}(a)\\ &\textsc{ChainResponse}(a,b)\\ &\textsc{NotResponse}(a,b)\}.\end{aligned}$$

As can be seen, $\mathcal{C}_3$ contains the constraint $\textsc{Participation}(\text{a})$, which states that the task a must occur in every execution trace. In result, the constraints $\textsc{ChainResponse}(a,b)$ and $\textsc{NotResponse}(a,b)$ must also always be activated in any execution trace. However, this constellation is inconsistent, i.e. there cannot exist a trace that satisfies the model in $\mathcal{C}_3$.

On the contrary, a model containing the constraints in $\mathcal{C}_1$ or $\mathcal{C}_2$ can accept an arbitrary amount of execution traces, namely any trace which does not contain the task a. For example, a trace "*bcbdbde*" would satisfy respective models in $\mathcal{C}_1$ or $\mathcal{C}_2$. Thus, these constraint sets are not inconsistent as defined in [7], but rather *quasi-inconsistent*. That is, certain tasks are implicitly inhibited by a set of contradictory constraints. Due to this different conceptualization, this implicit inhibition can however not be detected via automata products as in [7], as there can be a non-empty set of accepted traces. In result, this paper discusses a new form of problem in declarative process discovery. Intuitively, declarative process models should not contain sets of constraints as in $\mathcal{C}_1$ or $\mathcal{C}_2$, as the can potentially make models unusable. Yet, current discovery techniques can return such quasi-inconsistent constraint sets. Furthermore, as motivated above, such quasi-inconsistencies can currently not be detected by existing means such as automata products. This is underlined by our experiment results (cf. Sect. 5), where we analyzed real-life models and found more than 25.000 of such contradictory constraint sets as in $\mathcal{C}1$ or $\mathcal{C}_2$, which cannot be detected with the approach by Di Ciccio et al. [7]. In this work, we therefore introduce the notion of quasi-inconsistency and propose a first structured approach to detect and analyze all minimal implicit inhibition sets in declarative process models.

The remainder of this paper is as follows. Section 2 provides background information on declarative process models. In Sect. 3, we introduce the novel concept of *quasi-inconsistent subsets* and show how results from the scientific field of inconsistency measurement can be adapted to detect and analyze such subsets. In Sect. 4, we present an algorithm for the feasible computation of quasi-inconsistent subsets. The proposed capabilities for detection and analysis are evaluated in Sect. 5, followed by a conclusion in Sect. 6.

2 Background

Traditional process models define a clear imperative structure of how exactly company activities should be executed. To allow for more flexibility, declarative process models have received increasing attention [1,6,7]. Here, a declarative process model defines constraints which must be upheld or not be violated, and thus process execution is flexible within this set of constraints.

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 constraint templates, and* $\boldsymbol{C}$ *is the set of actual constraints, which instantiate the template elements in* $\boldsymbol{T}$ *with tasks in* $\boldsymbol{A}$*.*

In this paper, we consider Declare [14], which is a widely acknowledged declarative process modelling language and notation. Declare allows to define constraints by using predefined templates and passing tasks as parameters to respective templates, cf. the examples in Sect. 1. In this way, modelers can use the rather intuitive templates to define constraints, with the formal semantics "hidden" from the user. Formally, the semantics of Declare can be defined with temporal logic [1,4]. This allows to use the amenities of temporal logic checking, as well as to create custom Declare constraint templates.

We define the semantics of Declare constraints with LTL$_p$ [13], a linear-time temporal logic with past. An LTL$_p$ formula is given by the grammar

$$\varphi ::= a|(\neg\varphi)|(\varphi_1 \wedge \varphi_2)|(\bigcirc\varphi)|(\varphi_1 \mathbf{U} \varphi_2)|(\ominus\varphi)|(\varphi_1 \mathbf{S} \varphi_2).$$

Each formula is built from atomic propositions $\in \mathbf{A}$ (relative to a declarative process model), and is closed under the boolean connectives, the unary temporal operators $\bigcirc$ (next) and $\ominus$ (previous), and the binary temporal operators $\mathbf{U}$ (until) and $\mathbf{S}$ (since). Given a declarative process model $\mathbf{M} = (\mathbf{A},\mathbf{T},\mathbf{C})$, a sequence t (with length n) of tasks in $\mathbf{A}$, where $t(i)$ denotes the i^{th} element of the sequence t, the semantics of LTL$_p$ formulae are defined as follows:

t,i $\models$ *True*/t,i $\not\models$ False	t,i $\models a$ iff $t(i) = a$
t,i $\models \neg\varphi$ iff $t,i \not\models \varphi$	t,i $\models \varphi_1 \wedge \varphi_2$ iff $t,i \models \varphi_1$ and $t,i \models \varphi_2$
t,i $\models \bigcirc\varphi$ iff $i < n$ and $t, i+1 \models \varphi$	t,i $\models \ominus\varphi$ iff $i > 1$ and $t, i-1 \models \varphi$

t,i $\models \varphi_1$ $\mathbf{U}$ φ_2 iff $t, j \models \varphi_2$ with $i \leq j \leq n$, and t, $k \models \varphi_1$ for all k s.t. $i \leq k < j$

t,i $\models \varphi_1$ $\mathbf{S}$ φ_2 iff $t, j \models \varphi_2$ with $1 \leq j \leq i$, and $t, k \models \varphi_1$ for all k s.t. $j < k \leq i$

From the above syntax and semantics, we furthermore derive $\varphi_1 \vee \varphi_2$ as $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2$ as $\neg\varphi_1 \vee \varphi_2$, $\Diamond\varphi$ as $\mathit{True}\,\mathbf{U}\varphi$ (which indicates that φ will eventually hold true, possibly later and not directly following $t(i)$), $\diamondminus\varphi$ as $\mathit{True}\ \mathbf{S}\ \varphi$ (which indicates that φ holds true sometime before $t(i)$, but not necessarily directly before $t(i)$), and $\Box\varphi$ as $\neg\Diamond\neg\varphi$ (which indicates that there is no future $t(i)$ which does not satisfy φ).

Based on such LTL$_p$ formulae, the semantics of individual Declare constraints can be defined. For instance, the exemplary constraints used in $\mathcal{C}_1$ and $\mathcal{C}_2$ are defined as ChainResponse$(a, b) \equiv \Box(a \rightarrow \bigcirc b)$, NotChainResponse$(a, b) \equiv \Box(a \rightarrow \neg \bigcirc b)$, NotResponse$(a, b) \equiv \Box(a \rightarrow \neg\Diamond b)$. A standard set of Declare templates and corresponding semantics have been defined derived from the work of [8]. Please see [1,7] or [12] for further details.

An interesting gist about such constrains is that Declare seems to capture *activation-response* relations between tasks. For instance, ChainResponse(a, b) can be interpreted such that, *if* there is an activation a, then this entails a reaction $\bigcirc b$. Therefore, following [2], we use the notion of reactive constraints, which make the activation and reaction semantics of LTL$_p$ constraints explicit.

Definition 2 (Reactive Constraints [2]). *Given a declarative process model $\boldsymbol{M} = (\boldsymbol{A}, \boldsymbol{T}, \boldsymbol{C})$, and a constraint $\in \boldsymbol{C}$ with activation α and reaction φ, a reactive constraint (RCon) Ψ is a pair (α, φ). We denote $\Psi = (\alpha, \varphi)$ as $\alpha \Rightarrow \varphi$. We say that α activates the constraint and the reaction φ.*

Table 1 provides an overview of Declare constraints used in this work, as well as the corresponding RCon and activation. Please refer to [2,7] for a further discussion and classification of activations in Declare constraints.

Table 1. Reactive constraints corresponding to exemplary declare constraints

Constraint	Reactive constraint	Activation
Response(a, b)	$a \Rightarrow \Diamond b$	a
ChainResponse(a, b)	$a \Rightarrow \bigcirc b$	a
AlternateResponse(a, b)	$a \Rightarrow \bigcirc(\neg a\ \mathbf{U}\ b)$	a
Precedence(a, b)	$b \Rightarrow \diamondminus a$	b
ChainPrecedence(a, b)	$b \Rightarrow \ominus a$	b
AlternatePrecedence(a, b)	$b \Rightarrow \ominus(\neg b\ \mathbf{S}\ a)$	b
NotResponse(a, b)	$a \Rightarrow \neg\Diamond b$	a
NotChainResponse(a, b)	$a \Rightarrow \neg \bigcirc b$	a
NotPrecedence(a, b)	$b \Rightarrow \neg\diamondminus a$	b
NotChainPrecedence(a, b)	$b \Rightarrow \neg\ominus a$	b

In result, a quasi-inconsistency is present if we have a constraint set containing multiple RCons with the same activation, but contradictory reactions. In the following, we will show how such quasi-inconsistencies in declarative process models can be detected and analyzed.

3 Detecting and Assessing Quasi-Inconsistencies

3.1 Detection

As declarative constraints are inherently of reactive nature, they underly the principle of ex falso quodlibet: no conclusions can be made without knowledge of activation. As motivated in Sect. 1, this means that the exemplary constraint sets $\mathcal{C}_1$ and $\mathcal{C}_2$ are not inconsistent per se, as it is not known whether these constraints will actually be activated (i.e., there is no constraint like PARTICIPATION(a) which dictates the occurrence of a task a in an execution). In turn, there can be an arbitrary amount of traces that satisfy models as in $\mathcal{C}_1$ and $\mathcal{C}_2$, thus it is not possible to detect quasi-inconsistency by the existing means of automata products as in [7], which detects inconsistency as an empty set of acceptable input traces.

Thus, we present a novel means for detecting quasi-inconsistency. In the following, we use the RCon representation, but sometimes provide specific DECLARE templates for readability. Furthermore, let a constraint c, we denote $out(c)$ as the outcome of a constraint, i.e. φ of the respective RCon.

Definition 3 (Individual Constraint Activation). *A set of activations A activates an individual constraint $c : a \Rightarrow \varphi$ iff $a \in A$.*

Quasi-inconsistencies can arise, if we have a set of activations A', such that A' activates at least two different constraints, and these constraints have contradictory outcomes, e.g. in example $\mathcal{C}_1$, the activation set {a} activates two contradictory constraints. However, as the conclusions of some constraints might be an activation to other constraints themselves via transitive relations, the activation set A' might activate a multitude of constraints. In order to analyze quasi-inconsistencies, all these activated constraints must be considered.

Definition 4 (Constraint Set activation). *A set of activations A activates a set of constraints C iff $\forall c \in C : A \cup \{out(c) | c \in C\}$ activates c.*

Example 1. Consider the constraint set $\mathcal{C}_4$, defined via

$$\mathcal{C}_4 = \{a \Rightarrow b, b \Rightarrow c, c \Rightarrow d\}.$$

For each individual constraint, the activation set is simply the premise of the constraint, i.e. a is the activation set of the individual constraint $a \Rightarrow b$. Furthermore, the activation a also activates the entire set of constraints in $\mathcal{C}_4$ via the transitive relations.

Given a declarative set of constraints, the introduced notions allow to define *quasi-inconsistent subsets.*

Definition 5 (Quasi-Inconsistent Subset). *For a constraint set C, the set of quasi-inconsistent subsets QI is defined as a set of pairs ($\boldsymbol{A}$, $\boldsymbol{C}$), s.t.*

1. $\boldsymbol{C} \subseteq C$

2. $\boldsymbol{A}$ *activates* $\boldsymbol{C}$
3. $\boldsymbol{A} \cup \boldsymbol{C} \models \perp$

To clarify, we consider a set of activations **A**, which activate **C**. Then, the entirety of all activations and activated constraints is inconsistent. Our proposition of quasi-inconsistent subsets allows to determine the "inconsistent subsets" of arbitrary declarative constraints sets, by augmenting activations and thus determining those constraints which will (a) always be activated together, and (b) yield an inconsistency, should they be activated. Consequently, we define minimal quasi-inconsistent subsets analogously.

Definition 6 (Minimal Quasi-Inconsistent Subset). *For a constraint set* C*, the set of minimal quasi-inconsistent subsets* MQI *is defined as set of pairs* $t = (\boldsymbol{A}, \boldsymbol{C})$*, s.t.*

1. t *is a quasi-inconsistent subset in* C
2. for any $t' \subset t$*, where exactly one element is deleted from exactly one of the sets in* t*,:* $t' \not\models \perp$

A minimal quasi-inconsistent subset is a quasi-inconsistent subset which is minimal w.r.t. set inclusion, i.e., removing exactly one constraint resolves the quasi-inconsistency. As we are mostly interested in the distinct constraints which are quasi-inconsistent to each other, we use M^C to denote the set of constraints **C** from any $M \in \mathsf{MQI}$.

Example 2. Consider the following Declare constraint set $\mathcal{C}_5$, defined via

$$\begin{aligned}\mathcal{C}_5 = \{&\textsc{ChainResponse}(a,b), && \textsc{Response}(b,d), && \textsc{NotChainPrecedence}(a,b)\\ &\textsc{ChainResponse}(d,e), && \textsc{NotResponse}(a,b), && \textsc{ChainResponse}(e,c)\\ &\textsc{ChainResponse}(b,c), && \textsc{Response}(a,b) && \textsc{NotResponse}(a,c) \quad \}\end{aligned}$$

Then[1],

$$\begin{aligned}\mathsf{MQI}(\mathcal{C}_5) &= \{M_1, M_2, M_3, M_4, M_5, M_6, M_7\}\\ M_1^C &= \{\textsc{NotChainPrecedence}(a,b), \textsc{ChainResponse}(a,b)\}\\ M_2^C &= \{\textsc{ChainResponse}(a,b), \textsc{NotResponse}(a,b)\}\\ M_3^C &= \{\textsc{Response}(a,b), \textsc{NotResponse}(a,b)\}\\ M_4^C &= \{\textsc{Response}(a,b), \textsc{NotResponse}(a,c), \textsc{ChainResponse}(b,c)\}\\ M_5^C &= \{\textsc{ChainResponse}(a,b), \textsc{ChainResponse}(b,c), \textsc{NotResponse}(a,c)\}\\ M_6^C &= \{\textsc{Response}(a,b), \textsc{Response}(b,d), \textsc{ChainResponse}(d,e),\\ &\qquad \textsc{ChainResponse}(e,c), \textsc{NotResponse}(a,c)\}\\ M_7^C &= \{\textsc{ChainResponse}(a,b), \textsc{Response}(b,d), \textsc{ChainResponse}(d,e),\\ &\qquad \textsc{ChainResponse}(e,c), \textsc{NotResponse}(a,c)\}\\ M_1 &= (\{a\}, \{\textsc{NotChainPrecedence}(a,b), \textsc{ChainResponse}(a,b)\})\end{aligned}$$

[1] M_2–M_7 are omitted due to space restrictions, but are analogously to M_1 (all with activation set $\{a\}$).

This example shows the minimal quasi-inconsistent subsets of the constraint set $\mathcal{C}_5$. As can be seen, all such subsets implicitly inhibit certain tasks in an unsatisfiable way. Thus, in the example, should the task a occur, the resp. model is unsatisfiable and can thus not be used for simulation or to govern compliant process execution. Intuitively, declarative process models should therefore not contain such inhibiting subsets. Through our novel definition of quasi-inconsistent subsets, we are able to detect all such problematic subsets within a set of constraints. Furthermore, our definition of quasi-inconsistent subsets enables a further assessment of resp. subsets.

3.2 Analysis

In order to understand potential inconsistencies, companies should be provided with a careful analysis of detected quasi-inconsistencies. To this aim, results from the scientific field of inconsistency measurement can be adapted [10]. Inconsistency measurement is a discipline concerned with the analysis of inconsistent information. Here, the 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. These measures can be distinguished into so-called *inconsistency measures*, and *culpability measures*. The former is used to assess the inconsistency of the entire constraint set, while the latter is used to assess the degree of blame that individual constraints carry in the context of the overall inconsistency. As some of these measures are based on set-theoretic principles, we propose to adopt these measures to analyze quasi-inconsistent subsets as follows.

Quasi-Inconsistency Measures. Let $\mathfrak{C}$ denote the universe of all declarative constraint sets. Then, an inconsistency measure $\mathcal{I}$ is a function

$$\mathcal{I} : \mathfrak{C} \rightarrow [0, \infty)$$

which assigns a non-negative real value to a constraint set, with the informal meaning that a higher value reflects a higher severity of inconsistency.

Following recent surveys [16,17], there are four measures based on minimal inconsistent subsets which have been proposed, namely the MI-*inconsistency measure*, the MI^C-*inconsistency measure*, the *problematic inconsistency measure* and the *mv-inconsistency measure*. Currently these measures are only defined for inconsistencies (and not for quasi-inconsistencies). To analyze the degree of quasi-inconsistency of declarative process models, these can easily be adapted to fit the use-case of quasi inconsistencies. For further evaluation, we present this for the example of the MI-*inconsistency measure*. We omit a detailled discussion of all measures due to space limitations.

Let $\mathcal{C}$ be a set of constraints and $\mathcal{A}(\mathcal{C})$ denote the tasks in a set $\mathcal{C}$. Then, the adapted versions of the abovementioned measures are defined as follows.

Definition 7 (MQI-inconsistency measure). *Define the* MQI-inconsistency measure *via*

$$\mathcal{I}^{Q}_{\mathsf{MI}}(\mathcal{C}) = |\mathit{MQI}(\mathcal{C})|$$

This measure counts the number of minimal quasi-inconsistent subsets in $\mathcal{C}$.

Example 3. We revisit the constraint set $\mathcal{C}_5$ from Example 2. Then

$$\mathcal{I}^{Q}_{\mathsf{MI}}(\mathcal{C}_4) = 7$$

Culpability Measures. Next to assessing the degree of inconsistency for an entire constraint set, results from inconsistency measurement also allow to quantify the degree of inconsistency for individual constraints. This allows to pinpoint constraints with a high degree of blame for the overall inconsistency. Let $\mathfrak{c}$ denote the universe of all possible constraints, and $\mathfrak{C}$ the universe of declarative constraint sets. Then, a culpability measure Γ is a function

$$\Gamma : \mathfrak{C} \times \mathfrak{c} \to [0, \infty)$$

which assigns a non-negative number to a mapping of an individual constraint to a constraint set, and can thus assess the culpability that an individual constraint represents w.r.t. the constraint set. There are two culpability measures based on minimal inconsistent subsets which have been proposed, namely the *cardinality based culpability measure* $\Gamma_{\#}$, and the *normalized culpability measure* Γ_c [11]. Again, these can be easily adopted for the use-case at hand, which we show for the cardinality-based culpability measure.

Definition 8 (Cardinality-Based Culpability Measure). *Define the* cardinality based culpability measure $\Gamma^{Q}_{\#}$ *via*

$$\Gamma^{Q}_{\#}(\mathcal{C}, \alpha) = |M \in \mathit{MQI}(\mathcal{C}) | \alpha \in M^{C}|$$

This measure counts the number of minimal quasi-inconsistent subsets that a constraint α appears in.

Example 4. We revisit the constraint set $\mathcal{C}_5$ from Example 2. Then

(i)	$\Gamma^{Q}_{\#}(\mathcal{C}_5, \textsc{ChainResponse}(a, b)) = 4$	*(vi)*	$\Gamma^{Q}_{\#}(\mathcal{C}_5, \textsc{Response}(b, d) = 2$
(ii)	$\Gamma^{Q}_{\#}(\mathcal{C}_5, \textsc{NotChainPrecedence}(a, b) = 1$	*(vii)*	$\Gamma^{Q}_{\#}(\mathcal{C}_5, \textsc{ChainResponse}(d, e) = 2$
(iii)	$\Gamma^{Q}_{\#}(\mathcal{C}_5, \textsc{NotResponse}(a, b) = 2$	*(viii)*	$\Gamma^{Q}_{\#}(\mathcal{C}_5, \textsc{ChainResponse}(e, c) = 2$
(iv)	$\Gamma^{Q}_{\#}(\mathcal{C}_5, \textsc{ChainResponse}(b, c) = 2$	*(ix)*	$\Gamma^{Q}_{\#}(\mathcal{C}_5, \textsc{Response}(a, b) = 3$
(v)	$\Gamma^{Q}_{\#}(\mathcal{C}_5, \textsc{NotResponse}(a, c) = 4$		

Culpability measures provide quantitative insight that can help companies to understand and resolve problems in their models [15]. The intuition here is that a higher culpability reflects a higher degree of blame that an individual constraint carries in the context of the overall inconsistency [9]. For example, the $\Gamma^{Q}_{\#}$ is essentially a scoring function which quantifies how many quasi-inconsistent subsets can be resolved, if a constraint is deleted.

4 Computation of Quasi-Inconsistent Subsets

The basis for the proposed detection and analysis are quasi-inconsistent subsets. In the following, we therefore propose an novel approach for the feasible computation of MQI. Algorithm 1 shows our approach to compute minimal quasi-inconsistent subsets for declarative constraints. As a central object of study, we utilize reactive constraints to construct a so-called *reactive entailment graph.*

4.1 Reactive Entailment Graph

Definition 9 (Reactive Entailment Graph). *Given a declarative process model* $\boldsymbol{M} = (\boldsymbol{A}, \boldsymbol{T}, \boldsymbol{C})$*, its reactive entailment graph (REG) is defined as a graph* $G = (A, E, \tau, n)$*, where* $A = \boldsymbol{A} \cup \overline{\boldsymbol{A}}$ *are the tasks in* M *in two forms (with and without overline symbol),* $E \subseteq A \times A$ *is the set of directed edges between tasks in* A*,* τ *is a function* $\tau : E \rightarrow \boldsymbol{T}$ *assigning an individual edge in* E *to a template type in* $\boldsymbol{T}$*, and* n *is a function* $n : E \rightarrow \mathbb{N}$ *which assign a natural number to an edge to allow for multiple edges between the same tasks in* A*.*

The reactive entailment graph is a graph representation of reactive constraints. For example, given the declarative constraint ChainResponse(a, b), this can be represented as two nodes a and b, related by an edge of type $\bigcirc$. In the following, we ommit edge numbering for simplicity.

An important detail is that we include two "forms" of tasks, explained as follows. As can be seen in Table 1, one could argue there are essentially two types of declarative constraints. First, there are constraints such as ChainResponse, which are aimed to ensure that, should some event occur, then another event *must* occur (in a certain way). Then, there are other constraints such as NotChainResponse, which are aimed to ensure that, should some event occur, then another event *must not* occur (in a certain way). The reactive entailment graph captures these two types of *demanding* and *prohibiting* constraints, with the intuition that the overlined form of a task relates to a prohibition and vice versa. Then, the edges, respectively the edge types convey information on how exactly a task is demanded or prohibited, w.r.t a node which is the activation.

Example 5. We revisit the exemplary constraint set from Example 2. Then, this yields the following reactive entailment graph:

$$\begin{tikzcd}
\overline{c} & \overline{a} & \\
a \arrow[u, "\neg\Diamond"] \arrow[d, "\neg\Diamond"'] \arrow[r, bend left, "\bigcirc"] \arrow[r, bend right, "\Diamond"'] & b \arrow[u, "\neg\ominus"'] \arrow[r, "\bigcirc"] \arrow[d, "\Diamond"] & c \\
\overline{b} & d \arrow[r, "\bigcirc"] & e \arrow[u, "\bigcirc"']
\end{tikzcd}$$

This graph encodes the relations between tasks of a declarative process model, as well as their relation type. For example, it can be seen that $a \Rightarrow \Diamond b$, and $a \Rightarrow \neg\Diamond\overline{b}$. This encodes that the activation a *demands* task b, resp. *prohibits* a later occurence of task b. An advantage of including two forms of tasks to encode the demanding, resp. prohibiting, nature of reactive constraints is an efficient way to scan the REG for potential inconsistencies, by searching for pairs of nodes n and n', where $n = \overline{n'}$, as will be discussed in the subsequent section.

The graph relations can be transformed back into the original constraints, where if $(\alpha, t_i) \in E$, then the original reactive constraint c_i is defined as $c_i = \alpha \Rightarrow \tau(\alpha, t_i)t_i$. For an edge $e \in E$, we denote the corresponding constraint as e^C. Given a path p being a sequence of edges in the REG, we denote the set of corresponding constraints captured by p as p^C.

4.2 Algorithm for Computing MQI in Declarative Constraint Sets

Following Definition 5, quasi-inconsistency can only occur if

1. There is at least one task Δ
2. Δ is the outcome of at least two constraints c_1 and c_2
3. $out(c_1) = \overline{out(c_2)}$

Furthermore, the constraints c_1 and c_2 have to be activated simultaneously, thus

4. c_1 and c_2 have the same activation set a

Algorithm 1 computes MQI of declarative constraint sets by exploiting the reactive entailment graph to search for subsets satisfying 1–4. In the following, we explain our algorithm based on the constraint set from Example 2 and the corresponding REG from Example 5.

Algorithm 1. Computation of minimal quasi-inconsistent subsets

```
Input  : Set of constraints C
Output: MQI(C)
1 $qmis \leftarrow \emptyset$;
2 $compConstraints = findComplements(C)$;
3 foreach $n$:$compConstraints$ do
4     $\alpha \leftarrow n.activation$;
5     $\omega \leftarrow n.reactionTask$ ;
6     $\mathbf{P} = findPaths(\alpha, \overline{\omega}) \cup findPaths(\overline{\omega}, \alpha)$;
7     foreach $P$:$\boldsymbol{P}$ do
8         if $\alpha \cup n \cup P^C \models \perp$ then
9             $mis \leftarrow mis \cup n \cup P^C$;
```

In line 1, a set to store minimal quasi-inconsistent subsets ($mqis$) is initialized. Then, we start by identifying all nodes n' of the REG which are a complement to another node n'' (line 2). In the example, there are three such cases, namely a vs $\overline{a}$, b vs $\overline{b}$ and c vs $\overline{c}$ (cf. the corresponding REG). Due to space limitations, we focus on $\overline{c}$ in the following, i.e., we assume the current iterated node $n_i = \overline{c}$. Its activation α is its predecessor in the REG, here a. The algorithm subsequently search for all shortest paths from α to the inverse of the current n_i via a breadth-first search, stored in P (i.e., in our example the algorithm searches for all shortest paths from a to c, and from c to a in the REG). We store possible paths from $\overline{n_i}$ to α, to cope with constraints such as precedence. Also, note that these can be transitive paths with multiple hops. As can be seen in the REG in Example 5, there are four paths from a to c. Subsequently, the algorithm verifies whether the constraints pertaining to a found path P contradict the original constraint $c_i = a \Rightarrow \neg\Diamond\overline{c}$. To this aim, we verify if $\alpha \cup c_i \cup P^C \models\perp$, in which case we have found a minimal quasi-inconsistent subset. In the example, the conditions verified in line 8 are respectively:

$$\begin{aligned}
&(1) && a \cup \textsc{NotResponse}(a,c) \cup \{\textsc{ChainResponse}(a,b), \textsc{ChainResponse}(b,c)\} \models\perp\\
&(2) && a \cup \textsc{NotResponse}(a,c) \cup \{\textsc{Response}(a,b), \textsc{ChainResponse}(b,c)\} \models\perp\\
&(3) && a \cup \textsc{NotResponse}(a,c) \cup \{\textsc{ChainResponse}(a,b), \textsc{Response}(b,d),\\
& && \textsc{ChainResponse}(d,e), \textsc{ChainResponse}(e,c)\} \models\perp\\
&(4) && a \cup \textsc{NotResponse}(a,c) \cup \{\textsc{Response}(a,b), \textsc{Response}(b,d),\\
& && \textsc{ChainResponse}(d,e), \textsc{ChainResponse}(e,c)\} \models\perp
\end{aligned}$$

Note that the activation α is augmented in line 8 to allow for this detection of quasi-inconsistent subsets via Definition 5. Concluding the example, as all 4 cases return true, we have successfully found four *mqis* based on the reactive entailment graph (cf. the formalization of these four *mqis* in Example 2, specifically M_4–M_7).

5 Evaluation

We implemented an MQI-solver for Declare constraints. Our implementation takes as input a DECLARE constraint set C and returns as output $\mathsf{MQI}(C)$ and the introduced (quasi) inconsistency measures. We then performed run-time experiments on the following real-life data sets:

- BPI challenge 2017[2]. This data set contains an event log of a loan application process of a Dutch financial institute. The log is constituted of 1,202,267 events corresponding to 31,509 loan application cases.
- BPI challenge 2018[3]. This data set contains an event log of a process at the level of German federal ministries of agriculture and local departments. The log comprises 2,514,266 events corresponding to 43,809 application cases.

[2] https://www.win.tue.nl/bpi/doku.php?id=2017:challenge.
[3] https://www.win.tue.nl/bpi/doku.php?id=2018:challenge.

– Sepsis 2016[4]. This data set contains an event log of a hospital process concerning the treatment of sepsis, which is a life threatening condition. The log contains around 1000 cases with 15,000 events.

We selected these data-sets because they provide recent data from real-life processes. Also, we selected these data-sets to analyze data of domains which are subject to a high degree of regulatory control and sensible to compliant process execution (e.g., financial-, government- and medical sector).

From these logs, we mined declarative process models using Minerful, which is a state-of-the-art tool for declarative constraint discovery [7]. As configuration for mining, we considered the three parameters of *support*, *confidence* and *interest*. The support threshold indicates the minimum number of traces a constraint has to be fulfilled in for it to be included in the discovered model. Confidence scales the support by the ratio of traces in which the activation occurs, resp. interest scales by the ratio of traces both the constrained tasks occur in. We ran Minerful with a support of 75%, confidence of 12.5% and interest factor of 12.5%, as proposed in the experiment design by [7]. We then applied our implementation to (a) compute all minimal quasi-inconsistent subsets, and (b) compute the $\mathcal{I}^Q_{\mathsf{MI}}$ quasi-inconsistency measures, as well as the $\Gamma^Q_{\#}$ culpability measures for all constraints. The experiments were run on a machine with 3 GHz Intel Core i7 processor, 16 GB RAM (DDR3) under macOS High Sierra Version 10.13.6.

Table 2. Results of run-time experiments for the analyzed data-sets

Log	BPI Challenge '17	BPI Challenge '18	Sepsis '16
Constraints	305	70	207
$\mathcal{I}^Q_{\mathsf{MI}}$ (or # of *mqis*)	28954	25303	7736
Runtime	27074 ms	10930 ms	4379 ms

Table 2 shows an overview of the resp. mined constraints, as well as the number of detected *mqis*, and resp. quasi-inconsistency measures. For the model mined from the BPI'17 log, nearly 29.000 *mqis* were detected. The largest *mqi* had 17 elements. Here, the REG could efficiently be used to detect this subset via a path-based search. In the BPI'18 model, the largest *mqi* contained 22 elements. Also, 62 of the 70 discovered constraints were part of the overall inconsistency (as opposed to only 87/305 constraints in the BPI'17 log). Interestingly, only 70 constraints still lead to a high amount of *mqis*. In the Sepsis'16 log, there were roughly 7.700 *mqis*.

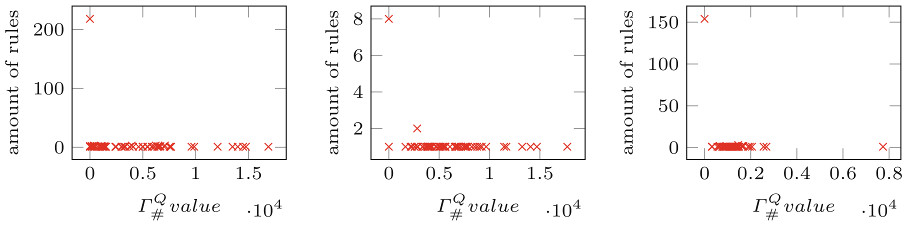
[4] https://data.4tu.nl/repository/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460.

Example 6. For illustration, the following shows an actual *mqi* that we detected in the BPI'17 model.

CoExistence(*A_Accepted*, *A_Concept*),
ChainResponse(*A_Concept*, *W_Validateapplication*).
ChainResponse(*W_Validateapplication*, *W_PersonalLoancollection*),
CoExistence(*A_Accepted*, *A_CreateApplication*),
Response(*A_CreateApplication*, *O_Sent*),
NotCoExistence(*W_PersonalLoancollection*, *O_Sent*)

This actual constraint set returned by the discovery algorithm is quasi-inconsistent. First, *A_Accepted* and *A_Concept* are constrained to appear together. Then, *A_Concept* transitively entails *W_PersonalLoancollection* via two ChainResponse constraints. Also, *A_Accepted* and *A_CreateApplication* are constrained to appear together. Then, because *A_CreateApplication* occurs, the task *O_Sent* must occur later. However, the last constraint demands that *O_Sent* and *W_PersonalLoancollection* never occur together, both of which are however entailed. In result, this is a quasi-inconsistent subset with the activation *A_Accepted*. Note that the discovery algorithm however did not return a constraint such as Participation(*A_Accepted*). Thus, this set of constraints returned by the miner is not inconsistent per se and thus cannot be detected as problematic with existing approaches. Yet, we argue that such a set of constraints should not be contained in any declarative process model, as it is highly confusing and potentially makes the model unusable in practice. Here, our approach allows to detect such problematic sets of constraints as quasi-inconsistent subsets. Table 2 shows that a high number of these *mqis* was actually returned by the miner for all three analyzed logs. As identifying such amount of problematic subsets manually is unfeasible, our approach therefore contributes a feasible means to detect problematic constraints and thus to improve model quality.

In the scope of identifying the actual causes of inconsistency, culpability measures can be used to quantify the degree of blame that individual constraints carry [9]. For the three discovered models, we therefore computed the respective $\varGamma^{Q}_{\#}$ values for all constraints.



(a) (a) BPI challenge 2017 (b) (b) BPI challenge 2018 (c) (c) Sepsis 2016

Fig. 1. Distribution of culpability values for the constraints in the respective models, using the $\varGamma^{Q}_{\#}$ measure.

Figure 1 shows the distribution of the $\varGamma^{Q}_{\#}$ culpability values for the constraints mined from the respective logs. The x-axis shows the respective culpability value, while the y-axis shows the number of constraints with this value. What can be seen for all analyzed models is that we have a high number of constraints with a culpability value of 0 (i.e. they are not part of any *mqi*), and only a few number of individual constraints which are highly responsible in the context of the overall inconsistency (i.e. they are part of many *mqi*). For example, in the constraint set mined from the BPI'2017 log, there are around 200 constraints with a culpability of 0, which can thus be seen as unproblematic. This equates to $\frac{2}{3}$ of all constraints. It is thus possible to identify those (roughly) 100 constraints, which should be attended to. We argue that this is a valuable piece of business intelligence and increases efficiency in managing constraints. Here, the corresponding culpability ranking is a further driver for understanding inconsistencies in the context of resolution strategies. That is, for all the considered models, a few number of constraints can be identified that have the highest culpability values. Thus, these constraints can be strategically targeted first to allow for an effective inconsistency resolution. This is evident in the model mined from the Sepsis 2016 log. There was one specific constraint which was part of all *mqis*, namely $\textsc{Response}(\mathit{AdmissionNC}, \mathit{ReleaseA})$. If one would delete this constraint, all quasi-inconsistencies would be resolved. This information could therefore be exploited for effective resolution means. As a further example, the model derived from the BPI'17 log contained the constraint $\textsc{Response}(\mathit{A\_Incomplete}, \mathit{O\_Accepted})$, which had the highest $\Gamma^{Q}_{\#}$ value of 16890, meaning one could eliminate over 60% of all *mqis* while deleting only one constraint.

To summarize, due to the distribution of culpability values, it would be possible to resolve all quasi-inconsistent subsets through targeting selected constraints via the culpability ranking and deleting only these few elements. This would allow to mitigate all *potential* inconsistencies, i.e. implicit inhibition sets, with a low amount of information loss. As mentioned, this is clearly shown for the model of the Sepsis'16 log, where it would be possible to resolve all *mqis* while deleting only one constraint. In result, we argue that our analysis capabilities by the means of culpability measures provide valuable business insights that can be used as a basis for an informed resolution strategy.

6 Related Work

Our work is related to the discipline of business rules management, i.e., ensuring a consistent set of business rules. In this context, companies have to be supported with means to ensure design-time compliance of declarative process models. While there are some approaches that are aimed to solve problems as discussed in this work by design, i.e. during modelling, this work is related to works that assess an existing set of constraints. This is relevant when existing constraints have to be analyzed, which can be often the case, e.g. analyzing the constraints discovered in process discovery, analyzing a previously modelled set of constraints or analyzing a merged set of constraints after company mergers. A closely related work is that by Di Ciccio et al. [7], who focus on resolving redundancies and inconsistencies in declarative process models. However, as discussed in the introduction, those authors define inconsistency as a model which cannot accept any traces, i.e. it is unsatisfiable. To detect such inconsistencies, those authors represent declarative constraints as finite state automata $\mathcal{A}$, and denote $\mathcal{L}(\mathcal{A})$ as the set of strings accepted by $\mathcal{A}$. Then, those authors can detect inconsistent constraints by identifying those constraint sets that are unsatisfiable via automata products, i.e. $\mathcal{L}(\mathcal{A}') = \emptyset$. As motivated in our introduction, quasi-inconsistent constraint sets can still accept an arbitrary set of traces. Thus, quasi-inconsistency cannot be detected by existing means. Our contribution relative to [7] can be seen in the analysis of the BPI'17 log, which was also analyzed by those authors. Where our approach found nearly 29.000 *potential* inconsistencies, those authors reported 2 inconsistencies. While not inconsistent per se, we argue that quasi-inconsistent sets of constraints such as in $\mathcal{C}_1$ or $\mathcal{C}_2$ should still not be contained in declarative process models, as they can potentially make the model unusable and are highly confusing to modelers. Here, to the best of our knowledge, our approach is the first to offer a tractable solution for detecting all sets of potentially contradictory constraints, i.e. minimal implicit inhibition sets, in declarative process models.

7 Conclusion

In this work, we presented the novel concept of quasi-inconsistencies in declarative process models. As quasi-inconsistencies *potentially* make the model unusable, it is important to detect such problems. Here, we proposed a first approach for such a detection. Through the proposed inconsistency measures, companies are presented with quantitative insights regarding model quality. Element-based culpability measures furthermore allow to prioritize problematic constraints and pin-point individual constraints which should be attended to. Through a computation of MQI based on the reactive entailment graph, our approach is applicable to arbitrary reactive constraints.

Future work could be directed towards the integration of our results, especially the proposed analysis capabilities. In the scope of process discovery, inconsistency measures could be used to as a metric to evaluate the quality of discovered constraint sets. For example, users could define a threshold of allowed quasi-inconsistency. Also, the quantitative insights provided by culpability measures could be used to pin-point the actual causes of quasi-inconsistency, and could thus be integrated into existing methods for resolving errors in declarative process models, e.g. [7], or as a basis for cost-analysis in trace alignment [5].

References

1. Burattin, A., Maggi, F.M., Sperduti, A.: Conformance checking based on multi-perspective declarative process models. Expert Sys. Appl. **65**, 194–211 (2016)
2. Cecconi, A., Di Ciccio, C., De Giacomo, G., Mendling, J.: Interestingness of traces in declarative process mining: the janus LTLp$_f$ approach. In: Weske, M., Montali, M., Weber, I., vom Brocke, J. (eds.) BPM 2018. LNCS, vol. 11080, pp. 121–138. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98648-7_8
3. Corea, C., Delfmann, P.: Supporting business rule management with inconsistency analysis. In: Proceedings of the BPM 2018 Industry Track (2018)
4. De Giacomo, G., De Masellis, R., Montali, M.: Reasoning on LTL on finite traces: insensitivity to infiniteness. In: AAAI (2014)
5. De Giacomo, G., Maggi, F.M., Marrella, A., Patrizi, F.: On the disruptive effectiveness of automated planning for LTL f-based trace alignment. In: Thirty-First AAAI Conference on Artificial Intelligence (2017)
6. van Der Aalst, W.M., Pesic, M., Schonenberg, H.: Declarative workflows: balancing between flexibility and support. Comput. Sci.-Res. Dev. **23**(2), 99–113 (2009)
7. Di Ciccio, C., Maggi, F.M., Montali, M., Mendling, J.: Resolving inconsistencies and redundancies in declarative process models. Inf. Syst. **64**, 425–446 (2017)
8. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering (1999)
9. Grant, J., Hunter, A.: Measuring consistency gain and information loss in stepwise inconsistency resolution. In: Liu, W. (ed.) ECSQARU 2011. LNCS (LNAI), vol. 6717, pp. 362–373. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22152-1_31
10. Grant, J., Martinez, M.V.: Measuring Inconsistency in Information. College Publications, London (2018)
11. Hunter, A., Konieczny, S., et al.: Measuring inconsistency through minimal inconsistent sets. KR **8**, 358–366 (2008)
12. Maggi, F.M., Di Ciccio, C., Di Francescomarino, C., Kala, T.: Parallel algorithms for the automated discovery of declarative process models. Inf. Syst. **74**, 136–152 (2018)
13. Markey, N.: Past is for free: on the complexity of verifying linear temporal properties with past. Acta Informatica **40**(6–7), 431–458 (2004)
14. Pesic, M.: Constraint-based workflow management systems: shifting control to users (2008)
15. Sadiq, S., Governatori, G.: Managing regulatory compliance in business processes. In: vom Brocke, J., Rosemann, M. (eds.) Handbook on Business Process Management 2. IHIS, pp. 265–288. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-642-45103-4_11
16. Thimm, M.: On the expressivity of inconsistency measures. Artif. Intell. **234**, 120–151 (2016)
17. Thimm, M.: On the evaluation of inconsistency measures. In: Grant, J., Martinez, M.V. (eds.) Measuring Inconsistency in Information. Studies in Logic, vol. 73. College Publications, London (2018)