Advanced Auditing of Run-Time Conflicts in Declarative Process Models using Time Series Clustering

Carl Corea
Universität Koblenz-Landau, Germany, ccorea@uni-koblenz.de

Rana Mansour
Universität Koblenz-Landau, Germany, ranamansour@uni-koblenz.de

Patrick Delfmann
Universität Koblenz-Landau, Germany, delfmann@ercis.de

Advanced Auditing of Run-Time Conflicts in Declarative Process Models using Time Series Clustering

Carl Corea, Rana Mansour, and Patrick Delfmann

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

Abstract. We present a novel approach for auditing conflicts between declarative constraints that arise during process execution, i.e., relative to observed traces. As a main advantage, taking a post-execution perspective allows to consider *all* observed traces and their interrelations, and to assess conflicts from a global perspective. Our approach allows to classify and prioritize conflicts as a basis for re-modelling, e.g., which conflicts are an outlier, and which require an urgent change to the model. Also, our approach provides means for quantitative root-cause analysis, i.e., prioritizing which rules need to be changed. We implement our approach and show that it can be applied in settings of industrial scale by means of runtime experiments with real-life data-sets.

Keywords: Inconsistency Measurement, Declarative Process Models, Time Series Clustering

1 Introduction

In theory, declarative process models, i.e., sets of constraints, can be used to reason about ongoing process instances. To this aim, during process execution ("at run-time"), the constraints of a declarative process model are evaluated against *traces*, i.e., sequences of company activities. Unfortunately, in this setting, it can occur easily that traces activate a set of *contradictory* constraints. For example, consider the constraints in M_0, with

$$M_0 = \{\textsc{Response}(a, c), \textsc{NotResponse}(b, c)\}$$

This declarative process model consists of two constraints: $\textsc{Response}(a, c)$ indicates that if an activity a occurs, it must be eventually followed by an activity c. Vice versa, $\textsc{NotResponse}(b, c)$ indicates that if activity b occurs, it must never be followed by activity c. Note that M_0 is not inconsistent, as there can be many traces that satisfy the constraints. For example, a trace "*acb*" follows both constraint specifications. However, if we observe a sequence ab, we run into a run-time conflict [1]. In this case, both constraints are triggered, i.e., activated, by the trace, but the two constraints are in logical conflict: The first constraint demands that activity c "should" follow, where the second demands that c "should not". This cannot hold at the same time. In such cases, there can be no possible continuation of the trace s.t. the rules in M_0 are all satisfied, and the declarative process model cannot be used for any meaningful reasoning anymore, e.g., to govern what activities should follow next.

If (two or more) constraints become conflicting in regard to observed traces, companies may need to re-model the set of constraints, such that it can handle the occurring traces correctly. Deciding *how* to actually re-model can however be a complex task, as companies may encounter thousands of interrelated traces daily. As an example, consider the declarative process model M_1 and the three traces shown in Figure 1.

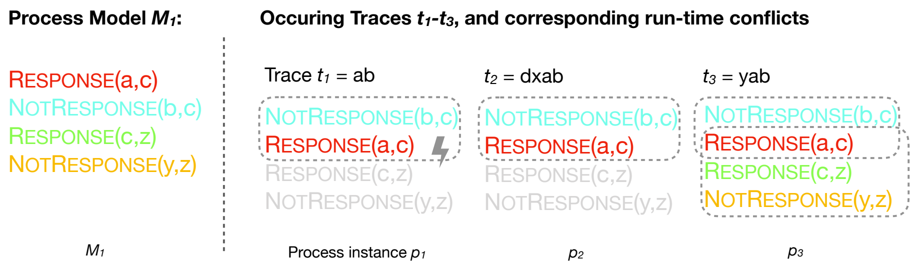


Figure 1. Exemplary run-time conflicts for a sequence of process instances $\rho_1 - \rho_3$.

In the example, there were 2 different types of conflicts, namely RESPONSE(a, c) vs. NOTRESPONSE(b, c) (denoted $\mathcal{I}_1$), and RESPONSE(a, c), RESPONSE(c, z) vs. RESPONSE(y, z) (denoted $\mathcal{I}_2$). However, we see that the *number* of times that the conflicts occurred differs (e.g., $\mathcal{I}_1$ occurs more often than $\mathcal{I}_2$). This could be an important insight for re-modelling, e.g., conflicts that occur more often may be more severe. Vice versa, specific conflicts that only occur a few times may be considered as outliers. Here, methods are needed that assess the *severity* of occurring run-time conflicts **(R1)**. Assessing the severity of run-time conflicts however does not only relate to "simply" counting the number of times that a specific conflict occurred, but also requires to consider *when* it occurred. For example, problems occurring increasingly in recent times may be more severe than problems that decreased over time. Here, methods are needed that can incorporate this *time* aspect when assessing the severity of run-time conflicts **(R2)**. Furthermore, in the example, we see that the constraint RESPONSE(a, c) contributes to both $\mathcal{I}_1$ and $\mathcal{I}_2$. This is an important insight, as it allows to pin-point the actual causes of the run-time conflicts, e.g., as a basis for re-modelling. Here, methods are needed that support companies in assessing which individual rules are highly problematic from a global perspective (over all traces) **(R3)**.

To address the above requirements, we present a novel approach for auditing run-time conflicts from a post-execution perspective. In particular:

- We propose an auditing approach for classifying observed run-time conflicts relative to a set of traces, which allows to decide whether the conflicts induced by the traces were outliers or require a change to the declarative process model (R1).
- Within this approach, we propose a method for identifying similar conflicts using time-series clustering (R2). This allows to pin-point which (groups of) constraints need to be actually changed (R3).

This work is structured as follows. We present preliminaries in Section 2. Our underlying research method is presented in Section 3. Then, our approach is presented in Section 4 and demonstrated in Section 5. We conclude in Section 6.

2 Preliminaries

2.1 Declarative Process Models and Sequences

Declarative process models can be used to confine the allowed company behavior. The intuition is that *any* behavior within the bounds of the constraints is allowed. As opposed to traditional (imperative) process models, this allows for a high degree of flexibility.

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

In this work, we consider the declarative modelling language Declare [2]. Declare comes with a set of "standard" templates, i.e., predicates, which can be used to define constraints for company activities. In this work, we will use the templates shown in Table 1:

Table 1. Declare template types considered in this work.

Template	**Description**	**Activation**
RESPONSE(a,b)	*"If a occurs, it must be eventually followed by b"*	a
NOTRESPONSE(a,b)	*"If a occurs, it must not be followed by b"*	a
EXISTENCE(a)	*"a must occur in the trace (somewhere/at least once)"*	-

We acknowledge there are many other types of Declare templates, e.g., related to cardinalities, but refer the reader to [3] for an overview of further Declare templates. Due to space limitations, we omit a detailed discussion of Declare syntax and semantics and refer the reader to [3].

As motivated in Section 1, declarative process models are evaluated against traces, i.e., sequences of activities. For this, we define a trace as a sequence of activities $t = \langle a_1, ..., a_n \rangle$, where every a_i refers to an activity of a declarative process model.

Definition 2 (Process Instance (Evaluation)). *Given a declarative process model* M *and a trace* t_i*, we define a process instance evaluation as a pair* $\rho_i = (t_i, M)$.

We will refer to a process instance evaluation as a "process instance", for readability. Importantly, the trace t_i can activate certain constraints in the process model M, namely if the activation of a constraint is explicitly mentioned in the trace. For example, recall the constraint RESPONSE(a, b) with the activation a (cf. Table 1), then any trace containing a activates this constraint (In other words, if a occurs in the trace, then this constraint is "active", i.e., it is "waiting" for b to occur in order to be satisfied).

Finally, we do not only consider single traces, but rather consider *multiple* traces from an auditing perspective. For this, we consider a sequence of process instances, which is constructed by matching the individual traces with a shared declarative model. We assume the traces in the sequence to have a partial order, e.g., ordered by the first or average timestamp of the activities of the traces[2].

[1] For readability, we will denote declarative process models as a set of constraints (**C**), cf. M_1

[2] Note that there is no total order between traces, as there is no restriction on when the individual activities in a trace can occur.

Definition 3 (Sequence of Process Instances). *For a declarative model M and sequence of traces $t_1, ..., t_n$, we define a sequence of process instances as an n-tuple $\mathcal{P} = ((t_1, M), ...(t_n, M)) = (\rho_1, ..., \rho_n)$.*

2.2 (Run-Time) Conflicts and Inconsistency Measurement

Run-Time Conflicts. The focus of this paper is on logical conflicts between constraints that can occur at run-time. As an example, consider again the model $M_0 = \{\textsc{Response}(a, c), \textsc{NotResponse}(b, c)\}$. A run-time conflict can occur for M_0 relative to specific traces (here, whenever a sequence ab occurs in a trace). If ab occurs in the trace, there can be no possible continuation of the trace such that all constraints in M_0 can be satisfied. To define this notion of run-time conflict, we build on RV-LTL semantics [4,5], which is a multi-valued logic for runtime verification of declarative constraints in regard to a given trace. Following [5], the RV-LTL semantics of a constraint γ, relative to a trace t, denoted $[t \models \gamma]_{RV}$, can thus have one of four values:

1. $\top$ (γ is permanently satisfied by t for $\underline{\text{any}}$ possible continuation of t)
2. $\bot$ (γ is permanently violated by t for $\underline{\text{any}}$ possible continuation of t)
3. $\top^P$ (γ is possibly satisfied, i.e., γ is currently satisfied, but could become violated from a continuation of t)
4. $\bot^P$ (γ is possibly violated, i.e., γ is currently violated, but could become satisfied from a continuation of t)

For example, for a trace $t_i = a$, we have that $[t_i \models \textsc{Existence}(a)]_{RV} = \top$, as this constraint is permanently satisfied for any continuation of t_i . Likewise, $[t_i \models$ $\textsc{Response}(\text{a,b})]_{RV} =\bot^P$, as it is possibly violated (this can only be healed if b occurs).

Recalling M_0 , if a trace contains the sequence ab, we thus have that all constraints in M_0 are possibly violated, and there can be no possible continuation s.t. all constraints in M_0 leave this state (except to being permanently violated).

Definition 4 (Run-Time Conflict). *Given a declarative process model $\boldsymbol{M} = (\boldsymbol{A}, \boldsymbol{T}, \boldsymbol{C})$ and a trace t, a run-time conflict is a set of constraints $\mathcal{I} \subseteq \boldsymbol{C}$, s.t.*

1. *$\forall c_i \in \mathcal{I} : ([t \models c_i]_{RV} =\bot^P$ or c_i is activated by a constraint $c_j \in \mathcal{I}$).*
2. *There can be no possible continuation of t s.t. all $c_i \in \mathcal{I}$ become satisfied.*

In other words, a run-time conflict affects a set of constraints and is induced by a trace. It triggers a situation s.t. we can have no further continuation of the trace where all affected constraints can become satisfied. This notion was also introduced as a fifth RV-LTL truth value $\bot^c$ in [4] (cf. conflicting sets). A run-time conflict $\mathcal{I}$ is called a *minimal* run-time conflict, iff there is no $\mathcal{I}' \subset \mathcal{I}$ s.t. $\mathcal{I}'$ is also a run-time conflict. For a trace t_i and declarative process model M, we denote the set of all minimal run-time conflict of M relative to t_i as $\mathsf{MIN}^{t_i}(M)$. For readability, for any process instance $\rho_i = (t_i, M)$, we write $\mathsf{MIN}(\rho_i)$ to denote the same as $\mathsf{MIN}^{t_i}(M)$.

Example 1. We recall $M_2 = \{\textsc{Response}(a, c), \textsc{NotResponse}(b, c)\}$ and consider the traces $t_4 = a$ and $t_5 = ab$. Then we have

$$\mathsf{MIN}^{t_4}(M_2) = \{\emptyset\} \qquad \mathsf{MIN}^{t_5}(M_2) = \{\{\textsc{Response}(a, c), \textsc{NotResponse}(b, c)\}\}$$

In regard to t_4, $\textsc{NotResponse}(b, c)$ is not in a possibly violated state. However, in regard to t_5, it is not possible that both constraints in M_2 ever become $\top$ or $\top^P$ again (either they both stay possibly violated, or one will become permanently violated).

Inconsistency Measurement. The introduced notion of minimal run-time conflicts allows to quantify the "severity" of a conflict, e.g., counting how often certain problems occurred. This notion of quantifying the degree of a conflict has been established by the scientific field of inconsistency measurement [6], which studies means for assessing the degree of inconsistency, or conflict, in knowledge representation formalisms. Here, there can be two perspectives, namely a) counting on the level of run-time conflicts, and b) counting on a constraint-level [7].

For a), the level of run-time conflicts, given a process instance $\rho_i = (t_i, M)$, the overall number of conflicts that occurred is $\mathcal{I}_{\#} = |\mathsf{MIN}(\rho_i)|$. Note that every distinct run-time conflict can occur only once in every process instance.

For b), the level of constraints, given a single process instance $\rho_i = (t_i, M)$ and a constraint $\gamma \in M$, we denote $\mathcal{C}_{\#}$ as the number of minimal run-time conflicts in ρ_i that γ is a part of. To clarify, $\mathcal{C}_{\#}(\rho_i, \gamma) = |\{\mathcal{I} \in \mathsf{MIN}(\rho_i) \mid \gamma \in \mathcal{I}\}|$.

Example 2. We recall the model M_1 and traces $t_1 - t_3$ from Figure 1. Then we have

$\mathcal{I}_{\#}(\rho_1) = 1$ (# of inc. in process instance ρ_1) $\quad \mathcal{C}_{\#}(\rho_1, \textsc{Response}(a, c)) = 1$

$\mathcal{I}_{\#}(\rho_2) = 1 \quad \mathcal{C}_{\#}(\rho_1, \textsc{Response}(c, z)) = 0$

$\mathcal{I}_{\#}(\rho_3) = 2 \quad \ldots$

2.3 Related Work and Contributions

Handling logical conflicts in declarative process models has gained recent momentum and brought forward a series of approaches for design-time handling [3, 8, 9] or run-time handling [1,4]. From the requirements **R1** - **R3**, we however see that such approaches do not suffice when considering multiple process instances, as they cannot consider different traces and their interrelations. Therefore, it is important to devise *auditing* approaches.

In a previous work [10], those authors proposed a measure $\Sigma_{\#}$, which can count the number of run-time conflicts that a constraint γ is part of, over a sequence of process instances $\mathcal{P}$. While that measure allows to determine *how often* certain problems occurred relative to a set of traces, that approach cannot consider *when* the problems occurred (as it simply considers a sum). Therefore, in this work, we incorporate the time distribution of *when* conflicts occurred, to provide a more fine-grained analysis.

A further limitation of the work in [10] is that there is no visualization of the insights. As pointed out in a recent agenda-setting contribution on post-execution auditing approaches by PUFAHL AND REHSE [11], this is however an important aspect to *"provide actual value for business users and allow to initiate actions based on the results"* [11, p.27]. The proposed approach is therefore designed to be extendable by a visualization method. Such a visualization should encode the conflict analysis (including time aspect) and helps companies to determine a cutoff as to which constraints actually require re-modelling, and which constraints should be kept as-is. While this is not in the scope of this report, many existing results from time series clustering visualization indicate that heatmaps are highly suitable in the context of our approach [12–14].

2.4 Considered Time Series and Time Series Clustering

Considered Time Series. In essence, we are interested in how conflicts develop over time. Here, we consider two perspectives: a) how different (types of) conflicts occurred over time, and b) how individual constraints were involved in these conflicts, over time. To capture both of these distributions, we consider them as time series. In general, a time series is a sequence of values $T = \langle v_1, ..., v_n \rangle \in \mathbb{R}^n$ [12].

As an example, consider Figure 1. A conflict (type) that occurred in all process instances was $\mathcal{I}_1 = \{\textsc{Response}(a, c), \textsc{NotResponse}(b, c)\}$. So, for perspective a) (conflict type level), we could use $\langle 1, 1, 1 \rangle$ to denote that $\mathcal{I}_1$ occurred in all process instances. For perspective b) (constraint-level), considering the constraint $\textsc{Response}(a, c)$, we could use $\langle 1, 1, 2 \rangle$ to denote how many conflicts this constraint was part of in the individual process instances. In other words, we consider the individual process instances as time points. We consider two types of series, namely conflict-level time series $T^{\mathcal{I}}_{\mathcal{P}} \in \{0, 1\}^n$ and constraint-level time series $T^{\gamma}_{\mathcal{P}} \in \mathbb{N}^n$, where n is the length. For a process instances $\mathcal{P}$, every v_i of these time series is defined as follows

$$\text{Every } v_i \in T^{\mathcal{I}}_{\mathcal{P}} = \begin{cases} 1 \text{ if } \mathcal{I} \in \mathsf{MIN}(\mathcal{P}) \\ 0 \text{ otherwise} \end{cases} \qquad \text{Every } v_i \in T^{\gamma}_{\mathcal{P}} = \mathcal{C}_{\#}(\rho_i, \gamma)$$

To clarify, conflict-level series encode how a run-time conflict $\mathcal{I}$ occurred over time, and constraint-level series encode how many conflicts a constraint γ was part of over time (cf. the above examples $T^{\mathcal{I}_1}_{\mathcal{P}_{\text{Figure 1}}} = \langle 1, 1, 1 \rangle$ and $T^{\textsc{Response}(a,b)}_{\mathcal{P}_{\text{Figure 1}}} = \langle 1, 1, 2 \rangle$).

Time Series Clustering. To identify conflicts or constraints that behaved similarly, we try to identify similar time series. To this aim, we apply time series clustering [15], which is a data mining technique for grouping similar time series from a multivariate time series. The intuition is that the individual time series are to be partitioned into m non-overlapping clusters, where the time series in a cluster are "similar", and the distance to other clusters is maximized. Here, determining when two times series are considered "similar" is performed by means of a similarity measure $sim : T \times T \to \mathbb{R}^{\infty}_{\geq 0}$, that denotes the distance between two time series as a non-negative real value. As an exemplary similarity measure, consider the euclidean distance [16]. Assuming two time series C and Q, the euclidean distance is defined via $sim(C, Q) = \sqrt{\sum_{i=1}^{n} (c_i - q_i)^2}$. This similarity measure computes the distance between two time series C and Q by pair-wise comparing the absolute distance between each time point in the two series. While our approach can be used with any similarity measure (cf. [15] for an overview), we continue our discussion based on the euclidean distance measure.

3 Methodological Considerations: Design Science Research

The main aim of this work is to develop an approach for auditing run-time conflicts between constraints. Following [17], the main aim of this work is therefore a *goal of design*, i.e., to develop an artifact capable of solving specific and relevant problems. Due to this design-oriented focus, we therefore follow a design-science research approach, as this approach is widely acknowleded as a suitable method for works with a primary focus on the design and development of artifacts [18–21].

Following [22], artifacts can be subdivided into *constructs, models, methods* and *instantiations*. Constructs describe the language and terminology used to formalize information, while models use this terminology to represent problems [23]. Methods describe a problem-solving process [23]. Instantiations are aggregates of the above artifact types, e.g. an implemented tool using constructs and allowing to apply a method [23]. The focus of this work is on the development of constructs, methods and instantiations. Constructs, e.g. formal definitions of conflicts, will be specified as a basis for developing new methods for auditing conflicts. These methods will then be implemented, i.e. instantiated.

Regarding procedural approaches in Design Science research (DSR), there have been various proposals, cf. e.g. [20, 21, 24, 25]. Following [25], a design-science approach can typically be divided into the phases of *problem awareness, suggestion, development, demonstration* and *conclusion*. In the previous sections, we have presented our research problem and solution suggestion, i.e., we suggest to adapt and extend results from the field of inconsistency measurement to support companies in the scope of auditing, in particular in regard to the identified requirements R1-R3. In the following sections, we will present, develop and demonstrate a concrete approach that can satisfy these requirements. The result of this work–an artifact–is then to be seen as a first increment of multiple design science research cycles: In a future work, we will evaluate the artifact developed in this work, e.g., by means of experiments with domain experts. Then, that feedback will be used to iteratively refine the artifact.

4 Advanced Auditing of Run-Time Conflicts

In this section, we present our approach for auditing run-time conflicts.

4.1 Approach Overview

The goal of our proposed approach is to analyze run-time conflicts observed in a series of process instances $\mathcal{P}$. Our approach is divided into three steps, shown in Figure 2.

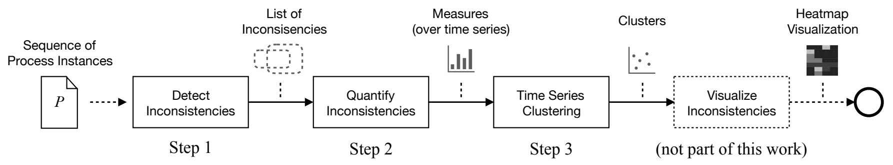


Figure 2. Approach Overview.

Step 1: Detecting Conflicts. Our approach takes as input a series of process instances $\mathcal{P}$. As a first step, we then compute the run-time conflicts for every process instance in $\mathcal{P}$. The result of step 1 is a list of all run-time conflicts.

Step 2: Quantifying Conflicts. In this step, we analyze how often certain conflicts occurred. This is performed from two perspectives (cf. Section 2.4): For every distinct conflict type $\mathcal{I}_i$, we compute a corresponding time series distribution $T_{\mathcal{P}}^{\mathcal{I}_i}$, that represents in which process instances the conflict $\mathcal{I}_i$ occurred in. For every constraint c, we compute a time series $T_{\mathcal{P}}^{c}$, which encodes how many run-time conflicts c was part of over the individual process instances.

Step 3: Time Series Clustering. Next, we cluster the time series computed in step 2. This allows to identify conflicts or constraints that behaved similarly over time.

Outlook: Explaining Conflicts. The previous results should then be visualized to supports modellers in understanding run-time conflicts (not part of this work).

In the following, we detail the individual steps of our approach.

4.2 Step 1: Detecting Conflicts

To compute run-time conflicts in process instances, we exploit the previously introduced notions of reactive constraints [26] and reactive entailment graphs [7]. Importantly, this work focuses on run-time conflicts regarding the constraint types in Table 1 (see below).

For declarative constraints, works such as [3,26] coined the concepts of *activation* and *reaction*, e.g., for RESPONSE(a, b), a can be seen as the activation, and b as the corresponding reaction. For any constraint c, we therefore denote $\alpha(c)$ as the activation of c and $\omega(c)$ as the reaction (also referred to as reactive constraint pair [26]). An overview of activations and reactions in Declare can be found in [3]. The activation- and reaction relations in a declarative process model can then be represented as a directed graph, also referred to as a reactive entailment graph [7].

Definition 5 (Reactive Entailment Graph, [7]). *Given a declarative model $\boldsymbol{M} = (\boldsymbol{A}, \boldsymbol{T}, \boldsymbol{C})$, its reactive entailment graph (REG) is defined as a graph $G(\boldsymbol{M}) = (N, E, \tau)$. N is a set of nodes corresponding to the activities, with $N \in \boldsymbol{A} \cup \overline{\boldsymbol{A}}$ (in two forms, with and without overline symbol, see below discussion). E is the set of directed edges between elements in N, where there is an edge (s, t) in E iff for any $c \in \boldsymbol{C} : s = \alpha(c)$ and $t = \omega(c)$. τ is a function $\tau : E \to \boldsymbol{T}$ that assigns a template type from $\boldsymbol{T}$ to individual edges as an edge type.*

Example 3. We recall the declarative process model M_1 from Figure 1. This yields the following reactive entailment graph, shown in Figure 3:

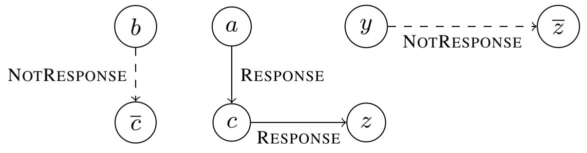


Figure 3. Reactive Entailment Graph (REG) for M_1.

An important detail is that we include two "forms" of nodes (with and without overline symbol), explained as follows. For the constraints RESPONSE(a, c) and NOTRESPONSE(b, c), the *reaction* is c in both cases. However, the nature of the reaction differs. The RESPONSE constraint is of *demanding* nature (the reaction *should* occur), and the NotResponse constraint is of *prohibiting* nature (the reaction should *not* occur). The REG can capture these deontic semantics, where the overlined version of a node corresponds to a prohibition, and vice versa. Based on these two forms of nodes, it can only occur that there is a run-time conflict due to an opposing demand and prohibition, if there exists a pair of nodes n and $\overline{n}$.

Corollary 1. *If there is no pair of nodes n and $\overline{n}$ in REG, there can be no run-time conflict wrt. the introduced constraints (cf. Table 1)*[3].

The REG can then also be used to compute actual run-time conflicts. Importantly, run-time conflicts always arise relative to specific traces. Here, for any trace t, it is easy to observe that any constraint that can never by activated via t cannot be part of that run-time conflict (e.g., for a trace $t = ab$, the constraint NotResponse(y, z) from M_1 cannot be part of the conflict, as y does not occur in the trace, and the constraint cannot be activated by other constraints). This means we can filter out nodes and edges from the REG that correspond to such constraints, allowing for a faster computation of run-time conflicts. To compute the run-time conflicts of a model M w.r.t. a trace t, we thus propose the following algorithm.

1. Relative to trace t, filter out all constraints γ that can never be activated via t.
2. For the remaining constraints, construct the REG.
3. Iterate through all pairs of nodes n, $\overline{n}$:
   (a) For each pair $(n, \overline{n})$, (recursively) search for all direct paths from any node to n (denoted as a set of paths P), and search for all direct paths to $\overline{n}$ (denoted P').
   (b) Every combination $P \times P'$ pertains to a run-time conflict

Algorithm 1. Proposed algorithm for computing run-time conflicts of a model M wrt. a trace t.

Example 4. We recall M_1 and trace $t_1 = ab$. Our proposed algorithm then performs as follows. First (1.), we filter out the constraints from M_1 that can never be activated via t_1. Here, we can consequently drop NotResponse(y, z). Then (2.), the REG is computed for the remaining constraints, shown in Figure 4:

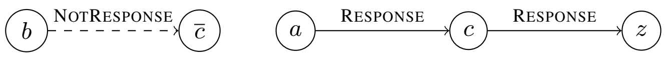


Figure 4. Filtered REG for M_1 w.r.t. t_1.

For the computed graph, we then iterate through all pairs of nodes $n, \overline{n}$ (3.). In this case, these are the nodes c and $\overline{c}$. For this pair, we then search for all direct paths to c, or

[3] We acknowledge there can be other constraint combinations that lead to "dead-ends" in process execution, e.g., if an activity x is expected a certain number of times but there is a constraint limiting the number of occurrences too strictly, however, this is beyond the scope of this report - Corollary 1 is meant to show the relation between the REG and conflicts arising due to opposite deontic "demands".

to $\overline{c}$, respectively. This can be performed by a simple recursive breadth-first search [27]. In result, we obtain the following sets of paths (3a.):

$$P \text{ (paths } \textit{to} \text{ c): } = \{a \rightarrow c\} \qquad P' \text{ (paths } \textit{to } \overline{c}\text{): } = \{b \rightarrow \overline{c}\}$$

Last, via algorithm step 3b.), every combination in $P \times P'$ pertains to a run-time conflict of M_1 w.r.t. t_1. The graph paths can then be transformed back into constraints, i.e., for an edge $e = (\alpha, \omega)$ in the graph, the corresponding Declare constraint is $\tau(e)(\alpha, \omega)$. In result, we have found a set of minimal run-time conflicts, here: $\mathsf{MIN}^{t_1}(M_1) = \{\{\textsc{Response}(a, c), \textsc{NotResponse}(b, c)\}\}$.

As mentioned, if other constraint types as in this work are considered, there can be other types of run-time conflicts, however, we leave this discussion for future work.

4.3 Step 2: Quantifying Conflicts and Computing Time Series

We continue our discussion based on the example from Figure 1. Assume that for all process instances, we have computed the set of run-time conflicts:

$$\begin{aligned} &\mathsf{MIN}^{t_1}(M_1) = \{\mathcal{I}_1\}, \qquad \mathsf{MIN}^{t_2}(M_1) = \{\mathcal{I}_1\}, \qquad \mathsf{MIN}^{t_3}(M_1) = \{\mathcal{I}_1, \mathcal{I}_2\} \\ &\text{with } \mathcal{I}_1 = \{\textsc{Response}(a, c), \textsc{NotResponse}(b, c)\} \\ &\quad \mathcal{I}_2 = \{\textsc{Response}(a, c), \textsc{Response}(c, z), \textsc{NotResponse}(y, z)\} \end{aligned}$$

We can then quantitatively assess the occurred conflicts (cf. also Example 2), and encode this information as time-series, both on a conflict-type level and on a constraint-level (cf. Section 2.4). In the example, denoting $\mathcal{P}_1 = \rho_1, ..., \rho_3$, this yields the following time series, shown in Figure 5.

$$T_{\mathcal{P}_1}^{\mathcal{I}_1} = \langle 1, 1, 1 \rangle \qquad T_{\mathcal{P}_1}^{\mathcal{I}_2} = \langle 0, 0, 1 \rangle$$

(a) Conflict-Level Time Series

$$T_{\mathcal{P}_1}^{\textsc{Response}(a,c)} = \langle 1, 1, 2 \rangle \quad T_{\mathcal{P}_1}^{\textsc{NotResponse}(b,c)} = \langle 1, 1, 1 \rangle$$
$$T_{\mathcal{P}_1}^{\textsc{Response}(c,z)} = \langle 0, 0, 1 \rangle \quad T_{\mathcal{P}_1}^{\textsc{NotResponse}(y,z)} = \langle 0, 0, 1 \rangle$$

(b) Constraint-Level Time Series

Figure 5. Example of the considered types of time-series.

On a conflict-type level, the time-series convey at what points in time the conflicts $\mathcal{I}_1$ and $\mathcal{I}_2$ occurred at. This can be useful for identifying conflicts that occur frequently (such as $\mathcal{I}_1$), or assess whether a conflict was an outlier.

Likewise, the constraint-level time series allow to identify constraints that contribute largely to the occurring conflicts. This can be an important driver for re-modelling. For example, from the shown time-series, it can be seen that $\textsc{Response}(a, c)$ is highly problematic and therefore represents a good candidate for re-modelling.

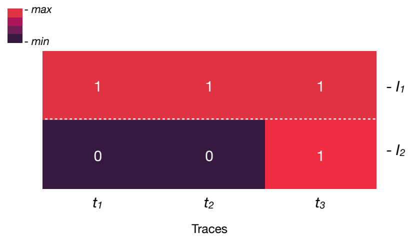
When dealing with large numbers constraints, comparing the time series manually may become unfeasible. Therefore, our approach implements time series clustering in Step 3, to identify groups of highly problematic conflicts and constraints.

4.4 Step 3: Time Series Clustering

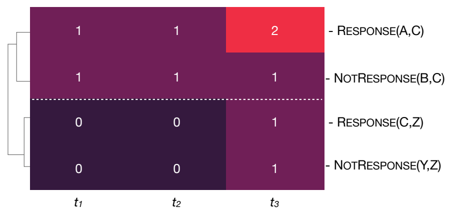
Several time series clustering approaches have been proposed in literature (cf. [15]). However, many of these approaches require a predefined number of clusters (set by the user). As the number of clusters is not known in our setting, we therefore use *hierarchical clustering*, as this method is highly suitable for the problem of clustering a set of time series without knowledge of the number of clusters [12, 15]. The hierarchical clustering algorithm proceeds as follows: For a set of time series $D = \{T_1, ..., T_n\}$, each time series is initialized in an own cluster. First, the distances between between each pair of time series are computed. Then, the two time series (i.e., clusters) with the smallest distance are grouped into one cluster. This is performed iteratively, until the distance between two clusters exceeds a given cutoff value, in which case clustering is stopped.

4.5 Outlook: Explaining Conflicts

To explain the results from steps 1-3, the results should be visualized. While a concrete visualiszation technique is beyond the scope of this work, we show an example how the results from steps 1-3 could be visualized via heatmaps, shown in Figure 6.



(a) Conflict-type heatmap



(b) Constraint-level heatmap

Figure 6. Resulting heatmap visualizations for the data from Figure 1.

The x-axis is the time axis (e.g., every process instance). For the conflict-type heatmaps, every point on the y-axis is a distinct conflict type that occurred. For the constraint heatmaps, every point on the y-axis corresponds to an individual constraint. The heatmaps convey all insights generated by our approach, i.e., the respective quantitative time series, as well as the computed clusters (clusters are separated with a white line; the results of the hierarchical clustering (hierarchies) are indicated with black lines).

Heatmaps allow users to classify how conflicts behaved over time. For example, based on drift type patterns [12], shown in Figure 7, the conflict-type heatmaps allow to identify outliers, or to identify if certain conflicts increased over time. Also, seasonal effects or recurring patterns can be detected.

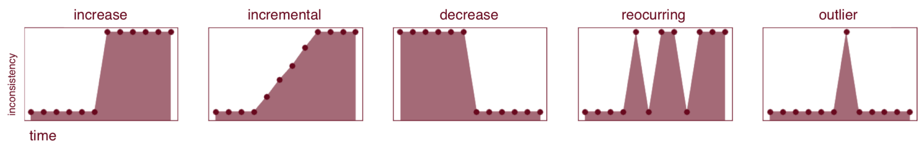


Figure 7. Exemplary drift pattern types (Adapted from [12]).

5 Tool Support and Demonstration

We implemented our approach as a proof-of-concept[4]. Our tool takes as input an event log (set of traces) and a declarative process model (Declare) and can then perform all the steps of our approach. For demonstration, we performed runtime experiments with the real-life data sets of the *Business Process Intelligence (BPI) challenge*. Here, we considered the data-sets of 2017-2020, cf. Table 2.

For every data-set, we mined a set of Declare constraints, using the miner from [3]. The mined constraint sets can be found online[5]. Then, for each mined set and the corresponding log, we applied our approach to compute clusters.

Table 2 shows the runtimes needed to apply our tool to the considered data-sets. As can be seen, even for logs of industrial complexity, e.g. BPI 19, with over 250.000 traces, the tool could perform all introduced steps of our approach in less than 80 minutes.

Table 2. Runtime results for the considered BPI data sets

Data-set	# of Rules	# of Cases	Run-time	# of Run-Time Conflicts
BPI'17	50	31.509	6270s	31.509 (100%)
BPI'18	84	43.809	4542s	0 (0%)
BPI'19	51	251.734	4560s	434 (0.17%)
BPI'20	330	10.500	1620s	323 (3.07%)

6 Conclusion

In this paper, we presented an approach for auditing run-time conflicts in declarative processes by means of time series clustering. The clusters allow experts to understand how different conflicts behaved over time, and provide means for quantitative root-cause analysis, as a basis for identify similar groups of (highly) problematic constraints. Our demonstration shows that our approach can be applied to real-life data-sets in a feasible runtime.

A limitation of our approach is that we assume some form of order of traces. While it is possible to have a partial order of traces (e.g., by starting timestamp), no relation is known between elements of different traces. In our tool, we try to counteract this problem by sorting the traces by average time stamp. In this sense, some concept of "earlier" and "later" traces can be assumed. Also, our approach could be easily extended to calculate conflicts by day, which could yield a time series over absolute scales. A further limitation of our approach is that we consider a trace as the smallest unit of time. In this sense, the causes of run-time conflicts cannot be matched to sub-parts of traces. While our approach can already compute the causes of problems as a set, being able to identify a concrete subtrace may be useful in the scope of drill-down analysis (cf. [12]). In future work, we therefore plan to extend our approach with growing window techniques, to consider time-units of more fine-grained granularity.

As a following work, we will extend our approach as shown in Figure 2 and devise/evaluate a concrete visualization technique. Based on the suggestions in [14], we intend to use heatmaps as a visualization technique.

[4] `https://bit.ly/3xVjY2N`
[5] `https://bit.ly/365Vs4C`

References

1. Maggi, F.M., Montali, M., Westergaard, M., Van Der Aalst, W.M.: Monitoring business constraints with linear temporal logic: An approach based on colored automata. In: International Conference on Business Process Management (BPM 2011). pp. 132–147. Springer (2011)
2. Pesic, M., Schonenberg, H., Van der Aalst, W.M.: Declare: Full support for loosely-structured processes. In: 11th IEEE international enterprise distributed object computing conference (EDOC 2007). pp. 287–287. IEEE (2007)
3. Di Ciccio, C., Maggi, F.M., Montali, M., Mendling, J.: Resolving inconsistencies and redundancies in declarative process models. Inf. Systems 64, 425–446 (2017)
4. Maggi, F.M., Westergaard, M., Montali, M., van der Aalst, W.M.: Runtime verification of ltl-based declarative process models. In: International Conference on Runtime Verification (RV 2011). pp. 131–146. Springer (2011)
5. Bauer, A., Leucker, M., Schallhart, C.: Comparing ltl semantics for runtime verification. Journal of Logic and Computation 20(3), 651–674 (2010)
6. Thimm, M.: Inconsistency measurement. In: International Conference on Scalable Uncertainty Management. pp. 9–23. Springer (2019)
7. Corea, C., Delfmann, P.: Quasi-inconsistency in declarative process models. In: Business Process Management Forum, Vienna, Austria, September 1-6, 2019. Lecture Notes in Business Information Processing, vol. 360, pp. 20–35. Springer (2019)
8. Corea, C., Nagel, S., Mendling, J., Delfmann, P.: Interactive and minimal repair of declarative process models. In: Proceedings of the BPM Forum 2021 co-located with the 19th International Conference on Business Process Management (BPM 2021). Rome (2021)
9. Corea, C., Deisen, M., Delfmann, P.: Resolving inconsistencies in declarative process models based on culpability measurement. In: 15. Internationale Tagung Wirtschaftsinformatik (WI 2019). Siegen (2019)
10. Corea, C., Thimm, M., Delfmann, P.: Measuring inconsistency over sequences of business rule cases. In: Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning (KR 2021). Hanoi (2021)
11. Pufahl, L., Rehse, J.R.: Conformance checking with regulations–a research agenda. In: 11th Int. Workshop on Enterprise Modeling and Information Systems Architectures (2021)
12. Yeshchenko, A., Di Ciccio, C., Mendling, J., Polyvyanyy, A.: Comprehensive process drift detection with visual analytics. In: International Conference on Conceptual Modeling. pp. 119–135. Springer (2019)
13. Yeshchenko, A., Di Ciccio, C., Mendling, J., Polyvyanyy, A.: Visual drift detection for sequence data analysis of business processes. IEEE Transactions on Visualization and Computer Graphics (2021)
14. Ware, C.: Information visualization: perception for design. Elsevier (2004)
15. Aghabozorgi, S., Shirkhorshidi, A.S., Wah, T.Y.: Time-series clustering–a decade review. Information Systems 53, 16–38 (2015)
16. Batista, G.E., Keogh, E.J., Tataw, O.M., De Souza, V.M.: Cid: an efficient complexity-invariant distance for time series. Data Mining and Knowledge Discovery 28(3), 634–669 (2014)
17. Becker, J., Holten, R., Knackstedt, R., Niehaves, B.: Forschungsmethodische positionierung in der wirtschaftsinformatik: epistemologische, ontologische und linguistische leitfragen. Tech. rep., Westfälische Wilhelms-Universität Münster (2003)
18. Baskerville, R., Lyytinen, K., Sambamurthy, V., Straub, D.: A response to the design-oriented information systems research memorandum. European journal of information systems 20(1), 11–15 (2011)
19. Gericke, A., Robert, W.: Entwicklung eines bezugsrahmens fuer konstruktionsforschung und artefaktkonstruktion in der gestaltungsorientierten wirtschaftsinformatik. In: Wissenschaftstheorie und gestaltungsorientierte Wirtschaftsinformatik, pp. 195–210. Springer (2009)

20. Vaishnavi, V., Kuechler, W., Petter, S.: Design science research in information systems. January 20, 2004 (2004)
21. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design science in information systems research. MIS quarterly pp. 75–105 (2004)
22. March, S.T., Smith, G.F.: Design and natural science research on information technology. Decision support systems 15(4), 251–266 (1995)
23. Winter, R.: Design science research in europe. European Journal of Information Systems 17(5), 470–475 (2008)
24. Peffers, K., Tuunanen, T., Rothenberger, M.A., Chatterjee, S.: A design science research methodology for information systems research. Journal of management information systems 24(3), 45–77 (2007)
25. Kuechler, B., Vaishnavi, V.: On theory development in design science research: anatomy of a research project. European Journal of Information Systems 17(5), 489–504 (2008)
26. Cecconi, A., Di Ciccio, C., De Giacomo, G., Mendling, J.: Interestingness of traces in declarative process mining: The janus ltlpf approach. In: International Conference on Business Process Management. pp. 121–138. Springer (2018)
27. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to algorithms. MIT press (2009)