Mining Behavioral Patterns for Conformance Diagnostics

Eduardo Goulart Rocha[1,2(✉)], Sebastiaan J. van Zelst[1], and Wil M. P. van der Aalst[1,2]

[1] Celonis Labs GmbH, Munich, Germany
{e.goulartrocha,s.vanzelst}@celonis.com
[2] Process and Data Science (PADS) Chair, RWTH Aachen University, Aachen, Germany
wvdaalst@pads.rwth-aachen.de

Abstract. Conformance checking is the field of process mining concerned with the monitoring and reporting of discrepancies between event logs and process models. An often overlooked issue is the entry barrier for conformance checking techniques. On the one hand, constraint-checking methods provide intuitive conformance diagnostics, yet require a significant manual effort and expertise from the users to elicit the corresponding constraints. On the other hand, procedurally-oriented techniques, e.g., alignments, provide low-level conformance results that require a significant interpretation effort from the end-user. Therefore, in this paper, we propose to combine the best of both worlds and present an automated method to generate conformance diagnostics in the form of higher-level behavioral patterns, derived from a procedural model. The approach is implemented as a standalone tool and evaluated against real-life datasets, where it is shown to explain nearly all deviations with good scalability.

Keywords: Process Mining · Conformance Checking · Conformance Diagnostics · Declarative Process Mining

1 Introduction

Process mining is the field of computer science combining traditional process science with data science to analyze *event data*. A core task of process mining is *conformance checking*, which consists of comparing desired behavior (i.e., modeled in some process modeling formalism) and observed behavior (i.e., as captured in the event data) to quantify their differences and identify frequent patterns of deviation. In conformance checking, process models may be of a *procedural* or *declarative* nature. Procedural models, e.g., BPMN diagrams [8], describe the exact sequence of steps that are allowed in a process, i.e., a *closed-world assumption*. Declarative models, e.g., DECLARE [25], specify only the constraints upon which a process must operate, i.e., an *open-world assumption*.

A. Marrella et al. (Eds.): BPM 2024, LNCS 14940, pp. 291–308, 2024.
https://doi.org/10.1007/978-3-031-70396-6_17

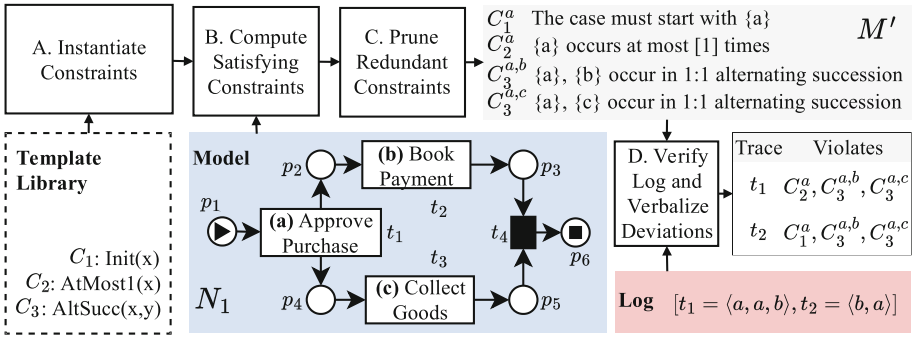

Fig. 1. Overview of the framework steps. The constraint template library contains a set of constraint templates capturing common behavioral patterns. The template library is reusable across users and processes and is populated by the tool provider/vendor, but it can be extended with user-defined patterns.

Table 1. Example log with all optimal alignments and violated constraints. An optimal alignment returns the minimal number of activity insertions (-) and deletions (/) needed to make a trace fit the given model.

Trace	Optimal Alignments	Violated Constraints
$t_1 : \langle a, a, b \rangle$	a / b c; a / c b; a c / b; / a b c; / a c b	$C_2^a, C_3^{a,b}, C_3^{a,c}$
$t_2 : \langle b, a \rangle$	a b / c; a b c /; a c b /; / a b c; / a c b	$C_1^a, C_3^{a,b}, C_3^{a,c}$

Procedural process models are widely available in organizations, e.g., the SAP reference model [14]. Organizations maintain large repositories of procedural models that have been designed for enactment or documentation purposes. Declarative models are used less frequently in practice. While their ease of use is graded positively by practitioners [27], their open-world assumption often yields too flexible or incomplete models.

Both modeling formalisms can be used for conformance checking [1,17]. Conformance checking artifacts based on procedural models, e.g., *trace alignments*, require a significant cognitive effort from the user to be interpreted. To show that, we consider the purchase process from Fig. 1. The process starts with the approval of the purchase (a). After that, the payment must be booked (b) and the goods must be collected (c), which can happen in any order. Only two activity sequences are allowed: $\langle$*Approve Purchase*, *Book Payment*, *Collect Goods*$\rangle$ and $\langle$*Approve Purchase*, *Collect Goods*, *Book Payment*$\rangle$. Even for this simple process, state-of-the-art procedural techniques such as trace alignments produce confusing results. To see that, consider optimal alignments from Table 1.

Trace $t_1 = \langle$*Approve Purchase*, *Approve Purchase*, *Book Payment*$\rangle$, has 2 issues: the purchase is approved twice and the goods are not collected. These are indicated by constraints $C_2^a, C_3^{a,b}$, and $C_3^{a,c}$. Techniques such as trace alignments suggest edits to "fix" these constraints. Removing one *Approve Purchase* (a) fixes C_2^a and $C_3^{a,b}$ and inserting a *Collect Goods* (c) fixes $C_3^{a,c}$, but these edits can occur at difference places of the trace, leading to 5 optimal alignments.

Similarly, trace $t_2 = \langle Book\ Payment, Approve\ Purchase \rangle$ has two issues: the payment is booked before approval and the goods are not collected. These are indicated by the violated constraints C_1^a, $C_3^{a,b}$, and $C_3^{a,c}$. The trace also has five optimal alignments. It can be "fixed" by either moving a to an earlier point $\boxed{a} \cdots \boxed{\not a}$ (fixes C_1^a and $C_3^{a,b}$) and inserting $\boxed{c}$ (fixes $C_3^{a,c}$), as in the first three alignments; or moving b to a later point $\boxed{\not b} \cdots \boxed{b}$ (fixes C_1^a and $C_3^{a,b}$) and inserting $\boxed{c}$ (fixes $C_3^{a,c}$), as in the last two alignments.

In the examples above, multiple edit operations might be related to the fix of a single constraint. Similarly, multiple constraints might be fixed by a single edit operation. Furthermore, the same edit type might mean different things depending on the context, e.g. the deletion $\boxed{\not a}$ for t_1 and t_2. Consequently, the only way of obtaining insights on the patterns of deviations is by manually inspecting each trace. In summary, the diagnostics provided by trace-alignments are low-level, non-deterministic, and its interpretation depends on context. In contrast, the equivalent declarative model is able to directly explain the deviations in a more understandable form by pointing out the violated business constraints.

Therefore, while on the one hand procedural models are often preferred by practitioners, the conformance diagnostics returned by declarative models are better suitable for direct interpretation. To bridge this gap, this paper proposes a novel conformance checking framework that generates declarative constraints from a procedural input model, which are subsequently used to present conformance diagnostics in a declarative fashion.

Figure 1 presents a schematic overview of our proposed framework. As input, the user provides a *procedural model* and an *event log*. The tool internally maintains a *constraint template library* consisting of high-level descriptions of constraints, i.e., at the meta-level, to be checked against the reference model. The procedural model is used to compute constraint instances based on the template library (Steps A and B). For example, assume that the library contains the template `Init(x)`. If the procedural model specifies that some activity, e.g., activity *'a'*, should always be executed first, the constraint `Init(a)` is instantiated. The result is an "equivalent" declarative version of the originally procedural model. Then, techniques from the field of declarative process mining are used to prune redundant constraints (C) and verify the log to provide diagnostics (D).

Using the discovered constraints to analyze the log ensures that the returned diagnostics are on a higher level, and, thus, more understandable than other automated techniques. Furthermore, the approach presents a series of interesting properties such as *determinism*, *monotonicity* of the reported deviations, and *flexibility* to add new patterns. The approach is implemented as a standalone tool and evaluated on two real-life datasets, where it is shown to be *scalable* and to generate *nearly complete diagnostics*. Our main contributions are:

1. We present the first framework to generate declarative-like conformance diagnostics on top of procedural models
2. We provide a method for the efficient verification of a subclass of constraint patterns, ensuring the scalability of the approach

The remaining of this paper is organized as follows: Sect. 2 introduces basic definitions, Sect. 3 formalizes each step of the framework. Sections 4 and 5 present the tool and evaluates the approach, Sect. 6 presents related work. Finally, Sect. 7 concludes the paper with directions for future work.

2 Preliminaries

This section fixes basic mathematical notations. We assume familiarity with automata [10] and Petri net [5] theories. We use $\mathcal{U}_a$ to denote the universe of activity labels and $\tau \notin \mathcal{U}_a$ to denote the invisible (empty) label. A *trace* is a finite sequence of activities $\sigma \in \mathcal{U}_a^*$. An *event log* is a multi-set of traces $L \in \mathcal{B}(\mathcal{U}_a^*)$ describing the sequences of observed activities and their frequencies. Similarly, a process model M is a set $M \in \mathcal{P}(\mathcal{U}_a^*)$ describing the set of allowed variants. For this work, we consider models defined as Petri nets and note that BPMN diagrams can be converted into Petri nets.

Definition 1 ***(Labeled Accepting Petri Net).*** *A labeled accepting Petri net is a sextuple $N = (P, T, F, l, m_0, m_f)$ where P is the set of places, T the set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation, $l : T \to \mathcal{U}_a \cup \{\tau\}$ is the labeling function, and $m_0, m_f \in \mathcal{B}(P)$ are its initial and final markings. Transitions t for which $l(t) = \tau$ are called invisible transitions.*

The state of a labeled accepting Petri net $N = (P, T, F, l, m_0, m_f)$, called *marking*, is a multiset of places $m \in \mathcal{B}(P)$. It is possible to move from one state into another by firing enabled transitions using the occurrence rule [10]. For a firing sequence $\sigma = \langle t_1, \cdots, t_n \rangle \in T^*$, we write $m[\sigma\rangle m'$ to denote that σ is enabled at $m \in \mathcal{B}(P)$ and that firing transitions in σ in sequence leads to $m' \in \mathcal{B}(P)$. The *state space* of a Petri net is the set $\mathcal{R}(N) = \{m' | \sigma \in T^*, m_0[\sigma\rangle m'\}$ of states reachable by firing any enabled firing sequence and its *accepted language* is the set $\mathcal{L}(N) = \{l(\sigma) \mid \sigma \in T^*,\ m_0[\sigma\rangle m_f\}$, where $l(\sigma) \in \mathcal{U}_a^*$ is the concatenation of the labels of transitions in σ.

Figure 1 shows a Petri net N accepting the language $\{\langle a, b, c\rangle, \langle a, c, b\rangle\}$ with $|\mathcal{R}(N)| = 6$. This work focuses on the class of *regular languages*, which is the class of languages that can be recognized by a *Deterministic Finite Automaton*.

Definition 2 ***(Deterministic Finite Automaton (DFA)).*** *A Deterministic Finite Automaton (DFA) is a quintuple $D = (Q, \Sigma, \delta, q_0, F)$ where Q is the set of states, $\Sigma \subseteq \mathcal{U}_a$ is the alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, q_0 is the initial state and $F \subseteq Q$ is the set of final states.*

A trace $\sigma = \langle a_1, \cdots, a_n \rangle \in \mathcal{U}_a^*$ is accepted by a DFA $D = (Q, \Sigma, \delta, q_0, F)$ if there exists a sequence of states $\langle s_0, \cdots, s_n \rangle$ s.t. $s_0 = q_0$, $\forall_{1 \leq i \leq n} q_i = \delta(q_{i-1}, a_i)$ and $s_n \in F$. We call the set $\mathcal{L}(D) = \{\sigma \in \mathcal{U}_a^* \mid \sigma \text{ is accepted by } D\}$ its language. In general, for a Petri net N, if $|\mathcal{R}(N)| < \infty$, i.e., its state space is finite, then $\mathcal{L}(N)$ can be expressed by a DFA called its *behavioral automaton*.

Definition 3 ***(Behavioral Automaton).*** *The behavioral automaton of a bounded labeled accepting Petri net $N = (P, T, F, l, m_0, m_f)$ is the unique (up to state renaming) minimal DFA $B_N = (Q, \Sigma, \delta, q_0, F)$ such that $\mathcal{L}(N) = \mathcal{L}(B_N)$.*

3 Mining Behavioral Patterns

This section details the framework steps from Fig. 1. For the remainder, we consider $\mathcal{U}_a = \{a, b, c\}$ and the event log $L_1 = [\langle a, a, b\rangle, \langle b, a\rangle]$. We first formalize the concepts of *constraint*, *constraint template* and *constraint template library*:

Definition 4 ***(Constraint).*** *A constraint c is an object describing a language $\mathcal{L}(c) \subseteq \mathcal{U}_a^*$. A trace $\sigma \in \mathcal{U}_a^*$ satisfies c if $\sigma \in \mathcal{L}(c)$. We denote the universe of all constraints as $\mathcal{U}_C$.*

The core idea of this work is centered around the automatic instantiation of constraints from a set of constraint templates (called a template library).

Definition 5 ***(Constraint Template).*** *A constraint template is a function $C : \mathcal{U}_a^k \to \mathcal{U}_C$ returning a constraint for each k-tuple of activities $(a_1, \cdots, a_k) \in \mathcal{U}_a^k$, where k is the number of template parameters. We write $C^{a_1,\cdots,a_k} = C(a_1, \cdots, a_k)$.*

Definition 6 ***(Constraint Template Library).*** *Let $\mathcal{C}$ be the universe of constraint templates. A template library $\mathbb{L}$, is a set of templates $\mathbb{L} \subseteq \mathcal{C}$.*

This work focuses on models expressed as bounded Petri nets and constraints expressing regular languages. The DECLARE [25] language is an example of a set of constraints that can be expressed using regular expressions. Table 2 shows an example constraint template library containing three constraint templates $\mathbb{L}_1 = \{C_1, C_2, C_3\}$ (in practice, more extensive libraries are used). Replacing the template parameters x, y with concrete activities yields a constraint. For example, $C_3(a, b)$ returns $C_3^{a,b} \in \mathcal{U}_C$ describing all traces where a and b occur in one-to-one succession. Furthermore, each constraint is associated to a verbalization describing it in human-understandable way.

3.1 Step A: Instantiating Constraints

The first step of the framework is to instantiate candidate constraints from the template library following an *instantiation strategy*. The most straightforward strategy is the brute force instantiation of all templates using all parameter combinations, but notice that more sophisticated instantiation strategies are also possible. For $\mathcal{U}_a = \{a, b, c\}$ and $\mathbb{L}_1 = \{C_1, C_2, C_3\}$, the brute force instantiation yields the set of constraints $Inst = \{C_1^a, \cdots, C_1^c, C_2^a, \cdots, C_2^c, C_3^{a,b}, \cdots, C_3^{c,b}\}$.

3.2 Step B: Computing Satisfied Constraints

From the set of instantiated constraints in the previous step, we are interested in the constraints that are *satisfied* by the model. A constraint is satisfied by a process model if and only if the constraint holds for all traces in the model's accepted language. For the Petri net of Fig. 1 and the instantiated set *Inst*, the set of satisfied constraints is $Sat = \{C_1^a, C_2^a, C_2^b, C_2^c, C_3^{a,b}, C_3^{a,c}\}$.

Table 2. Example constraint template library $\mathbb{L}_1 = \{C_1, C_2, C_3\}$ with templates expressed as regular expressions. x, y denote the constraint parameters and k the number of template parameters.

	Template	k	Verbalization	RegEx
C_1	Init(x)	1	The case must start with $\{x\}$	x.*
C_2	AtMost1(x)	1	$\{x\}$ occurs at most [1] times	(!x)*x(!x)*
C_3	AltSucc(x, y)	2	$\{x\}$, $\{y\}$ occur in 1:1 alternating succession	(!(x|y)|(x(!(x|y))*y))*

Definition 7 ***(Satisfying Constraint).*** *Let $c \in \mathcal{U}_C$ be a constraint and let M be a process model. c is said to be satisfied by $M \iff \mathcal{L}(M) \subseteq \mathcal{L}(c)$.*

For a bounded Petri net N with behavioral automaton B_N and constraint c expressed as a regular expression with associated minimal DFA B_c, it holds that $\mathcal{L}(N) \subseteq \mathcal{L}(c) \iff \mathcal{L}(B_N) \subseteq \mathcal{L}(B_c)$. Checking this condition is in $\mathcal{O}(|B_{SN}| * |B_c| * |\mathcal{U}_a|)$ [10]. Therefore, one can check if a constraint is satisfied by a model as long as the model and the constraint's corresponding DFAs can be obtained.

3.3 Scalable Conformance Diagnostics with Γ-Invariant Constraints

As discussed in the previous section, it is possible to check if a constraint is satisfied by computing the Petri net's behavioral automaton. This becomes expensive for large Petri nets due to the state explosion problem. In this section, we introduce a class of constraints for which constraint checking is significantly accelerated. The method consists of reducing the Petri net's state space by pruning irrelevant activities. We start with the definition of Γ-invariance for a constraint's language, which intuitively means that modifying a trace by inserting or removing activities in Γ does not influence language inclusion.

Definition 8 ***(Γ-Invariance).*** *Let $c \in \mathcal{U}_C$ and let $\Gamma \subseteq \mathcal{U}_a$. c is Γ-invariant if for any $\sigma, \hat{\sigma} \in \mathcal{U}_a^*$ s.t. $\sigma{\restriction}_{\mathcal{U}_a \setminus \Gamma} = \hat{\sigma}{\restriction}_{\mathcal{U}_a \setminus \Gamma}$, it holds that $\sigma \in \mathcal{L}(c) \iff \hat{\sigma} \in \mathcal{L}(c)$.*

Γ-invariance is closely related to the notion of stutter-invariance and next-free formulas in model checking [24]. In DECLARE, Γ-invariant constraints are essentially constraints that do not express a directly-follows relation. In our running example, constraint C_2^a is $\{b, c\}$-invariant, i.e., inserting or removing activities b and c to a trace does not change whether C_2^a holds for that trace. Meanwhile, C_1^a is not Γ-invariant for any $\Gamma \subseteq \mathcal{U}_a$. Lemma 1 below shows that one can compute the maximal set Γ for which a constraint c is Γ-invariant by looking into the self-loops of c's corresponding minimal DFA.

Lemma 1 ***(Γ-Invariance and Self Loops).*** *Let $c \in \mathcal{U}_C$ be a constraint and let $D_c = (Q, \mathcal{U}_a, \delta, q_0, F)$ be the unique minimal DFA such that $\mathcal{L}(c) = \mathcal{L}(D)$. c is Γ-invariant $\iff \forall q \in Q, \gamma \in \Gamma : \delta(q, \gamma) = q$.*

Proof. The $\Leftarrow$ direction is straightforward to see. We prove the $\Rightarrow$ direction by contradiction. Assuming that c is Γ-invariant but $\exists q, q' \in Q, \gamma \in \Gamma, q \neq q'$ s.t. $\delta(q,\gamma) = q'$. Let $x \in \mathcal{U}_a^*$ be a prefix leading to state q. Then $x\gamma$ leads to state q'. From the minimality of D_c, it follows that q and q' correspond to different equivalence classes of the Nerode equivalence. Therefore, there exists a suffix $z \in \mathcal{U}_a^*$ such that either $xz \in \mathcal{L}(D_c)$ and $x\gamma z \notin \mathcal{L}(D_c)$ or $xz \notin \mathcal{L}(D_c)$ and $x\gamma z \in \mathcal{L}(D_c)$. But this contradicts Γ-invariance. □

Definition 9 ***(Projecting Petri nets).*** *Let $N = (P, T, F, l, m_0, m_f)$ be a labeled accepting Petri net and $A \subseteq \mathcal{U}_a$ a set of activities. The projection of N into A is the net $N_A = (P, T, F, l_A, m_0, m_f)$ with* $l_A(t) = \begin{cases} l(t) \text{ if } l(t) \in A \\ \tau \text{ otherwise} \end{cases}$.

It holds that $\mathcal{L}(N_A) = \{\sigma\restriction_A \mid \sigma \in \mathcal{L}(N)\}$, i.e., the accepted language of the projected Petri net is the projection of the language of the original net in A. Furthermore, oftentimes $B_{N_A} \ll B_N$. We can use this fact to speed up the validity check of Γ-invariant constraints using the lemma below.

Lemma 2 ***(Validity Check for Γ-invariant Constraints).*** *Let $c \in \mathcal{U}_C$ be Γ-invariant and N a Petri net. Then* $\mathcal{L}(N) \subseteq \mathcal{L}(c) \iff \mathcal{L}(N_{\mathcal{U}_a \setminus \Gamma}) \subseteq \mathcal{L}(c)$

Proof. $\Rightarrow$ follows from $\mathcal{L}(N_{\mathcal{U}_a \setminus \Gamma}) \subseteq \mathcal{L}(N)$. $\Leftarrow$ follows from Γ-invariance of c. □

Lemma 2 means that we can check if a Γ-invariant constraint c is satisfied by a Petri net N by checking if it is satisfied by $N_{\mathcal{U}_a \setminus \Gamma}$. For the template library of Table 2, constraint C_2^a is $\{b, c\}$-invariant. By projecting Petri net N_1 from Fig. 1 into $\{a\}$, we obtain the Petri net $N_{1_{\{a\}}}$ from Fig. 2. By applying language preserving reduction rules [23], we obtain the reduced net $N'_{1_{\{a\}}}$. C_2^a is satisfied by $N'_{1_{\{a\}}}$. Therefore, it follows from Lemma 2 that it is also satisfied by N_1. Notice that $|\mathcal{R}(N_1)| = 6$, while $|\mathcal{R}(N'_{1_{\{a\}}})| = 2$. This reduction in the state space size can lead to dramatic speedups as will be shown in Sect. 5.1.

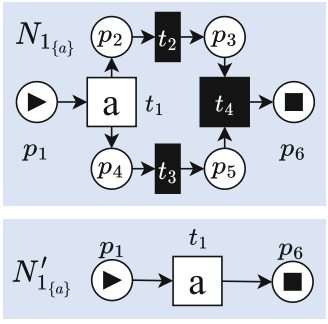

Fig. 2. Projected and reduced nets $N_{1_{\{a\}}}$ and $N'_{1_{\{a\}}}$.

3.4 Step C: Pruning Redundant Constraints

The result of the previous step is the set of satisfied constraints for the given model. However, some constraints are redundant. For the satisfied constraints *Sat*, C_2^a ($\{a\}$ occurs at most [1] times) and $C_3^{a,b}$ ($\{a\}$, $\{b\}$ occur in 1:1 alternating succession) imply C_2^b ($\{b\}$ occurs at most [1] times). Redundant constraints produce redundant diagnostics. Therefore, we add a redundancy resolution step.

Definition 10 ***(Redundant Constraints).*** *Let $C \subseteq \mathcal{U}_C$. A constraint c is redundant in C if $\exists\ C' \subseteq C \setminus \{c\}$ s.t. $\left(\bigcap_{c'_i \in C'} \mathcal{L}(c'_i)\right) \subseteq \mathcal{L}(c)$. We write $C' \rightarrow c$.*

Algorithm 1: An approximate model minimization algorithm.

input A set of constraints C, a total ordering criterion $<_c$, a maximum intersection size m
output A set of constraints $C' \subseteq C$ s.t. $C' \Rightarrow C$
$C' \leftarrow C$;
foreach $(c_1, c_2, \cdots, c_m, c_{m+1}) \in C^{m+1}$ s.t. $c_i <_c c_j \ \forall 1 \leq i < j \leq m+1$ **do**
  if $\{c_1, \cdots, c_m\} \rightarrow c_{m+1}$ **then** $C' \leftarrow C' \backslash \{c_{m+1}\}$;
return C'

We are interested in a minimal set of redundancy-free constraints (in our example, $M' = \{C_1^a, C_2^a, C_3^{a,b}, C_3^{a,c}\}$ is such a set). Formally, for $C \subseteq \mathcal{U}_C$, we want to solve the optimization $min_{C' \subseteq C} |C'| \ s.t. \ \forall c \in C, \ C' \rightarrow c$. In the declarative process mining literature, there exist approaches to prune redundant constraints [6,13]. However, existing techniques require the intersection of all constraints' DFAs, which is in $\mathcal{O}\left(\prod_{c \in C} |B_c|\right)$. Instead, to ensure the scalability of the pipeline, we use an approximated approach specified in Algorithm 1. The approach orders all constraints according to a predefined order [6][1] and checks all combinations $(c_1, c_2, \cdots, c_m, c_{m+1}) \in C^{m+1}$ s.t. $c_i <_c c_j$ for all $1 \leq i < j \leq m+1$. If c_{m+1} is found to be redundant, it is removed. The algorithm's correctness follows directly from the fact that we use a total ordering criterion, which ensures that we do not introduce cycles of constraint implications. Furthermore, its complexity is in $\mathcal{O}\left((\max_{\{c \in C\}} |B_c|^{m+1}) * |C|^{m+1}\right)$, i.e. a polynomial with exponent m in the maximum constraint DFA size and the number of constraints.

3.5 Step D: Verifying the Log and Verbalizing Deviations

The previous step produces a set of redundancy-free constraints that must be satisfied by all traces in the model. If a trace violates a constraint, the trace is also deviant. Hence, we can use the discovered constraints as a way to provide higher-level diagnostics to the user and use the constraint's associated verbalization to explain the trace's deviation. For that, standard constraint monitoring techniques can be leveraged such as replay-based or alignment-based [17]. As discussed in Sect. 1, using the set of discovered constraints leads to more understandable diagnostics.

3.6 Runtime and Properties

If a trace violates a constraint c, then it must also violate the model (since $\mathcal{L}(M) \subseteq \mathcal{L}(c)$). Therefore, the approach produces only "correct" diagnostics. Also, given the same template library, the computed set of satisfied constraints

[1] The order is specific to the template library and hence configured by the tool provider/vendor.

is always the same and only depends on $\mathcal{L}(M)$. Therefore, the approach is deterministic. Moreover, given models M_1 and M_2, with satisfied constraints Sat_{M_1} and Sat_{M_2}, it holds that $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2) \Rightarrow Sat_{M_2} \subseteq Sat_{M_1}$, i.e., *monotonicity* holds for the provided set of diagnostics. This last property ensures that the diagnostics are consistent as the user edits the model. Adding/removing behavior to a model can only prune/extend the set of reported violations. These properties do not hold after the pruning step. However, this is not harmful as the set of satisfied constraints can be logically inferred from the set of minimal constraints.

The method instantiates $\mathcal{O}(|\mathcal{U}_a|^k)$ constraints. The validity check depends on the size of the model's behavioral automaton, which is worst-case exponential. However, for Γ-invariant constraints the state-space can be reduced to make it manageable in practice. The pruning step performs $\mathcal{O}(|\mathcal{U}_a|^{mk+k})$ DFA operations. By limiting the maximum size of a constraint's DFA, we can ensure a constant time for this operation, therefore the pruning step can be made in $\mathcal{O}(|\mathcal{U}_a|^{mk+k})$ too. Last, checking the constraints is in $\mathcal{O}(|E| * |\mathcal{U}_a|^k)$ where $|E|$ is the number of events in the log. At first, the exponents look too high, but as shown by the experiments in Sect. 5.1 they are tractable in practice.

4 Implementation and Qualitative Evaluation

The proposed framework is implemented as a web app using the PM4Py [4] and automata-lib [11] libraries for the backend (both written in pure Python)[2] Fig. 3 shows an overview of its UI. On the sidebar, the user can set the framework's parameters. The tool displays the model, the set of discovered constraints, and the log. Furthermore, it has a trace explorer that allows the user to drill down on single traces of the log. As a template library, we implement the set of control-flow behavioral patterns presented in [26], which includes all DECLARE [25] templates as well as templates with more than two parameters. For the minimization step, we adapt the DECLARE constraint ordering criteria proposed in [6] to this library.

We demonstrate how the framework can be used to gain insights into the well-known Italian Road Fines [18] process. We extract the normative description provided in [20]. The process starts with the creation of a fine. In at most 90 days, the fine is sent to the offender's address. The date on which the offender is notified is inserted into the system. From the moment of notification, the offender has 60 days to appeal to the judge or the prefecture, which might dismiss the case. After these 60-days, no further appeal is possible and the offender must pay the fine. If the payment takes too long, a penalty is added to the total amount. Payments are possible at any moment and the case can also terminate at any point if the owned amount is fully paid. Otherwise, if it takes too long, the process also terminates with the case being sent for credit collection. The entire case must be completed within a year of the fine creation.

The original model from [20] is designed as a data Petri net. Since our approach can only handle control flow, we enrich the original event log with events to

[2] Evaluation code is available at https://zenodo.org/doi/10.5281/zenodo.10824798.

Fig. 3. Overview of the user interface, consisting of 5 main components (modified to better fit the printed version).

capture temporal marks (90 Days After Creation, 1 year After Creation, and 60 Days After Notification) and data attributes such as dismissal type (Dismissed by Judge and Dismissed by Prefecture) and payment status (Partial Payment and Full Payment). A reference BPMN model is drawn from the process description (model view in Fig. 3). For the alignment cost function, we assign very high costs to the removal and addition of temporal marks. Since these are not proper process steps, we do not want to allow them to be edited. The data is loaded into the implemented tool. We consider a max parameter of two, and the proposed "approximated" constraint minimization method from Sect. 3.4.

The most frequently violated constraint is **65:** *Each {Add Penalty} must be preceded by {60 Days After Notification}*, occurring in 40% of the cases. The constraint description suggests that penalties are being added too early. To drill down on that, the tool provides a set of frequently co-occurring constraints that positively correlate (i.e., lift >1) with this violation. Among them, constraint **60:***Each {Send fine} must be followed by a {90 Days After Creation}* suggests a causal relation. If fines are sent too late, then the penalty is also added earlier than it should. This is plausible since the time to add a penalty is computed considering the time elapsed since the fine creation. So a delay in sending the fine also causes the penalty to be added too early. This suggests that sending the fines more speedily will also improve other areas of conformance.

Another interesting violation refers to constraint **3:***{Full Payment} occurs at most [1] times.* The enriched log distinguishes between Partial and Full payments, so repeated full payments can only occur in situations where the outstanding amount is fully paid, but further payments are still effectuated. This only happens for 5% of the cases but is arguably a serious problem. To understand its root cause, we again look into the constraints that co-occur with it and see that constraint **117:***{Full Payment} cannot be followed by {Add Penalty}* has a strong correlation with it (lift > 10). Constraints 3 and 117 are violated in only 5% and 2.5% of the cases, but constraint 117 is violated with a 50% chance if 3 is also violated. This suggests that double payments mostly happen to pay the added penalty. A possible root cause for that are communication delays between the agents adding the penalty and processing the payment.

Overall, this analysis shows how the high-level diagnostics combined with understandable constraint descriptions help the analyst to make sense of deviations and formulate potential root causes and improvement areas. In principle, similar insights could be obtained by following the methodology proposed in [26]. However, the analyst would need to guess and manually configure each pattern of interest. By mining the constraints from the process model, our approach saves the analyst significant time. To the best of our knowledge, [12] is the only existing work that generates natural-language diagnostics from procedural models. The approach works by matching a series of hard-coded patterns in the synchronized error-correcting product of the event log and process model.

Table 3 shows a snippet of the behavioral statements produced by [12]. While our approach produces 143 constraints (out of which only 76 are violated), [12] produces 703 distinct behavioral statements. The log contains 995 violating variants, i.e. there is almost one unique statement for every violating variant. The

Table 3. Snippet of diagnostics produced by [12] (simplified to fit the table) for the enriched road fines dataset. In total, 703 statements are produced.

ID	Statement
S1	{Creation + 60 Days } occurs after {Add Penalty} and before {Creation + 1 year}
S2	After {Partial Payment}, {Full Payment} is repeated
S3	After {Send Fine}, {Full Payment} is repeated
S4	{Add Penalty} occurs after {Full Payment} and before {Creation + 60 Days }

high number of distinct statements make it difficult for users to understand common deviation patterns. Many statements are slight variations of each other, e.g. statements **S2** and **S3** only differ in their "reference point". The tool from [12] does not provide the frequency of each violation and does correlate violations, making it difficult to drill down through them. Among the returned set of behavioral statements, **S1** is equivalent to constraint **65** reported by our tool (unfortunately, we could not find an equivalent statement to constraint **60**). Similarly, constraints **3** and **117** are (indirectly) reflected by statements **S2**/**S3** and **S4**.

In summary, the method described in [12] requires more extensive drill-down efforts (due to the higher number of violations) to obtain similar insights. Furthermore, our approach differs from [12] in three key ways: First, we offer a flexible method for defining new patterns; second, we can explain deviations caused by long-term dependencies, e.g. via the "AlternatingSuccession" template; last, we provide better runtime guarantees. A merit of [12] is to identify precision issues, i.e. behaviors allowed by the model but absent in the log. This could be achieved in our framework by "swapping" the roles of the log and the model.

5 Quantitative Evaluation

This section evaluates the approach for its scalability and completeness. We compare two scenarios: considering all constraints (ALL) and considering only Γ-invariant constraints (Γ). For each scenario, we vary the maximum number of template parameters max_k from 2 to 4. The framework is evaluated on two real-world datasets: The enriched Italian road fines dataset (RF) described in Sect. 4 and the BPI Challenge 2015 event log [9] (BPI-15) with a model discovered with the Inductive Miner-infrequent variant [15] with a noise threshold of 0.9. We filter the BPI-15 log for municipality 1, and subprocess 8, and remove repeated activities. This is needed as otherwise the used process discovery method would only return flower constructs. The models are of small to middle size. Both contain 20 transitions and 21 resp. 561 states in their behavioral automata. The experiments are run single-threaded on a Ubuntu 22.04 notebook with an Intel Xeon E-2276M processor and 32 GB of main memory.

5.1 Scalability and Pruning Efficiency

This experiment evaluates the scalability of the approach and the effectiveness of the pruning step. The results are summarized in Table 4. For both datasets,

Table 4. Evaluation of constraint check and minimization steps measuring the number of instantiated constraints (#inst), and the number of satisfied/minimal constraints (#sat/#min) and the time in seconds to compute them (t_{sat}/t_{min}).

DS	max_k	ALL Templates					Γ-Invariant Templates				
		$\#inst$	$t_{sat}(s)$	$\#sat$	$t_{min}(s)$	$\#min$	$\#inst$	$t_{sat}(s)$	$\#sat$	$t_{min}(s)$	$\#min$
RF	2	8531	2.58	1295	11.8	227	3386	0.75	511	7.9	143
	3	50075	17.0	19653	87.7	223	13772	3.74	1244	13.2	145
	4	683315	285	336991	1699	454	140420	46.4	21927	830	468
BPI-15	2	9442	16.8	444	6.02	242	3763	0.75	148	1.31	109
	3	49042	71.3	11147	83.4	1112	13663	3.45	170	1.38	109
	4	555922	1328	221355	3281	1953	115039	35.7	9632	1075	555

increasing the maximum number of template parameters max_k from 3 to 4 leads to a significant increase in the number of instantiated constraints (#inst), which causes a significant increase in the total runtime for the satisfiability check (t_{sat}). Nevertheless, increasing max_k only slightly increases the number of satisfied constraints. For the road fines dataset, the average time to check if a constraint is satisfied ($= t_{sat}/\#inst$) is approximately the same for both scenarios (ALL and Γ). For the BPI-15 dataset, the check of each constraint takes significantly longer for the ALL than for the Γ scenario. This is explained by its larger state space, for which the techniques presented in Sect. 3.3 pay off.

In all experiment setups, the constraint minimization step takes a significant portion of the total time, with up to an hour for BPI-15 when $max_k = 4$. This might sound high, but in comparison, the method proposed in [6] goes out of memory for certain scenarios. Still, the approximated minimization step is very effective at pruning redundant constraints, pruning 99.999% of the satisfied constraints for the road fines dataset in the ALL scenario with $max_k = 4$. Finally, when considering the absolute runtime numbers, all experimental setups could be computed within two hours, with the majority of them being computed within two minutes. Last, notice that the approach is implemented in pure Python. A more careful implementation in a compiled language is expected to bring at least an order of magnitude of improvement, making the approach applicable to interactive scenarios.

5.2 Completeness and Redundancy

We measure the completeness of the produced diagnostics, i.e., the share of deviations that can be explained, and the amount of redundancy of the returned diagnostics. For each scenario, we align the event log and the process model and pinpoint constraints to model/log moves using the method presented in [17]. Constraints associated with a move *explain* the move, i.e., the constraints justify the insertion/deletion at the given position. Table 5 summarizes the results.

On the trace level, most non-conforming traces violate at least one discovered constraint. On the alignment move level, the discovered constraints explain most

Table 5. Completeness and redundancy results. *dev* and *mov* are the numbers of deviant variants and non-synchronous moves. We report the number of traces violating at least one constraint (*det*), the number of moves explained by at least one constraint (*expl*), and the average number of constraints per move (*avg*)

DS		*dev*	*det*			*mov*	*expl*			*avg*		
			k=2	k=3	k=4		k=2	k=3	k=4	k=2	k=3	k=4
RF	ALL	995	994	995	995	4489	4448	4489	4489	3.79	4.34	9.47
	Γ		992	995	995		3914	4238	4238	3.49	3.88	8.76
RF	$\Gamma+$	995	992	995	995	4489	3914	4238	4489	3.42	3.97	8.14
BPI-15	ALL	70	70	70	70	110	108	108	108	2.34	2.46	7.30
	Γ		70	70	70		108	108	108	2.17	2.17	5.86

of the moves, with the road fines dataset performing the worst with 13% of its moves unexplained for the Γ scenario with $k = 2$. The experiment also shows that by using ALL constraints or increasing the maximum number of template parameters, more moves can be explained.

Finally, Table 5 also shows the average number of constraints explaining each alignment move. The method must report as few constraints per move as possible, to not overwhelm the user. For the BPI-15 dataset, in all but two scenarios, the average number of constraints per alignment move is below three. Furthermore, we observe only a slight variation in the number of constraints per move when varying max_k. In contrast, for the road fines dataset, all scenarios report over three constraints per move, with a peak of 9 constraints per move for the ALL scenario with $max_k = 4$.

In summary, the experiments show the scalability of the method, with Γ-invariant constraints being particularly efficient for models with large state spaces. This increased efficiency of Γ-invariant constraints comes at the cost of a slight decrease in completeness. By increasing the maximum number of constraint parameters, more deviations can be explained, but this comes at the expense of an increase in the total runtime. This exposes the tradeoff between scalability, completeness, and redundancy, which the user can select.

Enriching the Template Library. As shown in the section above, the discovered constraints provide nearly complete diagnostics. Still, some violations cannot be explained by any constraint. One example is the alignment displayed in Fig. 3. No constraint explains the insertion of Send for Credit Collection. However, the template library can be easily extended to become more complete.

By analyzing the model, one notices that the insertion happens because at least one of the activities "Full Payment", "Send for Credit Collection", "Dismissed by Judge" or "Dismissed by Prefecture" must occur. By adding the constraint template $.^*(\alpha_1|\alpha_2|\alpha_3|\alpha_4).^*$ to the library, meaning *"either* $\alpha_1, \alpha_2, \alpha_3$*, or* α_4 *must happen in the case"*, all moves are explained (see $\Gamma+$ in Table 5). Of course, this

does not mean that the library is complete, but shows that it can be easily extended to capture new patterns.

6 Related Work

The most basic form of conformance diagnostics consists of quantifying behavioral differences between process models and event logs in terms of a single metric [1,16,22,28,29]. Some methods go beyond that to provide insights into the nature of deviations. In [28] and [29], diagnostics are provided as a set of differences in the behavioral profiles of the log and the model. Techniques based on alignments [1] provide diagnostics as a set of moves (insertions or deletions) on a trace, corresponding to undesired or missing behavior. In [22], decomposition techniques are used to detect regions of the model that are non-conforming. Similarly, [16] detects problematic subsets of activities by comparing log and model projections. These techniques offer too low-level feedback, often limited to a set of activities/edits, and do not effectively explain the deviations.

The issue with low-level feedback has been known for long. In [2], the authors of the alignment technique present a method to detect higher-level deviation patterns via high-level alignments. However, this requires users to specify each pattern to be detected, which is a similar drawback as in [26] (discussed in Sect. 4). In comparison, our method requires no further user input besides the model and the log. Other methods focusing on returning high-level deviations are [12] (discussed in Sect. 4) and [21]. The latter proposes a set of deviation patterns that can be detected on top of a purpose-built multi-layer synchronous product net using trace alignments. The approach incorporates the resource and privacy perspectives and can provide the context in which deviations occur. Both techniques are based on alignments, therefore they suffer from some of its limitations such as non-determinism and poor scalability. Furthermore, it is unclear how these approaches can be extended to support new patterns.

Finally, our work closely relates to the field of declarative process mining. Our framework leverages multiple declarative techniques such as automatically computing constraint vacuity conditions [7], constraint discovery [19] (for steps A & B), constraint minimization [6,13] (step C), and monitoring [17] (step D). Our setup is different in that we use procedural models as a starting point. Moreover, both approaches are complementary. For example, the declarative model generated after step C could be loaded into tools such as RuM [3] for further analysis (but one would lose the procedural perspective).

7 Conclusion and Future Work

We presented a method to generate understandable conformance diagnostics based on behavioral patterns. The method automatically instantiates the constraints from a template library, checks for satisfying constraints, and prunes redundant constraints. It presents a series of interesting properties such as returning higher-level diagnostics, absence of false positives, and monotonicity

of the reported diagnostics. The qualitative evaluation demonstrates how it can be applied to analyze an event log, uncovering drill-down directions. The quantitative evaluation shows that it can cover most of the deviations in real-world logs in a feasible runtime. The user can control the scalability of the method by restricting the set of deviation patterns and varying the maximum number of template parameters at the expense of slightly less complete diagnostics.

In future work, we plan to improve the different steps of the framework with smarter constraint instantiation strategies, and more efficient constraint satisfiability checking and minimization techniques. Another direction is to consider time and data-aware models and constraint templates, to overcome the limitation to the control flow only, but this requires more sophisticated reasoning techniques. Last, we would like to explore root-cause analysis techniques to explain and correlate deviations.

Acknowledgements. We thank the Alexander von Humboldt (AvH) Stiftung for supporting our research.

References

1. Adriansyah, A., van Dongen, B.F., van der Aalst, W.M.P.: Conformance checking using cost-based fitness analysis. In: 15th International Enterprise Distributed Object Computing Conference, pp. 55–64 (2011). https://doi.org/10.1109/EDOC.2011.12
2. Adriansyah, A., van Dongen, B.F., Zannone, N.: Controlling break-the-glass through alignment. In: International Conference on Social Computing, SocialCom 2013, pp. 606–611. IEEE (2013). https://doi.org/10.1109/SocialCom.2013.91
3. Alman, A., Di Ciccio, C., Haas, D., Maggi, F.M., Nolte, A.: Rule mining with RuM. In: 2nd International Conference on Process Mining, ICPM 2020, Padua, Italy, pp. 121–128. IEEE (2020). https://doi.org/10.1109/ICPM49681.2020.00027
4. Berti, A., van Zelst, S.J., van der Aalst, W.M.P.: Process Mining for Python (PM4Py): bridging the gap between process- and data science. CoRR abs/1905.06169 (2019). http://arxiv.org/abs/1905.06169
5. Desel, J., Esparza, J.: Free Choice Petri Nets. Cambridge University Press, Cambridge (1995)
6. Di Ciccio, C., Maggi, F.M., Montali, M., Mendling, J.: Resolving inconsistencies and redundancies in declarative process models. Inf. Syst. **64**, 425–446 (2017). https://doi.org/10.1016/j.is.2016.09.005
7. Di Ciccio, C., Maggi, F.M., Montali, M., Mendling, J.: On the relevance of a business constraint to an event log. Inf. Syst. **78**, 144–161 (2018). https://doi.org/10.1016/j.is.2018.01.011
8. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Inf. Softw. Technol. **50**(12), 1281–1294 (2008). https://doi.org/10.1016/j.infsof.2008.02.006
9. van Dongen, B.F.: BPI Challenge 2015 (2015). https://doi.org/10.4121/UUID:31A308EF-C844-48DA-948C-305D167A0EC1
10. Esparza, J., Blondin, M.: Automata Theory: An Algorithmic Approach. MIT Press, Cambridge (2023)

27. Reijers, H.A., Slaats, T., Stahl, C.: Declarative modeling–an academic dream or the future for BPM? In: Daniel, F., Wang, J., Weber, B. (eds.) BPM 2013. LNCS, vol. 8094, pp. 307–322. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40176-3_26
28. Rozinat, A., van der Aalst, W.M.P.: Conformance checking of processes based on monitoring real behavior. Inf. Syst. **33**(1), 64–95 (2008). https://doi.org/10.1016/j.is.2007.07.001
29. Weidlich, M., Polyvyanyy, A., Desai, N., Mendling, J., Weske, M.: Process compliance analysis based on behavioural profiles. Inf. Syst. **36**(7), 1009–1025 (2011). https://doi.org/10.1016/j.is.2011.04.002