Aligning Data-Aware Declarative Process Models and Event Logs

Giacomo Bergami[1], Fabrizio Maria Maggi[1], Andrea Marrella[2], and Marco Montali[1]

[1] Free University of Bozen-Bolzano, Italy
gibergami@unibz.it, {maggi,montali}@inf.unibz.it
[2] Sapienza - University of Rome, Italy
marrella@diag.uniroma1.it

Abstract. Alignments are a conformance checking strategy quantifying the amount of deviations of a trace with respect to a process model, as well as providing optimal repairs for making the trace conformant to the process model. Data-aware alignment strategies are also gaining momentum, as they provide richer descriptions for deviance detection. Nonetheless, no technique is currently able to provide trace repair solutions in the context of data-aware declarative process models: current approaches either focus on procedural models, or numerically quantify the deviance with no proposed repair strategy. After discussing our working hypotheses, we demonstrate how such a problem can be reduced to a data-agnostic trace alignment problem, while ensuring the correctness of its solution. Finally, we show how to find such a solution leveraging Automated Planning techniques in Artificial Intelligence. Specifically, we discuss how to align traces with data-aware declarative models by adding/deleting events in the trace or by changing the attribute values attached to them.

Keywords: Conformance Checking · Alignments · Data-Aware Declarative Models · Multi-Perspective Process Mining · Automated Planning

1 Introduction

Conformance checking is a branch of process mining assessing whether a sequence of distinguishable events (i.e., a *trace*) conforms to the expected process behavior represented as a *process model* [21]. When a trace does not conform to the model, we say that the trace is *deviant*. In this case, techniques based on cost-driven alignments additionally provide minimal repair strategies to make the trace conformant to the model [2]. Alignments represent a valuable instrument for business analysts, as the combined provision of alternative repair strategies, ranked by alignment cost, supports the business analyst in choosing among different process improvement strategies. In conformance checking, models can be described by either procedural or declarative languages; while the former fully enumerate the set of all the possible allowed traces, the latter list the constraints

delimiting the expected behavior. Declarative process models like Declare models [19], whose semantics can be expressed in Linear Time Logic on finite traces (LTL_f) [9] can always be transformed into constraint automata. The representation of Declare models as automata can be adopted for aligning traces with this type of models [13,8].

Multi-perspective checking for process conformance is gaining momentum, as conformance checking techniques considering both control flow and data annotations as "first-class citizens" enable to discover more deviations [16]. This reflects the essence of real-world business processes, which are inherently described by both processes and their different domain objects [20] (e.g., employees, products, etc.), which can be encoded as traces and event data. While alignment-based data-aware conformance has been already investigated in the context of procedural models, most of the conformance checking approaches for data-aware declarative models [7] focus on a numerical approximation of the degree of conformance of a trace against the model and do not provide repair strategies.

To tackle this research gap, we propose a novel approach for aligning event logs and data-aware declarative models based on the reduction of this problem into a data-agnostic alignment problem. This solution exploits the following considerations: *a*) to represent the process model, we use a sub-set of the data-aware extension of Declare presented in [7]. After representing the data-aware Declare model using a data-agnostic LTL_f semantics, *b*) we exploit the data predicates in the data-aware Declare clauses to partition the data space. This provides propositions representing data in addition to event labels. Then, *c*) we combine each event label with the propositions generated in *b*) and transform the model in *a*) into its data-aware counterpart. The automata-based representation of such a model is used to align traces (seen as sequences of events with a payload of data attribute-value pairs) with the model. In particular, we show that the alignment problem can be expressed as a planning problem in Artificial Intelligence, which can be efficiently solved by selected state-of-the-art planners [8,17].

Despite the resulting data-agnostic alignment via planning is semantically equivalent to customary cost-based aligners [2], our previous work [8] showed that the former outperforms the latter in terms of computational performance and scalability in the presence of models of considerable size, which is the case of this paper. In fact, as a consequence of the reduction of the data-aware alignment problem into a data-agnostic one, the automata-based process models used as input for our approach have several more transitions and states than in traditional alignment problems. Therefore, as we needed to show the feasibility of our approach, we decided to resort to planning-based alignments for both presenting our framework outline and performing the experiments. Planners generate repair strategies able to align traces and a data-aware declarative model based on changes at the level of control flow (such as adding/deleting events) or at the level of the data flow (such as changing the attribute values attached to them).

The rest of the paper is structured as follows: after providing relevant related work (§2), we introduce the notion of event log (§3.1) and the data-aware declarative language used to represent the model (§3.2); we also provide hints

on Automated Planning, as we will later exploit the SymBA*-2 optimal planner [24] to compute the alignments (§3.3). These preliminary notions guide us into the definition of our working assumptions adhering to the literature of reference (§4). After deep-diving into the technical details providing the solution to the data-aware declarative alignment problem (§5), we benchmark SymBA*-2 over a synthetic dataset and discuss its performance in this context (§6). Last, we draw our final conclusions and propose some future work (§7).

2 Related Work

Most of the conformance checking techniques reported in the scientific literature are based on procedural models. In [2], for the first time, the authors introduce conformance checking augmented with the notion of alignments. A multi-perspective alignment-based approach has been presented in [16], where the authors propose techniques for conformance checking with respect to data-aware procedural models. This work combines the A* algorithm for alignment-based control-flow conformance checking with Integer Linear Programming for data conformance checking. [3]

The work described in [13] presents a (data-agnostic) conformance checking approach based on the concept of alignment for declarative models. It converts a Declare model into an automaton and performs conformance checking of a log with respect to the generated automaton. As a result of the analysis each trace in the log is converted into the most similar trace that the model accepts. This approach is similar to the procedural one presented in [16]. Our first attempt was, therefore, to extend this data-aware procedural approach to the declarative case. However, procedural models allow for a divide-and-conquer approach where, when searching the alignment space for the optimal alignment computation, the contribution of the control flow and of the data can be separately analyzed at first, then combining the obtained results. This is, in general, possible since removing data conditions from a procedural models leads to a more relaxed resulting model. The situation is completely different for declarative models, since removing data conditions from negative constraints could make them stronger, restricting the traces that the model accepts. Therefore, it is not possible to search in the space of traces that the model accepts, constructed by only considering the control-flow, and then refine the search considering the contribution of data.

More recently, in [6], the authors have presented an approach where the data perspective for conformance checking with Declare is expressed in terms of conditions on global process variables disconnected from the specific Declare constraints expressing the control flow. In other words, data constraints are not bound to control flow constraints and thus it is not possible to bind the behavior to specific data attributes. The only truly multi-perspective approach based on

[3] Note that, by design, Integer Linear Programming is not suitable to support the lexicographic order of strings, which is instead supported by our approach.

declarative models is the one presented in [7], in which the authors present an algorithmic framework to efficiently check the conformance of data-aware Declare constraints with respect to event logs. This approach numerically characterizes the degree of conformance of a log trace against the model without, however, providing repair strategies to the user. To go beyond the numerical evaluation of the conformance and build an alignment of a deviant trace, a boolean answer to the constraint-satisfaction problem is not sufficient since we need to solve an optimization problem with respect to a specific cost function. Therefore, in the current paper, Automated Planning has been chosen as a technique to formalize this optimization problem and translate it into an operational framework.

3 Preliminary Definitions

3.1 Event Logs

(Data) *payloads* are finite functions $p \in V^K$, where K is a finite set of keys and V is a (finite) set of data values. We consider also the case in which the value of a certain key k is missing in a payload. In particular, we denote as ε an element $\varepsilon \notin V$, such that $p(k) = \varepsilon$ for $k \notin \mathrm{dom}(p)$. Given a finite set of activity labels Act, an event σ_j is a pair $\langle \mathtt{A}, p \rangle$, where $\mathtt{A} \in \mathsf{Act}$ is an activity label, and p is a payload; we denote with λ (and ς) the first (and second) projection of such pair, i.e., $\lambda(\sigma_j) = \mathtt{A}$ (and $\varsigma(\sigma_j) = p$). A *trace* σ is a temporally-ordered and finite sequence of distinct events $\sigma_1 \cdots \sigma_n$, modeling a process run. We distinguish the trace keys (K_t) from the event keys (K_e), such that $K = K_t \cup K_e$ with $K_t \cap K_e = \emptyset$: all events within the same trace associate the same values to the same trace keys, i.e., $\forall \langle \mathtt{A}_i, p_i \rangle, \langle \mathtt{A}_j, p_j \rangle \in \sigma.\ \forall k \in K_t.\ p_i(k) = p_j(k)$. A log $\mathcal{L}$ is a finite set of traces. This characterization is compliant with the eXtensible Event Stream (XES) format, which is the *de facto* standard for representing event logs within the Business Process Management community [1].

3.2 Data-Aware Declare

Declare is a declarative process modeling language [19]. A Declare model $\mathcal{M}$ is described as a set of constraints $\{ c_1, \ldots, c_m \}$ that must be simultaneously satisfied throughout a process execution. Such constraints express either positive (or negative) dependencies between two events having labels in Act, or quantify the occurrence of events having a specific label in Act. In the first case, one of the two clause labels is called *activation*, and the other *target*; while testing a trace σ for conformance over this clause, the presence of the activation label in σ triggers the clause verification, requiring the (non-)execution of an event containing the target label in the same trace.

Declare has been extended to include conditions over data in the Declare constraints [7]. In this paper, we will consider two types of data predicates ϕ^d (*conditions*) decorating activations (i.e., activation conditions) and targets (i.e., target conditions), respectively. While activation conditions must be valid when

Table 1: Semantics for MP-Declare constraints in LTL_f.

Template	**LTL_f Semantics**
existence	$\top \rightarrow \mathbf{F}(\mathtt{A} \wedge \phi^d) \vee \mathbf{O}(\mathtt{A} \wedge \phi^d))$
responded existence	$\mathbf{G}((\mathtt{A} \wedge \phi^d) \rightarrow (\mathbf{O}(\mathtt{B} \wedge \phi^d) \vee \mathbf{F}(\mathtt{B} \wedge \phi^d)))$
response	$\mathbf{G}((\mathtt{A} \wedge \phi^d) \rightarrow \mathbf{F}(\mathtt{B} \wedge \phi^d))$
alternate response	$\mathbf{G}((\mathtt{A} \wedge \phi^d) \rightarrow \mathbf{X}(\neg(\mathtt{A} \wedge \phi^d)\mathbf{U}(\mathtt{B} \wedge \phi^d))$
chain response	$\mathbf{G}((\mathtt{A} \wedge \phi^d) \rightarrow \mathbf{X}(\mathtt{B} \wedge \phi^d))$
precedence	$\mathbf{G}((\mathtt{B} \wedge \phi^d) \rightarrow \mathbf{O}(\mathtt{A} \wedge \phi^d))$
alternate precedence	$\mathbf{G}((\mathtt{B} \wedge \phi^d) \rightarrow \mathbf{Y}(\neg(\mathtt{B} \wedge \phi^d)\mathbf{S}(\mathtt{A} \wedge \phi^d))$
chain precedence	$\mathbf{G}((\mathtt{B} \wedge \phi^d) \rightarrow \mathbf{Y}(\mathtt{A} \wedge \phi^d))$
not responded existence	$\mathbf{G}((\mathtt{A} \wedge \phi^d) \rightarrow \neg(\mathbf{O}(\mathtt{B} \wedge \phi^d) \vee \mathbf{F}(\mathtt{B} \wedge \phi^d)))$
not response	$\mathbf{G}((\mathtt{A} \wedge \phi^d) \rightarrow \neg\mathbf{F}(\mathtt{B} \wedge \phi^d))$
not precedence	$\mathbf{G}((\mathtt{B} \wedge \phi^d) \rightarrow \neg\mathbf{O}(\mathtt{A} \wedge \phi^d))$
not chain response	$\mathbf{G}((\mathtt{A} \wedge \phi^d) \rightarrow \neg\mathbf{X}(\mathtt{B} \wedge \phi^d))$
not chain precedence	$\mathbf{G}((\mathtt{B} \wedge \phi^d) \rightarrow \neg\mathbf{Y}(\mathtt{A} \wedge \phi^d))$

an event exhibiting the activation label occurs, target conditions impose value limitations on the payload of events containing the target label.

We use atom $\mathtt{A}$ as a shorthand for $\lambda(\sigma_i) = \mathtt{A}$ for each $\mathtt{A} \in \mathsf{Act}$ given an event σ_i to be assessed, while ϕ^d is a propositional formula containing as atoms either the universal truth ($\top$), or the falsehood ($\bot$), or a binary relation "$\mathtt{A}.k \; \Re \; c$", where c is a constant value representing either a number or a string, $\Re$ is either an equality or a precedence/subsequent relation over values in V or their negation, and $k \in K$ acts as a placeholder for $\varsigma(\sigma_i)(k)$, where $\varsigma(\sigma_i)$ is the payload associated to the event σ_i and k is associated to a value $\sigma(\sigma_i)(k)$. E.g., "$\mathtt{RP}.\mathit{quality} \leq 3$" is formally represented as $\varsigma(\sigma_i)(k) \leq 3$ for key $k = \mathit{quality}$ and for any event σ_i having $\lambda(\sigma_i) = \mathtt{RP}$. This is a widely adopted assumption, that spans from data-aware procedural models [16] to data-aware declarative models [7]. Furthermore, this assumption can also be adapted to categorical data, as strings are ordered via lexicographical orderings over the single characters. We denote the *compound conditions*, namely the conjunction of label requirements and data conditions, as $\psi = \mathtt{A} \wedge \phi^d$.

The semantics of the Declare constraints we consider here is represented in Table 1. Here, the $\mathbf{F}$, $\mathbf{X}$, $\mathbf{G}$, and $\mathbf{U}$ LTL_f future operators have the following meanings: formula $\mathbf{F}\psi_1$ means that ψ_1 holds sometime in the future, $\mathbf{X}\psi_1$ means that ψ_1 holds in the next position, $\mathbf{G}\psi_1$ says that ψ_1 holds forever in the future, and, lastly, $\psi_1\mathbf{U}\psi_2$ means that sometime in the future ψ_2 will hold and until that moment ψ_1 holds (with ψ_1 and ψ_2 LTL_f formulas). The $\mathbf{O}$, $\mathbf{Y}$ and $\mathbf{S}$ LTL_f past operators have the following meaning: $\mathbf{O}\psi_1$ means that ψ_1 holds sometime in the past, $\mathbf{Y}\psi_1$ means that ψ_1 holds in the previous position, and $\psi_1\mathbf{S}\psi_2$ means that ψ_1 has held sometime in the past and since that moment ψ_2 holds.

3.3 Automated Planning

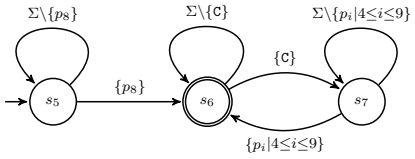

Fig. 1: Representation of the LTL$_f$ formula $\mathbf{G}(\neg \mathtt{C} \vee (\mathbf{F}(p_4 \vee p_5 \vee p_6 \vee p_7 \vee p_8 \vee p_9))) \wedge \mathbf{F}p_8$ as a constraint automaton [25], where Σ contains all the non-$\bot$ and non-$\top$ atoms.

Planning systems are an Artificial Intelligence technology showing how to reach a prefixed goal configuration given an initial world: the goal is met by exploiting a set of actions that change the initial world to reach the goal configuration [11]. PDDL is the standard Planning Domain Definition Language [10]; it allows us to formulate such problems as $\mathcal{P} = (I, G, \mathcal{P}_\mathcal{D})$, where I is the description of the initial world, G is the goal configuration, and $\mathcal{P}_\mathcal{D}$ is the planning domain. The domain is built upon a set of propositions describing the state of the world (i.e., the set of valid propositions) and a set of actions Ω that can be performed to reach the goal configuration. An action schema $a \in \Omega$ is in the form $a = \langle Par_a, Pre_a, Eff_a \rangle$, where Par_a is the list of the input parameters for a, Pre_a defines the preconditions under which a can be performed, and Eff_a specifies the effects of the action on the current world. Both Pre_a and Eff_a are represented as propositions in $\mathcal{P}_\mathcal{D}$ via boolean predicates and numeric fluents (i.e., global numeric variables).

Recently, the planning community has developed several planners implementing scalable search heuristics, which enable the solution of challenging problems in several Computer Science domains [17]. Walking in the footsteps of [8], we focus on planning techniques characterized by fully observable and static domains providing a perfect world description. In these scenarios, a sequence of actions whose execution transforms the initial state into a state satisfying the goal is the desired solution. In order to represent numeric alignment costs, we exploit the former formalization enhanced with the numeric features provided by PDDL 2.1 [10], thus keeping track of the costs of planning actions and synthesizing plans satisfying pre-specified metrics.

4 Working Assumptions

In this section, we outline some working assumptions that can be inferred from the literature of reference. First, we assume that *a*) compliance requirements of Declare models can be expressed in a formal language such as Linear Time Logic on Finite Traces (LTL$_f$) [9], as business process logs contain only traces of finite length; *b*) we restrict the possible log trace repairs to the traces generated by the automaton representation of the Declare model [8]; *c*) differently from [15,16], we can avoid to model reading and writing operations, as the entirety of

our analysis will be conducted once traces reach their completion; *d*) last, each event (which is part of the) trace must be represented by one single proposition: similarly to the non-data aware scenario [8], each event is associated with just one label. As we will see in the incoming section, the latter consideration will require us to partition the possible data space into distinct atoms.

Given an appropriately chosen set Σ of atoms, it is always possible to represent a trace $\sigma = \sigma_1 \cdots \sigma_n$ as a finite sequence $t_\sigma = t_1 \cdots t_n$, where, for $1 \leq i \leq n$, t_i is a unique atom $t_i \in \Sigma$ such that $\sigma_i \vDash t_i$ [8]. Contextually, any LTL_f formula $\varphi_\mathcal{M}$ representing a Declare model $\mathcal{M}$ can be represented as a deterministic finite-state automaton (DFA) $\mathcal{A}_{\varphi_\mathcal{M}}$ [25] accepting all the sequences t_σ from traces σ satisfying $\varphi_\mathcal{M}$ (see Figure 1). A DFA $(\Sigma, Q, q_0, \rho, F)$ is defined [12] over a finite set of states Q reading as input symbols from a finite alphabet Σ that are consumed by traversing the automaton from a starting state $q_0 \in Q$ via a transition function $\rho\colon Q \times \Sigma \to Q$; the input sequence is accepted once the input sequence is completely digested and an accepting state in $F \subseteq Q$ is reached through navigation. Since in the non data-aware Declare scenario the atoms within LTL_f could be either $\top$, or $\bot$, or $\psi = \texttt{A}$, Σ corresponds to the activity set Act, as each event is associated to one single label.[4] For data-aware Declare we will extend Σ to take into consideration propositional formulas representing data conditions.

We also want to show that our conceptual framework can be translated into an operational framework by taking existing solid techniques and extending them appropriately. After reducing the data-aware alignment problem into a data-agnostic one, we choose to operationalize it using Automated Planning, as our previous work [8] already showed that such a strategy outperforms customary cost-based trace aligners in terms of computational performance and scalability.

Last, we freely assume that all the events having the same label will always contain the same set of keys, with possibly differently associated values.[5] This is a common assumption in the relational database field, where all the rows belonging to the same table contain the same number of values.

5 Data-Aware Declarative Conformance Checking as Planning

In this section, we study the problem of aligning log traces $\sigma \in \mathcal{L}$ and a (data-aware) Declare model $\mathcal{M}$ for data-aware declarative conformance checking: to do so, we firstly reduce such problem to a mere automaton sequence acceptation task via a specific set of atoms Σ (*Cf.* §4) generated from the compound atoms in $\mathcal{M}$: the finite sequence t_σ generated from the log trace σ is accepted by the automaton $\mathcal{A}_{\varphi_\mathcal{M}}$ iff. σ is conformant to the model $\mathcal{M}$ (§5.1). Next, we code t_σ

[4] To allow multiple labels as customary of big-data scenarios [5], we could simulate such a situation by choosing only the most relevant label as the actual label and using other fields in the payload to hold the remaining ones.

[5] As both strings and floating point numbers have non-strict partial orders, we assume that missing values are represented as an extra minimal element ε being neither a string nor a specific floating point number.

Table 2: Intermediate steps for generating distinct atoms for B labeled events by partitioning the data space via intervals in Declare clauses.

$\mu(\mathtt{B}, x)$		$\mu(\mathtt{B}, y)$	
$\mathtt{B}.x > 3$	$\mathtt{B}.x > 3$	$\mathtt{B}.y = 0$	$0 \leq \mathtt{B}.y \leq 0$
$\mathtt{B}.x > 0$	$0 < \mathtt{B}.x \leq 3,\ \mathtt{B}.x > 3$	$\mathtt{B}.y \neq 0$	$\mathtt{B}.y < 0 \vee \mathtt{B}.y > 0$
$\mathtt{B}.x \leq 0$	$\mathtt{B}.x \leq 0$		

(a) Interval decomposition in $\mu(\cdot, \cdot)$

B	$\mathtt{B}.y < 0$	$\mathtt{B}.y = 0$	$\mathtt{B}.y > 0$
$\mathtt{B}.x \leq 0$	p_1	p_2	p_3
$0 < \mathtt{B}.x \leq 3$	p_4	p_5	p_6
$\mathtt{B}.x > 3$	p_7	p_8	p_9

(b) Atom generation for B by data space partitioning via $\times_{k \in K} \mu(\mathtt{B}, k)$

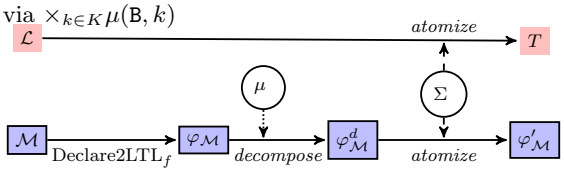

Fig. 2: Intermediate steps required for obtaining $\Sigma = \{\ p_i \mid 1 \leq i \leq 9\ \} \cup \{\mathtt{C}\}$ from $\mathcal{M}$ and transforming $\mathcal{L} = \{\ \sigma, \sigma'\ \}$ into a set of finite sequences $T = \{\ t_\sigma, t_{\sigma'}\ \}$, as well as replacing atoms in $\varphi_{\mathcal{M}}$ with equivalent atoms in Σ ($\varphi'_{\mathcal{M}}$).

and $\mathcal{A}_{\varphi_{\mathcal{M}}}$ as specific automata (§5.2) that are exploited by a planner to generate the minimally repaired sequence $\hat{t_\sigma}$ of t_σ (§5.3), out of which we generate the minimally repaired trace $\hat{\sigma}$ which is conformant to $\mathcal{M}$ (§5.4).

5.1 Σ-encoding for Conformance Checking

As per the previous considerations, we want to show that, to solve the trace alignment problem for data-aware declarative conformance checking, it is sufficient to provide a specific characterization of Σ. Σ will be used to generate an automaton accepting symbols in Σ and the automaton will be used to test log traces represented as finite sequences in Σ^*. The proposed approach for obtaining Σ from a (data-aware) Declare model $\mathcal{M}$ is sketched in Figure 2, and described in detail in the following.

In the first *Declare2LTL$_f$* step, we exploit the usual conversion of each single Declare clause into an LTL$_f$ formula (see Table 1) in the *negated normal form* [14], where negations are possibly pushed inside atoms "$\mathtt{A}.k\ \Re\ c$" by replacing $\Re$ with its negation.

Example 1. *The Declare model $\mathcal{M}$ containing clauses* $\mathsf{Response}(C, B,\ B.x > 0)$ *and* $\mathsf{Existence}(B,\ B.x > 3 \wedge B.y = 0)$ *is represented as the intermediate LTL$_f$ formula* $\varphi_{\mathcal{M}} = \mathbf{G}(\neg C \vee \mathbf{F}(B \wedge B.x > 0)) \wedge \mathbf{F}(B \wedge B.x > 3 \wedge B.y = 0)$.

In the second *decomposition* step, for each compound condition $\psi = \mathtt{A} \wedge \phi^d$ over labels $\mathtt{A} \in \mathsf{Act}$, we collect all the atoms in ϕ_d in the form "$\mathtt{A}.k\ \Re\ c$" for

each $k \in K$ in a map $\mu(\mathtt{A}, k)$. Contextually, we represent atoms as intervals, and we *decompose* them into a disjunction of maximal non-overlapping data-aware predicates. This task can be efficiently computed via interval trees [4]. Last, we replace the atoms in each LTL_f formula by its decomposed representation.

Example 1 (continued)**.** *Table 2a shows the interval decomposition results for the ψ extracted from $\mathcal{M}$. E.g., predicates $B.x > 3$ and $B.x > 0$ are first represented as intervals $(3, +\infty)$ and $(0, +\infty)$, and then decomposed into disjoint sub-intervals $(-\infty,0]$, $(0,3]$, and $(3,+\infty)$. As a result, $\varphi_{\mathcal{M}}$ is decomposed into $\varphi^d_{\mathcal{M}} = \mathbf{G}(\neg C \vee \mathbf{F}(B \wedge B.x > 0)) \wedge \mathbf{F}(B \wedge (0 < B.x \leq 3 \vee B.x > 3) \wedge B.y = 0)$.*

In the third *atomization* step, we put an atom $\mathtt{A} \in \mathsf{Act}$ in Σ if the map $\mu(\mathtt{A}, k)$ is empty for each key $k \in K$; otherwise, given all the keys $k_{\mathtt{A}_1}, \ldots, k_{\mathtt{A}_h} \in K$ for which the map $\mu(\mathtt{A}, k_{\mathtt{A}_i})$ is not empty, we partition the data space by combining the non-overlapping intervals obtained from the previous step as $\mu(\mathtt{A}, k_{\mathtt{A}_1}) \times \cdots \times \mu(\mathtt{A}, k_{\mathtt{A}_h})$. For each of these interval combinations, we generate a fresh atom and put it in Σ.

Example 1 (continued)**.** *Label C is never associated to a data condition, and therefore it will be associated to one single atom C. On the other hand, label B is associated to several atoms obtained by partitioning the data space via the intervals in Table 2a. Table 2b shows the atom decomposition of B via data intervals over keys x and y, which induce a space partitioning of 9 intervals, for which we generate nine distinct atoms $p_1 \ldots p_9$. As a result, we obtain $\Sigma = \{\, p_i \mid 1 \leq i \leq 9 \,\} \cup \{\, C \,\}$ in Figure 2.*

Starting from these atoms, we firstly replace the compound conditions in the LTL_f interpretation $\varphi_{\mathcal{M}}$ of $\mathcal{M}$ with a disjunction of atoms from Σ as described in Table 2b, thus obtaining an equivalent LTL_f formula $\varphi'_{\mathcal{M}}$. Secondly, we generate a finite sequence $t_\sigma \in T$ for each log trace $\sigma \in \mathcal{L}$ by replacing each event σ_i in σ with the only atom $t_i \in \Sigma$ such that $\sigma_i \vDash t_i$.

Example 1 (continued)**.** *With reference to our running example, we replace the compound conditions in $\varphi^d_{\mathcal{M}}$ with the previously generated atoms; the compound condition $B \wedge B.x > 0$ is replaced by all the possible configurations of y and data intervals $0 < B.x \leq 3$ and $B.x > 3$, which are identified by the disjunction $p_4 \vee p_5 \vee p_6 \vee p_7 \vee p_8 \vee p_9$. On the other hand, $B \wedge B.x > 3 \wedge B.y = 0$ can be directly replaced by atom p_8: this results into an equivalent formula $\varphi'_{\mathcal{M}} = \mathbf{G}(\neg C \vee (\mathbf{F}(p_4 \vee p_5 \vee p_6 \vee p_7 \vee p_8 \vee p_9))) \wedge \mathbf{F}p_8$. Given a log $\mathcal{L} = \{B\{x{=}1,y{=}0\}C\{x{=}6\}C\{x{=}4\},\ C\{x{=}8\}B\{x{=}10,y{=}0\}\}$, all the events labeled as C are replaced with atom C, as there are no (data) conditions related to C in $\mathcal{M}$ that we can exploit to partition the data space. On the other hand, each event labeled as B is replaced by an equivalent atom in Σ: event $B\{x{=}1,y{=}0\}$ is uniquely represented by p_5, while event $B\{x{=}10,y{=}0\}$ is uniquely represented by p_8. This transformation results into a set of string sequences $T = \{p_5\ C\ C, C\ p_8\}$.*

After generating $\varphi'_{\mathcal{M}}$, we can exploit existing approaches [25] to generate a DFA that only accepts sequences satisfying $\varphi'_{\mathcal{M}}$. With reference to the previous

example, the first trace is not conformant to $\mathcal{M}$, since the first sequence is not accepted by the associated automaton. Similarly, the second trace is conformant to $\mathcal{M}$, since the second sequence is accepted by the associated DFA. In the forthcoming subsection, we will discuss how to generate repaired sequences that are accepted by the reference model.

5.2 Automaton Manipulation for Trace Alignment

Consider a sequence $t_\sigma = t_1 \cdots t_n$ generated from a trace σ, and the constraint automaton $\mathcal{A}_{\varphi_\mathcal{M}}$ generated from the Declare model $\mathcal{M}$. If the trace is deviant with respect to the model, we are interested in generating a repair sequence $\varrho = \varrho_1 \cdots \varrho_m$ from t_σ describing the operations to perform over σ to make it conformant to $\mathcal{M}$.

To realize this transformation, we consider two types of atomic violations, which can be caused by wrong (*deletion*) or missing (*insertion*) atoms in Σ. Differently from the non-data aware case [8], we also need to model *replacement* operations, defined as a data update within one single trace event: these operations can be mimicked by a delete operation followed by an insertion, as they substitute an event within a trace with the same event where a data value has been updated. The above operations, that will be later on encoded as PDDL actions, can be defined as follows:

- *deletion*/`del` $[\#\sigma_k \leftarrow \phi] ::= \sigma_1 \cdots \sigma_{k-1}\sigma_{k+1} \cdots \sigma_n$, for $n = |\sigma|$, $1 \leq k \leq n$, and $\phi = \sigma_k$
- *insertion*/`ins` $[@\sigma_k \leftarrow \phi] ::= \sigma_1 \cdots \sigma_{k-1}\phi\sigma_k \cdots \sigma_n$, for $n = |\sigma|$ and $1 \leq k \leq n$
- *replacement*/`repl` $[\sigma_k[\phi \mapsto \phi']] ::= \sigma_1 \cdots \sigma_{k-1}\phi'\sigma_{k+1} \cdots \sigma_n$ for $n = |\sigma|$, $1 \leq k \leq n$, and $\phi = \sigma_k$

Similarly to customary cost-based trace aligners, each of these operations has an associated cost, either quantifying the severity of the found violation or determining which operations shall be preferred. E.g., by assigning a higher cost to insertions and deletions and a lower one to replacements, we will favor replacements when possible. The *alignment cost* is defined as the number of deletions multiplied by their cost, plus the number of insertions multiplied by their cost, plus the number of replacements multiplied by their cost.

We can now define the conformance checking problem as follows:

Definition 1 (Log/Declare Conformance Checking). *Given a trace σ and a Declare model $\mathcal{M}$, checking the conformance of σ against $\mathcal{M}$ is the task of verifying whether σ conforms to $\mathcal{M}$, or σ is deviant and there exists a repair sequence ϱ making σ non-deviant for $\mathcal{M}$ and guaranteeing a minimal transformation cost.*

The process of generating a repair sequence can be addressed by resorting to DFAs (§4). Let $t_\sigma = t_1 \cdots t_n$ be a string sequence generated from a log trace σ via Σ, $\mathcal{A}_{\varphi_\mathcal{M}} = (\Sigma, Q, q_0, \rho, F)$ the constraint automaton to check t_σ

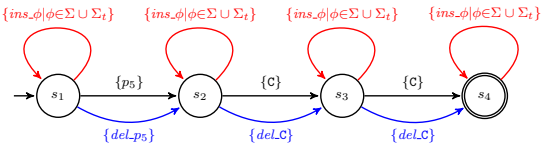

Fig. 3: Augmented trace automaton $\mathcal{T}^+$ for $t_{\sigma'} = p_5$ C C.

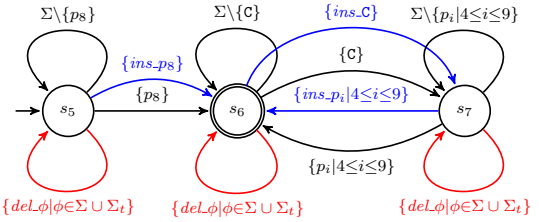

Fig. 4: Augmented constraint automaton $\mathcal{A}^+_{\varphi_\mathcal{M}}$ for $\mathcal{A}_{\varphi_\mathcal{M}}$.

against. From t_σ, we define a further automaton, called the *trace automaton* $\mathcal{T} = (\Sigma_t, Q_t, q_0^t, \rho_t, F_t)$ having *a)* $\Sigma_t = \{\, t_i \mid t_i \in t_\sigma \,\}$, *b)* $Q_t = \{\, q_0^t, \cdots, q_n^t \,\}$ as a set of $|t_\sigma| + 1$ states, *c)* $\rho(q_i^t, e_{i+1}) = q_{i+1}^t$ for $0 \leq i \leq n-1$, and *d)* $F_t = q_n^t$. By definition, such a graph accepts only t_σ.

Next, we augment $\mathcal{T}$ and $\mathcal{A}_{\varphi_\mathcal{M}}$ by adding transitions related to the atomic operations of insertions and deletions. Thus, from $\mathcal{T}$ we generate the automaton $\mathcal{T}^+ = (\Sigma_t^+, Q_t, q_0^t, \rho_t^+, F_t)$ having:

- Σ_t^+ extending $\Sigma_t \subseteq \Sigma$ by adding an insertion ins_ϕ for each atom $\phi \in \Sigma_t \cup \Sigma$ and a deletion del_ϕ for each atom $\phi \in \Sigma_t$.
- ρ_t^+ extending ρ_t by adding deletions $\rho_t^+(p, del\_\phi) = q$ for each transition $\rho_t(p, \phi) = q$, and insertions $\rho_t^+(q, ins\_\phi) = q$ for all atoms $\phi \in \Sigma \cup \Sigma_t$ and states $q \in Q_t$.

Figure 3 shows the trace automaton generated from the deviant trace σ_1 from Example 1. Similarly, from $\mathcal{A}_{\varphi_\mathcal{M}}$, we obtain $\mathcal{A}^+_{\varphi_\mathcal{M}} = (\Sigma^+, Q, q_0, \rho^+, F)$ having:

- Σ^+ extending Σ by adding an insertion ins_ϕ for each atom $\phi \in \Sigma$ and a deletion del_ϕ for each atom $\phi \in \Sigma \cup \Sigma_t$.
- ρ^+ extending ρ_t by adding insertions $\rho^+(p, ins\_\phi) = q$ for each transition $\rho(p, \phi) = q$ and deletions $\rho_t^+(q, del\_\phi) = q$ for all atoms $\phi \in \Sigma \cup \Sigma_t$ and states $q \in Q$.

Figure 4 shows the automaton augmented with the repair operations $\mathcal{A}^+_{\varphi_\mathcal{M}}$ obtained for the model $\mathcal{M}$ from Example 1. Intuitively, $\mathcal{A}^+_{\varphi_\mathcal{M}}$ accepts all the string sequences conformant to the model and have been obtained by adding/removing the missing/wrong atoms to/from t_σ, where atomic operations are explicitly marked. As required, both augmented automata do not accept $t_{\sigma'} = p_5$ C C. If insertions are associated to the lowest cost, the best alignment strategy adds

p_8 at the end on the trace; by explicitly marking such repair with *ins_p*$_8$, the augmented automata now accept $\hat{t_{\sigma'}} = p_5$ `C C` *ins_p*$_8$. On the other hand, if replacements are associated to the lowest cost, the best alignment strategy would replace the last `C` with p_8, thus requiring to first delete `C` and then insert p_8; the resulting $\hat{t_{\sigma'}} = p_5$ `C` *del_p*$_8$ *ins_p*$_8$ is also accepted by both automata.

Next, we show how automated planners can efficiently identify the repair operations ϱ needed to repair the trace σ using the augmented automata just defined.

5.3 Encoding in PDDL

In this section, we show how, given an augmented constraint automaton $\mathcal{A}^{+}_{\varphi_{\mathcal{M}}}$ obtained from an LTL$_f$ formula $\varphi_{\mathcal{M}}$, and an augmented trace automaton $\mathcal{T}^{+}$ obtained from a trace t, we build a cost-optimal planning domain $\mathcal{P}_{\mathcal{D}}$ and a problem instance $\mathcal{P}$ in PDDL. $\mathcal{P}_{\mathcal{D}}$ and $\mathcal{P}$ can be used to feed any state-of-the-art planners accepting PDDL 2.1 specifications, as discussed in Section 3.3. A solution plan for $\mathcal{P}$ amounts to the set of interventions of minimal cost to repair the trace with respect to $\varphi'_{\mathcal{M}}$, and generates a repair sequence ϱ that is going to be exploited in the forthcoming subsection for finally repairing the trace.

Planning Domain. In $\mathcal{P}_{\mathcal{D}}$, we provide two abstract types: `activity` and `state`. The first captures the activities involved in a transition between two different states of a constraint/trace automaton. The second is used to uniquely identify the states of the constraint automaton (through the sub-type `automaton_state`) and of the trace automaton (through the sub-type `trace_state`). To capture the structure of the automaton and to monitor its evolution, we defined five *domain propositions* as boolean predicates in $\mathcal{P}_{\mathcal{D}}$:

- `(trace ?t1 - trace_state ?e - activity ?t2 - trace_state)` holds if there exists a transition in the trace automaton between two states `t1` and `t2`, being `e` the activity involved in the transition.
- `(automaton ?s1 - automaton_state ?e - activity ?s2 - automaton_state)` holds if there exists a transition between two states `s1` to `s2` of a constraint automaton, being `e` the activity involved in the transition.
- `(atoms ?e1 - activity ?e2 - activity)` holds if `e1` and `e2` are two atoms in Σ associated to a same activity label.
- `(cur_state ?s - state)` holds if `s` is the current state of a constraint/trace automaton.
- `(final_state ?s - state)` holds if `s` is a final state of a constraint/trace automaton.

It is worth to notice that, if a generic activity `A` is associated to some data condition, `A` will be represented as a set of atoms p_1, p_2, p_3, etc. in $\mathcal{P}_{\mathcal{D}}$, see for example Table 2b. This means that, for any combination of atoms p_i - p_j associated to `A`, there will exist an instance of the predicate `(atoms)` that will hold for p_i and p_j. Furthermore, we define a *numeric fluent* `total-cost` to keep track of the cost of the violations. Notice that: *(i)* in PDDL, parameters are

written with a question mark character '?' in front, and the dash character '-' is used to assign types to parameters; and *(ii)* we remain consistent with the PDDL syntax, which allows the values of both predicates and fluents to change as a result of the execution of an action.

Planning actions are used to express the *repairs* on the original trace t. In our encoding, we have defined four actions to perform *synchronous moves* both in the trace/constraint automaton, or to add/remove/replace activities to/from/in the constraint and trace automata. In the following, we suppose that actions `ins`, `del` and `repl` have cost equal to 1. However, their cost can be customized to define the severity of a violation or to force priorities among actions.

```
(:action sync
 :parameters (?t1 - trace_state ?e - activity ?t2 - trace_state)
 :precondition (and (cur_state ?t1) (trace ?t1 ?e ?t2))
 :effect(and (not (cur_state ?t1)) (cur_state ?t2)
             (forall (?s1 ?s2 - automaton_state)
               (when (and (cur_state ?s1)
                          (automaton ?s1 ?e ?s2))
                     (and (not (cur_state ?s1))(cur_state ?s2))))))
```

```
(:action ins
 :parameters (?e - activity)
 :effect (and (increase (total-cost) 1)
          (forall (?s1 ?s2 - automaton_state)
            (when (and (cur_state ?s1)
                       (automaton ?s1 ?e ?s2))
                  (and (not (cur_state ?s1))
                       (cur_state ?s2))))))
```

```
(:action del
 :parameters (?t1 - trace_state
              ?e - activity
              ?t2 - trace_state)
 :precondition (and (cur_state ?t1)
                    (trace ?t1 ?e ?t2))
 :effect(and (increase (total-cost) 1)
    (not (cur_state ?t1))(cur_state ?t2)))
```

```
(:action repl
 :parameters (?t1 - trace_state ?e1 - activity ?t2 - trace_state ?e2 - activity)
 :precondition (and (cur_state ?t1) (trace ?t1 ?e1 ?t2) (atoms ?e1 ?e2))
 :effect(and (increase (total-cost) 1) (not (cur_state ?t1)) (cur_state ?t2)
             (forall (?s1 ?s2 - automaton_state)
               (when (and (cur_state ?s1)
                          (automaton ?s1 ?e2 ?s2))
                     (and (not (cur_state ?s1))(cur_state ?s2))))))
```

We modeled `sync` and `del` in such a way that they can be applied only if there exists a transition from the current state `t1` of the trace automaton to a subsequent state `t2`, being `e` the activity involved in the transition. Notice that, while `del` $[\#\texttt{t1} \leftarrow \texttt{e}]$ yields a *single* move in the trace automaton, `sync` yields, in addition, one move on the constraint automaton, to be performed synchronously. In particular, a synchronous move is performed in the constraint automaton if there exists a transition involving activity `e` connecting `s1` – the current state of the automaton – to a state `s2`. Then, `ins` $[@\texttt{t1} \leftarrow \texttt{e}]$ is performed only for transitions involving activity `e` connecting two states of the constraint automaton, with the current state of the trace automaton that remains the same after the execution of the action. Finally, `repl` $[\texttt{t1}[\texttt{e1} \mapsto \texttt{e2}]]$ can be seen as a synchronous combination of a `del` and an `ins`. It yields one move on the trace automaton and one on the constraint automaton, involving two atoms `e1` and `e2` associated to a same activity label, i.e., such that the predicate `(atoms ?e1 ?e2)` holds.

Planning Problem. In $\mathcal{P}$, we first define a finite set of constants required to properly ground all the domain propositions defined in $\mathcal{P}_{\mathcal{D}}$. In our case, constants

correspond to the state and activity instances involved in the trace/constraint automaton. Secondly, we define the *initial state* of $\mathcal{P}$ to capture the exact structure of the trace/constraint automaton. This includes the specification of all the existing transitions that connect two states of the automaton, and the definition of all the pairs of atoms belonging to a same activity label. The current state and the final states of the trace/constraint automaton are identified as well. Thirdly, to encode the goal condition, we first pre-process the constraint automaton by: *(i)* adding a new dummy state with no outgoing transitions; *(ii)* adding a new special action, executable only in the final states of the original automaton, which makes the automaton move to the dummy state; and *(iii)* including in the set of final states only the dummy state. Then, we define the goal condition as the conjunction of the final states of the trace automaton and of the constraint automaton. In this way, we avoid using disjunctions in goal formulas, which are not supported by all planners. Finally, as our purpose is to minimize the total cost of the plan, $\mathcal{P}$ contains the following specification: `(:metric minimize (total-cost))`. As the goal requires that in both augmented automata an accepting state is reached, the actions will encode the strategies to successfully visit both automata via their transition functions, while assigning different alignment costs to each of the strategies. When the goal is reached, the resulting action sequence (where `sync`s are stripped) represent the repair sequence ϱ that we are going to exploit in the next section.

5.4 Trace Repair

Last, we need to leverage the repair actions generated by the planner to repair the entire trace so to make it conformant to the model as a whole. In particular, the generated repair actions are always ordered based on their positions within the trace. By removing all the `sync` actions provided by the planner, we will obtain a sequence of insertions [@`t1` ← `e`], deletions [#`t1` ← `e`], and replacements [`t1`[`e1` ↦ `e2`]] for a trace σ via its associated t_σ. While deletions [#`t1` ← `e`] can be trivially implemented in the data-aware scenario by simply removing the problematical event, for insertions (or replacements), we need to add events with their associated payloads (or adapt the contained data values). Replacements [`t1`[`e1` ↦ `e2`]] can be implemented by replacing the values in `t1` violating the data condition `e2` with the nearest values to the values in `t1` satisfying `e2`. On the other hand, insertions require to generate totally new values: the insertion [@`t1` ← `e`] of a new event `t1` satisfying `e` can be modeled by generating a new event having the label induced by `e`, which is then instantiated with the same data values present in the last occurrence of an event similarly labeled if any, and instantiated with default values otherwise; then, such values are repaired by choosing the values satisfying `e` nearest to the default ones. E.g., the alignment result $\hat{t_\sigma} = p_5$ `C C` ins_p_8 of trace $\sigma =$`B{x=1,y=0}C{x=6}C{x=4}` generates the repair $\varrho = [@\sigma_4 \leftarrow p_8]$ after removing the `sync` operations. Then, we obtain a new trace $\sigma =$`B{x=1,y=0}C{x=6}C{x=4}B{x=4,y=0}`, where `4` is the nearest integer to `B.x=1` (taken from the first event) that satisfies $p_8 \equiv$ `B`.$x > 3 \wedge$`B`.$y = 0$.

Trace length	Alignment Time	Alignment Cost	Alignment Time	Alignment Cost	Alignment Time	Alignment Cost	Alignment Time	Alignment Cost
0 const. modified	3 constraints		5 constraints		7 constraints		10 constraints	
10	599.7	0	772.92	0	-	-	-	-
15	767.23	0	978.38	0	1,887.29	0	-	-
20	854.12	0	1,127.25	0	2,093.71	0	18,421.26	0
25	950.04	0	1,268.54	0	2,297.12	0	20,525.71	0
30	1,026.91	0	1,392.93	0	2,381.38	0	25,394.29	0
1 const. modified	3 constraints		5 constraints		7 constraints		10 constraints	
10	603.84	1	797.16	1	-	-	-	-
15	728.83	1	898.53	1	1,932.31	1	-	-
20	851.62	1	1,094.13	1	2,113.08	1	17,770.91	1
25	929.72	1	1,280.61	1	2,296.76	1	24,023.28	1
30	1,114.75	1	1,379.26	1	2,499.32	1	27,232.07	1
2 const. modified	3 constraints		5 constraints		7 constraints		10 constraints	
10	601.04	1.06	856.71	1.18	-	-	-	-
15	736.93	1.13	934.45	1.23	1,875.37	1.44	-	-
20	864.06	1.06	1,112.61	1.38	2,112.51	1.33	18,370.95	1.55
25	973.28	1.24	1,230.41	1.53	2,299.19	1.38	21,152.86	1.7
30	1,066.69	1.04	1,346.02	1.52	2,453.52	1.58	25,882.66	1.61
3 const. modified	3 constraints		5 constraints		7 constraints		10 constraints	
10	623.85	2.14	937.88	2.22	-	-	-	-
15	748.5	2.25	1,012.5	2.56	1,893.64	2.5	-	-
20	877.4	2.1	1,026.11	2.44	2,095.72	2.63	18,918.48	2.41
25	1,007	2.55	1,115.53	2.34	2,287.93	2.66	22,010.35	2.5
30	1,114.46	2.06	1,230.54	2.33	2,462.62	2.35	26,178.43	2.75

Table 3: Experimental results. The time (in *ms.*) is the average per trace.

6 Experiments

We have developed a planning-based alignment tool that implements the approach discussed in Section 5. The tool allows us to load existing logs formatted with the XES (eXtensible Event Stream) standard and to import data-aware models previously designed using RuM [3]. In order to find the minimum cost trace alignment against a pre-specified data-aware Declare model, our tool makes use of the SymBA*-2 [24] planning system. To produce optimal alignments, SymBA*-2 (winner of the sequential optimizing track at the 2014 Int. Planning Competition) performs a bidirectional A* search. We tested our approach on the grounded version of the problem presented in Section 5.3. We performed our experiments with a machine consisting of an Intel Core i7-4770S CPU 3.10GHz Quad Core and 4GB RAM. We used a standard cost function with unit costs for any alignment step that adds/removes activities in/from the input trace or changes a data value attached to them, and cost 0 for synchronous moves.

To have a sense of the scalability with respect to the "size" of the model and the "noise" in the traces, we have tested the approach on synthetic logs of different complexity. Specifically, we generated synthetic logs using the log generator presented in [23]. We defined four Declare models having the same alphabet of activities and containing 3, 5, 7 and 10 data-aware constraints respectively. Then, to create logs containing noise, i.e., behaviors non-compliant with the original Declare models, we changed some of the constraints in these models and generated logs from them. In particular, we modified the original Declare models by replacing 1, 2, and 3 constraints in each model using different strategies. In some cases, we replaced a constraint with its negative counterpart (see Table 1); in other cases, we replaced a constraint with a weaker constraint; in other cases, we replaced a data condition with its negation. Each modified

model was used to generate 5 logs of 1000 traces containing traces of different lengths (i.e., containing 10, 15, 20, 25, and 30 events, respectively).

The results of the experiments can be seen in Table 3. The alignment time (in ms.) and cost (that corresponds to the amount of `ins`/`del`/`repl` activities in an alignment) refers to the average per trace. The missing values in the table refer to experiments that could not be carried out because traces of certain lengths (e.g., 10) could not be generated by specific models (e.g., including 7 or 10 constraints), i.e., traces of those lengths satisfying those models do not exist. It is evident from the table that the alignment cost does not affect the performance of the alignment tool as, when the noise increases, the execution time does not change. As expected, however, the execution time is slightly sensible to the trace length, and grows exponentially with the number of (data-aware) constraints in the reference model. However, the results suggest that the heuristics adopted by the planner is able to efficiently cope with the above complexity enabling to perform off-line analysis with acceptable performance in case of a reasonably large number of data-aware constraints.

The declarative models and the event logs used for the experiments are available for testing and experiments repeatability at: https://tinyurl.com/ezd788bb.

7 Conclusions

In this paper, we presented an approach tackling conformance checking of log traces over data-aware Declare models. The proposed approach exploits Automated Planning for aligning the log traces and the reference model via a preliminary partitioning of the data space. The experiments show that the performance of the approach is acceptable even when the reference model contains a reasonably large number of data-aware constraints. In addition, since the implemented tool is independent of the planner used to solve the alignment problem, forthcoming improvements in the efficiency of the planners will be automatically transferred to the tool.

Future works will investigate the relationship between planners and approximate path matching techniques [18]. We will also investigate the possibility of performing alignments over data-aware knowledge bases [22], which potentially quicken the time required to test the satisfiability of the data conditions by conveniently indexing (i.e., pre-ordering) the payload space. The use of these approaches could allow us to tackle correlation conditions (i.e., data predicates involving attributes belonging to the payload of the activation and of the target, simultaneously) [7] that we did not consider in the current contribution. In fact, the presented approach is not able to cope with the state space explosion, caused by the presence of correlation conditions in the constraints to be checked, when searching for the optimal alignments.

References

1. Acampora, G., Vitiello, A., Di Stefano, B., van der Aalst, W., Gunther, C., Verbeek, E.: IEEE 1849: The XES Standard: The Second IEEE Standard Sponsored by IEEE

Computational Intelligence Society. IEEE Comp. Int. Mag. **12**(2) (2017)
2. Adriansyah, A., van Dongen, B.F., van der Aalst, W.M.P.: Conformance Checking Using Cost-Based Fitness Analysis. In: EDOC'11 (2011)
3. Alman, A., Di Ciccio, C., Haas, D., Maggi, F.M., Nolte, A.: Rule Mining with RuM. In: ICPM'20 (2020)
4. Bentley, J.: Solutions to Klee's rectangle problems. Tech. rep., Carnegie-Mellon Univ. (1977)
5. Bergami, G., Magnani, M., Montesi, D.: A Join Operator for Property Graphs. In: Workshops of the EDBT/ICDT 2017 Joint Conf. (2017)
6. Borrego, D., Barba, I.: Conformance checking and diagnosis for declarative business process models in data-aware scenarios. Expert Syst. Appl. **41**(11) (2014)
7. Burattin, A., Maggi, F.M., Sperduti, A.: Conformance checking based on multi-perspective declarative process models. Expert Syst. Appl. **65** (2016)
8. De Giacomo, G., Maggi, F.M., Marrella, A., Patrizi, F.: On the Disruptive Effectiveness of Automated Planning for LTLf-Based Trace Alignment. In: AAAI'17. AAAI press (2017)
9. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: IJCAI'13. AAAI press (2013)
10. Fox, M., Long, D.: PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. J. Artif. Intell. Res.(JAIR) **20**, 61–124 (2003)
11. Ghallab, M., Nau, D.S., Traverso, P.: Automated planning - theory and practice. Elsevier (2004)
12. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation, 3rd Edition. Addison-Wesley (2007)
13. de Leoni, M., Maggi, F.M., van der Aalst, W.M.P.: Aligning event logs and declarative process models for conformance checking. In: BPM'12. Springer (2012)
14. Li, J., Pu, G., Zhang, Y., Vardi, M.Y., Rozier, K.Y.: Sat-based explicit LTL$_f$ satisfiability checking. Artif. Intell. **289**, 103369 (2020)
15. Maggi, F.M., Montali, M., Bhat, U.: Compliance monitoring of multi-perspective declarative process models. In: EDOC'19. pp. 151–160. IEEE (2019)
16. Mannhardt, F., de Leoni, M., Reijers, H.A., van der Aalst, W.M.P.: Balanced multi-perspective checking of process conformance. Computing **98**(4) (2016)
17. Marrella, A.: Automated Planning for Business Process Management. Journal on Data Semantics **8** (2019)
18. Myers, E.W., Miller, W.: Approximate matching of regular expressions. Bulletin of Mathematical Biology **51**(1), 5–37 (Jan 1989)
19. Pesic, M., Schonenberg, H., van der Aalst, W.: DECLARE: Full Support for Loosely-Structured Processes. In: EDOC 2007. pp. 287–298 (2007)
20. Petermann, A., Junghanns, M., Müller, R., Rahm, E.: FoodBroker - Generating Synthetic Datasets for Graph-Based Business Analytics. In: WBDB'14 (2014)
21. Rozinat, A., van der Aalst, W.M.P.: Conformance checking of processes based on monitoring real behavior. Inf. Syst. **33**(1) (2008)
22. Schönig, S., Rogge-Solti, A., Cabanillas, C., Jablonski, S., Mendling, J.: Efficient and Customisable Declarative Process Mining with SQL. In: CAiSE'16 (2016)
23. Skydanienko, V., Di Francescomarino, C., Ghidini, C., Maggi, F.M.: A tool for generating event logs from multi-perspective declare models. In: BPM'18 (2018)
24. Torralba, A., Alcazar, V., Borrajo, D., Kissmann, P., Edelkamp, S.: SymBA*: A Symbolic Bidirectional A* Planner. In: Int. Planning Comp. (2014)
25. Westergaard, M.: Better algorithms for analyzing and enacting declarative workflow languages using LTL. In: BPM'11. Springer (2011)