Efficient Conformance Checking of Rich Data-Aware Declare Specifications

Abstract. Despite growing interest in process analysis and mining for data-aware specifications, alignment-based conformance checking for declarative process specifications has focused on pure control-flow specifications, or mild data-aware extensions limited to numerical data and variable-to-constant comparisons. This is not surprising: finding alignments is computationally hard, even more so in the presence of data dependencies. In this paper, we challenge this problem in the case where the reference model is captured using data-aware Declare with general data types and data conditions. We show that, unexpectedly, it is possible to compute data-aware optimal alignments in this rich setting, enjoying at once efficiency and expressiveness. This is achieved by carefully combining the two best-known approaches to deal with control flow and data dependencies when computing alignments, namely A* search and SMT solving. Specifically, we introduce a novel algorithmic technique that efficiently explores the search space, generating descendant states through the application of repair actions aiming at incrementally resolving constraint violations. We prove the correctness of our algorithm and experimentally show its efficiency. The evaluation witnesses that our approach matches or surpasses the performance of the state of the art while also supporting significantly more expressive data dependencies, showcasing its potential to support real-world applications.

Keywords: Multi-perspective conformance checking · Efficient optimal alignments · Data-aware Declare · Satisfiability modulo theories (SMT).

1 Introduction

Conformance checking [7] is a cornerstone task in process mining. It relates the observed behaviour contained in an event log to the expected behaviour described by a reference process model, with the goal of identifying and reporting deviations. A widely adopted approach substantiates conformance checking in the computation of so-called optimal *alignments*, where each non-conforming log trace is compared against the closest model trace(s), indicating where discrepancies are located and calculating a corresponding cost [5].

Lifting the computation of alignments to process models integrating multi-perspectives (most prominently data and control-flow), has been tackled with increasing interest [18,3,13], and so has been dealing with other forms of data-aware conformance checking. On the one hand, this reflects a growing prominence of multi-perspective (in particular, data-aware) process models in the foundations of process science. On the other hand, computing multi-perspective alignments provides more informative insights than in a pure control-flow setting [18].

Despite this growing interest, alignment-based conformance checking for declarative data-aware process specifications is still an open problem. Existing work mainly tackles constraint-based specifications on the process control-flow, concretely expressed using DCR graphs [9], Declare [17], or more expressive formulae in Linear Temporal Logic on finite traces (LTLf) [10]. Also trace analysis frameworks for multi-perpective Declare have been developed [4,6] that report activations and fulfillments of constraints, but without computing alignments (in [4] data conditions are also restricted in that they cannot refer to individual events). Work on process mining via SQL queries [20,19] covers conformance checking to some extent, in that Declare constraints expressed as SQL queries can filter traces from a database that match these constraints. However, no alignments are constructed, and only exactly matching traces are considered. To the best of our knowledge, the only attempt in lifting alignment computation to a data-aware setting is [3], which considers however a very limited data-aware extension of Declare where data are numerical and data conditions are restricted to variable-to-constant comparisons (such as $x > 5$).

The need for considering richer multi-perspective dependencies in constraints,[1] is well-known (see, e.g., [6,18,13]), and exemplified next.

Example 1. To highlight the expressivity of complex data conditions, consider a model for a shipping company that has a Declare *response* constraint between two events: "Package Shipment" (A) and "Delivery Confirmation" (B), such that after a package is shipped, delivery confirmation must be received. The constraint is equipped with a data condition that specifies that the delivery confirmation must be received within 3 days of shipping if the package weighs less than 10 kg and the delivery address is within a specific geographic region. In any other case, it must be received within 10 days of shipping. Our approach can handle this condition in SMT-LIB2 [2] syntax or simply as `(A.weight < 10.0 and B.region == "Europe") ? (B.time - A.time <= 3d) : (B.time - A.time <= 10d)`.

The fact that previous work did not consider rich constraints like the one shown in the example is not surprising: finding alignments is computationally hard, even more so in the presence of data dependencies. Also, this more expressive setting cannot be attacked relying on previous methods [10,3], based on the construction of an automaton capturing all and only the traces accepted by the reference Declare specification. In fact, this is not possible even for numerical datatypes going beyond mere comparison predicates, due to undecidability [14].

In this paper, we tackle this challenging problem by introducing a foundational algorithmic approach, paired with an effective implementation, to compute alignments for rich data-aware Declare specifications. Constraints are extended with expressive data conditions (both local to time instants and correlating time instances) over various datatypes, ranging from primitive datatypes such as strings and numbers to full-fledged data structures [15]—essentially, all types supported by state-of-the-art satisfiability modulo theory (SMT) solvers [11,12].

[1] As it will become clear in the paper, we treat timestamps as a specific datatype, hence data constraints also cover quantitative time constraints, such as deadlines.

This is achieved by casting the alignment problem as a search problem whose steps aim at identifying and repairing constraint violations, thereby intervening on model, trace, edit, and synchronous moves. This requires at once to reason on time and data conditions, and to efficiently explore the space of possible repairs. To tackle this problem we strategically integrate the two most effective methods for respectively handling control flow and data dependencies in alignment computation: A* search, in the variant adopted by one of the most recent methods for Declare [8], and SMT solving [2], so far employed for aligning data-aware procedural process models [13]. Our technique defines a novel search space that is explored using the A* search algorithm to find an optimal alignment. The initial state of the search space represents the original trace as an SMT formula. When a state is explored, an SMT solver is used to identify constraint violations and to generate the child states by repairing the parent state. This process continues until a goal state is found that has minimal cost and no violation to repair, from which the overall alignment can be reconstructed.

We establish the correctness of our algorithm through rigorous proofs and provide an extensive experimental evaluation, showing its ability to efficiently operate even when rich data conditions are employed. Our method matches or surpasses the performance of the state of the art, while providing for the first time concrete support for rich datatypes and data conditions that no other approach can handle.

The remainder of this paper is structured as follows: In Sec. 2 we recall the necessary preliminaries about data-aware Declare, and alignments. Sec. 3 is dedicated to our approach to conformance checking for data-aware Declare specifications. We describe its implementation in the tool DADA in Sec. 4. In Sec. 5, we provide a detailed evaluation and comparison with the state-of-the-art. In Sec. 6, we conclude and give some directions for future work. Additional material, including an extended version of this paper, is available online [1].

2 Preliminaries

In this section we introduce the required background about event logs, Declare with data, and alignments. We start with events and the data condition language.

Event logs. We consider an arbitrary infinite set Id of event identifiers. We consider the following notions of data-aware events, traces, and event logs:

Definition 1. *An* event *e is a triple $e = (\iota, a, \alpha)$ such that $\iota \in Id$, $a \in \mathcal{A}$ is an activity, and α is a partial assignment that maps variables in V to elements of their domain. Given a set of events E, a* trace *$\mathbf{e}$ is a finite sequence of events in E, that is, $\mathbf{e} \in E^*$; and an* event log *is a multiset of traces.*

Data conditions. We consider *sorts* $\Sigma = \{\texttt{bool}, \texttt{int}, \texttt{rat}, \texttt{string}\}$ for data payloads, with associated domains $\mathcal{D}(\texttt{bool}) = \mathbb{B}$, the booleans; $\mathcal{D}(\texttt{int}) = \mathbb{Z}$, the integers; $\mathcal{D}(\texttt{rat}) = \mathbb{Q}$, the rational numbers, and $\mathcal{D}(\texttt{string}) = \mathbb{S}$, finite strings. For a set of variables V and a sort $\sigma \in \Sigma$, V_σ denotes the subset of V of sort σ.

(1)	Existence(n, A):	A occurs at least n times.
(2)	Absence(n, A):	A occurs at most $n-1$ times.
(3)	Init(A):	A is the first activity.
(4)	End(A):	A is the last activity.
(5)	Choice(A, B):	Either A or B, or both, occur.
(6)	RespondedExistence(A, B):	If A occurs, B also occurs.
(7)	Response(A, B):	If A occurs, B follows.
(8)	AlternateResponse(A, B):	If A occurs, B follows without an A in between.
(9)	ChainResponse(A, B):	If A occurs, B is the next activity.
(10)	Precedence(A, B):	If B occurs, A precedes it.
(11)	AlternatePrecedence(A, B):	If B occurs, A precedes it without a B in between.
(12)	ChainPrecedence(A, B):	If B occurs, A is the previous activity.
(13)	NotResponse(A, B):	If A occurs, B does follow.
(14)	NotRespondedExistence(A, B):	If A occurs, B does not.
(15)	NotChainResponse(A, B):	If A occurs, B is not the next activity.

Table 1. Supported Declare templates.

Definition 2. *A* data condition *over a set of variables V is an expression according to the following grammar:*

$c = V_{\texttt{bool}} \mid \mathbb{B} \mid n \geq n \mid r \geq r \mid r > r \mid s = s \mid b \wedge b \mid \neg b \qquad s = V_{\texttt{string}} \mid \mathbb{S}$

$n = V_{\texttt{int}} \mid \mathbb{Z} \mid n + n \mid -n \qquad r = V_{\texttt{rat}} \mid \mathbb{Q} \mid r + r \mid -r$

The set of data conditions over a set of variables V is denoted by $\mathcal{C}(V)$.

We use data conditions as in Def. 2 in this paper to have a concrete language to refer to, but our implementation actually allows for the full SMT-LIB2 language [2] that is supported by the SMT solver of choice.

Declare. In the sequel, we assume that $\mathcal{A}$ is a fixed set of activities, denoted by lower-case letters. Tab. 1 lists the Declare templates used in this paper. We call a *Declare constraint* an expression that is obtained from a Declare template by substituting the upper-case template variables by activities in $\mathcal{A}$. Constraints based on templates (6)–(9) and (10)–(12) are called *response* and *precedence constraints*, respectively. Constraints using (13)–(15) are *negation constraints*.

Declare templates, as well as the derived constraints, have *activations* and *targets*. Intuitively, an activation is an event whose occurrence imposes the (non) occurrence of other events. These other events are called targets. For the templates in Tab. 1, in all *response* and *negation* templates variable A is the activation and B the target; while in all *precedence* templates, B is the activation and A the target. The remaining patterns use both A and B as targets. Given a Declare constraint φ, an activity is an activation (resp. target) activity in φ if it is substituted for an activation (resp. target) variable in the underlying template.

We consider *multi-perspective* Declare constraints that include data conditions. To that end, for the remainder of the paper we fix a set of sorted *process variables* V. Intuitively, these variables are considered the payload of activities. They are maintained along the entire trace, but may change their values. For a set *Set*, let $V^{Set} = \{v_s \mid v \in V \text{ and } s \in Set\}$ be a set of labelled variables that contains a copy of each variable in V for each element in *Set*. In particular, for $a \in \mathcal{A}$, the idea is that v_a represents the value of v while observing activity a.

Definition 3. *A* Declare constraint with data *is a quadruple $\langle \varphi, c_A, c_T, c_R \rangle$ consisting of a Declare constraint φ and data conditions c_A, c_T and c_R. Precisely,* *for a the activation and t the target activity in φ: (i) $c_A \in \mathcal{C}(V^{\{a\}})$ is called the* activation condition*, (ii) $c_T \in \mathcal{C}(V^{\{t\}})$ is called the* target condition*, and (iii) $c_R \in \mathcal{C}(V^{\{a,t\}})$ is called the* correlation condition*.*

Intuitively, c_A constrains the data variables while the activation activity is observed, c_T the data variables while the target activity is observed, and c_R expresses relationships between the data variables of both activities. For Declare constraints φ without activation, we assume that all but c_T are $\top$. For simplicity of presentation, we assume that the activation and target activity are different, (though in our implementation this is not required). A *Declare specification* $\mathcal{M}$ is a set of Declare constraints with data. In the sequel, if no confusion can arise, we refer to Declare constraints with data simply by *constraints*.

Example 2. As running example, we use the set of variables $V = \{x\}$, activities $\mathcal{A} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ and the specification $\mathcal{M}$ that consists of the following two constraints ψ_1 and ψ_2, where for readability we write $a.v$ instead of v_a, for $a \in \mathcal{A}$:
- $\psi_1 = \langle \text{ChainResponse}(\mathsf{a}, \mathsf{c}), \top, \top, \mathsf{c}.x > \mathsf{a}.x\rangle$: This specifies that each occurrence of a must be directly followed by an event with c such that the value of x associated with activity c is greater than the value of x seen with a.
- $\psi_2 = \langle \text{AlternatePrecedence}(\mathsf{c}, \mathsf{b}), \mathsf{b}.x \geq 0, \mathsf{c}.x \neq 0, \mathsf{c}.x < \mathsf{b}.x\rangle$: This states an alternate precedence relationship between the activation b and the target c, demanding that if the value of x seen with b is non-negative, an activity c must occur before activity b, without any other b activities with $x \geq 0$ in between. Furthermore, the x value of c must be lower than the x value of b.

The semantics of Declare constraints with data is the same as in [3], we recall it in [1, Def. 11]. The set of all traces that satisfy all constraints in $\mathcal{M}$ is denoted by *runs*$(\mathcal{M})$. We assume for our approach that *runs*$(\mathcal{M}) \neq \emptyset$.

Alignments. We aim to design a conformance-checking procedure that, given a trace and a Declare specification $\mathcal{M}$, finds an optimal alignment of $\mathbf{e}$ and a run of $\mathcal{M}$. Typically, when constructing alignments, not all events in the trace can be put in correspondence with an event in a run, and vice versa. Hence we use a "skip" symbol $\gg$ and consider the extended set of events is $E^{\gg} = E \cup \{\gg\}$.

For a set E of events as above, a pair $(e, f) \in E^{\gg 2} \setminus \{(\gg, \gg)\}$ is called *move* iff it is one of: (i) *log move* if $e \in E$ and $f = \gg$; (ii) *model move* if $e = \gg$ and $f \in E$; (iii) *edit move* if $(e, f) \in E^2$, $(e, f) = ((\iota, a, \alpha), (\iota, a, \alpha'))$, $\mathcal{D}(\alpha) = \mathcal{D}(\alpha')$ and $\exists v \in \mathcal{D}(\alpha)$ such that $\alpha(v) \neq \alpha'(v)$; (iv) *synchronous move* if $(e, f) \in E^2$ and $e = f$. We denote by *Moves* the set of all moves.

For a sequence of moves $\gamma = \langle (e_1, f_1), \dots, (e_n, f_n)\rangle$, the *log projection* $\gamma|_L$ of γ is the maximal subsequence $e'_1, \dots, e'_i$ of $e_1, \dots, e_n$ such that $e'_1, \dots, e'_i \in E^*$, that is, it contains no $\gg$ symbols. Similarly, the *model projection* $\gamma|_M$ of γ is the maximal subsequence $f'_1, \dots, f'_j$ of $f_1, \dots, f_n$ such that $f'_1, \dots, f'_j \in E^*$.

Definition 4 (Alignment). *Given a Declare model $\mathcal{M}$, a sequence of moves γ is an* alignment *of a trace $\mathbf{e}$ against $\mathcal{M}$ if $\gamma|_L = \mathbf{e}$, and $\gamma|_M \in$ runs$(\mathcal{M})$. The set of alignments for a trace $\mathbf{e}$ wrt. $\mathcal{M}$ is denoted by Align$(\mathcal{M}, \mathbf{e})$.*

Example 3. Consider the trace $\mathbf{e} = \langle(\#_1, \mathsf{a}, \{x = 0\}), (\#_2, \mathsf{b}, \{x = 2\})\rangle$. The following are two possible alignments for $\mathbf{e}$ against the model from Ex. 2:

$$\gamma_1 = \begin{array}{|l|c|l|}\hline \mathsf{a}\quad \{x=0\} & \gg & \mathsf{b}\quad \{x=2\} \\ \hline \mathsf{a}\quad \{x=0\} & \mathsf{c}\quad \{x=1\} & \mathsf{b}\quad \{x=2\} \\ \hline \end{array} \qquad \gamma_2 = \begin{array}{|l|l|c|}\hline \mathsf{a}\quad \{x=0\} & \gg & \mathsf{b}\quad \{x=2\} \\ \hline \mathsf{a}\quad \{x=0\} & \mathsf{c}\quad \{x=3\} & \gg \\ \hline \end{array}$$

Each move (e, f) is shown in a column, including e in the first row and f in the second row. Since event identifiers are irrelevant in alignments, we omit them.

A *cost function* is a mapping $\kappa\colon \mathit{Moves} \to \mathbb{R}^+$ that assigns a cost to every move. It is naturally extended to alignments as follows.

Definition 5 (Alignment cost). *Given $\gamma \in \mathit{Align}(\mathcal{M}, \mathbf{e})$ as before, the* cost *of γ is defined as the sum of costs of its moves, that is, $\kappa(\gamma) = \sum_{i=1}^{n} \kappa(e_i, f_i)$. Moreover, γ is* optimal for $\mathbf{e}$ and $\mathcal{M}$ *if $\kappa(\gamma)$ is minimal among all alignments for $\mathbf{e}$ and $\mathcal{M}$, namely there is no $\gamma' \in \mathit{Align}(\mathcal{M}, \mathbf{e})$ with $\kappa(\gamma') < \kappa(\gamma)$.*

In this paper, we will use the standard cost function κ that assigns $\kappa(e, f) = 1$ if (e, f) is a log or model move, $\kappa(e, f) = 0$ if (e, f) is a synchronous move, and for an edit move $(e, f) = ((\iota, a, \alpha), (\iota, a, \alpha'))$, $\kappa(e, f) = |\{\alpha(v) \neq \alpha'(v) \mid v \in V\}|$.

3 Data-Aware Declare Aligner

In this section, we outline the conceptual approach of the Data-Aware Declare Aligner. Given a Declare specification $\mathcal{M}$ and a trace $\mathbf{e}$, the aim is to find an optimal alignment of $\mathbf{e}$ wrt. $\mathcal{M}$. To that end, the basic idea is to start with the event sequence in $\mathbf{e}$, and subsequently *repair* it until an event sequence is obtained that satisfies all constraints in $\mathcal{M}$. To navigate through a large search space of possible alignments while ensuring an optimal solution, our approach leverages the A* algorithm. Each *state* represents a (partially) ordered set of events together with data conditions, effectively acting as a candidate alignment with a respective cost that may not yet satisfy all constraints.

An overview of the approach is sketched in Fig. 1: the initial state S_0 represents the set of events in the input trace, ordered as in $\mathbf{e}$, and with data conditions that reflect the variable assignments. The previously unvisited state S of minimal cost is selected. It is then checked whether there are remaining constraint violations in S. In this case, all possible *repairs* are applied to S creating a new child state from each repair, and another search iteration is performed. Otherwise, an optimal alignment for $\mathbf{e}$ is reconstructed from S.

States. As mentioned above, a state contains a partially ordered set of events, and data conditions on their payloads. In order to express conditions that involve variables in all events, we need, as a technicality, labelled variables: For an event $e = (\iota, a, \alpha)$, let $V^e = \{v_\iota \mid v \in V\}$ be a copy of the set V where each variable is labelled by the id of e. For a set of events E, let $V^E = \bigcup_{e \in E} V^e$ be the set of variables for all events in E. A state with set of events E can then use data conditions (cf. Def. 2) on V^E to refer to the events' payloads. In the sequel we also assume that V contains a special variable τ of type integer, and τ^ι will

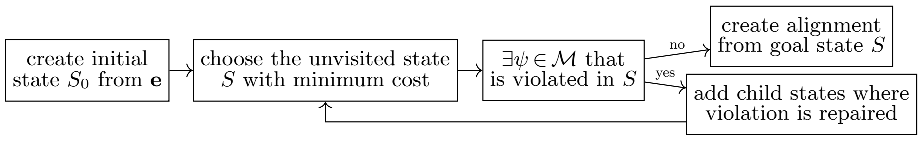
create initial state S_0 from $\mathbf{e}$
choose the unvisited state S with minimum cost
$\exists \psi \in \mathcal{M}$ that is violated in S
no
create alignment from goal state S
yes
add child states where violation is repaired

Fig. 1. Overview of the approach.

denote the timestamp of event with id ι. To reason about (partial) orderings of events in E, states use *ordering conditions*, defined next:

Definition 6. *An* ordering condition o *for a set of events* E *is of the following form, where* $e, e' \in E$, $a \in \mathcal{A}$ *is an activity, and* c *is a data condition as in Def. 2:*

$$o := e < e' \mid e \ll e' \mid \mathit{first}(e) \mid \mathit{last}(e) \mid e <^{a}_{[c]} e' \mid \neg o \mid \top$$

Here $e < e'$ expresses that e happens before e', $e \ll e'$ that e happens before e' without any other event in between, $\mathit{first}(e)$ that e is the first, $\mathit{last}(e)$ that e is the last element, and $\top$ is a condition that is always true. Somewhat more complex, $e <^{a}_{[c]} e'$ expresses that e happens before e' without an event e'' in between that has activity a and satisfies c, where c is supposed to be a data condition over $V^{e''}$. A set of ordering conditions on E in *satisfiable* if there exists a topological sort of E that satisfies all conditions. A set of ordering conditions O is said to *entail* an ordering condition q, denoted $O \models q$, if $\bigwedge O \wedge \neg q$ is unsatisfiable. Note that the ordering conditions are defined to closely align with the semantics of the supported Declare templates, as clarified in Def. 8.

We are now ready to give the definition of a state:

Definition 7. *A* state *is a pair* $S = \langle E, C\rangle$ *where* E *is a set of events, and* C *is a set of ordering conditions on* E *and data conditions over* V^E.

A state $\langle E, C\rangle$ thus represents a set of events E that is partially ordered by the ordering conditions in C, and where payloads of events are constrained by the data conditions in C.

For a trace $\mathbf{e} = \langle e_1, \dots, e_n\rangle$, let $E(\mathbf{e}) = \{e_1, \dots, e_n\}$ be its set of events, $O(\mathbf{e}) = \{e_i < e_{i+1} \mid 1 \le i < n\}$ be the set of ordering conditions that capture the event ordering in $\mathbf{e}$, and $D(\mathbf{e})$ the conjunction of all equations $v_\iota = \alpha(v)$ such that an event $e = (\iota, a, \alpha)$ occurs in $\mathbf{e}$ and $v \in \mathit{dom}(\alpha)$. The *initial state* is $\langle E(\mathbf{e}), O(\mathbf{e}) \cup D(\mathbf{e})\rangle$, it serves as the starting point for the exploration of the search space. Note that we use formulas that mix ordering and data conditions; we will explain in the next section how standard SMT solvers can be used to perform satisfiability checks of such formulas.

Example 4. Consider the trace $\mathbf{e}$ in Ex. 3 with events $e_1 = (\#_1, \mathsf{a}, \{x = 0\})$ and $e_2 = (\#_2, \mathsf{b}, \{x = 2\})$. If no confusion can arise, we write $\mathsf{a}.x$ rather than x_{ι_1} etc. for readability. The initial state is $S_0 = \langle\{e_1, e_2\}, (e_1 < e_2) \wedge (\mathsf{a}.x = 0) \wedge (\mathsf{b}.x = 2)\rangle$. Here $e_1 < e_2$ expresses that e_1 happens before e_2; the remaining conditions fix the values of x in the two events. Fig. 2 shows most of the search space for $\mathbf{e}$ and the

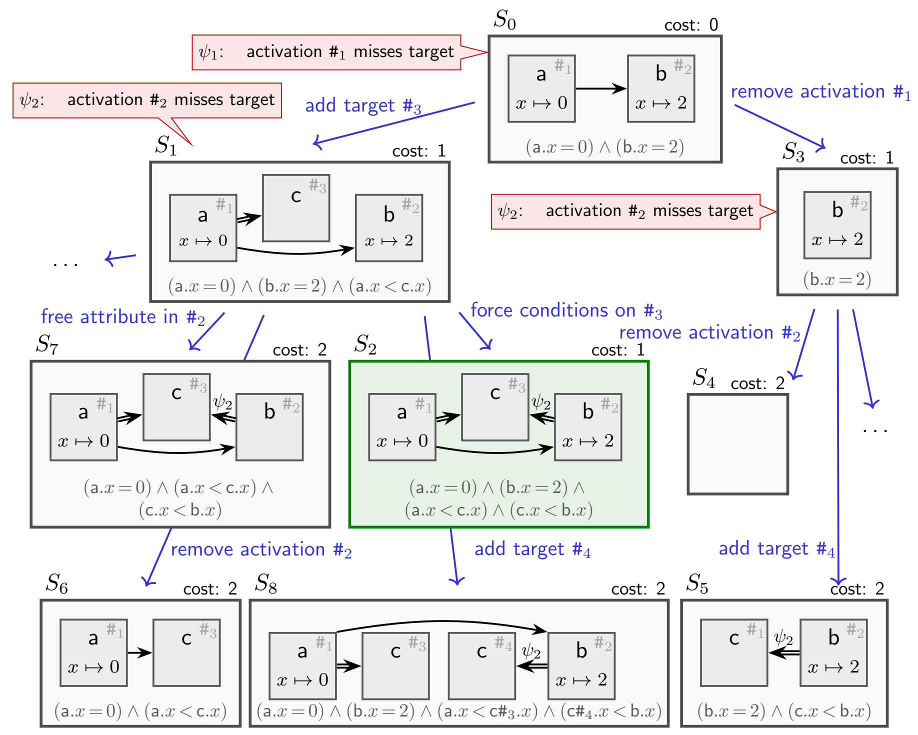
S_0 cost: 0
ψ_1: activation #$_1$ misses target
ψ_2: activation #$_2$ misses target
add target #$_3$
remove activation #$_1$
S_1 cost: 1
S_3 cost: 1
ψ_2: activation #$_2$ misses target
(a.$x=0$) ∧ (b.$x=2$)
(a.$x=0$) ∧ (b.$x=2$) ∧ (a.$x<$ c.x)
(b.$x=2$)
free attribute in #$_2$
force conditions on #$_3$
remove activation #$_2$
S_7 cost: 2
S_2 cost: 1
S_4 cost: 2
(a.$x=0$) ∧ (a.$x<$ c.x) ∧ (c.$x<$ b.x)
(a.$x=0$) ∧ (b.$x=2$) ∧ (a.$x<$ c.x) ∧ (c.$x<$ b.x)
remove activation #$_2$
add target #$_4$
add target #$_4$
S_6 cost: 2
S_8 cost: 2
S_5 cost: 2
(a.$x=0$) ∧ (a.$x<$ c.x)
(a.$x=0$) ∧ (b.$x=2$) ∧ (a.$x<$ c#$_3$.x) ∧ (c#$_4$.$x<$ b.x)
(b.$x=2$) ∧ (c.$x<$ b.x)

Fig. 2. Search space for the running example.

specification from Ex. 2 (the complete search space is shown in [1, Fig. 5]). States are shown as boxes, S_0 being the box on top. The events of a state are shown as boxes with activities within the state, and arrows in between them indicate ordering conditions. Here $e_1 < e_2$ is displayed by an arrow $e_1 \rightarrow e_2$, $e_1 \ll e_2$ by $e_1 \twoheadrightarrow e_2$, and the condition $e_1 <^{\mathsf{b}}_{[\mathsf{c}.x<\mathsf{b}.x]} e_2$ obtained from ψ_2 by $e_1 \overset{\psi_2}{\Rightarrow} e_2$. The formulas at the bottom of states specify data conditions. The states S_1–S_8 are obtained from S_0 by applying repairs; we will explain below how.

Constraint violations. We next define when constraints are violated in a state. To that end, we need some additional notation: given a Declare constraint ψ and events $e, e' \in E$, we denote by $Ord(\psi, e, e')$ the ordering conditions imposed by ψ between an activation event e and a target event e', defined as follows:

Definition 8. *Let e, e' be events and $\psi = (\varphi, c_A, c_T, c_R)$ a constraint. For templates φ with an activation, we define $Ord(\psi, e, e')$ as $e < e'$ (resp. $e' < e$) if φ is based on a* Response *(resp.* Precedence*) template, $e \ll e'$ (resp. $e' \ll e$) if φ has a* ChainResponse *(resp.* ChainPrecedence*) template, $\neg(e < e')$ for* NotResponse*, $\neg(e \ll e')$ for* NotChainResponse*, $e <^{a}_{[c_A]} e'$ for* AlternateResponse*, and $e' <^{a}_{[c_A]} e$ for* AlternatePrecedence*. In the last cases, a is the activation activity* *of φ. For constraints φ without activation, let $Ord(\psi, e)$ be first(e) or last(e) if φ is an* Init *or* Last *constraint, respectively. In all other cases, $Ord(\psi, e) = \top$.*

We also need to instantiate data conditions for events. To that end, given a Declare constraint $\psi = (\varphi, c_A, c_T, c_R)$ and an event $e = (\iota, a, \alpha)$ such that a is an activation activity for φ and b a target activity, we denote by $[c_A](e)$ (resp. $[c_T](e)$) the condition obtained from c_A (resp. c_T) by substituting v_a (resp. v_b) with v_ι for each $v \in V$. Similarly, for another event $e' = (\delta, b, \alpha')$, $[c_T \wedge c_R](e, e')$ denotes the condition obtained from $c_T \wedge c_R$ by substituting variable v_a by v_ι, and v_b by v_δ for all $v \in V$.

The first kind of violation is a *missing target*; intuitively, it applies if a constraint ψ can be activated but might lack a target that satisfies all conditions.

Definition 9. *A constraint (φ, c_A, c_T, c_R) has a* missing target *in state (E, C) if*
- *φ is a response or precedence constraint with activation activity a and there is an $e = (\iota, a, \alpha) \in E$ such that $C \wedge [c_A](e)$ is satisfiable, but no $e' \in E$ with target activity such that $\bigwedge C \wedge [c_A](e) \models Ord(\varphi, e, e') \wedge [c_T \wedge c_R](e, e')$; or*
- *φ is of the form* Existence(n, a)*, and $e_1, \dots, e_k$ are all events with activity a in E but $\bigwedge C \models \Sigma_{i=1}^{k} ite([c_T](e_i), 1, 0) \geq n$ does not hold; or*
- *φ is an* Init*,* End*, or* Choice *constraint and $e_1, \dots, e_k$ are all events with target activity in E but $\bigwedge C \models \bigvee_{i=1}^{k} Ord(\psi, e_i) \wedge [c_T](e_i)$ does not hold.*

Here $ite(b, d_1, d_2)$ abbreviates an if-then-else expression. In the first case of Def. 9, e is called *activation event*.

Fig. 2 shows three cases of missing target violations for the constraints in Ex. 2: in state S_0, ψ_1 is activated by the event $\#_1$ with activity a, but no target event with activity c is present. In S_1 and S_3, ψ_2 is violated: in S_3 since no event with activity c occurs, and in S_1 because, even though an event with activity c occurs, namely $\#_3$, its conditions do not entail $\mathsf{c}.x < \mathsf{b}.x$ and $\#_3 <^{a}_{[c_A]} \#_2$.

The second kind of violation is dual in that it signals too many targets.

Definition 10. *A constraint $\psi = (\varphi, c_A, c_T, c_R)$ has an* excessive target *in a state $S = (E, C)$ if*
- *φ is of the form* Absence(n, a) *and there are n events $e_1, \dots, e_n$ in E with activity a such that $\bigwedge C \wedge \bigwedge_{i=1}^{n} [c_T](e_i)$ is satisfiable;*
- *φ is a negation constraint, some $e_0, e_1 \in E$ have activation and target activity, resp., and $\bigwedge C \wedge Ord(\psi, e_0, e_1) \wedge [c_A](e_0) \wedge [c_T \wedge c_R](e_0, e_1)$ is satisfiable.*

The events $e_1, \dots, e_n$ in Def. 10 are called *excessive target events*.

For the states in Fig. 2, a constraint $\langle \text{NotResponse}(\mathsf{a}, \mathsf{b}), \mathsf{a}.x \geq 0, \top, \mathsf{b}.x > \mathsf{a}.x \rangle$ would have an excessive target violation in states S_0 and S_1, but not in S_3. On the other hand, $\langle \text{Absence}(\mathsf{b}), \top, \mathsf{b}.x = 3 \rangle$ would be violated in state S_7, but not in S_0 because there the conditions exclude $\mathsf{b}.x = 3$.

A constraint ψ is *violated* in a state if it has a missing or excessive target. A state $S = \langle E, C \rangle$ is a *goal state* if no constraint in $\mathcal{M}$ is violated in S and C is satisfiable. In Fig. 2, all leaves of the search tree (S_2 and S_4–S_8) are goal states.

Repairing violations. Our approach subsequently expands the search space by selecting a state where a constraint is violated and generating *child states* by repairing the violation in different ways. Four kinds of repairs are distinguished: (a) addition of an event, which will be reflected as a model move in the alignment; (b) removal of an event that stems from the trace, corresponding to a log move in the alignment; (c) freeing a data attribute in an event that stems from the trace, corresponding to an edit move; and (d) enforcement of conditions.

The applicable repairs and resulting states depend on the violated constraint $\psi = (\varphi, c_A, c_T, c_R) \in \mathcal{M}$ and current state $S = \langle E, C\rangle$. First, if ψ has an activation event $e_{act} \in E$, the following repairs are applied for both missing and excessive target violations to *disable* the activation:

(1) *Removing an activation event.* This applies if e_{act} stems from the trace $\mathbf{e}$. The resulting state is $S' = \langle E \setminus \{e_{act}\}, C'\rangle$ where C' is like C with conditions involving e_{act} removed.
(2) *Freeing a data attribute.* This applies to an event $e_{act} = (\iota, a, \alpha)$ from the trace $\mathbf{e}$ if α does not satisfy $\neg[c_A](e_{act})$. The repair removes an assignment $\alpha(v)$ of e_{act} for some $v \in V$. For $C' = C \setminus \{v_\iota = \alpha(v)\} \cup \{\neg[c_A](e_{act})\}$, the new state is $S' = \langle E, C'\rangle$.
(3) *Enforcing the negated activation condition.* The resulting state is $S' = \langle E, C'\rangle$ with $C' = C \cup \{\neg[c_A](e_{act})\}$.

If the violation is a missing target, then a target event can be added, or an existing event with the correct activity can be enforced to satisfy the data conditions, or in some cases events can be removed that block ordering conditions. More precisely, the following repairs apply:

(4) *Adding a target event.* A new state is of the form $S' = \langle E \cup \{e\}, C'\rangle$ where $e = (\iota, a, \emptyset)$ is a new event with fresh identifier ι. If ψ has an activation and e' is the activation event, then $C' = C \cup \{\mathit{Ord}(\psi, e', e), [c_A](e'), [c_T \wedge c_R](e', e)\}$. Otherwise, $C' = C \cup \{\mathit{Ord}(\psi, e), [c_T](e)\}$.
(5) *Freeing a data attribute.* This applies to events $e = (\iota, a, \alpha)$ in E that stem from the trace $\mathbf{e}$ and have the target activity but do not satisfy $[c_T \wedge c_R](e', e)$ if ψ has an activation event e' resp. $[c_T](e)$ otherwise. The repair removes an assignment $\alpha(v)$ of e for some $v \in V$, which can avoid the violation. The new state is $S' = \langle E', C \setminus \{v_\iota = \alpha(v)\}\rangle$ where E' is like E with e changed to (ι, a, α') such that α' is like α except being undefined for v.
(6) *Enforcing conditions.* This applies to an event $e \in E$ with activity a, i.e., a potential target event. The new state is $S = \langle E, C'\rangle$, where if ψ has an activation, and e' is the activation event that caused the missing target, then $C' = C \cup \{\mathit{Ord}(\psi, e', e)\} \cup \{[c_A](e'), [c_T \wedge c_R](e', e)\}$; otherwise, $C' = C \cup \{\mathit{Ord}(\psi, e), [c_T](e)\}$.
(7) *Removing a blocking event.* This applies if an event $e_t \in E$ with activity a is according to the ordering conditions in C not in the right position to act as target for ψ, but removing another event e from the trace can make room for e_t. E.g., Init constraints delete the first event, and ChainResponse constraints remove the events directly succeeding the activation. The resulting state is $S' = \langle E \setminus \{e\}, C'\rangle$ where C' is like C with conditions on e removed.

If ψ has an excessive target, we can either remove an excessive target event, or change the data conditions such that an event with target activity no longer acts as a target. Precisely, the following fixes apply:

(8) *Removing excessive target events.* This works like (1) above, but removes excessive target events if they stem from the trace.

(9) *Freeing a data attribute.* This repair is similar to (2) above, but it applies if there is an excessive target event $e = (\iota, a, \alpha)$ in E that stems from the input trace. However, we now enforce the negation of target and correlation conditions. The resulting state is $S' = \langle E', C' \rangle$ where E' is like E but where e is modified to (ι, a, α') such that α' is like α except that it is undefined for $v \in V$. Let $\widehat{C} = C \setminus \{v_\iota = \alpha(v)\}$. If there is an activation event e', we set $C' = \widehat{C} \cup \{\neg[c_T \wedge c_R](e', e)\}$; otherwise, $C' = \widehat{C} \cup \{\neg[c_T](e)\}$.

(10) *Enforcing negated conditions.* Let $e \in E$ be an excessive target event. There are two resulting states $S' = \langle E, C' \rangle$ and $S'' = \langle E, C'' \rangle$. If there is an activation event e', then $C' = C \cup \{\neg[c_T \wedge c_R](e', e)\}$; otherwise, $C' = C \cup \{\neg[c_T](e)\}$. Moreover, $C'' = C \cup \{\neg \mathit{Ord}(e', e)\}$.

Note that *all* applicable repairs are applied in all possible ways. For instance, when freeing a data attribute, a new state is generated for every event e and every variable assignment in e that satisfies the conditions in (2). Also, if there is a missing target violation and φ is a Choice constraint having two targets, a child state is created for each possible target.

For example, in Fig. 2, S_1 is obtained from S_0 by adding a target event #$_3$ (repair (4)); S_3 is obtained from S_0 by removing the activation event #$_1$ (repair (1)); S_7 is obtained from S_1 by freeing the data attribute x in event #$_2$ (repair (2)); and S_2 is obtained from S_1 by forcing conditions on event #$_3$ (repair (6)).

Alignment extraction. From a goal state $S = \langle E, C \rangle$, we extract an alignment as described by the pseudocode in Algorithm 1. The first step is to obtain an SMT model μ of the conditions $\bigwedge C$. This induces a list of model events $\mathbf{f} = \langle f_0, \dots, f_{m-1} \rangle$ that satisfies all ordering conditions, and where for each $f_j = (\iota_j, a_j, \alpha_j)$ the assignment α_j is given by $\alpha_j(v) = \mu(v_{\iota_j})$ for all $0 \leq j < m$. Algorithm 1 then walks simultaneously along $\mathbf{e}$ and $\mathbf{f}$, using i as an index for $\mathbf{e}$ and j for $\mathbf{f}$, and adds an edit or synchronous move if the current events e_i and f_j share the same id (so f_j stems from a trace event e_i), a model move if the id of the model event f_j does not occur in $\mathbf{e}$, and otherwise a log move. (For an event $e = (\iota, a, \alpha)$, we write $e.\mathit{id}$ to refer to ι.) In Line 6, the number of mismatching assignments in e_i and f_j determines whether the move is an edit or synchronous move. Note that the alignment of a state is in general not unique as there can be multiple SMT models. For instance, by applying Algorithm 1 to state S_2 resp. state S_6 in Fig. 2, one obtains the alignments γ_1 resp. γ_2 shown in Ex. 3.

A*-based search. Starting from the initial state that represents the input trace $\mathbf{e}$, our algorithm subsequently chooses a state with a violation and generates child states by applying all possible repairs. By a *search space* for $\mathbf{e}$ and $\mathcal{M}$, we mean below a graph of states where the root is S_0, and all states have as children the states obtained by all possible repairs, if any. To guide the search, the A* algorithm maintains for each state S a cost $\mathit{cost}(S) \in \mathbb{R}$, which can be shown

Algorithm 1 Extracting an alignment from a state

```
Require: State $S = \langle E, C\rangle$, trace $\mathbf{e} = \langle e_0, \dots, e_{n-1}\rangle$
Ensure: Alignment for $\mathbf{e}$
 1: $model \leftarrow$ SMT model of formula $\bigwedge C$
 2: $\langle f_0, \dots, f_{m-1}\rangle \leftarrow$ sort events in $E$ by assignment to ordering conditions in $model$
 3: $moves \leftarrow []$, $i \leftarrow 0$, $j \leftarrow 0$
 4: while $(i < n) \vee (j < m)$ do
 5:     if $(i < n) \wedge (j < m) \wedge (e_i.id = f_j.id)$ then
 6:         $moves.append(editOrSynchronousMove(e_i, f_j))$
 7:         $i \leftarrow i + 1$, $j \leftarrow j + 1$
 8:     else if $(j < m)$ and $f_j.id$ does not occur in $\mathbf{e}$ then
 9:         $moves.append(modelMove(f_j))$
10:         $j \leftarrow j + 1$
11:     else
12:         $moves.append(logMove(e_i))$
13:         $i \leftarrow i + 1$
14: return $moves$
```

to match exactly the cost of alignments extracted from S. The initial state has cost 0. When expanding a state S, the cost of a child state S' is determined by the applied repair: when adding or removing events, or freeing a data attribute, we have $cost(S') = cost(S) + 1$; when forcing condition satisfaction, $cost(S') = cost(S)$. In Fig. 2, each state is labelled with its respective cost.

Since repairs cause costs to increase, by fair exploration of the search space A* may conclude at some point that all goal states that might be further detected have a higher cost than the goal states found so far. At this point the search terminates, returning a goal state S_g with minimal cost K. Our correctness result below shows that the alignment extracted from S_g is optimal with cost K (cf. the proof in the extended version [1, Sec. A.2]). Our running example illustrates this result: in Fig. 2, the goal state with minimal cost is S_2, and indeed the alignment γ_1 extracted from it (cf. Ex. 3) is optimal with cost 1.

Theorem 1 (Correctness). *If S is a goal state with minimal cost K in a search space for $\mathcal{M}$ and $\mathbf{e}$ then γ returned by Algorithm 1 on input S and $\mathbf{e}$ is an optimal alignment of $\mathbf{e}$ wrt. $\mathcal{M}$ with cost K.*

4 Implementation

Our approach has been implemented in the tool DADA written in Kotlin, using the SMT solvers Z3 [11] and Yices [12] as backends. DADA requires two inputs: a multi-perspective event log in XES format [22] and a Declare specification with data $\mathcal{M}$. The model format is backward compatible with the one of [3]. Nevertheless, the syntax for data conditions has been significantly enhanced, allowing users to leverage the full expressiveness of the SMT-LIB2 language [2]. Also, the cost function can be customized, providing the cost of log, model, and edit moves as inputs. The Declare constraints language supported by our approach is, in fact, more expressive than initially introduced in Sec. 2. Specifically, branching in Declare constraints is enabled, as described in [8], and all Declare templates listed in [8, Tab. 2] have been implemented. Upon execution, the tool generates an optimal alignment, which can be output in either a human-readable format, similar to that illustrated in Ex. 3, or a compact machine-readable format. Additionally, the search space discovered during the computation can be exported as a graph, akin to the one depicted in Fig. 2.

Encoding. The SMT solver reasons on control flow and data dependencies in tandem, to identify violations and possible repairs for each constraint. We thus need to check satisfiability of formulas that mix ordering and data conditions. Data conditions as in Def. 2, but also much richer conditions, can be directly expressed in SMT-LIB2. Ordering conditions on a set E are encoded as follows: for every event $e \in E$ we use the SMT variable τ_e of integer type that encodes the event's timestamp. Then an ordering constraint $e_1 < e_2$ is directly translated to $\tau_{e_1} < \tau_{e_2}$; *first*(e) is translated to $\bigwedge_{e' \in E \setminus \{e\}} \tau_e < \tau_{e'}$ and similar for *last*(e); and $e \ll e'$ is translated to $\tau_{e_1} < \tau_{e_2} \wedge \bigwedge_{e \in E \setminus \{e_1, e_2\}} (\tau_e > \tau_{e_2} \vee \tau_e < \tau_{e_1})$. A constraint $e_1 <^a_{[c]} e_2$ is translated to $\tau_{e_1} < \tau_{e_2} \wedge \bigwedge_{e \in E_a} (\neg[c](e) \vee \tau_e > \tau_{e_2} \vee \tau_e < \tau_{e_1})$, where E_a is the set of all events in E with activity a, with e_1 and e_2 excluded. Moreover, for efficiency, the SMT solver is used in "online" mode: when checking the conditions of a state together with additional assertions, these are added temporarily and removed afterwards, using the push/pop mechanism of solvers.

Optimizations. We mention the most influential optimizations. The first is *selecting a violation to repair:* before detecting violations, events that meet the activation condition are identified for each constraint and state, and kept during search. As violations can be processed independently, the choice of the violation to repair next is crucial for performance. Currently, the tool selects the violation resulting in the fewest child states, to delay the state explosion.

The second optimization is about *detecting dead-ends:* in every state, all violations are precomputed, and for every violation it is checked which repairs are applicable. In case no repair is applicable for some violation, the state is a dead end, and the branch can be pruned from the search space. For example, this can happen when the only way to fix a violation is to enforce the target event to satisfy the data constraint of $x \geq 0$, but that same data attribute x was previously enforced to be less than 0 as a fix for another constraint.

Finally, we *prune unsatisfiable states.* If a state $S = \langle E, C \rangle$ was generated where $\bigwedge C$ is unsatisfiable, conflicting conditions were added while generating the state. Therefore, the state can be dropped from the search space.

5 Evaluation

We conduct our evaluation on the same Java Virtual Machine[2] as the state-of-the-art. The experiments are run on an Intel 5220R CPU with 8 GB of RAM. The source code, dataset, raw results, and executable are publicly available [1].

[2] OpenJDK 64-Bit VM Temurin 21.0.6+7-LTS

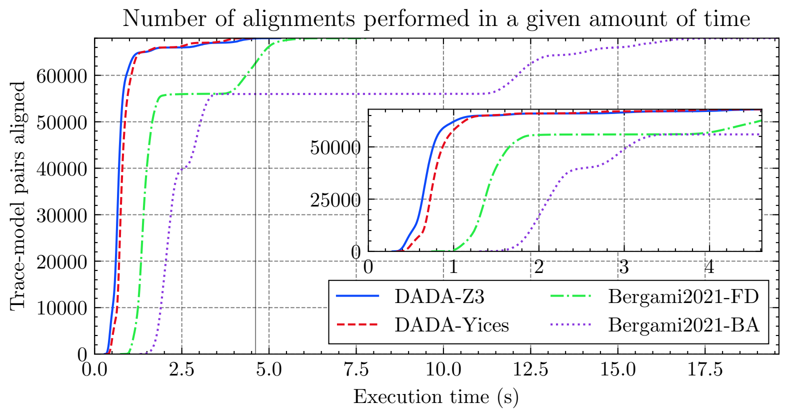

Fig. 3. Number of trace-model pairs aligned within a given time frame by each algorithm. The enlargement on the right highlights the differences for shorter time intervals.

Dataset. The evaluation utilizes a synthetic dataset that systematically varies in complexity, originally introduced by [3]. The complexity of the process models is influenced by the number of constraints (3, 5, 7, or 10) and constraint modifications (replacing 0, 1, 2, or 3 constraints). For each model, multiple event logs with varying trace lengths were generated (10, 15, 20, 25, or 30 events), resulting in 68,000 trace-model pairs. The models feature simple data conditions for both activations and targets of all their constraints. These are variable-to-constant conditions over `categorical` (with values c1, c2, or c3) and `integer` (ranging from 0 to 100), such as `categorical is c1` or `integer > 10`.

Performance comparison. We compare DADA, using either the Z3 [11] SMT solver or the Yices [12] SMT solver, to Bergami2021 [3], using the original SymBA* [21] planner or the Fast Downward [16] planner. Our experiments measure the execution time for each pair of model and trace of the dataset. To ensure all alignments are optimal, we validate that the alignment costs produced by DADA-Z3 and DADA-Yices match those generated by Bergami2021-BA and Bergami2021-FD.

Fig. 3 shows how, as the complexity of the trace-model pairs increases, the state-of-the-art algorithms exhibit a sharp increase in execution times, whereas our approach demonstrates better scalability. Notably, our approach aligns any trace-model pair in at most 5 seconds, and DADA-Z3 is on average 2.9 times faster than Bergami2021-FD and 5.9 times faster than Bergami2021-BA

Constraint flexibility. While the previous experiment was limited to the data conditions supported by [3], our approach can leverage the power of SMT solvers to define complex data dependencies such as the following correlation conditions. In these conditions, `A` refers to the activation and `T` to the target; `cat` is an abbreviation for the `categorical` attribute, and `timestamp` is the event's time.

(C1) `A.timestamp + A.integer * 1d > T.timestamp + T.integer * 1d`

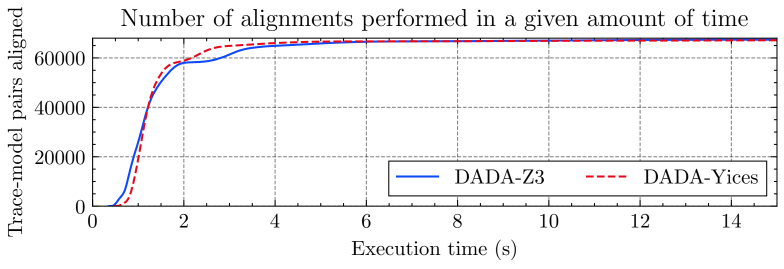

Fig. 4. Performance evaluation incorporating correlation constraints.

(C2) `(A.cat - "0") % 10 < (T.cat - "0") % 10`
(C3) `(A.cat == T.cat) ? (T.cat % 2 == 0) : (A.integer > T.integer)`

We create a new dataset by adding random negations, disjunctions and conjunctions of the previous correlation conditions to the original constraints, while retaining the original activation and target conditions, resulting in models like:

```
Response[act. 1, act. 2]|...|...|¬(¬C1 or ¬(¬C3 and ¬C2))|
Chain Response[act. 3, act. 4] |...|...|¬C1 or ¬C2 or C3 |
```

The added correlation conditions make the alignment problem even harder by potentially increasing (a) the number of repairs required to reach the optimal alignment, (b) the number of ways in which it is possible to repair them, and (c) the work performed by the SMT solver within each state. For these reasons, the models were simplified by only considering the ceiling of half of their constraints.

Fig. 4 shows that our approach can handle these advanced conditions, with only a small percentage of alignments timing out after 5 minutes or running out of memory (0.06% for DADA-Yices and 0.91% for DADA-Z3).

6 Conclusions

This paper presents a novel approach to computing data-aware optimal alignments between event logs and declarative process models, combining A* search and SMT solvers. Our key contributions include a new encoding scheme for the control flow, using an SMT solver to reason about control flow and data conditions, and an efficient A*-based search strategy that resolves constraint violations through repair actions. We prove its correctness and demonstrate its efficiency in experiments, matching or surpassing state-of-the-art performance while supporting more expressive data dependencies. Future work includes exploring optimization such as advanced pruning strategies and heuristic functions.

References

1. Anonymous: Efficient conformance checking of rich data-aware Declare specifications: Supplementary material (2025), https://tinyurl.com/DADAConformance

2. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard: Version 2.7. Tech. rep., University of Iowa (2025)
3. Bergami, G., Maggi, F.M., Marrella, A., Montali, M.: Aligning data-aware declarative process models and event logs. In: Proc. 19th BPM. LNCS, vol. 12875, pp. 235–251 (2021)
4. Borrego, D., Barba, I.: Conformance checking and diagnosis for declarative business process models in data-aware scenarios. Expert Syst. Appl. **41**(11), 5340–5352 (2014)
5. Bose, R.P.J.C., van der Aalst, W.M.P.: Process diagnostics using trace alignment: Opportunities, issues, and challenges. Inf. Syst. **37**(2), 117–141 (2012)
6. Burattin, A., Maggi, F.M., Sperduti, A.: Conformance checking based on multi-perspective declarative process models. Expert Syst. Appl. **65**, 194–211 (2016)
7. Carmona, J., van Dongen, B.F., Solti, A., Weidlich, M.: Conformance Checking - Relating Processes and Models. Springer (2018)
8. Casas-Ramos, J., Lama, M., Mucientes, M.: DeclareAligner: A leap towards efficient optimal alignments for declarative process model conformance checking (2025), arXiv 2503.10479
9. Christfort, A.K.F., Slaats, T.: Efficient optimal alignment between dynamic condition response graphs and traces. In: Proc. 21st BPM. LNCS, vol. 14159, pp. 3–19 (2023)
10. De Giacomo, G., Maggi, F.M., Marrella, A., Patrizi, F.: On the disruptive effectiveness of automated planning for LTLf-based trace alignment. In: Proc. 31st AAAI. pp. 3555–3561 (2017)
11. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Proc. 14th TACAS. LNCS, vol. 4963, pp. 337–340 (2008)
12. Dutertre, B.: Yices 2.2. In: Proc. 14th CAV. pp. 737–744 (2014)
13. Felli, P., Gianola, A., Montali, M., Rivkin, A., Winkler, S.: Cocomot: Conformance checking of multi-perspective processes via SMT. In: Proc. of BPM 2021. LNCS, vol. 12875, pp. 217–234. Springer (2021)
14. Felli, P., Montali, M., Patrizi, F., Winkler, S.: Monitoring arithmetic temporal properties on finite traces. In: Proceedings of the 37th AAAI Conference on Artificial Intelligence. pp. 6346–6354. AAAI Press (2023)
15. Gianola, A.: Verification of Data-Aware Processes via Satisfiability Modulo Theories, LNBIP, vol. 470. Springer (2023)
16. Helmert, M.: The fast downward planning system. Journal of Artificial Intelligence Research **26**, 191–246 (2006)
17. de Leoni, M., Maggi, F.M., van der Aalst, W.M.P.: An alignment-based framework to check the conformance of declarative process models and to preprocess event-log data. Inf. Syst. **47**, 258–277 (2015)
18. Mannhardt, F., de Leoni, M., Reijers, H., van der Aalst, W.: Balanced multi-perspective checking of process conformance. Computing **98**(4), 407–437 (2016)
19. Riva, F., Benvenuti, D., Maggi, F.M., Marrella, A., Montali, M.: An SQL-based declarative process mining framework for analyzing process data stored in relational databases. In: Proc. BPM Forum. LNBIP, vol. 490, pp. 214–231 (2023)
20. Schönig, S., Rogge-Solti, A., Cabanillas, C., Jablonski, S., Mendling, J.: Efficient and customisable declarative process mining with SQL. In: Proc. 28th CAiSE. pp. 290–305 (2016)
21. Torralba, A., Alcázar, V., Borrajo, D., Kissmann, P., Edelkamp, S.: SymBA*: A symbolic bidirectional A* planner. In: Planning Competition. pp. 105–108 (2014)
22. XES Working Group: IEEE standard for extensible event stream (XES) for achieving interoperability in event logs and event streams. IEEE Std 1849-2023 (2023)