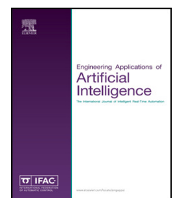
Engineering Applications of Artificial Intelligence 160 (2025) 111683

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai



Research paper

DeclareAligner: A leap towards efficient optimal alignments for declarative process model conformance checking

Jacobo Casas-Ramos *, Manuel Lama, Manuel Mucientes

Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidade de Santiago de Compostela, Santiago de Compostela, Spain

ARTICLE INFO

Dataset link: https://apps.citius.gal/ltgraph/

Keywords:
Process mining
Conformance checking
Optimal alignments
Declarative process models

ABSTRACT

Conformance checking is a crucial aspect of process mining, enabling organizations to identify deviations between actual process behavior and modeled expectations. At the heart of conformance checking lies the concept of optimal alignments, which provide a detailed, cost-minimized mapping of observed behavior to expected behavior. Optimal alignments facilitate the identification of root causes of non-conformity and guide corrective actions. This is a critical area where Artificial Intelligence (AI) plays a pivotal role in driving effective process improvement. However, computing optimal alignments poses significant computational challenges due to the vast search space inherent in declarative process models. Consequently, existing approaches often struggle with scalability and efficiency, limiting their applicability in real-world settings. This paper introduces DeclareAligner, a novel algorithm that uses the A* search algorithm, an established AI pathfinding technique, to tackle the problem from a fresh perspective leveraging the flexibility of declarative models. Key features of DeclareAligner include only performing actions that actively contribute to fixing constraint violations, utilizing a tailored heuristic to navigate towards optimal solutions, and employing early pruning to eliminate unproductive branches, while also streamlining the process through preprocessing and consolidating multiple fixes into unified actions. The proposed method is evaluated using 8054 synthetic and real-life alignment problems, demonstrating its ability to efficiently compute optimal alignments by significantly outperforming the current state of the art. By enabling process analysts to more effectively identify and understand conformance issues, DeclareAligner has the potential to drive meaningful process improvement and management.

1. Introduction

Process mining has become a crucial tool for organizations to analyze and improve their business processes, leveraging the increasing availability of event log data. Process mining encompasses a range of subjects (Aalst et al., 2012), including process discovery (reconstructing process models from event logs), conformance checking (comparing actual behavior with modeled behavior), and enhancement (improving process models based on insights gained).

Among these, conformance checking (Carmona et al., 2018) is particularly important as it compares an event log with a process model to identify deviations and discrepancies. One effective approach to conformance checking is through the use of alignments, which provide a detailed comparison of the executed steps in an event log with the expected behavior of the process model. Alignments visually represent the differences between the logged and modeled traces, highlighting discrepancies and mismatches. Optimal alignments (Adriansyah, 2014) further refine this approach by assigning a cost to each discrepancy and finding the alignment that minimizes the total cost.

Processes are often modeled using imperative approaches (Aalst, 1997), which spell out every possible allowed execution path in exhaustive detail. However, many real-world processes exhibit inherent variability, flexibility, and complexity, making it challenging to capture them using such rigid methods. In such cases, attempting to create an exhaustive model would likely result in a convoluted representation, characterized by numerous internal states — often referred to as a "spaghetti model" due to its intricate and cumbersome nature.

This is where declarative processes come into play (Back et al., 2018). Unlike their imperative counterparts, declarative process models focus on specifying the constraints that govern the behavior of a process, rather than dictating its exact execution flow. In essence, declarative models define "what" constraints must be satisfied during the execution of a process, without prescribing exactly "how" it should be done. This approach allows for greater flexibility in the actual

* Corresponding author.
E-mail addresses: jacobocasas.ramos@usc.es (J. Casas-Ramos), manuel.lama@usc.es (M. Lama), manuel.mucientes@usc.es (M. Mucientes).

https://doi.org/10.1016/j.engappai.2025.111683
Received 29 November 2024; Received in revised form 13 June 2025; Accepted 1 July 2025
Available online 31 July 2025
0952-1976/© 2025 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

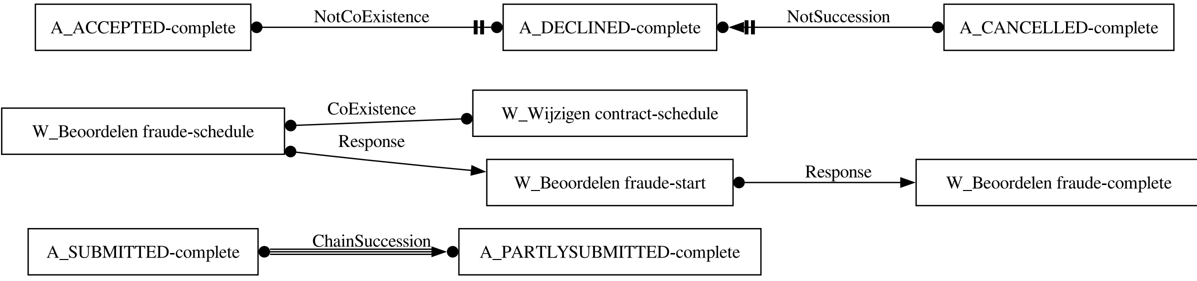


Fig. 1. Partial process model of a real-life Dutch financial institute.

process behavior, making it particularly suitable for capturing dynamic and complex processes.

Example. Consider a hospital's patient discharge process, which includes activities such as medication prescription, patient education, and follow-up appointment scheduling. A declarative model can specify constraints for this process, like "a patient must receive medication before being discharged" — `Precedence(ReceiveMedication, Discharge)` — or "a follow-up appointment must be scheduled after the discharge, without any other discharge in between" — `AlternateResponse(Discharge, Appointment)`. This allows for flexibility in the process while still enforcing key constraints.

However, this flexibility also introduces challenges when it comes to conformance checking. As modeled constraints only restrict some relationships among activities, all unrestricted behavior is allowed by default. This means that a declarative model implicitly permits a multitude of different execution paths. The DECLARE language (Pesic et al., 2007) is widely regarded as one of the leading approaches to declarative process modeling, offering a flexible and concise way to define constraints on business processes.

Example. A Dutch financial institute serves as a real-life case study for the application of optimal alignment during this paper's evaluation. The institute's loan application process involves a complex series of activities, including submission, fraud assessments, and approval decisions. Fig. 1 shows part of the process model for this real-life example, and includes constraints such as "a submission cannot be accepted and declined at the same time" and "a change of contract can be scheduled if a fraud assessment is also scheduled at some point and vice versa". By computing optimal alignments of this process against real logs, areas where actual process execution deviates from intended behavior can be identified, enabling the institute to refine its processes and improve customer satisfaction.

While much research has been devoted to computing optimal alignments for imperative process models (Casas-Ramos et al., 2024; Dongen et al., 2017; Dongen, 2018; Lee et al., 2018; Sani et al., 2020; Taymouri and Carmona, 2020), there is a noticeable gap in optimizing alignment techniques for declarative models.

Simple declarative conformance checking approaches do not compute optimal alignments. These methods, such as Chiariello et al. (2022) which is based on satisfiability problems, only report constraint failures without offering guidance on identifying the underlying issues. Others provide more detailed information, such as activations, fulfillments, and violations for each constraint (Burattin et al., 2016; Donadello et al., 2022; Maggi et al., 2019, 2011; Montali et al., 2013), but still fall short of computing optimal alignments. Another recent study by Riva et al. (2023) provides basic diagnostics, utilizing a database for both query and conformance checking. Although it acknowledges the need for richer feedback through alignments, this functionality is currently planned as future work. Notably, all these approaches fall short in identifying the root causes of non-conformances or providing minimal fixes, which is precisely the problem that optimal alignments aim to address.

More advanced diagnostics computing optimal alignments have been published using finite-state automata (Giacomo et al., 2017; Leoni et al., 2012). However, these methods can be computationally expensive due to the extensive search space they need to explore. Recent studies have introduced additional complexities, such as aligning data-aware declarative models, including Maggi et al. (2023), which employs SAT solvers, and Bergami et al. (2021), which leverages planning techniques. Furthermore, while Christfort and Slaats (2023) presents an efficient optimal alignment algorithm, its applicability is limited by its reliance on Dynamic Condition Response (DCR) graphs, which may pose interoperability challenges and suffer from a lack of widespread tool support. In addition, other proposals aim to provide more informative results, like Vespa et al. (2025), which checks conformance for probabilistic traces, acknowledging the inherent uncertainty in event log recordings, and Dongen et al. (2021), which aligns process models defined using both procedural and declarative paradigms. These advancements, while valuable, necessitate exploring even larger search spaces, thereby highlighting the need for efficient alignment algorithms for declarative process models.

Despite these advancements, there remains a significant need for more efficient and scalable approaches to computing optimal alignments for declarative conformance checking. To address this challenge, this paper introduces DECLAREALIGNER, a novel algorithm that extends the well-established A* search algorithm (Hart et al., 1968). A* is renowned for its ability to find the shortest path between two nodes in a graph. In the context of DECLAREALIGNER, these nodes are referred to as states, and the graph — which is termed the search space — is constructed incrementally as it is explored by the A* algorithm.

The proposed search space is initialized with a starting state that embodies the entire trace. From this foundation, new states are iteratively generated through the application of targeted corrective actions. As these actions are applied, each successive state exhibits increasing compliance with the process model, ultimately yielding a goal state that corresponds to a valid execution of the process. By comparing the original trace to a valid trace represented by this goal state, an optimal alignment can be extracted, highlighting the deviations between the actual and expected behavior.

A key advantage of DECLAREALIGNER lies in its ability to leverage the flexibility of DECLARE models, enabling efficient exploration of the vast search space of possible alignments and ultimately identifying the optimal alignment. The proposed approach offers several significant innovations that enhance its performance and scalability:

- It only considers taking actions that directly contribute to repairing violated constraints.
- It proposes several search optimization strategies in order to manage difficult alignments:
  - An heuristic that merges the suggested actions for all violated constraints of a state to provide accurate cost estimates.

– States that cannot reach the optimal alignment are pruned early to eliminate unfruitful branches of the search space.
– Actions that are required by chain constraints and do not interfere with other constraints are applied before the search.
– Actions perform multiple operations at once to avoid intermediate states.

The paper is structured as follows. Section 2 reviews existing approaches and highlights their limitations. Section 3 introduces the necessary concepts used by the algorithm. In Section 4 DeclareAligner is presented. Section 5 discusses the results of the empirical evaluation. Finally, Section 6 summarizes the contributions and suggests future work.

2. Related work

Declarative conformance checking, particularly using the DECLARE language (Pesic et al., 2007), has gathered significant attention in recent years due to its ability to model complex and flexible business processes and reason about their behavior. Several approaches have been proposed for declarative conformance checking. These can be broadly categorized into two types: those that report only basic conformance information and those that compute optimal alignments to provide detailed insights into process deviations.

Methods in the first category, such as Chiariello et al. (2022), simply execute recorded events in the process model and return a boolean result indicating conformance or non-conformance, without providing insights into specific issues. Other methods in this category, including Burattin et al. (2016), Donadello et al. (2022), Maggi et al. (2019, 2011), Montali et al. (2013) and Riva et al. (2023) report activations, satisfactions, and violations for each constraint within the DECLARE model and for every trace in the input log. While these results are faster to compute than optimal alignments, they do not identify root causes of non-conformance or suggest fixes. The main limitation of these approaches is that they only provide a high-level indication of conformance or non-conformance, without offering actionable insights for process improvement.

In contrast to simpler conformance checking methods, computing optimal alignments enables a more detailed analysis of process deviations and identification of opportunities for improvement. However, these approaches come at a higher computational cost. Currently, there are only two state-of-the-art approaches for computing optimal alignments of declarative models, namely those presented in Leoni et al. (2012) and Giacomo et al. (2017). A key characteristic shared by these approaches is the initial conversion of the DECLARE process model to finite-state automata. Specifically, standard DECLARE constraints can be translated into Linear-time Temporal Logic over finite traces (LTL_f) specifications, which in turn can be converted into finite-state automata for each constraint (Giannakopoulou and Havelund, 2001; Westergaard, 2011). The two state-of-the-art approaches differ primarily in their techniques for searching for the optimal alignment using these automata:

- The first approach (Leoni et al., 2012) employs the A* algorithm to find optimal alignments, providing metrics such as fitness, precision, and generalization. However, this method can be computationally expensive due to the need to explore a large state space using a relatively simple heuristic.
- The second approach (Giacomo et al., 2017) converts automata to a planning problem and uses classical planners to find optimal alignments. This approach has shown better efficiency but may face scalability issues due to the required conversion to a planning task or the lack of control over the approach that the planner takes.

The field of declarative conformance checking is continually evolving, with researchers exploring novel approaches to enhance its capabilities. For example, Maggi et al. (2023) proposes a method that leverages SAT solvers for data-aware declarative process mining, introducing an additional layer of complexity by incorporating data-aware constraints into the alignment problem. Similarly, Bergami et al. (2021) investigates the application of planning techniques for aligning data-aware declarative process models, which further complicates alignment computation by considering the impact of data attributes on process conformance. Additionally, Christfort and Slaats (2023) contributes to the understanding of efficient optimal alignment computation, although its focus on DCR Graphs may limit its broader applicability, as DCR Graphs are not as widely adopted as DECLARE. Moreover, Dongen et al. (2021) contributes to this area by examining the conformance checking of mixed-paradigm process models. However, it is noted that when such models contain solely declarative constraints without procedural constructs, they generate the largest possible search space, leading to performance degradation of the A* algorithm. The growing sophistication of these approaches creates a corresponding need to navigate increasingly vast search spaces, making efficient alignment algorithms for declarative process models a pressing requirement.

Table 1
Event log data collected for an e-learning process. Each row shows an event. An example trace with ID *Case234* is ⟨Enroll, Test, Exam⟩.

Trace ID	Activity	Timestamp
Case234	Enroll	2023-09-01 22:16:29
Case675	Class	2023-11-01 04:10:07
Case234	Test	2023-11-15 02:09:48
Case675	Exam	2023-12-13 07:24:41
Case234	Exam	2024-01-19 06:42:33
⋮	⋮	⋮

Existing methods for computing optimal alignments using DECLARE models struggle with large process models and logs. The proposed DeclareAligner algorithm addresses these limitations by introducing a novel approach that focuses only on actions that can potentially resolve violated constraints, preprocesses the problem to reduce unnecessary work, and groups multiple fixes together into single actions to minimize redundant effort. Additionally, the algorithm utilizes an advanced heuristic and detects and removes dead-ends from the search space. This results in improved performance and scalability, enabling the tackling of more complex optimal alignment tasks.

3. Preliminaries

Event logs record sequences of events that happened during the execution of a process. A single occurrence of an activity is called an Event. Each event is characterized by a trace identifier, activity name, timestamp, and optional additional information. Events are grouped into traces, i.e., sequences of events that occur in a specific context or scope. In essence, a trace provides a snapshot of how a particular process instance has unfolded over time. For the purposes of this paper, only the ordered sequence of activities contained within each trace is necessary. An example log is shown in Table 1.

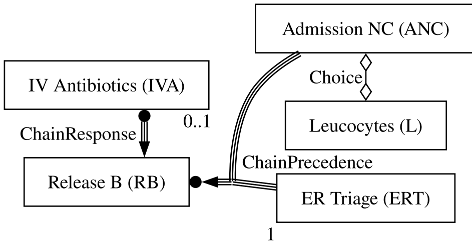
Definition 1 (*Trace*)**.** A trace is an ordered sequence of activities $\sigma = \langle A_1, \ldots, A_n \rangle$, where each activity A_i is extracted from events that belong to the same case, i.e., that share the same trace identifier. They are sorted according to the timestamp of the associated event.

Definition 2 (*Log*)**.** An event log $L = [\sigma_1, \ldots, \sigma_n]$ is a multiset of traces.

The most widely used declarative process modeling language is DECLARE (Pesic et al., 2007). It defines a list of parametrized templates that, when instantiated with specific activities, become constraints restricting the execution of those activities. Table 2 provides a comprehensive overview of all supported DECLARE templates. For formal

Table 2
Supported DECLARE constraints and their natural language descriptions.

Existence(n, A): A occurs at least n times.	*Participation(A)*: A occurs at least once.
Absence(n, A): A occurs at most n-1 times.	*AtMostOne(A)*: A occurs at most once.
Exactly(n, A): A occurs exactly n times.	*Init(A)*: A is the first activity.
End(A): A is the last activity.	*Choice(A, B)*: Either A or B, or both, occur.
ExclusiveChoice(A, B): Either A or B, but not both, occur.	*RespondedExistence(A, B)*: If A occurs, B also occurs.
Response(A, B): If A occurs, B follows.	*Precedence(A, B)*: If B occurs, A precedes it.
AlternateResponse(A, B): If A occurs, B follows without any other A in between.	*AlternatePrecedence(A, B)*: If B occurs, A precedes it without any other B in between.
ChainResponse(A, B): If A occurs, B is the next activity.	*ChainPrecedence(A, B)*: If B occurs, A is the previous activity.
CoExistence(A, B): If A occurs, B also occurs, and vice versa.	*Succession(A, B)*: Combines Response and Precedence.
AlternateSuccession(A, B): Combines AlternateResponse and AlternatePrecedence.	*ChainSuccession(A, B)*: Combines ChainResponse and ChainPrecedence.
NotRespondedExistence(A, B): If A occurs, B does not.	*NotCoExistence(A, B)*: If A occurs, B does not, and vice versa.
NotResponse(A, B): If A occurs, B does follow.	*NotPrecedence(A, B)*: If B occurs, A does not precede it.
NotSuccession(A, B): If A occurs, B does not follow, and if B occurs, A does not precede it.	*NotChainResponse(A, B)*: A is not immediately followed by B.
NotChainPrecedence(A, B): B is not immediately preceded by A.	*NotChainSuccession(A, B)*: A is not immediately followed by B and B is not immediately preceded by A.



(a) Graphical representation.

Choice(ANC, L)
ChainPrecedence([ERT, ANC], RB)
Absence(2, IVA)
Exactly(1, ERT)
ChainResponse(IVA, RB)

(b) Text version.

Fig. 2. Example of a DECLARE process.

definitions, refer to De Smedt et al. (2015). A DECLARE process model is simply a set of constraints.

A constraint is considered violated when the specified condition or rule is not met within the trace. This means that the activities in the process occur in a manner that breaches the constraint. If a constraint is not violated, it is considered satisfied.

A few constraints such as Init(A) are always active. However, most constraints have an activation activity. If the activation does not appear in the trace, the constraint is always (vacuously) satisfied. For example, the activity A of Response(A, B) is the activation, as this constraint ensures that B must appear at some point after A, only if A appears in the trace.

Branching is an important part of the DECLARE language as it allows defining more complex constraints by inserting multiple activities instead of just one in each parameter of the template. When multiple branched activities are applied to a parameter of a template, any of them can trigger the effect associated with the parameter. Branched activities are shown between square brackets to avoid confusion.

Example. Fig. 2 shows an example model from a hospital in both graphical and textual form with the following constraints:

- `Exactly(1, ERT)` specifies that `ERT` must occur exactly once in each trace.
- `Absence(2, IVA)` indicates that `IVA` cannot occur more than once.
- `Choice(ANC, L)` defines that either `ANC` or `L` must appear at some point in the trace.
- `ChainResponse(IVA, RB)` enforces that whenever `IVA` occurs, `RB` must also occur immediately after it.
- `ChainPrecedence([ANC, ERT], RB)` is an example of branching that specifies that whenever `RB` occurs, `ANC` or `ERT` must occur immediately before it.

Table 3
Alignment of the trace ⟨ANC, L, IVA, RB⟩ with the process from Fig. 2. Activities are shown as their initials.

LOG	ANC	L	≫	IVA	RB
MODEL	ANC	L	ERT	≫	RB

An alignment maps a trace to a process model, highlighting conformance issues and enabling applications like model repair and auditing. An alignment is a sequence of legal moves that traverse both the trace and the process model from start to finish.

Definition 3 (*Legal Move*)**.** Let M be a DECLARE model with activities A_M, L be a log with activities A_L, and i be the first index (zero-based) of the trace that still needs to be aligned such that $\sigma[i]$ is the next activity to align. A move is a tuple (a_L, a_M), where $a_L \in A_L \cup \gg$ and $a_M \in A_M \cup \gg$. Legal moves are moves of the following forms:

- **Synchronous moves** $(\sigma[i], \sigma[i])$ are available if the execution of the activity a_n does not permanently violate any constraint of M. This updates the state of the model accordingly and advances the index i to the next trace activity.
- **Log moves** $(\sigma[i], \gg)$ are available if $i < |\sigma|$. This move increments i by one, progressing in the trace without affecting the model.
- **Model moves** $(\gg, a)$ where $a \in A_M$ are available if all constraints of M would not become permanently violated if a is executed. This updates the state of the model by executing a.
Log and model moves may be referred to as asynchronous moves.

Definition 4 (*Alignment*)**.** An alignment $\gamma = \langle \gamma_1, \ldots, \gamma_n \rangle$ is a sequence of legal moves which reaches the end of the trace and a state of the model which satisfies all constraints.

Table 3 illustrates an example alignment, with each legal move as a column and the top and bottom rows represent the log and model parts respectively. A cost function assigns a penalty or cost to each move in an alignment, reflecting the degree of mismatch between the observed and modeled behavior. The standard cost function assigns a cost of 1 to asynchronous moves and a cost of 0 to synchronous ones. Table 3 has a total cost of 2.

Definition 5 (*Alignment Cost, Optimal Alignment*)**.** A cost function $C : (A_L \cup \{\gg\}) \times (A_M \cup \{\gg\}) \setminus (A_L \times A_M \cup \{\gg\} \times \{\gg\}) \to (0, \infty)$ assigns a cost c_{γ_i} to each possible legal asynchronous move, where A_L is the set

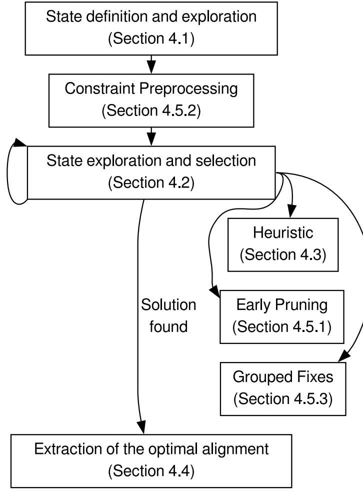


Fig. 3. High-level overview of DeclareAligner.

of activities in the trace and A_M is the set of activities in the declare model. Each valid alignment is assigned a cost $c_\gamma = c_{\gamma_1} + \cdots + c_{\gamma_n}$. An optimal alignment for a given trace and model is any of the alignments with the minimum total cost.

4. DeclareAligner algorithm

DeclareAligner is built on top of A* search, a popular pathfinding and graph traversal algorithm that efficiently finds the shortest path between two nodes in a weighted graph. The A* search algorithm, known for its completeness and optimality guarantees, is particularly well-suited for the optimal alignments task as this task can be expressed as a graph that repairs violated constraints until all constraints are satisfied. The nodes of this graph are referred to as states.

The DeclareAligner algorithm consists of several key components, as illustrated in Fig. 3. The process commences with an initial state, which represents the current trace (Section 4.1). Prior to initiating the search for the optimal alignment, the algorithm performs preprocessing on the initial state (Section 4.5.2) effectively reducing the workload by starting closer to the solution. Subsequently, the A* search algorithm recursively explores a search space that is tailored to the computation of optimal alignments. The graph is incrementally constructed by applying fixes to violated constraints, thereby generating neighboring states (Section 4.2). This exploration is facilitated by various optimizations, including a tailored heuristic (Section 4.3), early pruning of dead-ends (Section 4.5.1), and the grouping of multiple fixes into single actions (Section 4.5.3). When a state that has no violated constraints is explored, the optimal alignment can be extracted (Section 4.4).

4.1. State definition and notation

The algorithm explores a search space where each state is created by applying fixes to resolve constraint violations. A state contains the following attributes:

- **Cost** is a measure of the severity of fixes required to reach the state from the initial state.
- **Heuristic** is an estimate of the additional cost needed to transition from the current state to a goal state where all constraints are satisfied.
- **LTGraph** is a directed acyclic graph showing dependencies between activities or activity groups, with nodes connected by arcs that indicate precedence relationships. Activity groups can represent two types of relationships: branched activities, denoted as [A, B], where all activities have the same effect so they are interchangeable; or chained activity groups, denoted as A > B, where B directly follows A.
- **Violated activations** is a list of constraint activations that remain violated.
- **Will fix** is the activation that will be addressed next (Section 4.2).

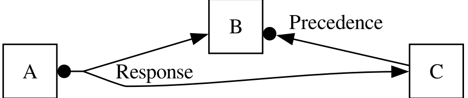


Fig. 4. Process model of the running example.

The algorithm commences with an initial state, denoted as *state0*, which serves as the starting point for the search process. The initial state is constructed by creating an LTGraph that reflects the original problem instance: each activity of the trace is added as a node and connected to the node for the next activity of the trace to indicate their strict precedence. The initial state has a cost of 0 since no edits have been made yet.

Example. To facilitate understanding of the concepts presented, a running example will be utilized throughout this section. The running example consists of the trace $\langle A, A \rangle$ and the process model shown in Fig. 4:

- `Response(A, [B, C])` enforces that if A appears in the trace (activation), then either B or C must occur at any point after the activation.
- `Precedence(C, B)` specifies that if B appears in the trace (activation), then C must occur at any point before the activation.

The initial state for the running example is depicted in Fig. 5, which shows from top to bottom:

- State identifier (*State0*), cost and heuristic values. The cost of the initial state is always 0, whereas the heuristic value is computed by identifying actions that are required to reach a goal state and adding their costs (Section 4.3).
- The LTGraph, which enforces the order between the two A activities present in the trace. Distinct subscripts identify each activity instance to avoid ambiguity.
- Lists of violated activations for each process model constraint. The initial state reveals that A_0 and A_1 are violated activations of the response constraint, stemming from the absence of B or C activities succeeding each A activity in the LTGraph. There are no violated activations for the precedence constraint, as there is no B in the LTGraph.
- The selected activation to repair, which in this case is A_1 (Section 4.2).

4.2. State exploration and selection

Neighbor generation is guided by identifying violated constraint activations and proposing repair actions. When multiple violations occur, those suggesting actions with the highest average costs are prioritized. This strategy of exploring high-cost actions first is grounded in the observation that non-zero cost actions result in asynchronous moves in the optimal alignment (Definition 3). Such actions can have a profound impact on other constraints, as they involve adding and removing activities. By prioritizing high-cost actions early in the search process, the risk of cascading effects that can arise when they are applied deeper in the search space, where numerous states are awaiting exploration, is mitigated. As a result of resolving high-cost actions first, subsequent repair actions suggested by other constraints tend to have localized effects

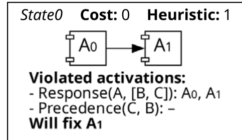
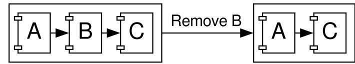
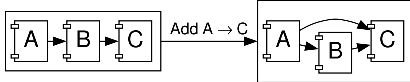


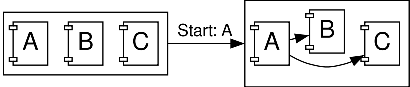
Fig. 5. Initial state for the running example.



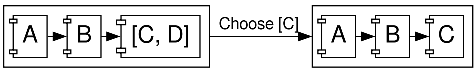
(a) Remove node example.



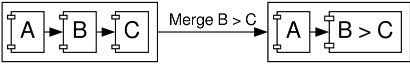
(b) Add arc example.



(c) Set start node example.



(d) Subset of branched activities example.



(e) Merge chains example.

Fig. 6. Examples of fix kinds applied to an LTGraph.

on the state. This makes them more likely to be successfully repaired without triggering additional conflicts, and ultimately facilitating a faster convergence towards an optimal alignment.

In cases where multiple violated activations have equal average costs, a secondary prioritization strategy to resolve ties is employed. Forward-looking constraints (e.g., Response) prioritize their last failed activation because all activations can be fixed if the target is inserted after this point. Analogously, backward-looking constraints favor their first failed activation. Negative constraints, however, require the opposite approach: since the goal is to avoid introducing targets altogether, removing the targets from the first activation effectively fixes all subsequent activations for forward-looking constraints and removing them from the last activation does so for backward-looking ones.

Generating neighboring states is accomplished by constructing actions, which comprise sequences of corrective modifications applied to the LTGraph in response to a violated constraint activation. All violations of declare constraints can be repaired by applying a combination of the following fix kinds:

- **Insert or remove node** (Fig. 6(a)). Adds or deletes a node of the LTGraph, updating adjacent arcs when removing a node.
- **Add an arc** (Fig. 6(b)). Enforces the origin node to precede the destination node.
- **Set start or end node** (Fig. 6(c)). Sets a node as the first or last node, adding arcs to all other nodes and ensuring it remains in that position.
- **Subset of branched activities** (Fig. 6(d)). In scenarios where a constraint is activated or violated by only a subset of branched activities present in a LTGraph node, this fix kind can be employed to avoid activation or enforce the target, respectively.
- **Merge or split chains** (Fig. 6(e)). To efficiently implement chain constraints, the LTGraph nodes are capable of representing lists of possibly branched activities (separated by >). This fix kind merges two nodes, preserving their chained relationship without the need to add arcs to all other nodes.

Only the insert or remove node fix kind has a non-null cost. This is because adding a node triggers a model move in the optimal alignment, while removing a node causes a log move (Section 4.4). In contrast, all other fix kinds only reduce the set of possible alignments that the LTGraph represents, without modifying their costs.

The standard cost function assigns a uniform cost of one to each insertion or removal operation. The cost of an action is the sum of the costs of its individual fixes. Each action taken contributes to increasing the cost of the current state.

Definition 6 (*State Cost*)**.** Let T be the complete set of actions applied to the initial state in order to reach the current state s, and let $ct(t)$ be the cost of the action t, the cost of s can be defined as:

$$cost(s) = \sum_{action \in T} ct(action)$$

The next state to explore is the previously unexplored state with the lowest total estimated cost ($cost + heuristic$). The algorithm continues to iteratively explore and select states until either a goal state is explored or no further states remain to be explored, in which case there is no optimal alignment.

Example (*Continued*)**.** Fig. 7 illustrates the complete search space explored by the algorithm to find the optimal alignment for the running example. The representation of an action is an arc connecting the parent state to the neighboring state. The arc is labeled with the fixes of the action. Goal states are highlighted with a green border, and the path to the optimal goal state is marked with green arrows.

As previously mentioned the initial state (*state0*) has two violated activations for the response constraint. In this case, the algorithm selects the A_1 activation from the LTGraph to repair. There are only two possible fixes, which create the neighboring states of *state1* and *state2*:

- *State1* removes the activation so that the constraint is never activated, with a cost of 1.
- *State2* inserts the expected target of the constraint, which can be either of the activities B or C. This results in the addition of the LTGraph node $[B, C]$ after the activation. This action results in satisfying both activations of the response constraint. However, it also activates the precedence constraint as one of the branched activities triggers it.

The algorithm selects *state2* to explore next based on its lower $cost + heuristic$ value. *state2* has only one violated activation to repair for the precedence constraint, resulting in the generation of the following neighboring states:

- *State3* removes the activation through a cost-free operation, which involves selecting a subset of branched activities to avoid triggering the precedence constraint.
- *State4* inserts the target activity before the activation. Note that the LTGraph now aggregates a series of possible activity orderings into a single state, reducing the size of the search space.

Ultimately, the algorithm reaches the goal state with the lowest cost, which in this case is *state3*. The optimal alignment can then be extracted from the goal state (Section 4.4).

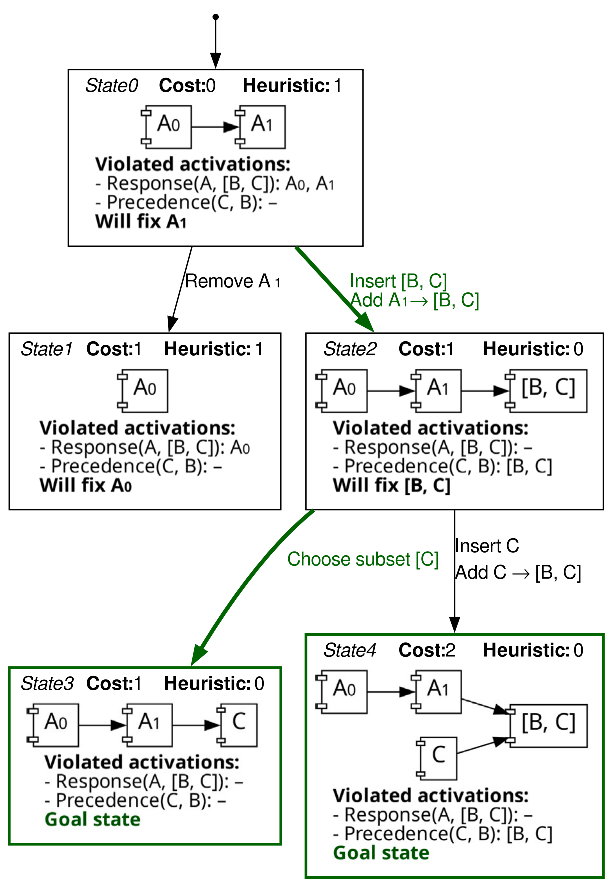


Fig. 7. Search space discovered for the running example. Best viewed in color.

4.3. Heuristic

The heuristic function used in DeclareAligner leverages knowledge about required repair actions to provide an accurate estimate of the remaining cost to reach a goal state. Given that all violated activations must be addressed, this approach is based on two key insights:

- **Minimum required actions**. At least one action from the set of repair actions suggested for each violated activation is necessary to resolve the violation.
- **Optimistic merging**. Actions that can fix multiple violations must be merged optimistically to avoid overestimating the remaining cost.

The heuristic function, formally defined in Algorithm 1, takes advantage of these insights to compute an accurate estimate of the remaining cost. The process begins by identifying all violated activations in the current state (alg. 1:2). For each violated activation, the set of proposed repair actions is retrieved (alg. 1:3).

Next, Dijkstra's algorithm (Dijkstra, 1959) is used to search for the combination of one action from each set that incurs the lowest total cost (alg. 1:4). The main loop of the search (alg. 1:8) involves four key functions:

- The RemoveLeastCost function (alg. 1:9) deletes and returns the unexplored heuristic state with the lowest total cost from the set of *hstates*, where the cost is determined by the sum of action costs. This state becomes the next candidate for exploration in the search process.
- The HActions function (alg. 1:12) selects the next violated activation that was not previously explored and adds all of its suggested fixes as new actions to discover new neighboring states.

Algorithm 1 Heuristic function

```
1:  function Heuristic(state)
2:      activations ← violatedActivations(state)
3:      actionSets ← {actions(a) | a ∈ activations}
4:      combination ← HDijkstra(actionSets)
5:      return ∑_{action∈combination} ct(action)
6:  function HDijkstra(actionSets)
7:      hstates ← {{}}                          ▷ Set of unexplored hstates
8:      loop                                    ▷ Main loop of Dijkstra's algorithm
9:          hstate ← RemoveLeastCost(hstates)
10:         if hstate is a goal state then
11:             return hstate
12:         hactions←HActions(hstate, actionSets)
13:         for all haction ∈ hactions do
14:             child ← HExplore(hstate, haction)
15:             AddOrReplace(hstates, child)
```

- The HExplore function (alg. 1:14) discovers a new *child* state from each of the proposed actions of the HActions function. It does so by adding the selected action to the current state and merging it with existing actions to avoid overestimating the cost.
- The AddOrReplace function (Alg. 1:15) checks whether the generated *child* state is already in *hstates*. If not, it adds the *child*; otherwise, it replaces the existing *hstate* only if the new one has a lower cost.

The search algorithm concludes upon exploring the first *hstate* that meets the goal criteria, namely, when the *hstate* includes at least one action from each activation (alg. 1:10). This is always found since any combination that includes actions from all activations is valid, and Dijkstra's algorithm ensures that the optimal one is found (alg. 1:11). Hence, the returned *hstate* is the combination of actions from all violated activations that yields the lowest cost. The resulting heuristic value is determined by this minimum possible cost of the combined actions (alg. 1:5). This value provides an informed estimate of the effort required to resolve all outstanding issues and achieve a valid solution.

Example (*Continued*)**.** Consider *state0* from Fig. 7. Two similar violated activations propose actions that either remove the activation or insert a new node after it, both of cost 1. However, the insert actions can be combined to solve both violations simultaneously. The heuristic searches for the lowest-cost combination of actions, which in this case is a single insert action of cost 1. This yields a heuristic value of 1 for *state0*.

4.4. Extraction of the optimal alignment

To extract the optimal alignment from the goal state, a topological sort for the LTGraph is found. A topological sort is a linear ordering of nodes in the LTGraph such that for every edge, the starting node comes before the ending node in the order. There can be multiple topological sortings for the nodes of an LTGraph, but the algorithm ensures that any one of them will have equivalent cost for a goal state. Since the objective is to obtain any optimal alignment, selecting any one of them results in the optimal alignment of the problem. This results in a sequence of activities executed in the model, where branched activities are resolved by selecting any one of them.

We then process both the nodes from the topological sort and the original trace simultaneously to extract the optimal alignment:

- If an LTGraph node was inserted when performing an action, a model move is added to the alignment and advance the topological sort.
- Otherwise:

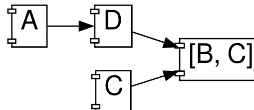


Fig. 8. Example of an LTGraph with branched activities.

Table 4
Example of optimal alignment extracted from the LTGraph shown in Fig. 8 and the trace $\langle A_0, D, A_1 \rangle$.

LOG	A	≫	D	≫	A
MODEL	A	C	D	C	≫

– If the next LTGraph node matches the next activity in the trace, a synchronous move is added to the alignment and advance both.
– Otherwise, a log move is added to the alignment and advance the trace.

Any remaining activities in the trace are handled by adding them as log moves. The collected sequence of moves is the optimal alignment.

Example. Let $\langle A, D, A \rangle$ be the trace and Fig. 8 be the LTGraph of the goal state from which the optimal alignment will be extracted. A valid topological sort of the LTGraph is $\langle A_0, C, D, C \rangle$, after selecting C from the branched activities $[B, C]$. This yields an optimal alignment where A_0 is a synchronous move, followed by a model move for the inserted activity C, another synchronous move for D, and another model move for the inserted activity C. Finally, since the last A from the trace was not processed, it results in a log move, producing the optimal alignment shown in Table 4.

4.5. Optimizations

Several optimization techniques can further improve the efficiency and effectiveness of the DeclareAligner algorithm. These enhancements build upon the foundation established in the previous sections and are designed to refine the performance of the algorithm.

4.5.1. Early pruning

The DeclareAligner algorithm improves search efficiency by pruning branches that cannot lead to a goal state. If any violated activation has no repair action, it is impossible to reach a goal state from that point. The reason behind this is that the violated activation will never be repaired even if all other violated activations are eventually repaired through the exploration of the search space. Since the heuristic already computes the suggested actions for all violated activations, the detection of dead-ends incurs no additional overhead.

To determine whether an action is feasible for a given state, a cycle detection algorithm is executed on the resulting LTGraph after applying the action. This is because the LTGraph represents a strict order, and any cycles would result in an impossible topological sort. In other words, if a cycle were present, it would indicate that there are conflicting requirements in the graph, making it impossible to extract a valid alignment from the given state. By detecting such cycles, actions that would lead to inconsistencies can be identified and pruned.

Example. The example illustrated in Fig. 9 demonstrates the benefits of early pruning. Initially, the end constraint is repaired by inserting a B activity at the end of the LTGraph. Upon exploring this second state, it is identified that the alternate succession constraint would require the insertion of a C after the B, and this cannot be satisfied without creating a cycle in the LTGraph ($B \rightleftarrows C$), indicating conflicting

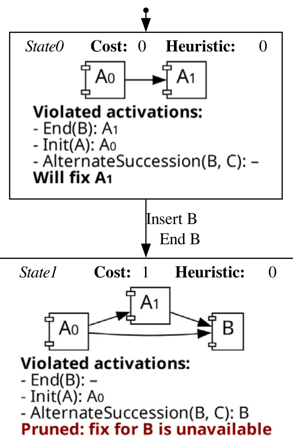


Fig. 9. Illustrative example highlighting the advantages of enabling the Early Pruning optimization, cutting down the search space from 7 discovered states to just 2. Best viewed in color.

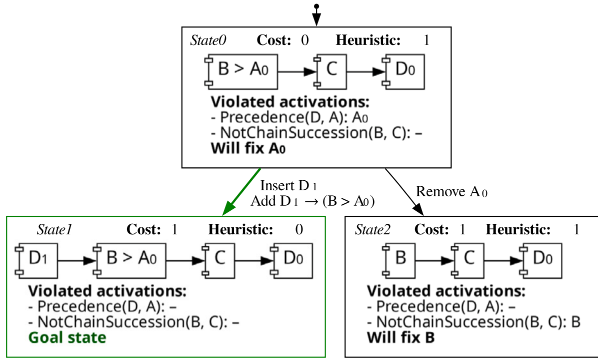


Fig. 10. This example shows how Constraint Preprocessing simplifies the search space by decreasing the number of discovered states from 9 to just 3. Best viewed in color.

requirements. As a result, this state is recognized as a dead-end, since repairing the alternate succession constraint is necessary to reach a goal state.

Without early pruning, the algorithm would proceed by addressing the violated A_0 activation of the init constraint, leading to the discovery of numerous additional neighbors before ultimately realizing that each of these states cannot reach a goal state due to the impossibility of solving the alternate succession constraint. In this simple scenario, enabling early pruning reduces the search space from 7 states to 2 states. For problems with more constraints, the difference is even more pronounced, as checking all constraints for dead-ends prevents unnecessary exploration and avoids realizing later that no solution exists.

4.5.2. Constraint preprocessing

The DeclareAligner algorithm can leverage preprocessing techniques to reduce problem complexity for certain constraints, specifically those containing "chain" in their name. By merging nodes in the LTGraph of the initial state that are affected by chain constraints, the algorithm can significantly improve efficiency while maintaining optimality. To achieve this, care must be taken to ensure that merged nodes can be split later if necessary.

This preprocessing step involves combining two consecutive chained activities into a single node in the initial state if they satisfy the constraint. By applying this preprocessing step before initiating the search, a substantial number of states can be eliminated from consideration, resulting in improved overall efficiency.

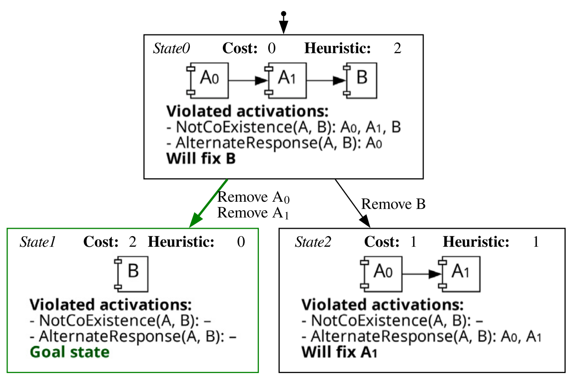


Fig. 11. Example search space demonstrating the benefits of the Grouped Fixes optimization, which reduces the number of discovered states from 7 to 3. Best viewed in color.

Example. Fig. 10 illustrates the impact of this optimization on the search space. In this example, the process model requires that immediately after an B, a C does not appear. Since this condition is met in the initial trace, the optimization merges the nodes for B and A_0 into a single LTGraph node.

If Constraint Preprocessing were disabled, the nodes of the LTGraph would still need to be merged to enforce the not chain succession constraint, necessitating an intermediate action. Consequently, the search space for this simple example would consist of 9 states instead of just 3 states. Notably, this optimization enables chain constraints to behave similarly to most other constraints: if there is no issue with the original trace, there is no need to generate repair actions during the search.

4.5.3. Grouped fixes

This optimization technique involves consolidating multiple fixes into a single action, thereby eliminating the need to propose individual fixes separately. This approach is particularly effective when dealing with complex actions that require a specific sequence of fixes. For instance, in cases where an activity must appear a certain number of times in the trace, grouping the necessary fixes into a single action allows for efficient insertion of all the required occurrences simultaneously.

Consolidating these fixes has a significant impact on the search space. By avoiding the generation of intermediate states, the algorithm reduces the need to explore many more unnecessary combinations of fixes.

Example. Fig. 11 illustrates the benefits of this optimization technique. Consider a scenario where the `NotCoexistence(A, B)` constraint is violated because both A and B appear in the trace. In this case, the only possible actions to resolve the violation are removing all instances of A or removing all instances of B. Thanks to grouped fixes, these actions can be performed in a single exploration step, allowing the algorithm to reach the goal state after discovering only 3 states.

In contrast, disabling this optimization would result in a significantly larger search space. For this simple example, the search space would consist of 7 states instead of just 3, highlighting the effectiveness of grouped fixes in improving the efficiency of the algorithm.

5. Evaluation

DeclareAligner is implemented in Kotlin and executed on the same Java Virtual Machine[1] as the other state-of-the-art algorithms that have been tested. Experiments are conducted on an Intel Xeon Gold 5220R CPU in single-threaded mode with 4GiB of RAM. The source code, dataset and binaries are available online.[2]

5.1. Datasets

The evaluation utilizes the same datasets as those in De Giacomo et al. (2023), extended with additional alignment problems to include all declare constraint templates.

- **D1. Real-life dataset from** De Giacomo et al. (2023). A personal loan application process log from a Dutch financial institute serves as the real-life dataset. Its process model contains 16 constraints and the dataset has a total of 854 trace-model pairs.
- **D2. Synthetic dataset from** De Giacomo et al. (2023). Synthetic logs are generated using three declare models with 10, 15, and 20 constraints. To introduce noise into the system, a modification strategy wherein a subset of constraints are replaced with their negative counterparts was employed. Specifically, 3, 4, or 6 constraints are randomly selected and substituted with their negated versions, while maintaining the same activities as arguments. For instance, the `RespondedExistence(A, B)` constraint may be replaced with its negation, `NotRespondedExistence(A, B)`. The log generator from Di Ciccio et al. (2015) produces four logs for each modified model, containing 100 traces of varying lengths (1–50, 51–100, 101–150, and 151–200 events). These noisy logs are aligned with the original models, totaling 3600 trace-model pairs.
- **D3. Extended dataset.** 16 declare templates were missing from the original test datasets, including alternate relation constraints, choice constraints, and various negative constraints. Additional synthetic log-model pairs are generated following the same procedure described in the previous paragraph, but ensuring all constraint templates are present in the results. This adds another 3600 trace-model pairs.

The complete dataset contains 8054 trace-model pairs.

5.2. Case study: Dutch financial institute

To demonstrate the practical utility of DeclareAligner, a concrete example from the loan application process of the Dutch financial institute, i.e., dataset D1, is considered. The complete process model contains 16 constraints that govern the behavior of the process. For simplicity, this case study focuses on three relevant constraints:

- `NotCoExistence(A_ACCEPTED, A_DECLINED)`: An application cannot be both accepted and declined at the same time.
- `NotSuccession(O_SELECTED, O_CREATED)`: An offer cannot be created at any point after it has been selected.
- `Succession(O_CREATED, O_SENT)`: After creating an offer, it must eventually be sent, and, if an order was sent, it must be eventually preceded by the creation of the order.

Since the recorded traces are too long to reproduce here, consider the following relevant subsequence of a trace extracted from the recorded log data: $\sigma = \langle$A_ACCEPTED, O_SELECTED, O_CREATED, O_SENT, O_SELECTED, O_CREATED, O_SENT, A_DECLINED$\rangle$. Running DeclareAligner on the full model and trace detects several errors in the traceand returns the optimal alignment shown in Table 5. This alignment includes three asynchronous moves: log move on A_ACCEPTED, and two log moves on O_SELECTED. These errors indicate that the activities corresponding to these moves should not have been performed in the trace, as they are incompatible with the constraints of the model. The algorithm determines that the cheapest way to reach conformance with the model is to not perform them.

[1] OpenJDK Temurin-22.0.1+8.

[2] https://apps.citius.gal/ltgraph/.

Table 5
Optimal alignment of the case study.

LOG	A_ACCEPTED	O_SELECTED	O_CREATED	O_SENT	O_SELECTED	O_CREATED	O_SENT	O_DECLINED
MODEL	≫	≫	O_CREATED	O_SENT	≫	O_CREATED	O_SENT	O_DECLINED

Table 6
Ablation study results. `Time` and `ExpStates` represent the average over the complete dataset. `Timeouts` is the number of trace-model pairs for which the time limit was reached. `Reduction` values indicate improvements relative to the baseline algorithm, expressed as percentages for `Time` and `ExpStates`, and absolute differences for `Timeouts`.

EP	CP	GF	Time (s)	Time reduction	ExpStates	ExpStates reduction	Timeouts	Timeout reduction
			105.8	–	3568.6	–	2674	–
✔			68.9	34.8%	510.4	85.7%	1689	985
	✔		76.8	27.4%	2664.7	25.3%	1973	701
		✔	54.2	48.8%	984.7	72.4%	1170	1504
✔	✔		55.2	47.8%	764.8	78.6%	1348	1326
✔		✔	36.4	65.6%	187.5	94.7%	706	1968
	✔	✔	17.3	83.7%	423.9	88.1%	359	2315
✔	✔	✔	**7.6**	**92.8%**	**107.0**	**97.0%**	**113**	**2561**

The optimal alignment chosen by DeclareAligner resolves the contradiction between accepting and declining an application, which violates the "an application cannot be both accepted and declined at the same time" business rule. Additionally, the trace shows two instances of an offer being selected, created, and sent, which contradicts the constraint "an offer cannot be created at any point after it has been selected". An alternative alignment would require a log move on the creation of the offer instead, but this would violate the "if an order was sent, it must be eventually preceded by the creation of the order" constraint, necessitating further repairs in the trace. DeclareAligner carefully considers all possible alignments to ensure that the returned solution is optimal.

The detected problems are common issues in this loan application process. Many applications are accepted and later declined after a lengthy review process, causing confusion and delays. Furthermore, offers are often selected before they are created, suggesting issues with internal procedures. These problems can have serious consequences, such as incorrect or unfair treatment of applicants, and can damage the reputation of the financial institute. By using DeclareAligner to analyze process execution and detect errors, businesses can improve the quality and consistency of their processes, providing better service to their customers.

5.3. Ablation study

This ablation study evaluates the impact of each component in DeclareAligner to determine how they individually contribute to its overall performance. Various metrics are collected for each trace-model pair, such as execution time (`Time`), expanded states (`ExpStates`), and the timeouts (`Timeouts`). The results of the ablation study are shown in Table 6.

Early Pruning (Section 4.5.1) results in a significant decrease of 34.8% in average execution times. Furthermore, it enables the computation of many more alignments, reducing the number of timeouts by 985.

This improvement is mainly due to the pruning of a large number of states that would have led to unsolvable constraints, reducing the number of expanded states by 85.7%. By avoiding unnecessary computations and reducing the search space, this optimization ultimately is capable of computing complex alignments that would not be possible otherwise.

Example. To demonstrate the effectiveness of the optimizations, a specific example from dataset D2[3] is considered. This process model comprises 10 constraints, representing a diverse range of constraint templates. The trace itself contains 50 events, and its optimal alignment has a cost of 4. Without any optimizations enabled, the algorithm computes the alignment in 1.44 s, expanding 530 states in the process. This provides a useful reference point for evaluating the benefits of each optimization, illustrating why each of the proposed optimizations is particularly effective for this concrete example.

Enabling only Early Pruning significantly reduces the search space, decreasing the number of expanded states to 46 and the search time to approximately 0.46 s. Although this optimization prunes dead-ends by an order of magnitude, its impact on execution time is limited to a factor of three due to the additional overhead of checking all constraints for violations and suggesting repairs at each state expansion. However, as demonstrated by the comprehensive evaluation, the benefits of Early Pruning outweigh this extra cost, leading to improved overall performance.

Constraint Preprocessing (Section 4.5.2) yields a significant decrease in execution time (27.4%). This improvement is accompanied by a considerable reduction in the number of timeouts, with 701 fewer cases timing out.

The primary reason for this improvement is that constraint preprocessing allows the algorithm to start from a more advanced initial state, thereby avoiding the generation of numerous unnecessary states (25.3%). By performing shortcuts for highly likely operations upfront, this optimization reduces the overall search space, making it possible to compute alignments more efficiently.

Example. Continuing the running example, enabling only Constraint Preprocessing yields a significant reduction in search space, decreasing the number of expanded states to 15 and the search time to approximately 0.09 s. This optimization is particularly effective in this case due to the presence of multiple chain constraints in the process model: `ChainResponse(act. 14, act. 15)`, `ChainPrecendence(act. 16, act. 17)`, and `NotChainSuccession(act. 22, act. 23)`. By merging activities that appear consecutively in the trace and satisfy these constraints into a single LTGraph node during preprocessing, the algorithm avoids suggesting multiple repair actions during the search process, thereby preventing unnecessary widening of the search space. The fact that each of these constraints is activated multiple times in the example trace further amplifies the benefits of this optimization, making it a key factor in achieving the observed performance gains.

Grouped Fixes (Section 4.5.3) leads to a substantial reduction in execution time (48.8%) and a decrease in timeouts by 1504 cases. This improvement is primarily due to the fact that grouped fixes enable the algorithm to prune the search space more effectively, reducing

[3] The example consists of the process model defined in `synthetic/10_constraints/10Constraints.xml` and the trace labeled as "Synthetic trace no. 33" from the log file `.../3_constraints_inverted/log-1-50.xes.gz`.

the number of expanded states by 72.4% and avoiding unnecessary exploration of intermediate states and their neighbors.

Example. Continuing the running example, enabling only Grouped Fixes yields a significant reduction in search space, decreasing the number of expanded states to 123 and the search time to approximately 0.53 s. A notable aspect of this example is the presence of a `Response(act. 11, act. 12)` constraint, which is initially violated six times in the given trace. Fortunately, each repair can be performed in a single, consolidated action that simultaneously inserts a node and adds an arc to enforce the desired behavior, effectively positioning the added activity in the correct region of the trace. By grouping these fixes together, the algorithm avoids generating intermediate states for this constraint, as well as several others in the example, resulting in a more efficient search process and improved overall performance.

Pairing any two optimizations generally leads to positive outcomes. This can be attributed to the fact that each optimization targets a distinct aspect of the search space, thereby reducing the number of expanded states required. Constraint Preprocessing focuses on moving the initial state closer to the goal, whereas Grouped Fixes avoid intermediate states when expanding neighbors. Meanwhile, Early Pruning eliminates dead-ends in the search space, preventing unnecessary exploration.

Combining all three optimizations yields the most substantial improvements across all tested metrics. The average execution time decreases by 92.8%, the expanded states are reduced by 97.0%, and timeouts are reduced by 2561 cases. By integrating these three optimizations, DeclareAligner is able to leverage their complementary strengths, resulting in a more efficient search strategy.

Example. To conclude the running example, enabling all three optimizations yields the most impressive performance gains, with a mere 5 expanded states and a search time of approximately 0.03 s. This shows how the optimizations are complementary and can be combined to achieve significant performance improvements.

5.4. State-of-the-art comparison

The remainder of the evaluation section assesses the performance of the DeclareAligner algorithm in comparison to the state-of-the-art methods. The state-of-the-art techniques for aligning event logs with declarative process models are DeclareReplayer (Leoni et al., 2012) and PlannerBA/PlannerFD (Giacomo et al., 2017). Both methods rely on an initial transformation of the Declare constraints into automata, which are then utilized to guide the search algorithm. This sequential nature of automata execution leads to a search space that is built by incrementally appending alignment moves from start to finish. A key difference between the two approaches lies in their implementation. Leoni et al. (2012) employs the A* algorithm directly, whereas Giacomo et al. (2017) transforms the generated automata into classical planning problems, leveraging the capabilities of planning technology to solve the alignment problem. This distinction gives Giacomo et al. (2017) an edge in certain scenarios, as the planning-based approach can efficiently handle complex search spaces. To provide a comprehensive understanding, this evaluation is supplemented by additional baselines:

- All proper subsets of optimizations described in Section 4.5 are considered, enabling only the optimizations in each subset.
- A naive algorithm that explores all possible alignments to find the optimal one.

To gain a deeper understanding of the interplay between optimization techniques, the DeclareAligner algorithm is evaluated under various configurations, each characterized by a unique combination of enabled optimizations. These are:

- B-None: no optimizations from Section 4.5 are applied.
- B-EP: only Early Pruning (EP) is enabled.
- B-CP: only Constraint Preprocessing (CP) is enabled.
- B-GF: only Grouped Fixes (GF) is enabled.
- B-EP-GF: both Early Pruning (EP) and Grouped Fixes (GF) are enabled.
- B-EP-CP: both Early Pruning (EP) and Constraint Preprocessing (CP) are enabled.
- B-CP-GF: both Constraint Preprocessing (CP) and Grouped Fixes (GF) are enabled.

Note that enabling all optimizations simultaneously results in the full algorithm, labeled as DeclareAligner. By systematically exploring these different configurations, this evaluation aims to elucidate the contribution of each optimization technique to the overall efficiency and effectiveness of the DeclareAligner algorithm, while also showcasing its improvements in the context of state-of-the-art performance.

A naive algorithm was implemented as another baseline that solves the optimal alignment problem for declare process models and event logs. It leverages the A* search algorithm, starting from an initial state that represents an empty alignment, where no moves have been made. From this state, it generates neighboring states by considering all possible moves:

- If there are remaining events in the trace, where *act* is the next activity:
  - Generate a neighboring state by appending a synchronous move on *act*.
  - Generate a neighboring state by appending a log move on *act*.
- For each activity *act* present in the model, generate a neighboring state by appending a model move on *act*.

A state within this search space is considered a goal state if two conditions are met: there are no remaining events in the trace, and the model accepts the given trace. This acceptance check relies on a simple Declare model checker, which determines whether a trace perfectly conforms to the process model. The A* search algorithm guides the selection and expansion of neighboring states, albeit without utilizing a heuristic (i.e., the heuristic function always returns 0). This procedure repeats until a goal state is reached, at which point the optimal alignment is directly represented by this state. However, due to its simplicity, this naive approach generates an excessively large search space, needing substantial resources for exploration and rendering it impractical for realistic alignment problems.

Table 7 summarizes the results of the state-of-the-art approaches and DeclareAligner divided by each of the used dataset sources, and the aggregated total for the complete dataset. The results unequivocally demonstrate the superiority of DeclareAligner over existing state-of-the-art techniques, as it emerges as the top performer across all evaluated metrics. Notably, DeclareAligner achieves a significant lead in all categories, with the sole exception of the number of timeouts in the D2 dataset, where it narrowly trails behind the leading algorithm. However, even in this instance, DeclareAligner delivers substantial reductions in average execution time, underscoring its efficiency and scalability. Further analysis shows that most of the cases where the execution time of DeclareAligner is close to the state of the art are the simplest alignment problems in the dataset. This can be attributed to the initialization phase required by some proposed optimizations, which can introduce a delay for trivial alignments but prove highly beneficial for complex instances.

As anticipated, the performance of the naive algorithm is notably subpar, with execution times substantially exceeding those of DeclareAligner and other state-of-the-art algorithms. Specifically, it reaches the 5 min time limit for 6974 trace-model pairs. In contrast, the ablated

Table 7
Comparison of the average execution times and number of timeouts reached for each tested algorithm on the evaluated dataset, presented split by data source and as an aggregated total.

Dataset:	D1		D2		D3		All	
Algorithm	Time (s)	Timeouts	Time (s)	Timeouts	Time (s)	Timeouts	Time (s)	Timeouts
Naive	56.0	345	281.9	3556	273.9	3073	260.0	6974
PlannerFD	6.3	411	117.4	832	124.2	703	111.4	1946
DeclareReplayer	0.2	**0**	102.1	804	55.0	384	72.8	1188
PlannerBA	8.5	355	34.3	**0**	65.6	45	46.2	400
B-None	1.4	117	182.2	1839	44.1	718	105.8	2674
B-EP	**0.1**	**0**	131.9	1403	15.0	286	68.9	1689
B-CP	1.4	121	120.8	1223	44.0	629	76.8	1973
B-GF	0.2	6	104.4	876	11.0	288	54.2	1170
B-EP-GF	**0.1**	2	73.9	566	**3.3**	138	36.4	706
B-EP-CP	**0.1**	37	102.9	1011	15.0	300	55.2	1348
B-CP-GF	0.2	6	26.2	196	11.0	157	17.3	359
DeclareAligner	**0.1**	**0**	**13.0**	88	**3.3**	**25**	**7.6**	**113**

Table 8
Ranking and statistical tests comparing execution times over the complete dataset.

(a) Average ranking of each tested algorithm.

Algorithm	Rank
DeclareAligner	**1.24**
DeclareReplayer	2.73
PlannerBA	3.12
PlannerFD	3.55
Naive	4.37

(b) Results of the Friedman test and Holm post-hoc analysis, indicating overall differences between algorithms and specific pairwise differences.

Statistical test	p-value
DeclareAligner vs. all	$<10^{-6}$
DeclareAligner vs. DeclareReplayer	$<10^{-6}$
DeclareAligner vs. PlannerBA	$<10^{-6}$
DeclareAligner vs. PlannerFD	$<10^{-6}$
DeclareAligner vs. Naive	$<10^{-6}$

versions of DeclareAligner manage even to surpass the state-of-the-art algorithms on some datasets and optimization combinations.

The results reveal that Early Pruning reduces average execution times from 105.8 to 68.9 s. This optimization also leads to a reduction in timeouts from 2674 to 1689, showcasing the effectiveness in reducing the search space that this optimization achieves. Constraint Preprocessing also yields significant improvements by allowing the algorithm to start from a more advanced initial state: it reduces the execution times from 105.8 to 76.8 s and the number of timeouts from 2674 to 1973. Lastly, Grouped Fixes avoids unnecessary exploration of intermediate states and their neighbors, reducing the execution times from 105.8 to 54.2 s and the number of timeouts from 2674 to 1170. Enabling any two optimizations at the same time contributes to improving the performance even further. Moreover, when all optimizations are combined (DeclareAligner), the algorithm achieves performance that is superior to its ablated counterparts across all metrics, with an average time of just 7.6 s and the number of timeouts to just 113. This suggests that the synergistic effect of combining the proposed optimizations yields a more efficient and effective alignment algorithm, capable of outperforming existing state-of-the-art solutions in various scenarios.

To further substantiate the comparison, a ranking of all algorithms based on their execution times is computed. For each sample, the algorithms are ranked, with ties being assigned the average rank. These ranks are then averaged across all samples to obtain an overall ranking for each algorithm. To ensure that the results are statistically significant, the Friedman test is performed to assess overall differences between the algorithms. Subsequently, the Holm post-hoc test to identify specific pairwise differences. Notably, to avoid skewing the results, the variants of the proposed algorithm were excluded from the rankings and statistical tests. This ensures the focus remains on the main algorithm and prevents artificially high ranks that could misrepresent comparative performance.

The ranking and statistical tests are presented in Table 8. Prior to conducting these tests, it was verified that the necessary assumptions for their validity were met. Specifically, the ranking data consists of matched samples, as each algorithm's performance is evaluated on the same set of logs and models. Additionally, the samples are independent of one another, meaning that the performance of one algorithm on a given sample does not influence its performance on another sample.

We also note that the Friedman test does not require normality of the data, which is beneficial in this case since execution times may not follow a normal distribution. Furthermore, the Friedman test does not assume variance homogeneity, making it suitable for these experiments even if the variances of the execution times differ across algorithms.

As expected, the ranks show the clear advantage in execution times of DeclareAligner. The p-values indicate that the differences in performance among the algorithms are statistically significant, demonstrating that the proposed approach outperforms others in terms of computational efficiency. To quantify the magnitude of this effect, Kendall's W was calculated, a measure of the degree of agreement between the different execution times. The resulting effect size is 0.014. This is very low, which indicates that the performances of the algorithms do not agree with each other, suggesting that there are substantial differences between them. In fact, this lack of agreement is confirmed by the post-hoc test, which reveals significant pairwise differences between the performances of DeclareAligner and the state of the art.

To illustrate the efficiency and scalability of DeclareAligner compared to state-of-the-art algorithms, Fig. 12 shows the number of trace-model pairs (y-axis) that can be solved within a given execution time (x-axis). Each algorithm is depicted as a distinct line, allowing for easy comparison of their performance across different time thresholds and number of logs solved.

Fig. 12 reveals that DeclareAligner significantly outperforms the state-of-the-art algorithms in terms of execution time. DeclareReplayer is also really fast to align the simplest trace-model pairs, even being capable of outperforming DeclareAligner for some of the most straightforward alignments. However, its performance quickly degrades with complexity of the input problem, showing the superior scalability of DeclareAligner. At around 17 s, PlannerBA overtakes DeclareReplayer in terms of the number of trace-model pairs solved, taking second place. This occurs because both planner-based techniques require some initial time to convert the alignment task into a classic planning problem and the planner also needs to perform some preprocessing optimizations in order to solve harder problems faster. The sequential construction of the search space, inherent to both state-of-the-art approaches due to their reliance on automata, can lead to inefficiencies in exploring the vast possible search space of alignments. In contrast, the more direct approach to generating repair actions chosen by DeclareAligner yields improved performance, as illustrated in this figure.

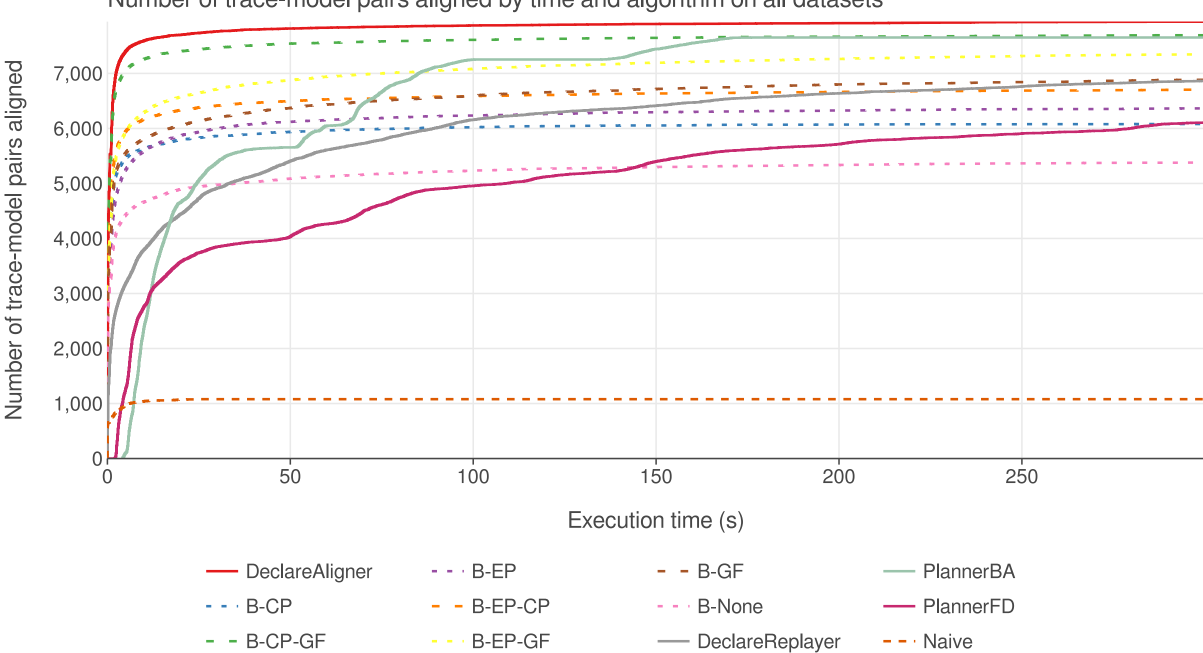


Fig. 12. Number of trace-model pairs successfully aligned by each algorithm. Best viewed in color.

The inclusion of ablated variants and the naive algorithm in the plots provides additional insights into the performance characteristics of DeclareAligner. Notably, the ablated variants exhibit reduced efficiency and scalability compared to DeclareAligner, highlighting the importance of the proposed optimizations. The naive algorithm, on the other hand, demonstrates poor performance across all datasets, aligning only 1080 pairs when all datasets are considered, with no significant improvement over time. In contrast, DeclareAligner maintains its superior performance over all the alternatives, effectively leveraging its optimizations to achieve efficient alignments even in complex scenarios. These results underscore the value of the carefully designed optimizations in achieving scalable and efficient optimal alignments.

A notable achievement of DeclareAligner is its ability to align 90% of the tested trace-model pairs in 3.5 s or less per alignment, outperforming the next best state-of-the-art algorithm (PlannerBA) which requires up to 100 s per problem to reach the same number of solutions. Furthermore, the proposed optimizations enable DeclareAligner to successfully compute alignments for 231 log-model pairs that remain unsolved by all other algorithms within a 5-min time limit.

To offer a clearer and more detailed interpretation of the results, Fig. 13 presents four supplementary subfigures that enhance the analysis provided in Fig. 12. These plots focus specifically on a shorter time limit of 20 s, making performance differences between the tested algorithms more apparent. Each subfigure isolates a specific dataset source, as introduced earlier in this section. Fig. 13(a) displays the aggregated results across all datasets, effectively serving as a zoomed-in version of Fig. 12 and clearly highlighting the efficiency gains achieved by DeclareAligner.

The remaining supplementary figures facilitate a detailed examination of performance differences within each dataset category. Fig. 13(b) highlights DeclareAligner's efficiency in handling real-life data, demonstrating its strong capability to align process models effectively in practical scenarios. While the differences between DeclareAligner and DeclareReplayer are relatively small in this case, the planning-based techniques exhibit noticeable delays, primarily due to the overhead introduced during the problem translation phase. Fig. 13(c) showcases DeclareAligner's performance on synthetic data previously employed in state-of-the-art evaluations, providing a controlled setting to assess the algorithm's scalability and robustness. Fig. 13(d) complements this by illustrating DeclareAligner's behavior on the extended synthetic dataset, which incorporates a wider variety of process model complexities through the use of additional constraint templates. In both cases, DeclareAligner demonstrates a clear advantage, successfully aligning nearly all trace-model pairs within the time limit of 20 s. In contrast, other state-of-the-art algorithms plateau significantly lower, typically aligning only about half of the available trace-model pairs, underscoring DeclareAligner's superior efficiency and reliability in more demanding scenarios.

6. Conclusions

This work introduces DeclareAligner, an A*-based algorithm for efficiently computing optimal alignments of declarative process models. The contributions of this research are twofold. Firstly, a novel algorithmic approach has been proposed to leverage the inherent flexibility of declarative processes in calculating optimal alignments. Secondly, additional search optimizations and techniques have been developed and integrated to effectively handle complex scenarios.

The evaluation, conducted on 8054 trace-model pairs from real-world and synthetic processes, demonstrates the substantial performance improvements of the proposed method over the state of the art. Notably, DeclareAligner successfully aligns 7941 pairs within the 5 minutes time limit, outperforming PlannerBA (7654), DeclareReplayer (6866), and PlannerFD (6108). Moreover, the proposed optimizations enable DeclareAligner to compute alignments for 231 pairs that are unsolvable by other algorithms within the same time frame. The results show that DeclareAligner achieves faster alignment times than any other algorithm for 7000 of the evaluated trace-model pairs. The significance of this research lies in its ability to facilitate efficient conformance checking while preserving the accuracy of optimal alignments, ultimately leading to improved process quality and reduced costs through effective detection and diagnosis of log-related issues.

While the proposed algorithm has demonstrated its effectiveness, it still has some limitations, particularly with regards to scalability for large-scale models comprising hundreds of constraints, where the explosion of intermediate states resulting from numerous possible paths to optimal alignments poses a significant challenge. To fully realize the algorithm's potential, additional research is necessary to investigate

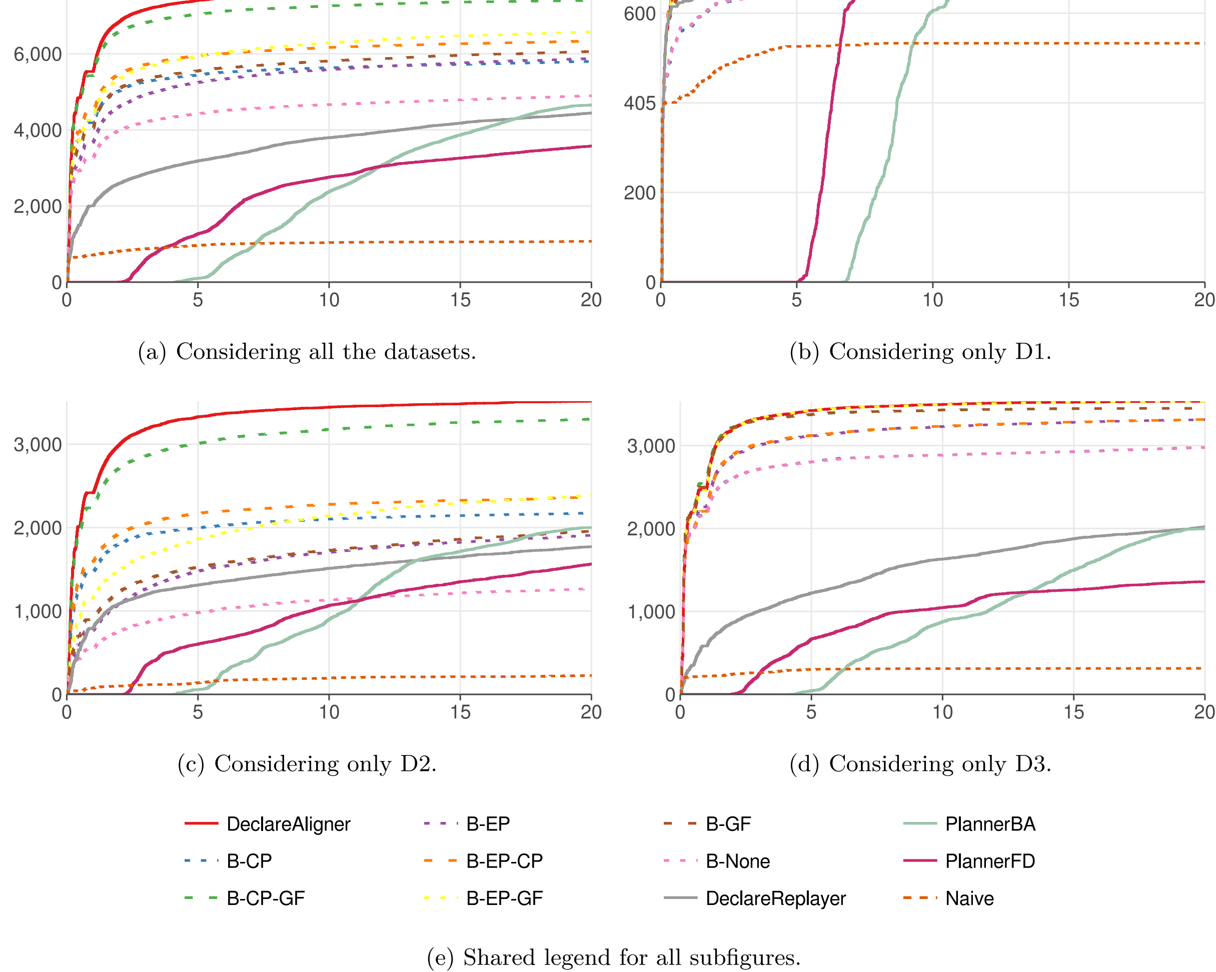


Fig. 13. Number of trace-model pairs successfully aligned by each algorithm in less than 20 s, also subdivided by dataset. Best viewed in color.

its performance on complex models and develop more strategies to enhance its efficiency. Furthermore, when applying the algorithm to real-world logs, it is essential to consider the ethical implications, particularly with regard to sensitive data, and ensure that privacy and security are maintained through measures such as anonymizing log data or aggregating it to obscure sensitive information, thereby prioritizing data protection and upholding the highest standards of ethical responsibility.

In future research, the focus will shift towards adapting the proposed algorithm to accommodate data-aware declarative process models, ensuring retention of its high performance. Through extension to manage intricate data attributes and relationships, the goal is to offer a scalable conformance checking solution suitable for real-world applications demanding both precision and efficiency.

CRediT authorship contribution statement

Jacobo Casas-Ramos: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Manuel Lama:** Writing – review & editing, Supervision, Resources, Methodology, Funding acquisition, Conceptualization. **Manuel Mucientes:** Writing – review & editing, Supervision, Resources, Methodology, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research was partially funded by the Spanish Ministerio de Ciencia e Innovación [grant numbers PID2023-149549NB-I00, TED2021-130374B-C21]. These grant are co-funded by the European Regional Development Fund (ERDF). Jacobo Casas-Ramos is supported by the Spanish Ministerio de Universidades under the FPU national plan [grant number FPU19/06668].

Data availability

Access to the research data and code is available at https://apps.citius.gal/ltgraph/.

References

Aalst, W.M.P. van der, 1997. Verification of workflow nets. In: Azéma, P., Balbo, G. (Eds.), Application and Theory of Petri Nets 1997. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 407–426.

Aalst, W. van der, Adriansyah, A., de Medeiros, A.K.A., Arcieri, F., Baier, T., Blickle, T., Bose, J.C., Brand, P. van den, Brandtjen, R., Buijs, J., et al., 2012. Process mining manifesto. In: Business Process Management Workshops: BPM 2011 International Workshops. Springer, pp. 169–194.

Adriansyah, A., 2014. Aligning observed and modeled behavior.
Back, C.O., Debois, S., Slaats, T., 2018. Towards an empirical evaluation of imperative and declarative process mining. In: Woo, C., Lu, J., Li, Z., Ling, T.W., Li, G., Lee, M.L. (Eds.), Advances in Conceptual Modeling. Springer International Publishing, Cham, pp. 191–198.
Bergami, G., Maggi, F.M., Marrella, A., Montali, M., 2021. Aligning data-aware declarative process models and event logs. In: Business Process Management. Springer International Publishing, pp. 235–251.
Burattin, A., Maggi, F.M., Sperduti, A., 2016. Conformance checking based on multi-perspective declarative process models. Expert Syst. Appl. 65, 194–211.
Carmona, J., van Dongen, B., Solti, A., Weidlich, M., 2018. Conformance Checking: Relating Processes and Models. Springer.
Casas-Ramos, J., Mucientes, M., Lama, M., 2024. REACH: Researching efficient alignment-based conformance checking. Expert Syst. Appl. 241, 122467.
Chiariello, F., Maggi, F.M., Patrizi, F., 2022. ASP-based declarative process mining. In: Thirty-Sixth AAAI Conference on Artificial Intelligence. AAAI 2022, AAAI Press, pp. 5539–5547.
Christfort, A.K.F., Slaats, T., 2023. Efficient optimal alignment between dynamic condition response graphs and traces. In: Business Process Management. Springer-Verlag, Berlin, Heidelberg, pp. 3–19.
De Giacomo, G., Fuggitti, F., Maggi, F.M., Marrella, A., Patrizi, F., 2023. A tool for declarative trace alignment via automated planning. Softw. Impacts 16, 100505.
De Smedt, J., Vanden Broucke, S.K.L.M., De Weerdt, J., Vanthienen, J., 2015. A full r/i-net construct lexicon for declare constraints. SSRN Electron. J..
Di Ciccio, C., Bernardi, M.L., Cimitile, M., Maggi, F.M., 2015. Generating event logs through the simulation of declare models. In: Barjis, J., Pergl, R., Babkin, E. (Eds.), Enterprise and Organizational Modeling and Simulation. Springer International Publishing, Cham, pp. 20–36.
Dijkstra, E.W., 1959. A note on two problems in connexion with graphs. Numer. Math. 1, 269–271.
Donadello, I., Riva, F., Maggi, F.M., Shikhizada, A., 2022. Declare4Py: A python library for declarative process mining. In: Proceedings of the Best Dissertation Award, Doctoral Consortium, and Demonstration & Resources Track BPM 2022, CEUR Workshop Proceedings. CEUR-WS.org, pp. 117–121.
Dongen, B.F. van, 2018. Efficiently computing alignments. In: Business Process Management. Springer International Publishing, Cham, pp. 197–214.
Dongen, B. van, Carmona, J., Chatain, T., Taymouri, F., 2017. Aligning modeled and observed behavior: A compromise between computation complexity and quality. In: 29th International Conference on Advanced Information Systems Engineering. CAiSE, Springer, pp. 94–109.
Dongen, B.F. van, De Smedt, J., Di Ciccio, C., Mendling, J., 2021. Conformance checking of mixed-paradigm process models. Inf. Syst. 102, 101685.
Giacomo, G.D., Maggi, F.M., Marrella, A., Patrizi, F., 2017. On the disruptive effectiveness of automated planning for LTLf-based trace alignment. In: Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 31.
Giannakopoulou, D., Havelund, K., 2001. Automata-based verification of temporal properties on running programs. In: Proceedings 16th Annual International Conference on Automated Software Engineering. ASE 2001, IEEE Comput. Soc..
Hart, P., Nilsson, N., Raphael, B., 1968. A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. 4, 100–107.
Lee, W.L.J., Verbeek, H.M.W., Munoz-Gama, J., Aalst, W.M.P. van der, Sepúlveda, M., 2018. Recomposing conformance: Closing the circle on decomposed alignment-based conformance checking in process mining. Inform. Sci. 466, 55–91.
Leoni, M. de, Maggi, F.M., Aalst, W.M.P. van der, 2012. Aligning event logs and declarative process models for conformance checking. In: Lecture Notes in Computer Science. Springer, Berlin Heidelberg, pp. 82–97.
Maggi, F.M., Marrella, A., Patrizi, F., Skydanienko, V., 2023. Data-aware declarative process mining with SAT. ACM Trans. Intell. Syst. Technol. 14, 1–26.
Maggi, F.M., Montali, M., Bhat, U., 2019. Compliance monitoring of multi-perspective declarative process models. In: Dijkman, R., Si-Said, S. (Eds.), Proceedings of the 23rd IEEE International Enterprise Distributed Object Computing Conference. EDOC 2019, IEEE Computer Society, pp. 151–160.
Maggi, F.M., Montali, M., Westergaard, M., Aalst, W.M.P. van der, 2011. Monitoring business constraints with linear temporal logic: An approach based on colored automata. In: International Conference on Business Process Management. Springer, Springer Berlin Heidelberg, pp. 132–147.
Montali, M., Maggi, F., Chesani, F., Mello, P., Aalst, W., 2013. Monitoring business constraints with the event calculus. ACM Trans. Intell. Syst. Technol. 5, 1–30.
Pesic, M., Schonenberg, H., Aalst, W.M.P. van der, 2007. DECLARE: Full support for loosely-structured processes.
Riva, F., Benvenuti, D., Maggi, F.M., Marrella, A., Montali, M., 2023. An sql-based declarative process mining framework for analyzing process data stored in relational databases. In: International Conference on Business Process Management. Springer, pp. 214–231.
Sani, M.F., Zelst, S.J. van, Aalst, W.M.P. van der, 2020. Conformance checking approximation using subset selection and edit distance. In: 32nd International Conference on Advanced Information Systems Engineering. CAiSE, Springer, pp. 234–251.
Taymouri, F., Carmona, J., 2020. Computing alignments of well-formed process models using local search. ACM Trans. Softw. Eng. Methodol. 29, 1–41.
Vespa, M., Bellodi, E., Chesani, F., Loreti, D., Mello, P., Lamma, E., Ciampolini, A., Gavanelli, M., Zese, R., 2025. Probabilistic traces in declarative process mining. In: AIxIA 2024 – Advances in Artificial Intelligence. Springer Nature, Switzerland, pp. 330–345.
Westergaard, M., 2011. Better algorithms for analyzing and enacting declarative workflow languages using LTL. In: Business Process Management: 9th International Conference, BPM 2011, Clermont-Ferrand, France, August 30-September 2, 2011. Proceedings 9. Springer, pp. 83–98.