Report Reasoning Agents

Alessia Carotenuto 1764400
Emanuele Iacobelli 1657799
Veronica Romano 1580844

July 27, 2021

Chapter 1

Introduction

The trace alignment problem comes from the Business Process Management (BPM) discipline. A definition of this problem can be found in the paper [van der Aalst, 2016][8]: "The Trace Alignment is the problem of "cleaning" and "repairing" dirty traces to make them compliant with the underlying process model". Basically, this problem consists in checking if sequences (called traces) of single actions (called events) conform to the expected process behavior and if not, changing as little as possible the traces, by adding or removing events, in order to align them to the standard process. In the paper "On the Disruptive Effectiveness of Automated Planning for LTL_f-based Trace Alignment" [De Giacomo,Maggi,Marrella,Patrizi] [1], that we have studied for this project, the authors provided a technique to synthesize the alignment instructions relying on finite automata theoretic manipulations. This technique can be effectively implemented by using planning technology which allows to outperform the results obtained with ad-hoc alignment systems. Specifically, in [1] the traces analyzed are sequences of activity names, while the process behavior is specified by using a declarative approach, i.e. through some constraints defined in LTL_f (Linear Temporal Logic over finite traces [De Giacomo and Vardi 2013][4]). The goal of the trace alignment problem is to make the traces conform with the process by satisfying the constraints defined in LTL_f. The proposed approach used the cost-optimal planning to find a successful plan of minimal cost to achieve this goal. In this type of planning the actions have a cost, in particular, adding and removing events have a non-zero cost while moving on the original trace has a zero cost. As previously said, the standard process is defined with a declarative approach, specifically through constraints defined with LTL_f formulas coming from DECLARE models. In detail, a DECLARE model consists of a set of constraints (i.e. rule templates applied to activities) and their semantics can be formalized using LTL_f in order to make them verifiable and executable. The encoding of the trace alignment problem, proposed in [1], is performed by using PDDL and the results are compared with a technique specifically tailored for the alignment problem (de Leoni, Maggi, and van der Aalst 2012; 2015)[6] [5] and previous approaches based on classical planning (De Giacomo et al. 2016) [2]. For our Reasoning Agent's project we implemented the solution described in [1] and we improved it by slightly modifying the proposed encoding. Also, for the experiments, differently from the analyzed paper, we created a general script that can be used with any kind of LTL_f formula, i.e not only with the one coming from DECLARE models. We were able to use any LTL_f formula thanks to the LTLf2DFA tool, developed by Francesco Fuggitti, which allows us to create from a general LTL_f formula the associated DFA. Finally, we compared the results obtained with our implementation of the encoding proposed in the paper, our modified version of the proposed encoding and the results shown in the paper.

Chapter 2

The Studied Paper

2.1 How to Define Business Processes and DECLARE Models

As said in the Section 1, the Trace Alignment problem comes from BPM discipline in which checking properties on finite traces is very important. This discipline works a lot with high-level processes that are used to organize finite tasks at a high-level. For instance, for an agent, a series of high-level tasks can be: “go to the office”, “pick a book”, “go to the boss’ office”, “leave the book” and so on. In detail, business processes can be defined with a procedural or a declarative approach. The procedural approach is well suited for the actual execution of the process which is fully specified through some models (i.e Finite-state automaton, Petri Net and so on) that describe what the process does. While, the declarative approach is well suited to various forms of analysis of the process which is specified by a set of constraints and any trace that satisfies these constraints is an allowed behaviour of the process. In particular, these constraints can be expressed in terms of DECLARE formulas or with LTL_f/LDL_f formulas. Going more into the details, DECLARE is a well-established declarative process modeling language, which can be seen as a dialect of LTL_f. This means that we can generalize the constraints by using only LTL_f formulas. To have a graphical visualization of the difference between the procedural and declarative approaches, the Figure 2.1 shows:

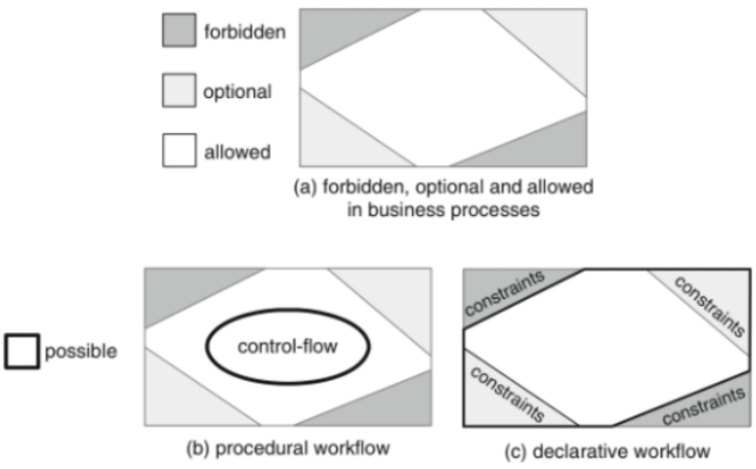


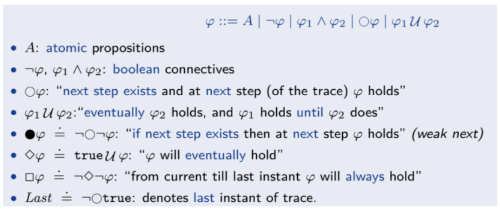
Figure 2.1: Business Processes

- the back-circle representing the traces obtained by the procedural description;
- the bigger white octagon representing the traces obtained with the declarative approach;
- the external triangles contain the traces that do not represent the allowed behaviour of the process.

In the paper [1] the declarative approach was used for solving the Trace Alignment problem and in particular the constraints were defined by using LTL_f.

2.2 The Logic LTL_f and the Associated NFA

The "Linear Temporal Logic over finite traces" comes from the "Linear Temporal Logic". The LTL is one of the formalisms used to express dynamic properties in formal methods. These formalisms are interested in traces representing the evolution of time (intended as what happens before or after something else). In particular, in the Linear Temporal Logic the evolution of things in time is represented as a single infinite time-line that is considered as the "real future" but, there is no certainty that it is the real one. Hence, LTL intrinsically defines infinite traces with which we can analyse tasks that require an infinite amount of time to be executed. However generally, an agent or a human has the need to execute tasks that are limited in time. For this reason, the LTL_f has been created and the traces analysed with this logic are finite. The syntax of LTL_f (Figure 2.2) is the same as the classic LTL but there is a difference in the meaning of these logics which is due to the fact that at a certain point in LTL_f the trace stops:



$$\varphi ::= A \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2$$

- A: atomic propositions
- $\neg\varphi$, $\varphi_1 \wedge \varphi_2$: boolean connectives
- $\bigcirc\varphi$: "next step exists and at next step (of the trace) φ holds"
- $\varphi_1 \,\mathcal{U}\, \varphi_2$:"eventually φ_2 holds, and φ_1 holds until φ_2 does"
- $\bullet\varphi \doteq \neg\bigcirc\neg\varphi$: "if next step exists then at next step φ holds" *(weak next)*
- $\Diamond\varphi \doteq \mathtt{true}\,\mathcal{U}\,\varphi$: "$\varphi$ will eventually hold"
- $\Box\varphi \doteq \neg\Diamond\neg\varphi$: "from current till last instant φ will always hold"
- $Last \doteq \neg\bigcirc\mathtt{true}$: denotes last instant of trace.

Figure 2.2: Syntax of the LTL_f

For example, if we consider the following formula "$\neg X(\varphi)$":

- in LTL it means "In the next step $\neg(\varphi)$ holds";
- in LTL_f it means "Not exist the next instant or $\neg(\varphi)$ holds there".

This particular formula is called in LTL_f "weak next". Talking about "the last step of a trace" has no sense in LTL but it is used in LTL_f to check properties in the last instant of the trace. A very important characteristic of each LTL_f formula is the fact that it can be associated with a NFA (Nondeterministic Finite Automaton: in this type of automata, from a state may exist more than one transition to another state with the same input symbol) that accepts exactly all the traces that satisfy the specific formula [De Giacomo and Vardi 2015] [3]. In particular this NFA is a tuple $A =< \Sigma, Q, q_0, \rho, F >$ where:

- $\Sigma = Prop$ is the input alphabet;
- Q is the finite set of automaton states;
- $q_0 \in Q$ is the initial state;
- $\rho \subseteq Q \times L_{Prop} \times Q$ is the transition relation;
- $F \subseteq Q$ is the set of final states.

For example, if we consider $t = e_1 \cdots e_n$ a trace and A the NFA associated with an LTL_f formula φ. A computation of A on t is a sequence $\delta = q_0 \xrightarrow{e_1} q_1 \cdots q_{n-1} \xrightarrow{e_n} q_n$ s.t., for $i = 0, ..., n-1$, there exists a transition $q_i \xrightarrow{\psi_i} q_{i+1} \in \rho$ s.t. $e_i \models \psi_i$. Since A is nondeterministic, there exist, in general, many computations of A on a trace t and we can say that A accepts t if there exists a computation δ on t s.t. the last state is final, i.e. belongs to F.

2.3 The Trace Alignment Problem

Generally speaking, a log trace is a trace such that the propositional interpretation associated with each position contains only one proposition. Each of these propositions is a singleton and a general log trace should be written as $t = \{a\}\{b\}\{c\}$. However, to simplify the notation, we can write a log trace as composed by simple propositions as $t = a\ b\ c$. Given an LTL_f formula φ and a trace t such that $t \not\models \varphi$, the goal of the Trace Alignment problem is to *"repair"* the trace t obtaining a new trace $\hat{t}$ such that $\hat{t} \models \varphi$. In [1], given a trace $t = e_1 \cdots e_k \cdots e_n$, two actions can be performed to *"repair"* it:

- addition: a proposition p can be *"added"* to t in position k only if $1 \leq k \leq n$, so that $\hat{t} = e_1 \cdots e_{k-1}\ p\ e_k \cdots e_n$;
- deletion: a proposition e_k with $1 \leq k \leq n$ can be *"deleted"* from t such that $\hat{t}' = e_1 \cdots e_{k-1}\ e_{k+1} \cdots e_n$.

After performing a *"repairing"* action, the cost function *cost* is generated as $cost(t, \hat{t}) = c$ where c is the minimal number of repairs needed for obtaining $\hat{t}$ from t. Using this function we can define the Trace Alignment problem as: given a trace t and an LTL_f formula φ such that $t \not\models \varphi$, find a trace $\hat{t}$ such that $\hat{t} \models \varphi$ and $cost(t, \hat{t})$ is minimal. Hence, if φ is satisfiable (i.e. there is a model in which the formula is true), a solution for the problem always exists because starting from the initial trace, by using the *repairing* actions (*addition*/*deletion*), it is possible to generate any log trace.
The approach proposed in [1] uses two automata to solve this problem. The first is the NFA $A =< \Sigma, Q, q_0, \rho, F >$ (called "constraint automaton") associated to the constraint φ. The second one is the DFA (Deterministic Finite Automaton: in this type of automata, for each state there is one and only one transition with the same symbol to another state) $T =< \Sigma_t, Q_t, q_0^t, \rho_t, F_t >$ (called "trace automaton") associated to the trace $t = e_1 \cdots e_n$, where:

- $\Sigma_t = \{e_1, \cdots, e_n\}$;
- $Q_t = \{q_0^t, \cdots, q_n^t\}$ is a set of $n + 1$ arbitrary states;
- $q_0^t \in Q_t$ is the initial state;
- $\rho_t = \bigcup_{i=0,\ldots,n-1} < q_i^t, e_{i+1}, q_{i+1}^t >$;
- $F^t = \{q_n^t\}$.

T is an automaton which accepts only the original trace t and A is an automaton which accepts all the traces that satisfy φ. However, to solve the Trace Alignment problem these two automata have to be augmented, in order to accept also traces modified with the *"repairing"* actions. The augmented version is denoted with the symbol $^+$:

- $T^+ =< \Sigma_t^+, Q_t, q_0^t, \rho_t^+, F_t >$ and accepts all the traces over Σ obtained by *"repairing"* t with the repairs explicitly marked. Moreover, T^+ remains a DFA as T.
- $A^+ =< \Sigma^+, Q, q_0, \rho^+, F >$ and accepts all the traces $\hat{t}$ which satisfy the constraint φ and which are obtained by *"repairing"* t with the repairs explicitly marked. Moreover, A^+ remains a NFA as A.

In particular:

- $\Sigma_t^+ = \Sigma^+$: contains all the propositions in Σ_t plus one proposition *del_p* for all propositions $p \in \Sigma$, and one proposition *add_p* for all propositions $p \in \Sigma \bigcup \Sigma_t$;
- ρ_t^+: contains all the transitions in ρ_t plus a new transition $< q, del\_p, q' >$ for all the transitions $< q, p, q' > \in \rho_t$; and a new transition $< q, add\_p, q >$ for all the proposition p in Σ_t and states $q \in Q_t$, if there is no transition $< q, p, q' > \in \rho_t$ (for all $q' \in Q_t$);
- ρ^+: contains all the transitions in ρ, plus one transition $< q, del\_p, q >$ for all $q \in Q$ and $p \in \Sigma_t$; and one transition $< q, add\_p, q' >$ for all the transitions $< q, \psi, q' > \in \rho$ s.t. $p \models \psi$;
- Q_t, q_0^t, F_t are the same of the automaton T and Q, q_0 and F are the same of the automaton A.

What has been explained above can be summarized with the following **Theorem**:

Consider a log trace t and an LTL_f formula φ, both over Prop, s.t. $t \not\models \varphi$. Let T^+ and A^+ be the automata obtained from t and φ, as described above. If t^+ is a trace accepted by both A^+ and T^+ containing a minimal number of repair propositions (with respect to all other traces accepted by A^+ and T^+), then a trace $\hat{t}$ with minimal cost $cost(t, \hat{t})$ s.t. $\hat{t} \models \varphi$ can be obtained from t^+ by removing all propositions of the form del_p and replacing all propositions of the form add_p with p.

Summing up, the problem can be also seen as: given a trace t and a constraint φ, find a trace accepted by both A^+ and T^+ with minimal number of repair propositions. However, in a real application there are more than one constraint. Thus, the proposed approach in [1] consists in creating the corresponding augmented automaton for each constraint and in finding the trace which is accepted by T^+ and all the A_1^+,...,A_n^+.

2.4 Trace Alignment as Planning

The technique explained in the previous section, based on automata-theoretic manipulations and used to solve the Trace Alignment problem, can be implemented as a "cost-optimal planning problem". Dealing with planning problems, we need to define the planning domain and the planning problem.
A (deterministic) planning domain with action costs is a tuple $D =< S, A, C, \tau >$ where:

- $S \subseteq 2^{Prop}$ is the finite set of domain states (i.e. the propositional interpretations);
- A is the finite set of domain actions;
- $C : A \rightarrow N^+$ is the cost function;
- $\tau : S \times A \rightarrow S$ is the transition function.

A possible plan for the domain D is a finite sequence $\pi \in A^*$ of actions and this plan is *executable* from a state $s_0 \in S$ if there is a sequence of states $\sigma = s_0 \cdots s_n$ s.t. for $i = 0, ..., n-1$, $s_{i+1} = \tau(s_i, a_{i+1})$. The cost of this plan is $C(\pi) = \sum_{i=1,...,n} C(a_i)$.
A cost-optimal planning problem is a tuple $P =< D, s_0, G >$ where:

- D is a planning domain with action costs;
- $s_0 \in S$ is the initial state of the problem;
- G is the problem goal.

A plan π is a solution of P if the last state s_n of trace induced by π is such that $s_n \models G$. Also, this solution is optimal if for all other solutions π', $C(\pi) \leq C(\pi')$.
Using the Theorem described in the Section 2.3, the Trace Alignment problem can be reduced to cost-optimal planning. In fact, given a trace t and an LTL_f formula φ and their corresponding augmented automata T^+ and A^+, the planning domain $D =< S, A, C, \tau >$ is defined in the following way:

- $S \subseteq 2^{Q_t \bigcup Q}$ (automata states are seen as propositions);
- $A = \{sync\_e, del\_e, add\_e \mid e \in \Sigma \bigcup \Sigma_t\}$ is the set of the repair propositions used as actions;
- for all $e \in \Sigma \bigcup \Sigma_t$, $C(sync\_e) = 0$, $C(add\_e) = C(del\_e) = 1$;
- τ is defined as follows, for all $e \in \Sigma \bigcup \Sigma_t$, $q_t, q'_t \in Q_t$, and $R, R' \subseteq Q$:
  - $\tau(\{q_t\} \cup R, sync\_e) = \{q'_t\} \bigcup R'$ iff $q_t \xrightarrow{e} q'_t \in \rho_t^+$ and, for all $q \in R$ and $q' \in R'$, there exists ψ s.t. $e \models \psi$ and $q \xrightarrow{\psi} q' \in \rho^+$;
  - $\tau(\{q_t\} \cup R, del\_e) = \{q'_t\} \cup R'$ iff $q_t \xrightarrow{del\_e} q'_t \in \rho_t^+$ and, for all $q \in R$ and $q' \in R'$, $q \xrightarrow{del\_e} q' \in \rho^+$;

- $\tau(\{q_t\} \cup R, add\_e) = \{q'_t\} \cup R'$ iff $q_t \xrightarrow{add\_e} q'_t \in \rho_t^+$ and, for all $q \in R$ and $q' \in R'$, $q \xrightarrow{add\_e} q' \in \rho^+$.

This type of domain is intended to represent the synchronous product of both T^+ and A^+ over the same input. The produced plans represent sequences of "repairing" actions applied on the original trace t to *align* it to the expected process behaviour. Specifically, the "repairing" actions:

- *sync_e* is used to move along the trace and it has a zero-cost. Hence, it should be the preferred action by the planner;
- *add_e* is used to add a new event e on the original trace t and it has a non zero-cost $(= 1)$;
- *del_e* is used to remove the event e of the original trace t and it has a non zero-cost $(= 1)$.

Since this type of domain allows moving synchronously on T^+ and A^+, when T^+ reaches its final state and A^+ reaches one of its final states, the "repairing" trace t^+ satisfy the constraint φ. Moreover, since a plan with minimal cost is required, this domain suggests the planner to create "repaired" traces $\hat{t}$ that are "closer" as possible to t. For this reason, the planning problem $P =< D, s_0, G >$ is defined in the following way:

- $s_0 = \{q_0, q_0^t\}$ is the initial state of T^+ and A^+;
- $G = q_t^f \wedge \bigvee_{q \in F} q$, for $q_t^f \in F_t$.

As said in the Section 2.3 after the definition of the Theorem, in reality there are more than one constraint and this planning problem can be easily generalized by adding in s_0 all the initial states of $A_1, ..., A_n$, in G all the final states of $A_1, ..., A_n$, in the planning domain add the new propositions used to capture the state of all automata and generalize the actions so that their execution progresses all the automata at once.

2.5 How to Implement it in PDDL

The encoding of this solution has been implemented in PDDL (Planning Domain Definition Language). In the following lines is shown how, given a set of augmented automata $A_1^+, ..., A_n^+$, obtained from n LTL_f formulas $\varphi_1, ..., \varphi_n$, and the augmented trace automaton T^+ obtained from a trace t, the planning domain D and the planning problem P can be built using the syntax of PDDL 2.1.

2.5.1 PLANNING DOMAIN

As regards the planning domain D, the authors of [1] defined:

- Two abstract types:
  - *activity*: represents the activities involved in a transition between two states, both for the trace states and for the automata states;
  - *state*: used for identifying the states of both the trace and the automata.

  Moreover, this last abstract type has two sub-types:
  - *automaton_state* used for identifying the state of the augmented constraint automata;
  - *trace_state* used for identifying the state of the augmented trace automaton.

```
(:types automaton_state trace_state - state  activity )
```

Figure 2.3: Definition of the types and sub-types in the PDDL domain file.

- Four predicates to capture the structure of the augmented trace and constraints automata:

  - *(trace ?t1 - trace_state ?e - activity ?t2 - trace_state)*: it represents a transition from the state $t1$ to the state $t2$ of the augmented trace automaton being 'e' the activity involved in the transition;
  - *(automaton ?s1 - automaton_state ?e - activity ?s2 - automaton_state)*: similarly to the previous proposition, it represents the transition from the state $s1$ to the state $s2$ of the augmented constraint automaton being 'e' the activity involved in the transition;
  - *(cur_state ?s - state)*: it represents the fact that s is the current state either of the sub-type *trace_state* or of the sub-type *automaton_state*;
  - *(final_state ?s - state)*: similarly to the previous proposition, it represents the fact that s is the final state either of the sub-type *trace_state* or of the sub-type *automaton_state*.

- A numeric fluent called *total-cost* to keep track of the cost of the "repairing" actions operation at each step.

```
(:functions (total-cost) )
```

Figure 2.4: Definition of the numeric fluent in the PDDL domain file.

- Three actions which represent the "repairing" actions:
  - *sync*: it is used to move forward along the trace t and synchronously to all the constraint automata. Basically, if a state of the trace automaton is the current one and there is a transition from this state with an activity e to another state, then the current state of the trace is updated and for all the constraint automata is checked if there is an activity "e" from their current states and if it exists, update their current states. This action has a zero cost;
  - *add*: for each current state of the constraint automata, if there is a transition with the activity "e" from the current state to another, then update the current state. This action increases the cost-function of 1;
  - *del*: it operates only on the trace automaton and not on the constraint automata. Basically, when a state of the trace automaton is the current state and there is a transition from this state with an activity e to another state, then update the current state. This action increases the cost-function of 1.

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

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

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

Figure 2.5: Definition of the actions *sync*, *add* and *del* in the PDDL domain file.

It is important to highlight that only the *del* action exclusively works on the trace automaton. While *add* and *sync* actions work with both the trace automaton and the constraint automata. Moreover, *sync* and *del* have preconditions (they can be executed only if there is a transition from the current state of the trace automaton to another one with the activity e). While the *add* action can always be performed.

2.5.2 PLANNING PROBLEM

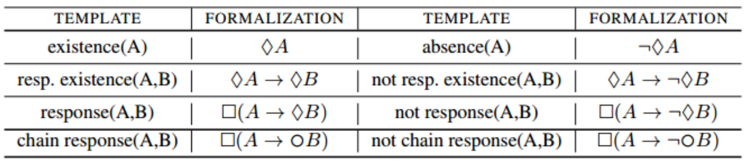
The planning problem has to contain all the objects needed to describe the trace automaton and the constraint automata (states and activities), the initialization of the objects at their initial conditions (the transitions of all the automata, the initial states and the final states) and the goal to be reached. Considering the definition of the goal given in the Section 2.4 $q_t^f \wedge \bigvee_{q \in F} q$, since the 'OR' ($\bigvee$) operator is not contemplated in the syntax of PDDL, the goal can be defined by using an 'AND' ($\wedge$) operator between the final state of the trace automaton and a *forall* operator that checks for each state of the constraint automata if the current state is a final state.
The implementation explained above generates traces that belong to the allowed behaviour of the process and have minimal cost but time required to find a solution is not very low. This is due to the fact that the constraint automata can have multiple final states in the same automaton and this increases the number of computations required for the planner to solve the problem. To speed up the process, the authors suggest another implementation of the goal which requires to preprocess each constraint automata in the following way:

- add a new state called *dummy* with no outgoing transitions. This operation is necessary when a constraint automaton has more than one final state;
- add a new action executable only in the final states of the original constraint automaton which allows them to move to the dummy state;
- include only the *dummy* state in the set of the final states of the corresponding constraint automaton, removing all the original ones.

Now, the goal can be defined as the conjunction of all the accepting states of the trace automaton and the constraint automata. Finally, in order to minimize the total cost of the plan, the planning problem has to contain the following specification *(:metric minimize (total-cost))*.

2.6 Experiment: Real and Synthetic Logs

The experiments proposed in [1] have been performed on a dataset which contains logs formatted in XES (eXtensible Event Stream) standard and DECLARE models formatted in XML. To find the minimum cost trace alignment the authors used two different planning systems to make comparisons. In particular, they used: FAST-DOWNWARD, which uses in the first iteration a best-first search and then a weighted A* search for iteratively decreasing weights of plans, and SYMBA*-2 which performs a bidirectional A* search. Then, they compared the results obtained with these two planning systems with a specified approach [6] and previous approaches [2]. Moreover, the dataset contains real-life logs and synthetic logs which have been both used for the experiments. In detail, the real-life log data come from real process executions and refers to an application process for personal loans in a Dutch financial institute. The original dataset contained 262200 events distributed across 36 activities and included 13087 traces but the authors used only 654 traces (real-life log) of various lengths for performing the experiments. With these traces a DECLARE model containing 16 constraints has been employed. On the real-life traces the plan cost obtained was between 0 and 2 for short traces (less than 50 events) and between 0 and 3 for long traces (more than 50 events). Instead, the synthetic logs have been used for having a sense of the scalability with respect to the size of the model and the noise in the traces. In particular, they defined 3 DECLARE models having the same alphabet of activities and containing respectively 10, 15 and 20 constraints. From these models, by replacing 3, 4 and 6 constraints in each model with their negative counterpart (Figure 2.6), they generated for each model, by using the log generator presented in [7], 4 synthetic logs of 100 traces with different lengths (from 1 to 50, from 51 to 100, from 101 to 150 and from 151 to 200).

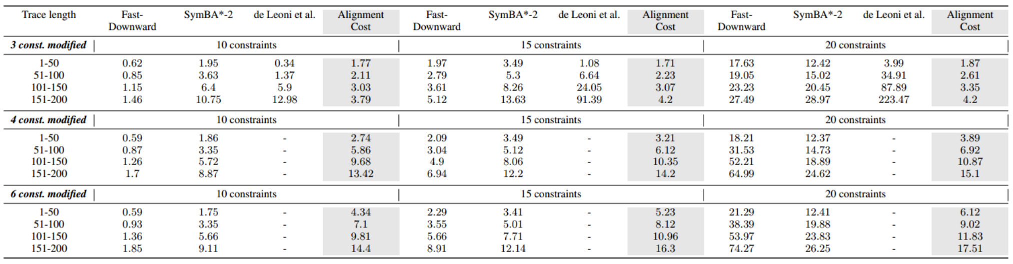


TEMPLATE	FORMALIZATION	TEMPLATE	FORMALIZATION
existence(A)	$\Diamond A$	absence(A)	$\neg\Diamond A$
resp. existence(A,B)	$\Diamond A \rightarrow \Diamond B$	not resp. existence(A,B)	$\Diamond A \rightarrow \neg\Diamond B$
response(A,B)	$\Box(A \rightarrow \Diamond B)$	not response(A,B)	$\Box(A \rightarrow \neg\Diamond B)$
chain response(A,B)	$\Box(A \rightarrow \bigcirc B)$	not chain response(A,B)	$\Box(A \rightarrow \neg\bigcirc B)$

Figure 2.6: LTL_f formalization of some DECLARE templates. The image shows also the negative counterpart of some constraints.

2.7 Results of the Experiments with the Synthetic Logs

Analyzing the results (Figures 2.7 and 2.8) of both real-life and synthetic logs it is easy to see that the de Leoni et al. approach is the fastest one for short traces with a small amount of noise (3 constraints inverted). But, looking at the results obtained using only the synthetic logs obtained by using 20 constraints with 3 constraints modified, while the FAST-DOWNWARD and SYMBA*-2 require respectively 27.49s and 28.97s, de Leoni et al. approach requires 223.47s. An explanation for this result can be the fact that the heuristics adopted by the planners are able to efficiently cope with the size of the state space, which is exponential with respect to the size of the model, the amount of noise and the trace length. Generally speaking, from these results, the FAST-DOWNWARD with blind A* performs better for 10 and 15 constraints, while for 20 constraints the SYMBA*-2 which uses bidirectional A* performs better.



Trace length	Fast-Downward	SymBA*-2	de Leoni et al.	Alignment Cost	Fast-Downward	SymBA*-2	de Leoni et al.	Alignment Cost	Fast-Downward	SymBA*-2	de Leoni et al.	Alignment Cost
3 const. modified	10 constraints				15 constraints				20 constraints			
1-50	0.62	1.95	0.34	1.77	1.97	3.49	1.08	1.71	17.63	12.42	3.99	1.87
51-100	0.85	3.63	1.37	2.11	2.79	5.3	6.64	2.23	19.05	15.02	34.91	2.61
101-150	1.15	6.4	5.9	3.03	3.61	8.26	24.05	3.07	23.23	20.45	87.89	3.35
151-200	1.46	10.75	12.98	3.79	5.12	13.63	91.39	4.2	27.49	28.97	223.47	4.2
4 const. modified	10 constraints				15 constraints				20 constraints			
1-50	0.59	1.86	-	2.74	2.09	3.49	-	3.21	18.21	12.37	-	3.89
51-100	0.87	3.35	-	5.86	3.04	5.12	-	6.12	31.53	14.73	-	6.92
101-150	1.26	5.72	-	9.68	4.9	8.06	-	10.35	52.21	18.89	-	10.87
151-200	1.7	8.87	-	13.42	6.94	12.2	-	14.2	64.99	24.62	-	15.1
6 const. modified	10 constraints				15 constraints				20 constraints			
1-50	0.59	1.75	-	4.34	2.29	3.41	-	5.23	21.29	12.41	-	6.12
51-100	0.93	3.35	-	7.1	3.55	5.01	-	8.12	38.39	19.88	-	9.02
101-150	1.36	5.66	-	9.81	5.66	7.71	-	10.96	53.97	23.83	-	11.83
151-200	1.85	9.11	-	14.4	8.91	12.14	-	16.3	74.27	26.25	-	17.51

Figure 2.7: The image shows the experimental results obtained in [1] for the *synthetic* case study. The table reports the average alignment time employed for the computations.

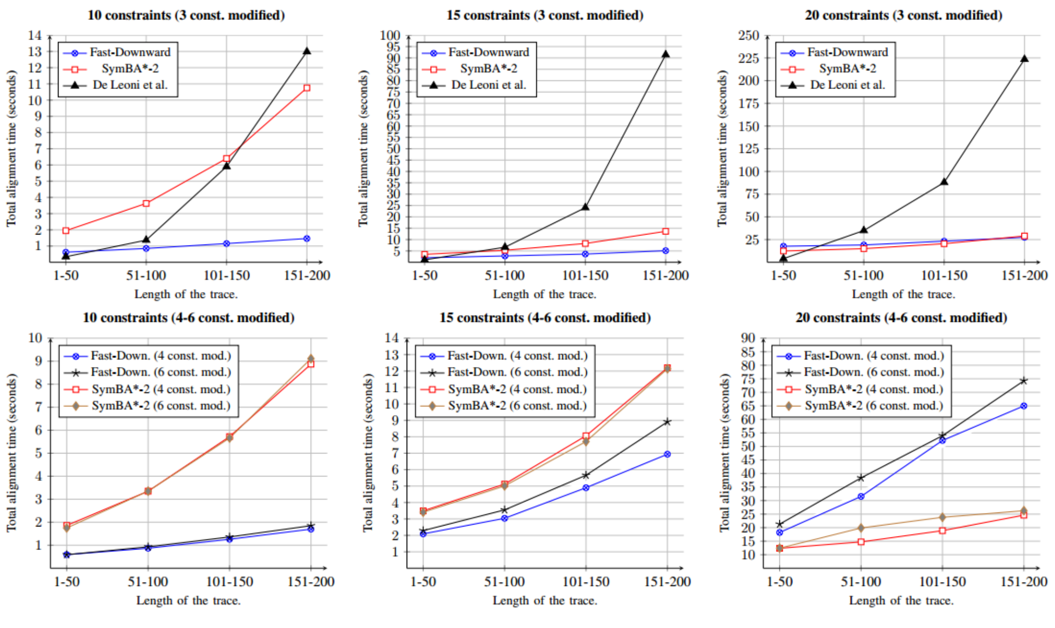


Figure 2.8: The image shows the plots obtained in [1] for the *synthetic* case study which report the total alignment time in seconds for each length of the traces. The results for the three planners are reported: FAST-DOWNWARD, SYMBA*-2 and de Leoni et al.

Chapter 3

Our Implementations

As said in the Chapter 1 we have developed two implementations of the proposed encoding by using the syntax of PDDL2.1. We started implementing the one illustrated in the paper [1] and then we created an improved version with the aim of achieving better results. For both the implementations we have used the FAST-DOWNWARD 20.06 to test the encoding since most of the best results, in the studied paper, were obtained with it. Moreover, we want to point out that the scripts we have created for this project are based on the structure of the files included in the provided dataset.

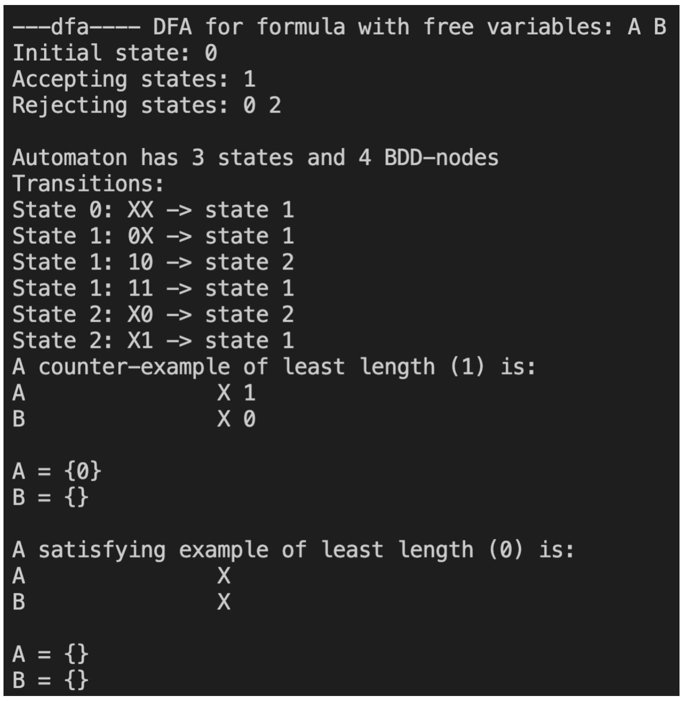
3.1 Encoding Provided in [1]

We developed a Python script called *createPddlDummyFree.py* which takes as arguments, in the CLI (Command Line Interface), the path of the file .xes/.txt containing the traces and the path of the file .xml/.txt containing the constraints defined in LTL_f. This script automatically generates a new planning problem file for each trace contained in the .xes/.txt file by using all the constraints contained in the .xml/.txt file. The planning domain file is unique for all the planning problems and we have created it by hand following the encoding illustrated in Section 2.4.
In particular, in each planning problem file:

- the objects defined are: all the states of the trace automaton correlated to the problem, all the states of the automata associated with the constraints and all the activities involved in both T^+ and all the A^+;
- the initial conditions are: the initial state of T^+ and all the initial states of the $A_1^+, .., A_n^+$, the final state of T^+ and all the final states of $A_1^+, .., A_n^+$ and all the transitions of both the trace and the constraint automata;
- the goal conditions are: the *and* between the final state of T^+ as current state and a *forall* that checks if the current states of the constraint automata *imply* final states.

Going into the details, from a general trace t it is easy to automatically write the corresponding DFA: if the trace is composed of n events, then there are $n+1$ trace states $(t_1, .., t_i, .., t_{n+1})$ and the transitions are *(trace t_i e_i t_{i+1})* with $i = 1, .., n$ where e_i is the activity in position i of the trace t. More problematic is how to automatically write the NFA corresponding to the constraints defining the process. As mentioned in the Section 1, our scripts accept general LTL_f formulas as constraints. To produce the corresponding augmented constraint automaton A^+ we have used the LTLf2DFA library, created by Francesco Fuggitti. This library generates from a generic LTL_f formula the corresponding DFA, instead of the corresponding NFA as used in [1]. Hence, from now on, we will be dealing only with DFA as augmented constraint automaton A^+. Before automatically encoding the automata, generated with this library, into the planning problem, we needed to perform another step. In fact, the DFA printed by LTLf2DFA were now suitable for being used as a constraint automaton A of a Trace Alignment problem. The reason is simple, these automata consider traces in which there are multiple events at a time, while an assumption of the BPs is that there is only one single event at a time. Therefore, we had to manipulate the transitions of the DFA generated by LTLf2DFA (which uses MONA as support library - https://www.brics.dk/mona/) before using them in our implementation.

An example of the output of this library, extracted from our experiments, is shown in the Figure 3.1.



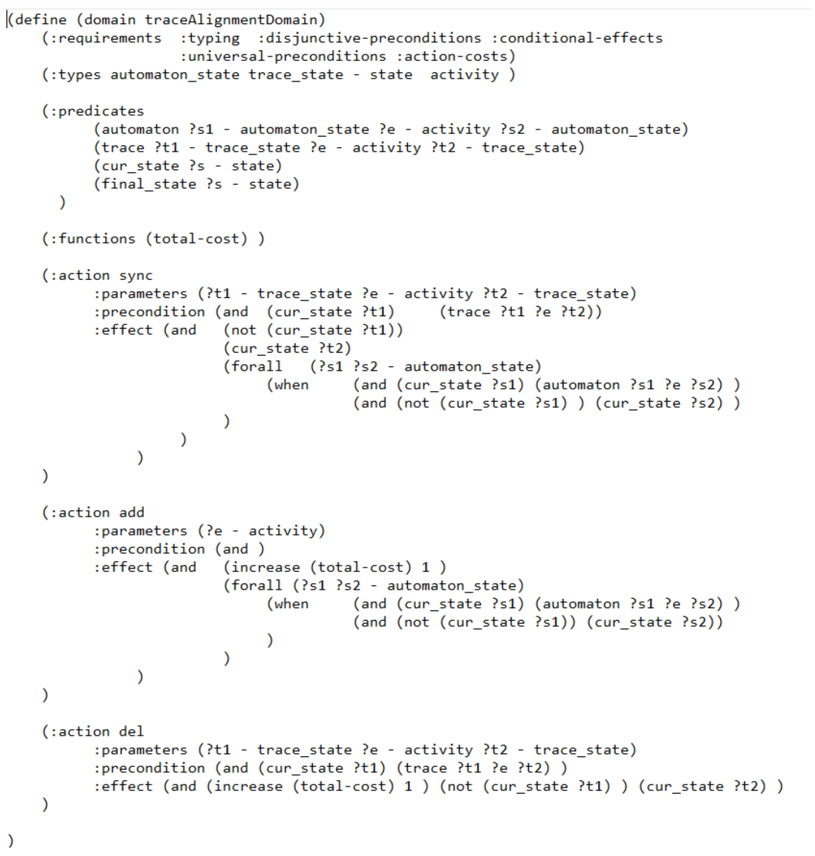
```
---dfa---- DFA for formula with free variables: A B
Initial state: 0
Accepting states: 1
Rejecting states: 0 2

Automaton has 3 states and 4 BDD-nodes
Transitions:
State 0: XX -> state 1
State 1: 0X -> state 1
State 1: 10 -> state 2
State 1: 11 -> state 1
State 2: X0 -> state 2
State 2: X1 -> state 1
A counter-example of least length (1) is:
A               X 1
B               X 0

A = {0}
B = {}

A satisfying example of least length (0) is:
A               X
B               X

A = {}
B = {}
```

Figure 3.1: The image shows the output of the library LTLf2DFA obtained by using MONA.

In the first row there is the list of the free variables (denoted as capital letters) of the LTL_f formula, ordered according to how they appear in the constraint. In the following lines, the initial state, the accepting states (= the final states), the rejecting states and all the transitions from each state are defined. We parsed this output to extract the information we needed and in particular: the *State 0* and the transitions starting from it can be discarded as well as all the loops (= transitions starting and arriving in the same state). For the remaining transitions the reasoning is different: since BPs are made by only one event at a time, all the transitions that require more than one event at a time can be discarded. According to the documentation of MONA the value "1" in position i means that the free variable in position i is involved in the transition, "0" means non-involved and "X" means that it can be either "1" or "0". Then:

- the transitions which have more than one "1" have to be discarded and not considered (more than one event at a time);
- the transitions which have an explicit "1", have to be taken with all the other "X" substituted by "0". Otherwise we will have more than one event at a time;
- the transitions in which there are not explicit "1" generates:
  - a transition with all "0": it means that no one of the free variables is involved in the transition. Since the free variables are not all the activities available on the trace, it means that this transition can be performed with the activity included in the set representing the difference between the set of activities used in the trace and the set containing the free variables;
  - a number of transitions equal to the number of "X". Each transition is obtained by replacing the "X" with all "0" and one "1" which will be put every time in a different position.

By using this approach we are able to generate a DFA compatible with the Trace Alignment problem and to automatically generate a planning problem file for each trace.



```
(define (domain traceAlignmentDomain)
    (:requirements  :typing  :disjunctive-preconditions :conditional-effects
                    :universal-preconditions :action-costs)
    (:types automaton_state trace_state - state  activity )

    (:predicates
          (automaton ?s1 - automaton_state ?e - activity ?s2 - automaton_state)
          (trace ?t1 - trace_state ?e - activity ?t2 - trace_state)
          (cur_state ?s - state)
          (final_state ?s - state)
      )

    (:functions (total-cost) )

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

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

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

)
```

Figure 3.2: The image shows how the PDDL domain file has been written starting from the encoding provided in [1].

3.2 Modified Version of the Encoding

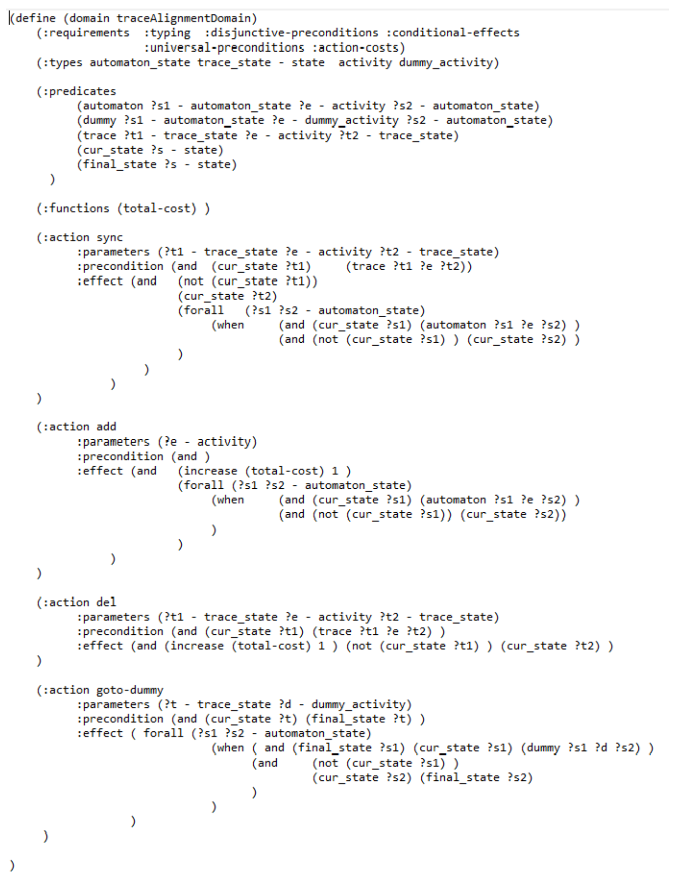
By looking at the results obtained with the implementation of the proposed encoding in [1], we realized that this implementation was slightly too expensive in terms of computations. One of the reasons was the way in which the goal was defined. The *forall* construct, which checks for all the states of the constraint automata if they are also final states, was very highly expensive. This observation has been the starting point for our modifications. In particular, we made two changes: the first one was the implementation of the dummy states in the constraint automata, in order to reduce the number of final states which have to be checked every time; the second one was the change of the expression of the goal in such a way to reduce the computational cost.

3.2.1 FIRST MODIFICATION

Regarding the first change, the purpose of the dummy state is to make sure that each constraint automaton has a unique final state without outgoing transitions. To implement it, for each constraint automaton, we needed to automatically add a transition from each final state to the dummy state. The dummy state is considered as an automaton state and it is unique for each automaton. Differently from the original implementation we modified both the domain and the problem file. In detail, the domain file has:

- an additional abstract type called *dummy_activity*, in order to not be confused with the "normal" activities used in the trace and in the constraints;
- an additional proposition *(dummy ?s1 - automaton_state ?e - dummy_activity ?s2 - automaton_state)* which defines the special transition that leads the final states to the dummy state;
- an additional action named *goto-dummy*, which executes the transition to the dummy state. Its precondition is that the trace automaton has to stay in its final trace state; the effect checks for all the states of the constraint automata if their current state is a

final state and if there is a *dummy* transition from this state to the dummy. If these conditions are satisfied, then the dummy state becomes the current state and is set as final state. This new action has a zero cost (as the *sync* action). Hence, it does not influence the cost-function.



```
(define (domain traceAlignmentDomain)
    (:requirements  :typing  :disjunctive-preconditions :conditional-effects
                    :universal-preconditions :action-costs)
    (:types automaton_state trace_state - state  activity dummy_activity)

    (:predicates
          (automaton ?s1 - automaton_state ?e - activity ?s2 - automaton_state)
          (dummy ?s1 - automaton_state ?e - dummy_activity ?s2 - automaton_state)
          (trace ?t1 - trace_state ?e - activity ?t2 - trace_state)
          (cur_state ?s - state)
          (final_state ?s - state)
      )

    (:functions (total-cost) )

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

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

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

    (:action goto-dummy
          :parameters (?t - trace_state ?d - dummy_activity)
          :precondition (and (cur_state ?t) (final_state ?t) )
          :effect ( forall (?s1 ?s2 - automaton_state)
                              (when ( and (final_state ?s1) (cur_state ?s1) (dummy ?s1 ?d ?s2) )
                                    (and     (not (cur_state ?s1) )
                                             (cur_state ?s2) (final_state ?s2)
                                    )
                              )
                   )
     )

)
```

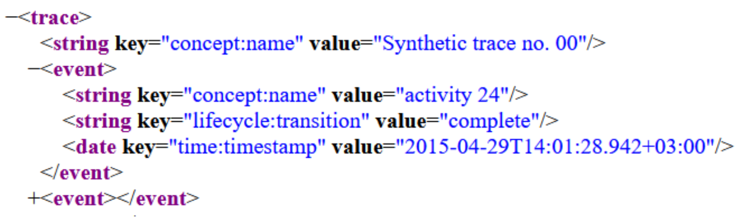
Figure 3.3: The image shows the modified PDDL domain file with the presence of the action *goto-dummy*.

3.2.2 SECOND MODIFICATION

We apply this second modification to change the way in which the goal is defined (in order to reduce the computational cost). As explained before, the disjunction in the formula $G = q_t^f \wedge \bigvee_{q \in F} q$ involves the use of a *forall* construct in PDDL. To eliminate this construct we simply changed the way in which the goal is defined. Now, it becomes the conjunction between: the final state of the trace automaton when it is the current state, all the dummy states when they are set as current states and all the final states of the automata which have only 1 final state, when they are set as current states. In this way, the computational time cost is extremely reduced because now the planner has to check a fewer number of predicates.

We want to highlight that we built our implementations basing on the structure of the .xes files and .xml files. In particular, in a .xes file a trace is defined inside the <trace> tag (Figure 3.4). Inside this tag, the events composing the trace are defined inside the <event> tag. In these .xes files each tag has a key *concept:name* and a value which allows to recognize respectively the name of the traces and the name of the events.

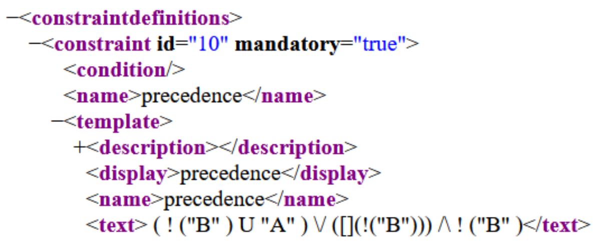
After generating the traces with their corresponding events we extracted from a .xml file the constraints defining the process. The information we were searching in this file were the LTL_f formulas and the activity involved in each formula (Figure 3.5). All the constraints were defined in the <constraintdefinitions> tag and all the information about each constraint was put inside the <constraint> tag. After extracting the string representing an LTL_f formula, we needed to process it since the LTLf2DFA library needed another semantics for working. Then, we extracted the activity involved in the formula, contained



```
−<trace>
   <string key="concept:name" value="Synthetic trace no. 00"/>
  −<event>
     <string key="concept:name" value="activity 24"/>
     <string key="lifecycle:transition" value="complete"/>
     <date key="time:timestamp" value="2015-04-29T14:01:28.942+03:00"/>
   </event>
  +<event></event>
```

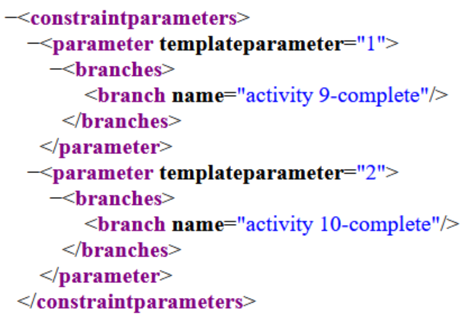
Figure 3.4: The image shows the structure of the .xes file. In particular, the tags <trace>, <event>, the key *concept:name* and its corresponding value.

in the <constraintparameters> tag (Figure 3.6) and we associated them to the variables in the formula itself.



```
−<constraintdefinitions>
  −<constraint id="10" mandatory="true">
     <condition/>
     <name>precedence</name>
    −<template>
      +<description></description>
       <display>precedence</display>
       <name>precedence</name>
       <text> ( ! ("B" ) U "A" ) ∨ ([](!("B"))) ∧ ! ("B" )</text>
```

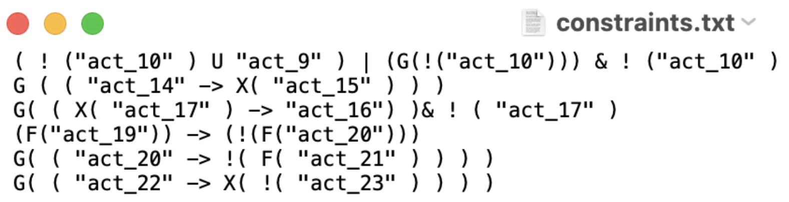
Figure 3.5: The image shows the LTL_f formula contained in the <constraintdefinitions> tag.



```
−<constraintparameters>
  −<parameter templateparameter="1">
    −<branches>
        <branch name="activity 9-complete"/>
     </branches>
   </parameter>
  −<parameter templateparameter="2">
    −<branches>
        <branch name="activity 10-complete"/>
     </branches>
   </parameter>
 </constraintparameters>
```

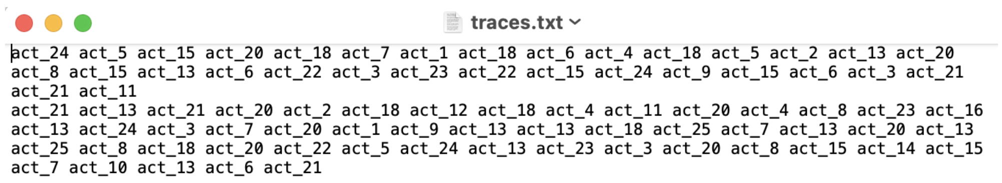
Figure 3.6: The image shows the name of the activities and the id corresponding to the associated parameters. In this case the parameter ”1” which is ”A” corresponds to ”activity 9-complete”, while the parameter ”2” which is ”B” corresponds to ”activity 10-complete”.

Moreover, we provided another type of file for storing the data. We created two simple .txt files: the first one contains the constraints in LTL_f representing the allowed process behaviour (Figure 3.7); the second one contains a trace written as an ordered list of the events' names (Figure 3.8). In particular, for the file representing the log traces, inspired on how the activities are written on the .xes and .xml files (i.e. *activity* {*id activity*}), we wrote one trace for line and in each trace the activities are written as *act_*{*activity id*}, all separated by ” ”. Regarding the file representing the constraints, they are written already in LTL_f with a syntax compatible with the LTLf2DFA library and the activities involved in the formula are written between “” (i.e. “*act_*{*activity id*}”).



```
( ! ("act_10" ) U "act_9" ) | (G(!("act_10"))) & ! ("act_10" )
G ( ( "act_14" -> X( "act_15" ) ) )
G( ( X( "act_17" ) -> "act_16") )& ! ( "act_17" )
(F("act_19")) -> (!(F("act_20")))
G( ( "act_20" -> !( F( "act_21" ) ) ) )
G( ( "act_22" -> X( !( "act_23" ) ) ) )
```

Figure 3.7: The file .txt for the constraints with each formula on a different line.


traces.txt

```
act_24 act_5 act_15 act_20 act_18 act_7 act_1 act_18 act_6 act_4 act_18 act_5 act_2 act_13 act_20
act_8 act_15 act_13 act_6 act_22 act_3 act_23 act_22 act_15 act_24 act_9 act_15 act_6 act_3 act_21
act_21 act_11
act_21 act_13 act_21 act_20 act_2 act_18 act_12 act_18 act_4 act_11 act_20 act_4 act_8 act_23 act_16
act_13 act_24 act_3 act_7 act_20 act_1 act_9 act_13 act_13 act_18 act_25 act_7 act_13 act_20 act_13
act_25 act_8 act_18 act_20 act_22 act_5 act_24 act_13 act_23 act_3 act_20 act_8 act_15 act_14 act_15
act_7 act_10 act_13 act_6 act_21
```

Figure 3.8: The file .txt for the traces with each trace on a different line.

Finally, for executing the script it's necessary to write the following lines on the terminal: *python <script Name>.py –Traces "<path of the trace file> (.xes or .txt)" –Constraints "<path of the constraint file> (.xml or .txt)"*.

In the following section, we are going to illustrate all the experiments performed. However we also tried to implement experiments by considering an hybrid solution between our two implementations: considering the presence of dummy states but maintaining the *forall* construct in the goal of the problem file. However we note that this computation required a lot of time with respect to the implementation which considers both the dummy state and the conjunction in the goal. For this reason we decided to not consider them in our results and make a comparison only between the two versions explained before.

3.3 Dataset, Experiments and Results: Our Work VS the Original One

The dataset used for the experiments was the same dataset used in [1]. As explained in the Section 2.6, this dataset contains two types of logs: real-life logs and synthetic logs. Since we needed to compare our results with the ones shown in [1], we decided to focus on the ones obtained by using the synthetic logs. Specifically, we decided to run:

- the implementation of the encoding provided in [1] with the traces contained in the files .xes generated by 10 constraints (with 3, 4 and 6 inverted) and by 15 constraints (with 3 inverted);
- our modified version of the encoding with the traces contained in the files .xes generated by 10 constraints (with 3, 4 and 6 inverted) and by 15 constraints (with 3, 4 and 6 inverted).

To run these implementations we used, as planning system, Fast-Downward 20.06 which performs a blind A^* search. Starting from the plots shown in the paper, we used as reference axes for our plots the total average time/cost and the lengths of the traces (i.e. 1-50, 51-100, 101-150, 151-200 and for some traces 201-250). In summary, we used the traces generated by the models with 10/15 constraints and 3, 4 and 6 constraints inverted to compare the results obtained with our best implementation and the results shown in the paper. While, to compare our implementations between each other we only used the traces generated by the models with 10/15 constraints with 3 constraints modified. For technical issues we have decided to not use the traces obtained from the models with 20 constraints because the time required to execute all the comparisons would have been too much. Looking carefully at the results obtained from our modified version of the encoding (Figures 3.9, 3.10 and 3.11) and the results proposed in [1] (Figure 2.8), it can be immediately seen that the computational time required for our implementation is generally higher compared to the results obtained in original encoding.
In particular, for short traces (1-50 and 51-100) our results are competitive with the original ones. While for long traces the time is significantly higher. Moreover, the computational time required from our implementation is always higher than the results shown in [1] when the number of inverted constraints is not equal to 3 (Figures 3.10 and 3.11). We thought that the differences between our implementation and the one in the original paper can depend on the device that we used for the computations. Another reason can be the fact that we used an updated version of the Fast Downward and we don't know how it has changed in these years. These observations come from the fact that, if we compare the costs of our implementation in Table 3.1, 3.2, 3.3, 3.4, 3.5 and 3.6 and the original ones in Table 2.7, they are the same for the traces generated from the models containing 10 and 15 constraints

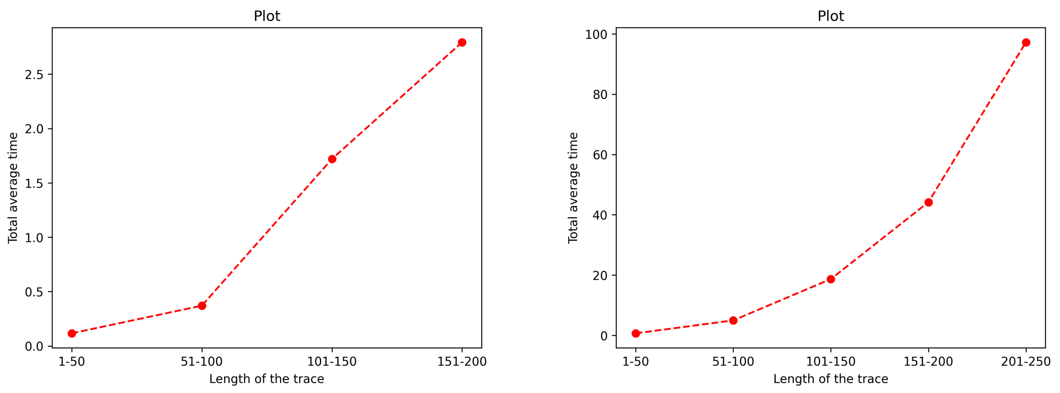


Figure 3.9: Plots of the total average time results obtained from our modified version of the encoding: 10 (left) and 15 (right) constraints with 3 constraints modified.

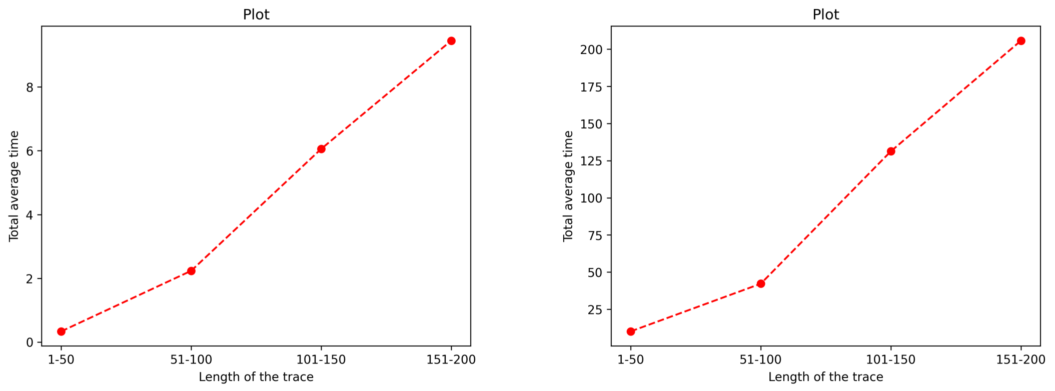


Figure 3.10: Plots of the total average time results obtained from our modified version of the encoding: 10 (left) and 15 (right) constraints with 4 constraints modified.

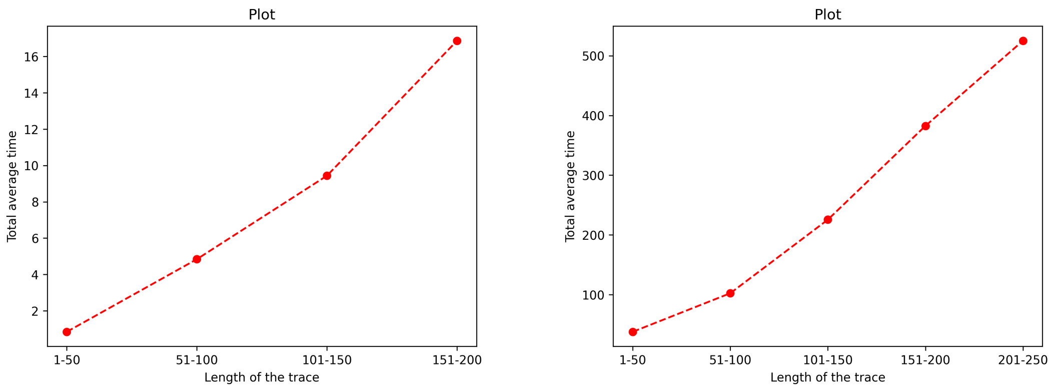
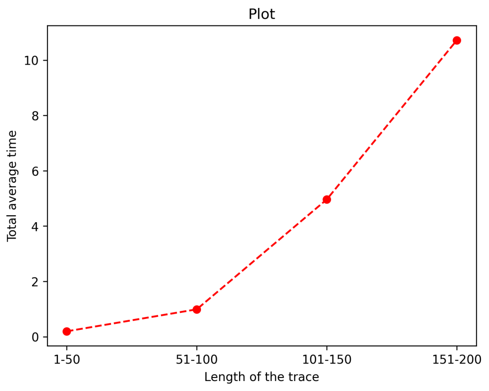


Figure 3.11: Plots of the total average time results obtained from our modified version of the encoding: 10 (left) and 15 (right) constraints with 6 constraints modified.

of which 3 constraints are inverted and from the models containing 10 constraints with 4 inverted. This means that the ”repairing” actions executed from both the implementations are equivalent. On the other hand, for the traces obtained from the models with 10 and 15 constraints of which 6 constraints inverted, our costs are higher than the originals. While, for the traces obtained from the models with 15 constraints of which 4 inverted our costs are lower.
Now, considering only the traces obtained from the models with 10 and 15 constraints of which 3 modified (Figures 3.12 and 3.13), the comparison between our two implementations is shown. We decided to use only these traces because the computations for the implementation without the dummy states and the *forall* construct in the goal required too much time.



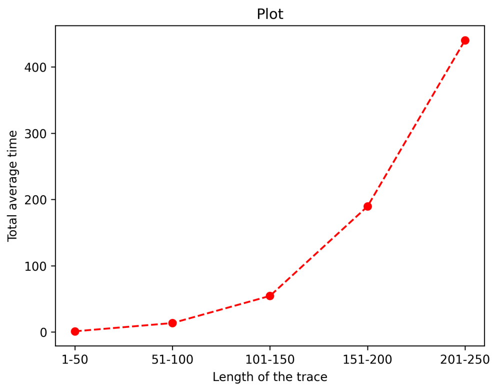
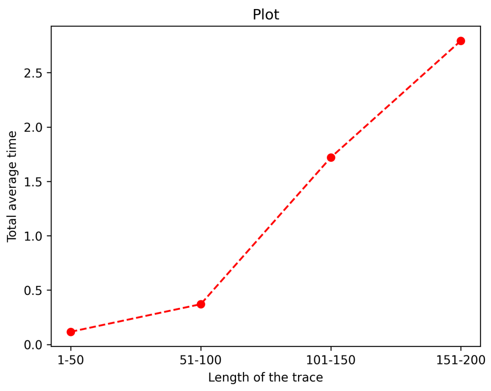


Figure 3.12: Plots of the total average time results obtained from our implementation without using the dummy state and considering the *forall* construct in the goal. (10 (left) and 15 (right) constraints with 3 constraints modified).



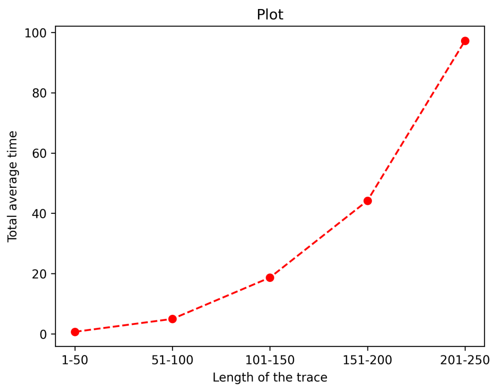


Figure 3.13: Plots of the total average time results obtained from our implementation by adding the dummy state and considering the conjunction in the goal. (10 (left) and 15 (right) constraints with 3 constraints modified).

The Figure 3.12 shows the results obtained from the implementation of the encoding proposed in [1] and Figure 3.13 our modified version. The traces used are: on the left the ones coming from the models with 10 constraints of which 3 inverted; on the right the ones coming from the models with 15 constraints of which 3 inverted. As expected, the time required to find a solution is lower for the modified version of the econding. It is easy to infer that since the goal written with the *forall* construct has to check a number of states much higher and it will need much more time for finding a solution. This fact is not very evident for short traces (in particular 1-50) but for long traces it is. For example, with the traces obtained from the model with 10 constraints the max average is 10.7193303s for the original implementation. While the average time for the modified version (dummy states and the conjunction in the goal) is 2.79383666s, hence 1/4 faster than the original one. The same behavior can be observed with the traces obtained from the model with 15 constraints, where the max time average is 440.37472431s for the original implementation, while the modified one requires 97.23643113s. These results show that the use of the dummy states and the conjunction in the goal are convenient for reducing the computational time cost. The following plots (Figure 3.14 and 3.15) show the average costs needed for the computations in Figures 3.12 and 3.13 respectively.

As expected, the costs required to find a solution are the same. This means that the solutions found are identical but the modified version is extremely faster. Also in this case all the values of the results, in terms of time and costs, are reported in the previous tables.

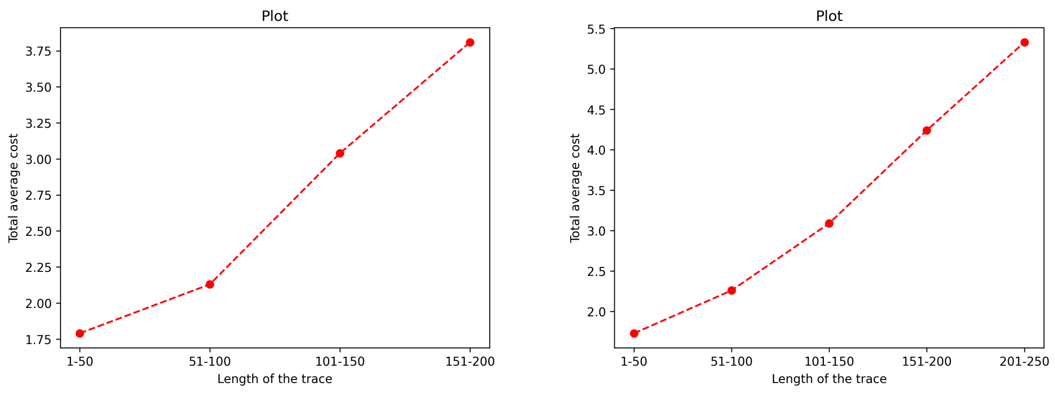


Figure 3.14: Plots of the total average cost results obtained from our implementation without using the dummy state and considering the *forall* construct in the goal. (10 (left) and 15 (right) constraints with 3 constraints modified).

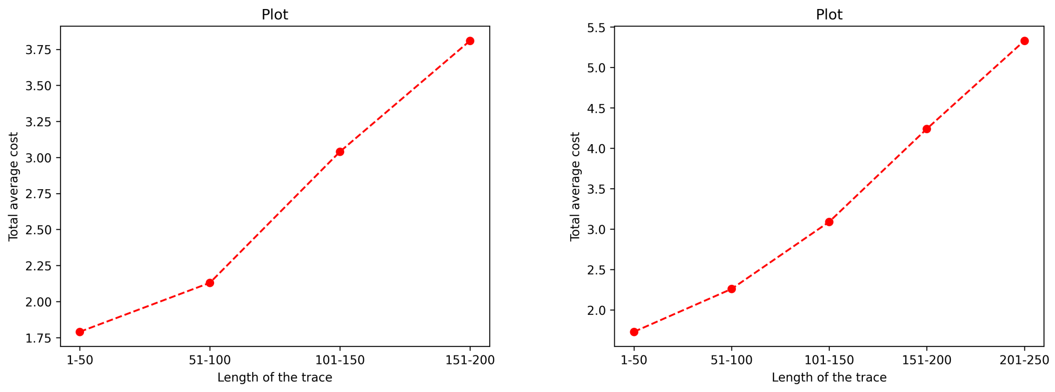


Figure 3.15: Plots of the total average cost results obtained from our implementation by adding the dummy state and considering the conjunction in the goal. (10 (left) and 15 (right) constraints with 3 constraints modified).

Trace length	Average time with dummy	Average time without dummy	Average cost
1-50	0.12	0.20	1.79
51-100	0.37	0.99	2.13
101-150	1.72	4.96	3.04
151-200	2.79	10.72	3.81

Table 3.1: The table shows the results obtained for 10 constraints with 3 constraints modified. It reports the trace length, the average time involved in the computations for the implementation with the dummy state and the conjunction in the goal, and the one without dummy state and the *forall* construct in the goal, and the average cost.

Trace length	Average time with dummy	Average time without dummy	Average cost
1-50	0.33	0.90	2.77
51-100	2.23	7.35	5.88
101-150	6.06	22.78	9.7
151-200	9.45	41.70	13.42

Table 3.2: Similarly to Table 3.1, the table shows the results obtained for 10 constraints with 4 constraints modified.

Trace length	Average time with dummy	Average time without dummy	Average cost
1-50	0.84	1.54	4.28
51-100	4.84	14.52	9.76
101-150	9.44	36.08	16.26
151-200	16.87	65.89	21.67

Table 3.3: Similarly to Table 3.1, the table shows the results obtained for 10 constraints with 6 constraints modified.

Trace length	Average time with dummy	Average time without dummy	Average cost
1-50	0.69	1.14	1.73
51-100	4.97	13.58	2.26
101-150	18.68	54.54	3.09
151-200	44.18	189.59	4.24
201-250	97.24	440.37	5.33

Table 3.4: 15 constraints (3 const. modified) - our implementations

Trace length	Average time with dummy	Average time without dummy	Average cost
1-50	10.11	-	3.8
51-100	42.18	-	5.95
101-150	131.29	-	9.51
151-200	205.73	-	12.34

Table 3.5: 15 constraints (4 const. modified) - our implementation

Trace length	Average time with dummy	Average time without dummy	Average cost
1-50	37.87	-	6.24
51-100	102.43	-	9.52
101-150	225.57	-	14.53
151-200	382.51	-	20.64
201-250	525.07	-	25.55

Table 3.6: 15 constraints (6 const. modified) - our implementation

Bibliography

[1] Giuseppe De Giacomo, Fabrizio Maria Maggi, Andrea Marrella, and Fabio Patrizi. On the disruptive effectiveness of automated planning for ltlf-based trace alignment. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[2] Giuseppe De Giacomo, Fabrizio Maria Maggi, Andrea Marrella, and Sebastian Sardina. Computing trace alignment against declarative process models through planning. In *Twenty-Sixth International Conference on Automated Planning and Scheduling*, 2016.

[3] Giuseppe De Giacomo and Moshe Vardi. Synthesis for ltl and ldl on finite traces. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[4] Giuseppe De Giacomo and Moshe Y Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

[5] Massimiliano De Leoni, Fabrizio M Maggi, and Wil MP van der Aalst. An alignment-based framework to check the conformance of declarative process models and to preprocess event-log data. *Information Systems*, 47:258–277, 2015.

[6] Massimiliano De Leoni, Fabrizio Maria Maggi, and Wil MP van der Aalst. Aligning event logs and declarative process models for conformance checking. In *International Conference on Business Process Management*, pages 82–97. Springer, 2012.

[7] Claudio Di Ciccio, Mario Luca Bernardi, Marta Cimitile, and Fabrizio Maria Maggi. Generating event logs through the simulation of declare models. In *Workshop on Enterprise and Organizational Modeling and Simulation*, pages 20–36. Springer, 2015.

[8] Wil MP Van Der Aalst, Marcello La Rosa, and Flávia Maria Santoro. Business process management, 2016.