Explaining Non-Compliance of Business Process Models through Automated Planning

Fabrizio Maria Maggi[1], Andrea Marrella[2],
Giuseppe Capezzuto[2], and Abel Armas Cervantes[3]

[1] University of Tartu, Estonia
f.m.maggi@ut.ee
[2] Sapienza Università di Roma, Italy
lastname@diag.uniroma1.it
[3] The University of Melbourne, Australia
abel.armascervantes@unimelb.edu.au

Abstract. Modern companies execute business processes to deliver products and services, whose enactment requires to adhere to laws and regulations. Compliance checking is the task of identifying potential violations of such requirements prior to process execution. Traditional approaches to compliance checking employ formal verification techniques (e.g., model checking) to identify which process paths in a process model may lead to violations. However, this diagnostics is, in most of the cases, not rich enough for the user to understand how the process model should be changed to solve the violations. In this paper, we present an approach based on finite-state automata manipulation to identify the specific process activities that are responsible to cause violations and, in some cases, suggest reparative actions to be applied to the process model to solve the violations. We show that our approach can be expressed as a planning problem in Artificial Intelligence, which can be efficiently solved by state-of-the-art planners. We report experimental results using synthetic case studies of increasing complexity to show the scalability of our approach.

1 Introduction

This paper falls within the scope of Business Process Management (BPM), an active area of research that is based on the observation that each product and/or service that a company provides to the market is the outcome of a number of activities [13]. Business processes (BPs) are the key instruments for organizing such activities and understanding their interrelationships.

BPs are described using *process models*, which capture the ways in which BP activities are carried out to accomplish a business objective, often with the help of an explicit control flow expressed through a suitable graphical notation, such as the ISO/IEC 19510:2013 standard BPMN. The BPM philosophy is built on the idea that there always exists a BP model that can be used to automate the BP execution. For this reason, process modeling is recognized as one of the most important steps in the BPM lifecycle [13], and the modeling task is nowadays supported by advanced techniques and tools that assist (human) process designers in the definition of the BP model. Nonetheless, despite all efforts, design flaws in BP models may still occur, and their impact may range from syntactically

incorrect models that cannot be properly executed to catastrophic faults that yield legal aftermaths [20]. Consequently, a large branch of research in BPM has focused on devising techniques for BP *verification*, with the aim of identifying and fixing errors prior to BP execution.

The bulk of the research on BP verification has been devoted to check domain-independent correctness criteria that depend exclusively on the structure of BP models, such as proper termination, absence of deadlocks, etc. (e.g., see [1, 7, 17]). However, the design of a BP requires also that its model adheres to several existing domain-dependent *compliance requirements* set by managers, laws, national and international regulations and standards. For instance, the Italian Ministry of Economy and Finance enforces rules to control financial processes in the public sector. The task of identifying potential violations of compliance requirements in a BP model is known as *compliance checking* [28].

Existing approaches to compliance checking employ formal verification techniques (e.g., model checking) to explain violations through a path in the BP from the initial to the error state [6] or by detecting which specific activity [2] or decision [20] has triggered a violation. However, such techniques do not identify *all the activities* of a BP path that can cause violations of compliance requirements. On the other hand, performing this task in a manual way can be time-consuming and error-prone since the amount of compliance requirements to be checked, as well as the number of activities that violate them, can be large.

In this paper, we tackle the above issues by presenting a compliance checking approach to *identify* all those activities of a BP model violating compliance requirements and *explaining* the causes that lead to the violations. The starting points of our approach are a BP model represented in BPMN and a list of compliance requirements expressed as temporal declarative rules with the well-established DECLARE [25] language, which enjoys formal semantics grounded in Linear Temporal Logic with finite execution semantics (LTL_f) [26]. Then, our approach leverages the fact that: *(i)* BPMN models can be converted into Petri nets [12], which can be automatically unfolded to derive execution paths of the BP model; and *(ii)* any LTL_f formula ϕ can be translated into a non-deterministic finite-state automaton (NFA) that accepts all paths satisfied by ϕ [10]. Based on this, we provide a technique, based on NFA manipulations, to detect all activities in any path of the BP model that violate a formula. We also show that such a technique can be expressed as a planning problem in Artificial Intelligence (AI), which can be efficiently solved by state-of-the-art planners.

We have implemented a plug-in of the Apromore BP analytics platform (cf. `http://apromore.org/`) that realizes our approach by employing the FAST-DOWNWARD planner (cf. `http://www.fast-downward.org/`) to solve the compliance checking problem. To motivate the employment of planning techniques, we performed several experiments that show how the complexity of the problem is large and that a technique is necessary that can scale up adequately.

The rest of the paper is organized as follows. Section 2 introduces a running example that will be used to explain our approach. In Section 3, we provide the relevant background necessary to understand the paper. Section 4 presents an overview of our compliance checking approach, while Section 5 discusses how to reduce the compliance checking problem to a planning problem in AI. Then, in Section 6, we report on experiment results performed on synthetic case studies of

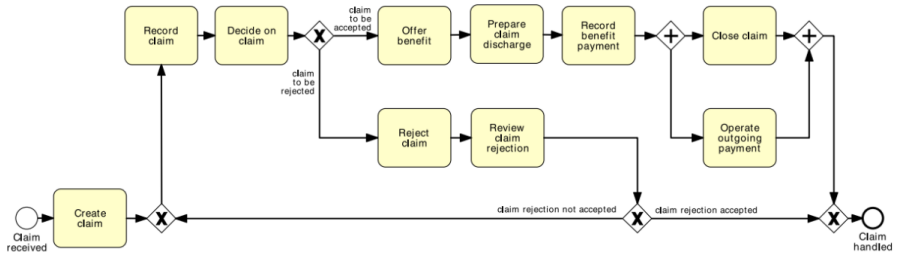

Fig. 1: BPMN model of the running example (original figure in [13])

growing complexity that show the *scalability* of our approach, while, in Section 7, we discuss related work. Finally, in Section 8, we conclude the paper.

2 Running Example

Fig. 1 shows the BPMN model of a running example taken from [13], which will be used in the rest of the paper. It describes a BP for claim handling. A claim is initially created and recorded. Then, a decision is taken on the claim. In case of acceptance, an offer of benefit is made, the claim discharge is prepared, and the benefit payment is recorded. Finally, in parallel, the claim is closed and the outgoing payment is made. In case of rejection, the claim is marked as "rejected". After this, it is possible to review the claim rejection, and in case the claim rejection is not accepted, the claim can be recorded again.

Supposing that this BP is executed in an insurance agency, we can realistically assume that, due to internal regulations or new governmental rules: *(i)* the review of the claim rejection is not allowed anymore; *(ii)* due to the previous rule, the rejection of the claim cannot be followed by a new recording; *(iii)* if the outgoing payment has not been executed, then the claim cannot be closed. Given these compliance requirements, we want to identify the parts of the BPMN model that are not compliant, explain the violations and, in some cases, suggest reparative actions to be applied to the BPMN model to make it compliant with the requirements. Note that techniques based on model checking produce a diagnostics based on counterexamples specifying, for example, that the path ⟨Create claim, Record claim, Decide on claim, Reject claim, Review claim rejection, Record claim, Decide on claim, Offer benefit, Prepare claim discharge, Record benefit payment, Close claim, Operate ongoing payment⟩ is not compliant with the requirements. However, starting from this information, it is hard to pinpoint what is the reason why each requirement is violated and what to do to solve such violations.

3 Background

3.1 BPMN and Petri nets

Many notations have been introduced to represent BPs, such as BPMN, EPC, YAWL or UML Activity Diagrams [13]. These languages allow process designers to specify aspects linked to different perspectives, ranging from expressing the

ordering with which activities need to be executed and the mutual exclusions among activities (control-flow perspective) to modeling the objects manipulated by activities and the resources allowed to execute them. In this paper, we focus on the *core sub-set* of BPMN,[4] which is considered the de-facto standard for modeling BPs. Specifically, to make our approach work, we impose the following syntactic restrictions: *(i)* the admissible flow objects for a BPMN model are activities, start and end events, intermediate events (if they can be translated into activities), exclusive and parallel gateways, and a flow relation is used to connect them; *(ii)* the model provides a finite number of start events and end events; *(iii)* any admissible flow object is on a path from a start to an end event. The use of core BPMN is not a significant limitation, since it realistically allows us to cover the majority of modeling needs [13].

If, on the one hand, BPMN provides an intuitive way for BPM users to model BPs, on the other hand, it is characterized by an ambiguous semantics. Therefore, in order to explain the technical aspects of our approach, we needed a simple language with clear semantics. For this reason, we opted for Petri nets (PNs) [24], which provide the formal foundations of the core sub-set of BPMN [12] and have proven to be adequate for modeling BPs [1]. This is especially true when the focus is only on the control-flow perspective, which is the case in this paper. A PN is a directed bipartite graph with two node types: *places* (graphically represented by circles) and *transitions* (graphically represented by squares) connected via directed arcs. Technically, a PN is a triple (P, T, F) where P and T are the set of *places* and *transitions*, respectively, such that $P \cap T = \emptyset$ and $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation.

At any time, places in a PN may contain a discrete number of marks called *tokens*, drawn as black dots. Any distribution of tokens over the places, formally represented by a total mapping $M : P \mapsto \mathbb{N}$, represents a configuration of the net called a *marking*. When PNs are used to represent BPs, transitions are associated with BP activities, and more specifically to activity labels, and markings indicate the BP state [1]. Since concrete executions of BPs have a start and an end, PNs need to be associated with an initial (respectively final) marking, characterized by the presence of one token in at least one of the starting (respectively ending) places of the PN and no tokens in any other place. The semantics of a PN defines how transitions route tokens through the net so that they correspond to a BP execution. Due to page limit, we refer to [1, 24] for the semantics of PNs. In this paper, we focus on 1-bounded PNs (a.k.a. *safe* PNs), which impose that the number of tokens in all places is at most 1 in all reachable markings, including the initial one. Safe PNs are the basis for best practices in BP modeling, and the behavior allowed by most of real-world BPs can be represented as safe PNs [18].

3.2 Declarative Temporal Rules and Finite State Automata

In order to provide automated support for compliance checking, compliance requirements need to be expressed in a formal language. This, in turn, allows for leveraging mature AI techniques. In this work, we focus on rules that can be expressed using Linear Temporal Logic with finite execution semantics (LTL_f). We refer to [10] for the complete syntax and semantics of LTL_f.

[4] Notice that our approach can easily be transferred to other BP modeling languages.

Since the direct use of temporal logics is inappropriate for most process analysts that conduct BP modeling, we decided to use DECLARE [25] for the specification of compliance requirements. DECLARE is a declarative modeling language that allows us to describe a set of (temporally extended) rules that must be satisfied throughout the BP execution. Unlike procedural models, where all allowed executions must be explicitly represented, in DECLARE, the orderings of activities are implicitly specified by rules and anything that does not violate them is possible during execution. The semantics of DECLARE is grounded on LTL_f. A DECLARE model $\mathcal{D} = (\mathcal{Z}, \pi_{\mathcal{D}})$ consists of a set of activities $\mathcal{Z}$ involved in a BP and a collection of temporal rules $\pi_{\mathcal{D}}$ defined over such activities.

Among all possible LTL_f rules, some specific *patterns* have been singled out as particularly meaningful for expressing DECLARE models. For instance, if we indicate as Rev activity Review claim rejection, *absence*(Rev) means that activity Rev cannot ever be performed; if we indicate as Rej activity Reject claim and as Rec activity Record claim, *not succession*(Rej, Rec) means that if Rej is performed, Rec cannot eventually be performed; finally, if we indicate as Pay activity Operate outgoing payment and as Close activity Close claim, *precedence*(Pay, Close) imposes that activity Pay must precede Close.

Given a BPMN model, the problem we want to address in this paper is to identify and explain *all potential violations* of DECLARE rules in the model. To this end, we exploit the well-known equivalence between (regular) languages and automata: any LTL_f formula ϕ can be associated with a non-deterministic finite-state automaton (NFA) $\mathcal{A}$ that accepts exactly all paths satisfying ϕ [10]. Formally, such NFA is a tuple $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$, where: *(i)* Σ is the *input alphabet*; *(ii)* Q is the finite set of *automaton states*; *(iii)* $q_0 \in Q$ is the *initial state*; *(iv)* $\delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*; and *(v)* $F \subseteq Q$ is the set of *final states*. Let $t = e_1 \cdots e_n$ be a path such that $e_i \in \Sigma$ (with $1 \leq i \leq n$) and $\mathcal{A}$ the NFA associated with an LTL_f formula ϕ. A *computation* of $\mathcal{A}$ on t is a sequence $\sigma = q_0 \xrightarrow{e_1} q_1 \cdots q_{n-1} \xrightarrow{e_n} q_n$ such that, for $i = 0, \ldots, n-1$, there exists a transition $q_i \xrightarrow{e_i} q_{i+1} \in \delta$. Since $\mathcal{A}$ is non-deterministic, there exist, in general, many computations of $\mathcal{A}$ on the path t. We say that $\mathcal{A}$ *accepts* t if there exists a computation σ on t such that the last state is final, i.e., belongs to F.

3.3 Automated Planning

Planning systems are problem-solving algorithms that operate on explicit representations of states and actions [16]. PDDL [15] is the standard Planning Domain Definition Language; it allows us to formulate a *planning problem* $\mathcal{P} = \langle I, G, \mathcal{P}_{\mathcal{D}} \rangle$, where I is the description of the initial state of the world, G is the desired goal state, and $\mathcal{P}_{\mathcal{D}}$ is the planning domain. A planning domain $\mathcal{P}_{\mathcal{D}}$ is built from a set of *propositions* describing the state of the world (i.e., the set of propositions that are true) and a set of *operators* Ω (i.e., *actions*) that can be executed. An *action schema* $a \in \Omega$ is of the form $a = \langle Par_a, Pre_a, Eff_a \rangle$, where Par_a is the list of *input parameters* for a, Pre_a defines the *preconditions* under which a can be executed, and Eff_a specifies the *effects* of a on the state of the world. Both Pre_a and Eff_a are stated in terms of *propositions* in $\mathcal{P}_{\mathcal{D}}$, represented as boolean predicates and numeric fluents.

In recent years, the planning community has developed a plethora of planners that embed very effective (i.e., scaling up to large problems) search heuristics,

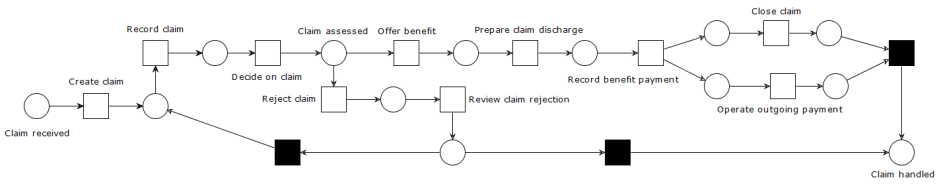

Fig. 2: Petri net derived from the BPMN model in Fig. 1

which have been employed to solve collections of challenging problems from several Computer Science domains [22]. There exist several forms of planning in the AI literature. In this paper, we focus on planning techniques characterized by *fully observable* and *static* domains, i.e., we rely on the *classical* planning assumption of a "perfect world description" [16]. A solution for a planning problem is a sequence of operators—a *plan*—whose execution transforms the initial state I into a state satisfying the goal G. To find a plan, we represent planning domains and problems making use of the STRIPS fragment of PDDL 2.1 [15] enhanced with the numeric features provided by the same language for keeping track of the costs of planning actions and synthesize plans satisfying pre-specified metrics.

4 The Approach

Our approach to compliance checking relies on 4 main steps to be performed in sequence. First of all, in order to be properly enacted, the approach requires that a process analyst provides as inputs: *(i)* a BPMN model to be checked for compliance; *(ii)* a list of compliance requirements expressed as temporal rules in DECLARE [25]; *(iii)* a severity function that assigns non-negative costs to the detected violations (see also Section 5).

With a BPMN model and a set of DECLARE rules as inputs, we rely on well-established transformation algorithms to convert these representations into their corresponding formal counterparts. In particular, we first translate the BPMN model into a PN by leveraging the technique described in [12]. In Fig. 2, we show the PN derived from the BPMN model of Fig. 1. The black-colored transitions are *invisible* transitions, i.e., they do not represent actual pieces of work, but their introduction is sometimes necessary to properly represent the process behavior. Then, we generate all the paths allowed by the PN, i.e., all the complete executions of the PN from the initial to the final marking(s). To this aim, we compute the *complete prefix unfolding* [23] of the PN, which provides a finite behavioral representation of the model. Since PNs can contain cycles, i.e., infinite paths (possibly of infinite length), we adopt the technique defined in [3] for computing the unfolding of the PN. This technique truncates the unfolding once all possible markings of the PN have been observed so that only a finite number of paths of finite length is generated.

We also translate the LTL_f formula ϕ, associated to each input DECLARE rule, into a NFA that accepts exactly all paths satisfied by ϕ, using the technique developed in [10]. For example, the DECLARE rules defined in Section 2 can be represented with the NFAs shown in Fig. 3.

At this point, we invoke an external planner that is in charge of identifying the violations of the compliance requirements in any path extracted from the PN.

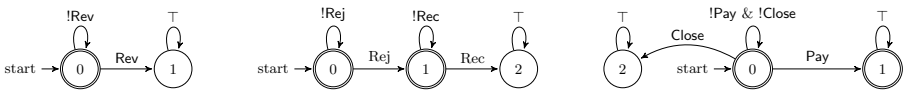

Fig. 3: Automata derived from the DECLARE rules introduced in Section 2: (a) *absence*(Rev), (b) *not succession*(Rej,Rec), (c) *precedence*(Pay, Close)

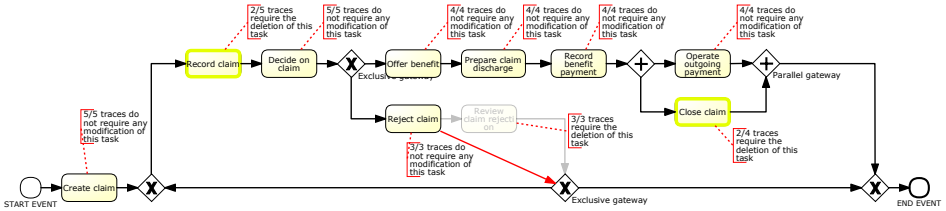

Fig. 4: Visualization in Apromore

In particular, the violations are identified both wrt. each *individual requirement* (by using the *local* automata represented in Fig. 3) and wrt. their *conjunction* (by using a *global* automaton given by the product of all the local automata).

We address this problem by resorting to *cost-optimal planning*, a form of classical planning where actions have costs, and where a successful plan of minimal cost (defined as the sum of the costs of the component actions) has to be found. The intuition behind our solution is that actions capture wrong/missing activities (having non-zero costs defined by the severity function) in the execution path under observation, and the goal is to make the path compliant with the behavior expressed in the automata at a minimal cost. To this aim, the planner implements a technique based on NFA manipulations, which is extensively presented in Section 5 together with its encoding in PDDL.

Finally, we visually present to the process analyst the violations of the compliance requirements on the input BPMN model. Fig. 4 shows the resulting visualization of the running example. The figure is a screenshot of a plug-in implementing the proposed technique available in the Apromore advanced BP analytics platform.[5] The visualization uses a color coding to suggest the amount of paths that support a given change. If 100% of paths support the removal of an activity, then the activity is greyed out; if from 66% to 99% of paths support this removal, then the activity is highlighted in orange; if from 33% to 66% of paths support the removal, then the activity is highlighted in yellow. Similarly, the plug-in shows the percentage of paths supporting the addition of an activity and, in case all paths agree with the addition, the plug-in suggests to add the activity to the model. Note that the plug-in provides a way to automatically repair the model. In particular, the grayed out elements can be removed and the suggested additions can be confirmed. In the running example, the tool suggests to remove Review claim rejection, since all paths support this operation, whereas this is not the case for the violations related to Close claim and Record claim, since these operations are not supported by all paths.

[5] Apromore is an open source platform, and the code as well as the links to the cloud versions of Apromore can be found at `http://apromore.org/`

A screencast showing how the plug-in developed in Apromore works is publicly available at: `https://youtu.be/pVcv5DSSt5A`.

5 Compliance Checking as Planning

In this section, we first demonstrate that the problem of identifying violations of compliance requirements in a BP path can be solved with a technique based on NFA manipulations (Section 5.1), and then we show how this technique can be encoded as a planning problem in PDDL (Section 5.2).

5.1 A NFA Manipulations Technique for Compliance Checking

Let us consider a BP path $t = \langle e_1, ..., e_{k-1}, e_k, e_{k+1}, ..., e_n \rangle$ and an LTL$_f$ formula ϕ. We are interested in "transforming" t into a new path $\hat{t}$ that is compliant with ϕ. To realize this transformation, we consider two kinds of violations, which can be caused by *wrong* or *missing* activities, respectively. For example, suppose that $e_k \in t$ violates ϕ: e_k is said to be *wrong* wrt. ϕ, and its deletion from t results in a new path $\hat{t} = \langle e_1, ..., e_{k-1}, e_{k+1}, ..., e_n \rangle$ that is compliant with ϕ. Similarly, a *missing* activity p can be added to a non-compliant path t at position k (with $1 \leq k \leq n+1$) to make it compliant wrt. ϕ. After the addition, the resulting compliant path is $\hat{t} = \langle e_1, ..., e_{k-1}, p, e_k, e_{k+1}, ..., e_n \rangle$.

The *addition* and *deletion* actions allow us to understand if the reason of a non-compliance path is due to the absence/presence of missing/wrong activities in t. Furthermore, these two actions are characterized by two values for the cost quantifying the severity of the violation found. The final cost of the transformation will be the sum of the number of deletion multiplied by the deletion cost plus the number of addition multiplied by the addition cost. Given what explained above, we can define the compliance checking problem as follows:

Definition 1 (Compliance Checking). *Given a BP path t and an* LTL$_f$ *formula ϕ such that t violates ϕ, find a path $\hat{t}$ that satisfies ϕ and such that the transformation cost is minimal.*

Compliance checking can be addressed by resorting to NFA. To see this, let $t = \langle e_1, ..., e_n \rangle$ be a BP path, ϕ the LTL$_f$ rule to check t against, and $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$ the corresponding NFA, which we call the *constraint automaton*. From t, we define a further automaton, called the *path automaton*, $\mathcal{T} = \langle \Sigma_t, Q_t, q_0^t, \delta_t, F_t \rangle$, where: *(i)* $\Sigma_t = \{e_1, ..., e_n\}$; *(ii)* $Q_t = \{q_0^t, \ldots, q_n^t\}$ is a set of $n+1$ states; *(iii)* $\delta_t = \bigcup_{i=0,\ldots,n-1} \langle q_i^t, e_{i+1}, q_{i+1}^t \rangle$; *(iv)* $F^t = \{q_n^t\}$. By construction, $\mathcal{T}$ is deterministic and accepts only t.

Next, we augment $\mathcal{T}$ and $\mathcal{A}$ to make them suitable to our definition of compliance checking, i.e., by adding transitions related to *addition* and *deletion* of activities. From $\mathcal{T}$, we generate the automaton $\mathcal{T}^+ = \langle \Sigma_t^+, Q_t, q_0^t, \delta_t^+, F_t \rangle$, where:

- Σ_t^+ contains all the activities in Σ_t, plus: one new activity *del_p*, for all activities $p \in \Sigma_t$; and one new activity *add_p*, for all activities $p \in \Sigma \cup \Sigma_t$;
- δ_t^+ contains all the transitions in δ_t, plus: a new transition $\langle q, del\_p, q' \rangle$, for all transitions $\langle q, p, q' \rangle \in \delta_t$; and, for all activities $p \in \Sigma \cup \Sigma_t$ and states $q \in Q_t$, a new transition $\langle q, add\_p, q \rangle$.

For example, if we indicate as Crt activity Create claim, and as Dec activity Decide on claim, the augmented path automaton $\mathcal{T}^+$ associated to path $t_{ex} = \langle \mathsf{Crt}, \mathsf{Rec}, \mathsf{Dec}, \mathsf{Rej}, \mathsf{Rev} \rangle$, derived from the BP model of our running example, is shown in Fig. 5.

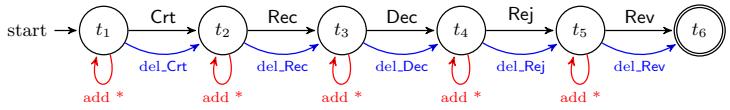

Fig. 5: Example of augmented path automaton

Similarly, from $\mathcal{A}$, we obtain an automaton $\mathcal{A}^+ = \langle \Sigma^+, Q, q_0, \delta^+, F \rangle$, such that: *(i)* Σ^+ contains all the activities in Σ, plus: one new activity add_p, for all activities $p \in \Sigma$; and one new activity del_p, for all activities $p \in \Sigma \cup \Sigma_t$; and *(ii)* δ^+ contains all the transitions in δ, plus: one new transition $\langle q, del\_p, q \rangle$ for all $q \in Q$ and $p \in \Sigma \cup \Sigma_t$; and one new transition $\langle q, add\_p, q' \rangle$ for all transitions $\langle q, p, q' \rangle \in \delta$. For instance, the DECLARE rules defined in our running example can be represented with the three augmented constraint automata shown in Fig. 6.

Intuitively, $\mathcal{A}^+$ accepts all paths $\hat{t}$ that satisfy ϕ and have been obtained by adding/removing missing/wrong activities to/from t, with the additions/deletions explicitly marked. For instance, if we consider path t_{ex} and its augmented path automaton in Fig. 5, neither the constraint automata in Fig. 3 nor their augmented versions in Fig. 6 accept t_{ex}. However, if we "repair" t_{ex} by removing Rev at the end, and we explicitly mark the repair with $del_$Rev, then all the augmented automata accept the new path $\hat{t}_{ex} = \langle \mathsf{Crt}, \mathsf{Rec}, \mathsf{Dec}, \mathsf{Rej}, del\_\mathsf{Rev} \rangle$.

Thus, given a BP path t and many LTL$_f$ rules $\phi_1, \ldots, \phi_n$, compliance checking is equivalent to searching for a "repaired path" $\hat{t}$ accepted by both $\mathcal{T}^+$ and $\mathcal{A}_1^+, \ldots, \mathcal{A}_n^+$ (i.e., the augmented constraint automata for all LTL$_f$ rules), with a minimal number of add/delete activities. We next show how to take advantage of the planning technology to efficiently search for the desired repaired path.

5.2 Encoding in PDDL

In this section, we show how, given a set of augmented constraint automata $\mathcal{A}_1^+, .., \mathcal{A}_n^+$ obtained from n LTL$_f$ formulas $\phi_1, .., \phi_n$, and an augmented path automaton $\mathcal{T}^+$ obtained from a path t, we build a cost-optimal planning domain $\mathcal{P}_\mathcal{D}$ and a problem instance $\mathcal{P}$ in PDDL. $\mathcal{P}_\mathcal{D}$ and $\mathcal{P}$ can be used to feed any state-of-the-art planners accepting PDDL 2.1 specification, as discussed in Section 3.3. A solution plan for $\mathcal{P}$ amounts to the set of interventions of minimal cost to repair the path wrt. the LTL$_f$ formulas.

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

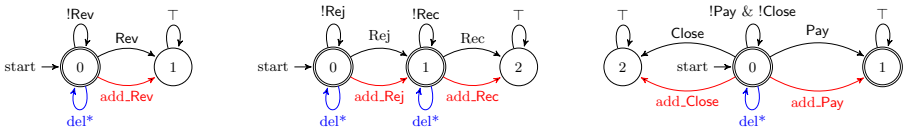

Fig. 6: Augmented constraint automata for DECLARE rules: (a) *absence*(Rev), (b) *not succession*(Rej,Rec), (c) *precedence*(Pay, Close)

- `(path ?t1 - path_state ?e - activity ?t2 - path_state)` holds if there exists a transition in the path automaton between two states `t1` and `t2`, being `e` the activity involved in the transition.
- `(automaton ?s1 - automaton_state ?e - activity ?s2 - automaton_state)` holds if there exists a transition between two states `s1` to `s2` of a constraint automaton, being `e` the activity involved in the transition.
- `(cur_state ?s - state)` holds if `s` is the current state of a constraint/path automaton.
- `(final_state ?s - state)` holds if `s` is a final state of a constraint/path automaton.

Furthermore, we define a *numeric fluent* `total-cost` to keep track of the cost of the violations. Notice that: *(i)* in PDDL, parameters are written with a question mark character '?' in front, and the dash character '-' is used to assign types to parameters; and *(ii)* we remain consistent with the PDDL terminology, which allows for both the values of predicates and fluents to change as a result of the execution of an action.

Planning actions are used to express the *repairs* on the original path t. Each action is characterized by its *preconditions* and *effects*, stated in terms of the domain propositions. In our encoding, we have defined three actions to perform *synchronous moves* in the path automaton and in the constraint automata, or to add/remove activities to/from the path automaton. In the following example, we suppose that both actions `add` and `del` have cost equal to 1. We notice that their cost can be customized to properly define the severity of a violation.

```
(:action sync
 :parameters (?t1 - path_state ?e - activity ?t2 - path_state)
 :precondition (and (cur_state ?t1) (path ?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                                    (:action del
 :parameters (?e - activity)                     :parameters (?t1 - path_state
 :effect (and (increase (total-cost) 1)                       ?e - activity
         (forall (?s1 ?s2 - automaton_state)                  ?t2 - path_state)
           (when (and (cur_state ?s1)            :precondition (and (cur_state ?t1)
                      (automaton ?s1 ?e ?s2))                      (path ?t1 ?e ?t2))
                 (and (not (cur_state ?s1))      :effect(and (increase (total-cost) 1)
                      (cur_state ?s2))))))          (not (cur_state ?t1))(cur_state ?t2)))
```

We modeled `sync` and `del` in such a way that they can be applied only if there exists a transition from the current state `t1` of the path automaton to a subsequent state `t2`, being `e` the activity involved in the transition. Notice that, while `del` yields a *single* move in the path automaton, `sync` yields, in addition, one

Path Length	Search Time (isolated rules)	Search Time (entire model)	Search Time (isolated rules)	Search Time (entire model)	Search Time (isolated rules)	Search Time (entire model)
0 const. violated	5 constraints		10 constraints		15 constraints	
10	615	146	1,189	282	1,763	1,764
25	725	164	1,454	295	2,181	1,792
35	732	165	1,508	300	2,256	1,799
50	774	172	1,542	311	2,292	1,801
1 const. violated	5 constraints		10 constraints		15 constraints	
10	599	147	1,173	414	1,769	3,716
25	646	156	1,304	421	1,971	3,665
35	719	166	1,450	439	2,181	3,889
50	747	172	1,481	458	2,250	4,091
3 const. violated	5 constraints		10 constraints		15 constraints	
10	596	158	1,164	558	1,727	5,700
25	725	187	1,422	586	2,160	6,186
35	752	196	1,518	618	2,286	6,432
50	771	201	1,535	631	2,319	6,710
5 const. violated	5 constraints		10 constraints		15 constraints	
10	614	162	1,214	625	1,838	6,113
25	760	207	1,552	626	2,310	6,362
35	767	211	1,569	628	2,364	6,401
50	801	219	1,651	641	2,506	6,612

Table 1: Experimental results: the time (in milliseconds) is the average per path

move per constraint automaton (all to be performed synchronously). In particular, a synchronous move is performed in each constraint automaton for which there exists a transition involving activity `e` connecting `s1` – the current state of the automaton – to a state `s2`. Finally, `add` is performed only for transitions involving activity `e` connecting two states of any constraint automaton, with the current state of the path automaton that remains the same after the execution of the action.

Planning Problem. In $\mathcal{P}$, we first define a finite set of constants required to properly ground all the domain propositions defined in $\mathcal{P}_{\mathcal{D}}$. In our case, constants correspond to the state and activity instances involved in the path automaton and in any constraint automaton. Secondly, we define the *initial state* of $\mathcal{P}$ to capture the exact structure of the path automaton and any constraint automaton. This includes the specification of all the existing transitions that connect two states of the automata. The current state and the final states of any path/constraint automaton are identified as well. Thirdly, to encode the goal condition, we first pre-process each constraint automaton by: *(i)* adding a new dummy state with no outgoing transitions; *(ii)* adding a new special action, executable only in the final states of the original automaton, which makes the automaton move to the dummy state; and *(iii)* including in the set of final states only the dummy state. Then, we define the goal condition as the conjunction of the final states of the path automaton and of all the constraint automata. In this way, we avoid using disjunctions in goal formulas, which are not supported by all planners. Finally, as our purpose is to minimize the total cost of the plan, $\mathcal{P}$ contains the following specification: `(:metric minimize (total-cost))`.

6 Evaluation

In order to investigate the *level of scalability* of our planning-based approach, we performed (with our tool) several synthetic experiments employing BP models and compliance requirements of increasing complexity. First, we created 4 BPMN

models with an increasing number of parallel branches.[6] Then, we converted the BPMN models into Petri nets and unfolded them thus obtaining 1121, 800, 961 and 1025 execution paths of average length 10, 25, 35 and 50, respectively.

Secondly, we defined 3 DECLARE models (having the same alphabet of activities) containing 5, 10 and 15 rules that are known to be compliant with the tested BPMN models. Then, to create DECLARE models non-compliant with the BPMN models, we changed some of the rules in the original DECLARE models by replacing 1, 3 and 5 rules in each model with their negative counterparts. In this way, we were able to understand how performance scales up with longer paths and more DECLARE rules, considering a growing amount of violations. We point out that real execution paths involve most often less than 50 activities, and compliance requirements with 15 DECLARE rules are considered to be large. So, one should consider our test not only as a practical one based on realistic settings, but also as one that is challenging. We used a standard cost function with cost 1 for any step that adds/removes activities to/from the input path, and cost 0 for synchronous moves. We tested our approach on the grounded version of the problem presented in Section 5.2. The experiments were performed with a machine consisting of a 2,7 GHz Intel Core i5 CPU and 8GB RAM. We configured the FAST-DOWNWARD planner, which is integrated in our tool, to employ the A* searching algorithm to guarantee the optimality of the solution.

The results of our experiments can be seen in Table 1 and Fig. 7. Concerning the table, for each combination "path length - size of DECLARE model", the table reports the search time (averaged over all paths) required by the planner for checking the compliance of all the rules included in the DECLARE model: *(i)* when tested in isolation (column "Search Time (isolated rules)"); and *(ii)* when tested as a conjunction (column "Search Time (entire model)"). Notice also that the rows of the table are split into clusters according to the number of DECLARE rules violated by any of the tested paths.

By analyzing Table 1 and Fig. 7, some conclusions can be drawn. Regarding the experiments performed to check the compliance of a path against the individual rules included in a DECLARE model, it is evident that the amount of rule violations has a small influence on the performance, differently from the size of the DECLARE models and the length of the paths. The latter two factors have a major influence on the search time, though not in a dramatic way to preclude from practical applicability. Conversely, the tests performed against the entire DECLARE models suggest that the presence of a large number of violations becomes the key factor that influences the search time. This is more evident with the largest DECLARE model consisting of 15 rules. However, the results show that the approach is feasible for offline reasoning and scales up well when DECLARE models and paths grow in size.

7 Related Work

In the context of process mining, some techniques have approached the problem of BP model repair based on event logs [21, 14, 27, 4]. These techniques aim at

[6] To guarantee the repeatability of our experiments, at `https://goo.gl/rwNRw7`, we provide: the instructions to configure and run the tool; the BPMN and DECLARE models used for the experiments.

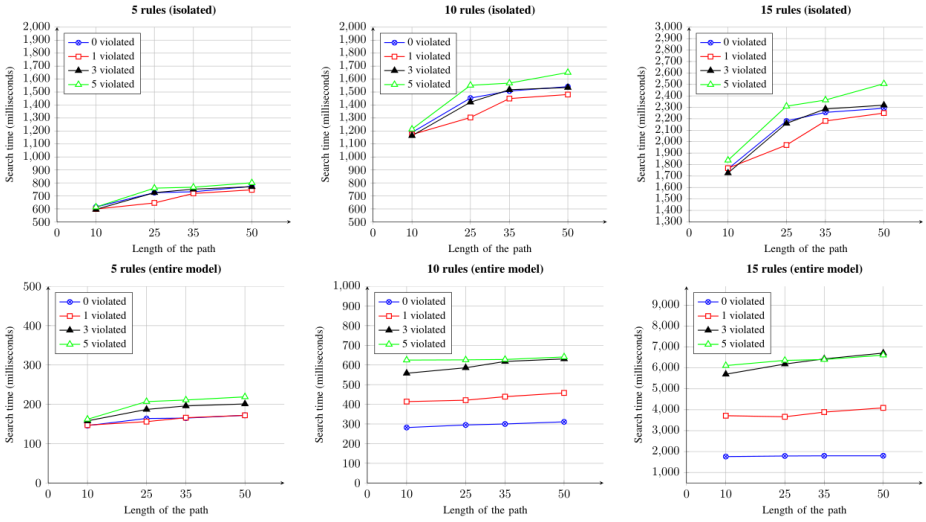

Fig. 7: Performance of repairing violations through planning

keeping the repaired model as similar as possible to the original model, while capturing all the possible behaviors from the event log. Some research works also exist that use planning techniques to deal with trace alignment in the context of conformance checking [9, 19, 8]. Readers should observe that the aforementioned approaches use event logs, while our approach is able to compare BP models and temporal rules.

Approaches addressing a problem that is closer to the one we face in this paper are presented in [6, 20]. These approaches provide insights about violations of temporal rules in BP models through model checking. In [6], the authors use BPMN-Q to express compliance requirements visually and computational tree logic (CTL) expressions to formally represent their semantics. Based on these requirements a counterexample is provided to explain possible discrepancies wrt. an input BPMN model. As already mentioned, model checkers provide counterexamples to explain non-compliance, but the diagnostics can be difficult to interpret. In [20], the authors aim at reducing the error paths produced by model checkers to make their diagnostics more user-friendly. This approach provides the user with diagnostics to understand the root cause of the violations, but it does not provide detailed feedback to the user in terms of where each compliance requirement is violated and why, and what to do to solve the non-compliance. Instead, we try to provide a richer feedback that can be used to adjust the input BP model in a semi-automatic way.

In [5], the authors introduce a compliance checking approach that identifies 4 pre-defined violation types and, for each of them, generates a resolution strategy using automated planning. While we adopt a similar technique to the synthesis of repair strategies, our approach is able to detect violations of any rule that can be represented as an NFA.

In [11], the authors still deal with compliance checking of a BP model wrt. temporal rules. However, this work is more focused on finding reparative actions to be applied to the model to solve the violations. This problem is addressed by

making some assumptions on the input BP model and compliance requirements. In particular, the input model needs to be a block-structured Workflow net without loops. In addition, the compliance requirements can be expressed with a sub-set of DECLARE and the intersection of the set of the behaviors of the BP model and the set of the behaviors of the DECLARE rules needs to be non-empty. These assumptions are needed to solve the repair problem, which is, in general, extremely challenging. In our contribution, we find a different trade-off in addressing this problem by providing general insights on where and why a violation occurs and providing only in some specific cases suggestions about reparative actions. This allows us to relax the assumptions made in [11]. Our approach can indeed be applied to any safe PN and any compliance requirement that can be expressed in LTL_f.

For a general introduction to the topic of regulatory compliance checking, the reader is referred to [29].

8 Concluding Remarks

Existing approaches to compliance checking employ verification techniques to explain violations through counterexamples, which can be extremely unintuitive in the presence of a large number of violations. Consequently, explaining violations using counterexamples could not be the most suitable solution to understand how to change BP models to solve non-compliance. To tackle this issue, in this paper, we have shown how automated planning can be used to efficiently solve the problem of checking compliance requirements expressed in terms of LTL_f rules, by pointing at the BP activities where compliance is breached.

It is worth noticing that the objective of the evaluation was to stress the scalability of planning techniques for checking the compliance of *single* BP paths and DECLARE models of *growing size*. However, when BP models include a large number of parallel gateways and cycles, the number of unfolded BP paths quickly explodes. As a consequence, part of the complexity moves from the size of BP paths and DECLARE models to the total amount of BP paths to be checked for compliance. Therefore, as future work, we aim at improving the presented technique to make it able to suggest reparative actions in larger sets of paths wrt. the ones considered in this paper. In addition, we plan to detect non-compliance that relates to time, resource and data aspects, such as activities that are not performed by authorized actors or within given deadlines. This problem is far from being trivial since it is, in general, undecidable. Thus, we need to explore what kind of expressiveness limitations are required to ensure decidability.

References

1. van der Aalst, W.M.P.: The application of Petri nets to workflow management. Journal of Circuits, Systems and Computers 08, 21–66 (1998)
2. van der Aalst, W.M.P., De Beer, H.T., van Dongen, B.F.: Process mining and verification of properties: An approach based on temporal logic. In: CoopIS (2005)
3. Armas-Cervantes, A., Baldan, P., Dumas, M., García-Bañuelos, L.: Diagnosing behavioral differences between business process models: An approach based on event structures. Inf. Sys. pp. 304 – 325 (2016)

4. Armas-Cervantes, A., van Beest, N.R.T.P., La Rosa, M., Dumas, M., García-Bañuelos, L.: Interactive and incremental business process model repair. In: CoopIS (2017)
5. Awad, A., Smirnov, S., Weske, M.: Resolution of Compliance Violation in Business Process Models: A Planning-Based Approach. In: CoopIS (2009)
6. Awad, A., Weidlich, M., Weske, M.: Visually specifying compliance rules and explaining their violations for business processes. Visual Lang. & Comp. 22(1) (2011)
7. Clempner, J.: Verifying soundness of business processes: A decision process Petri nets approach. Expert Systems with Applications 41(11), 5030–5040 (2014)
8. De Giacomo, G., Maggi, F.M., Marrella, A., Patrizi, F.: On the Disruptive Effectiveness of Automated Planning for LTLf-Based Trace Alignment. In: AAAI (2017)
9. De Giacomo, G., Maggi, F.M., Marrella, A., Sardiña, S.: Computing Trace Alignment against Declarative Process Models through Planning. In: ICAPS (2016)
10. De Giacomo, G., Vardi, M.Y.: Synthesis for LTL and LDL on finite traces. In: IJCAI. vol. 15, pp. 1558–1564 (2015)
11. De Masellis, R., Di Francescomarino, C., Ghidini, C., Lapõnin, A., Maggi, F.M.: Rule propagation: Adapting procedural process models to declarative business rules. In: EDOC (2017)
12. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and Analysis of Business Process Models in BPMN. Inf. Softw. Technol. 50(12), 1281–1294 (2008)
13. Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management. Springer (2013)
14. Fahland, D., van der Aalst, W.M.P.: Model repair - aligning process models to reality. Inf. Syst. 47, 220–243 (2015)
15. Fox, M., Long, D.: PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. J. Artif. Intell. Res.(JAIR) 20, 61–124 (2003)
16. Ghallab, M., Nau, D.S., Traverso, P.: Automated Planning: Theory and Practice. MK (2004)
17. Kheldoun, A., Barkaoui, K., Ioualalen, M.: Formal verification of complex business processes based on high-level Petri nets. Information Sciences 385, 39–54 (2017)
18. Kiepuszewski, B., ter Hofstede, A.H.M., van der Aalst, W.M.P.: Fundamentals of control flow in workflows. Acta Informatica 39(3) (2003)
19. de Leoni, M., Marrella, A.: Aligning Real Process Executions and Prescriptive Process Models through Automated Planning. Exp. Syst. with App. 82 (2017)
20. Lohmann, N., Fahland, D.: Where Did I Go Wrong? Explaining Errors in Business Process Models. In: BPM (2014)
21. Maggi, F.M., Corapi, D., Russo, A., Lupu, E., Visaggio, G.: Revising process models through inductive learning. In: BPM Workshops - BPI. pp. 182–193 (2010)
22. Marrella, A.: What Automated Planning can do for Business Process Management. In: BPM Workshops - BPAI (2017)
23. McMillan, K.L., Probst, D.K.: A Technique of State Space Search Based on Unfolding. Formal Methods in System Design 6(1), 45–65 (1995)
24. Murata, T.: Petri nets: Properties, analysis and applications. IEEE (1989)
25. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: DECLARE: Full Support for Loosely-Structured Processes. In: EDOC. pp. 287–300 (2007)
26. Pnueli, A.: The temporal logic of programs. In: Found. of Computer Science (1977)
27. Polyvyanyy, A., van der Aalst, W.M.P., ter Hofstede, A.H.M., Wynn, M.T.: Impact-driven process model repair. ACM Trans. Softw. Eng. Methodol. 25(4), 1–60 (2016)
28. Sadiq, S.W., Governatori, G., Namiri, K.: Modeling control objectives for business process compliance. BPM (2007)
29. Sadiq, S.W., Governatori, G.: Managing regulatory compliance in business processes. In: Handbook on Business Process Management 2, Strategic Alignment, Governance, People and Culture, 2nd Ed., pp. 265–288 (2015)