Resolution of Compliance Violation in Business Process Models: A Planning-Based Approach

Ahmed Awad, Sergey Smirnov, and Mathias Weske

Business Process Technology Group
Hasso Plattner Institute at the University of Potsdam
D-14482 Potsdam, Germany
{Ahmed.Awad,Sergey.Smirnov,Mathias.Weske}@hpi.uni-potsdam.de

Abstract. Keeping business processes compliant with regulations is of major importance for companies. Considering the huge number of models each company possesses, automation of compliance maintenance becomes essential. Therefore, many approaches focused on automation of various aspects of compliance problem, e.g., compliance verification. Such techniques allow localizing the problem within the process model. However, they are not able to resolve the violations. In this paper we address the problem of (semi) automatic violation resolution, addressing violations of execution ordering compliance rules. We build upon previous work in categorizing the violations into types and employ automated planning to ensure compliance. The problem of choosing the concrete resolution strategy is addressed by the concept of context.

Keywords: Business process modeling, compliance checking, process model parsing, process model restructuring.

1 Introduction

In today's business being compliant with regulations is vital. Process models provide enterprises an explicit view on their business. Thus, it is rational to employ process models for compliance checking. Companies hire experts to audit their business processes and to evidence process compliance to external/internal controls. Keeping processes compliant with constantly changing regulations is expensive [12].

Compliance requirements (rules) originate from different sources and address various aspects of business processes. For instance, a certain order of execution between activities is required. Other rules force the presence of activities under certain conditions, e.g., reporting banking transactions to a central bank, if large deposits are made. Violations of compliance requirements originating from regulations, e.g., the Sarbanes-Oxley Act of 2002 (see [1]), lead to penalties, scandals, and loss of reputation. Several approaches have been proposed to handle the divergent aspects of compliance on the level of process models. Most of them are focused on model compliance checking, i.e., on model verification problem [2,7,11,15].

Although the problem of compliance violation resolution was discussed in literature, it is usually perceived as a human expert task. However, it would be possible to (semi-) automate the task of resolving compliance violations. This would be valued as an aid to the human expert to speed up the process of ensuring compliance. In [3], we made the

R. Meersman, T. Dillon, P. Herrero (Eds.): OTM 2009, Part I, LNCS 5870, pp. 6–23, 2009.

first step towards resolving violations of compliance rules regarding execution ordering of activities by identifying the different violation patterns.

In this paper we show how automated planning techniques can be used for resolution of compliance violations. We present resolution algorithms for violation patterns identified in [3] and explain the role of *resolution context*. The developed approach assumes that compliance violations can be resolved sequentially, one after another. This implies that there are no contradictions between compliance rules: for any two rules r_1 and r_2, resolution of r_1 violation does not lead to violation of r_2 and vice versa.

The rest of the paper is organized as follows. Section 2 provides the necessary formalism. Section 3 describes a motivating example. Section 4 discusses a set of resolution algorithms for the different violation patterns. The related work is presented in Section 5. Section 6 concludes the paper and discusses future work.

2 Preliminaries

In this section we introduce the basic concepts, supporting the violation resolution approach. As resolutions are realized on a structural level, we introduce a supporting formalism—the concept of process structure trees. Further, we show how the problem domain can be described by a *resolution context* and the task of violation resolution can be interpreted in terms of automated planning.

2.1 Process Structure Tree

Correction of compliance violations assumes analysis and modification of business process models on the structural level. Hence, we need a technique efficiently supporting these tasks. We rely on the concept of a process structure tree (PST), which is the process analogue of abstract syntax trees for programs. The concept of a PST is based on the unique decomposition of a process model into fragments. Fragments, which are the decomposition result, are organized into a hierarchy according to the nesting relation. This hierarchy is called a process structure tree. PSTs can be constructed using various algorithms. One approach is a decomposition into *canonical single entry single exit (SESE) fragments*, formally described in [25]. Informally, SESE fragments can be defined as fragments with exactly one incoming and one outgoing edge. The node sets of two canonical SESE fragments are either disjoint, or one contains the other. Following [25], we consider the maximal sequence of nodes to be a canonical SESE fragment. As we assume process models to be structured workflows, the SESE decomposition suits the task well.

Definition 1. A *process structure tree* $P = (N, r, E, type)$ is a tree, where:

- N is a finite set of nodes, where nodes correspond to canonical SESE fragments
- $r \in N$ is the root of the tree
- $E \subseteq (N \times (N \setminus \{r\}))$ is the set of edges. Let tree nodes $n_1, n_2 \in N$ correspond to SESE fragments f_1 and f_2, respectively. An edge leads from n_1 to n_2 if SESE fragment f_1 is the direct parent of f_2
- $type : N \rightarrow \{\texttt{act}, \texttt{seq}, \texttt{and}, \texttt{xor}, \texttt{or}, \texttt{loop}\}$ is a function assigning a type to each node in N: `act` corresponds to activities, `seq`—sequences, `and`, `xor`, `or`—blocks of corresponding type, `loop`

- $N_{<type>} \subseteq N$ denotes the set of nodes with specific type, e.g., N_{seq} are the nodes of type seq.

A process model may contain several occurrences of one activity (e.g., activity A). Then, the model's PST has the set of nodes which correspond to occurrences of A. To address such a set of nodes we denote it with $N_a \subseteq N$. Since a tree has no cycles, a path between two nodes is a unique sequence of nodes.

Definition 2. A *path* between two nodes $n_0, n_k \in N$, is a sequence of nodes $path(n_0, n_k) = (n_0, n_1, \ldots, n_k)$ where $(n_i, n_{i+1}) \in E, 0 \leq i < k$. If there is no path between n_0 and n_k we denote it with $path(n_0, n_k) = \perp$. We write $n \in path(n_0, n_k)$ to express the fact that n lies on the path from n_0 to n_k.

Definition 3 formalizes the notion of the least common ancestor of two nodes.

Definition 3. The *least common ancestor* of two nodes $n, m \in N$ in the $P = (N, r, E, type)$ is a node $lca(n, m) = \{p : p \in N \wedge p \in path(r, n) \wedge p \in path(r, m) \wedge \nexists p'(p' \neq p \wedge p' \in path(r, n) \wedge p' \in path(r, m) \wedge p \in path(r, p'))\}$.

Depending on the type of the least common ancestor of two nodes, we can determine the behavioral relation between them, either sequence, choice, parallel, etc. If a node is of type seq the execution order for its direct children is defined.

Definition 4. The *order* of execution of a node $n \in N$ with respect to node $p \in N_{seq}$ is a function $order : N_{seq} \times N \rightarrow \mathbb{N}$, defined if $(p, n) \in E$ and where the first argument is the parent node and the second—its child.

To reflect the effect of data on formulating conditions for branches, we define a function $condition$ as follows.

Definition 5. *Let Pr be a set of predicates representing data conditions. A function $condition : N_{seq} \cup N_{loop} \rightarrow 2^{Pr}$ associates with each sequence or loop fragment a condition. An empty condition is evaluated as $true$.*

2.2 Catalog of Violations

We are concerned with resolving violations to execution ordering rules. Generally, execution ordering rules can be divided into *leads to* and *precedes* rules [2]. Informally, a rule A *leads to* B requires that after every execution of A, B must *eventually be executed*. On the other hand, a rule A *precedes* B requires that before executing B, A must have been executed before. Variants of these rules can be derived [4]. In this paper, we develop algorithms for resolving the *leads to* rules. These algorithms can be symmetrically applied to the *precedes* case and adapted to the variants. In general, four types of violation can be identified:

Splitting Choice. Activity A executes and B can be skipped due to the choice of an alternative branch.

Different Branches. If A and B are on different threads of the process.

Inverse Order. If A and B appear in order different then specified by the rule.

Lack of Activity. Depending on the rule type, for instance A *leads to* B, if a process model lacks activity B.

Since a process model might contain more than one occurrence of the activities A and B under investigation, we assume a priori knowledge about the pairing of such occurrences. In this paper we study in detail the different violations to the *leads to* rules. All results achieved can be symmetrically applied to the case of *precedes* rules.

2.3 The Resolution Context

The resolution context represents a global process independent description of the business activities. This context describes various relations between activities. We call these relations $aspects$ of the context. With the notion of the context, we try to simulate the knowledge needed while process models are first composed or later on modified.

Definition 6. The *resolution context C* is a 7-tuple $(N_{act}, A, T, asptype, con_{t\in T}, pre_{t\in T}, post_{t\in T})$, where:

- N_{act} is the set activities
- A is the set of objects, which define model aspects
- T is the set of aspect types
- $asptype : A \rightarrow T$ is the function relating each object to a particular aspect
- $con : N_{act} \times N_{act}$ is the relation between two activities, indicating contradiction
- $pre_{t\in T} \subseteq N_{act} \times 2^{\{a:\forall a\in A, type(a)=t\}}$ is the relation defining the prerequisites of activity execution in terms of aspects
- $post_{t\in T} \subseteq N_{act} \times 2^{\{a:\forall a\in A, type(a)=t\}}$ is the relation defining the result of activity execution in terms of aspects. Postconditions of an activity can be divided into positive and negative, i.e.,$post_{t\in T} = post^{+}_{t\in T} \cup post^{-}_{t\in T}$

One *aspect* of a context is a tuple $(N_{act}, A_t, t, asptype, con_t, pre_t, post_t)$, where $t \in T \wedge A_t = \{a : a \in A, asptype(a) = t\}$.

Definition 6 captures aspects with three elements: sets A and T and function $asptype$. Set A is the set of objects, describing the business environment from a certain perspective, e.g., dependencies of activities on data objects or their semantic annotations. Set T consists of the object types; an example is $T = \{activity, data, semantic\ Annotation\}$. Function $asptype$ specifies a type (element of set T) for an element of A. Typification of objects allows distinguishing aspects, e.g., distinguishing data flow from semantic description of a process.

The three basic relations are pre, $post$, and con. They respectively describe preconditions, postconditions, and contradiction relations. The pre relation describes what objects with different (types) aspects that are required for a certain activity in order to execute it. Similarly, the $post$ relation specifies what are the effects of executing a certain activity. For each activity there might be different sets of pre/post conditions to resemble the notion of alternation. Taking data as an aspect to describe such relations, pre_{data} would describe the precondition of each activity in terms of data elements. The con relation describes contradictions between activities. If two activities are known to be contradicting, at most one is allowed to appear in any process instance.

We assume process models to be consistent with the context: at least one precondition for any activity in the model must be satisfied in the process model. Also, activities are assumed to produce the effect, post condition(s), as described in the context. Moreover, for any two contradicting activities, there must not be a chance to execute both of them in a single instance.

Definition 7. *A process model and its PST* $P = (N, r, E, type)$ *are consistent with a resolution context* $C = (N_{act}, A, T, asptype, con_{t\in T}, pre_{t\in T}, post_{t\in T})$ *if:*

- *Preconditions are satisfied:* $\forall n \in N_{act} \wedge \forall t \in T \wedge pre_t \neq \emptyset \wedge \forall a \in A \wedge asptype(a) = t \wedge (n, a) \in pre_t : n \in N \rightarrow \exists m \in N_{act} : m \in N \wedge (m, a) \in post_t \wedge ((type(lca(m, n)) = seq \wedge order(lca(m, n), m) < order(lca(m, n), n)) \vee (type(lca(m, n)) = loop \wedge$ *m is on the mandatory part of the loop*$)$
- *No two contradicting execute in the same instance:* $\forall n, m \in N_{act} \wedge (n, m) \in con : N_n = \emptyset \vee N_m = \emptyset \vee \forall n' \in N_n \forall m' \in N_m type(lca(n', m')) = xor$

Similarly, the requirement imposed by a compliance rule has to be consistent with the context. For instance, a compliance rule must not impose order between two activities that are known to be contradicting according to the context. In this paper, we consider only rules that are consistent with the context; while might be violated by some process models. The resolution context plays central role assuring model consistency once changes have been applied to make them compliant.

2.4 Automated Planning

A violation resolution often implies that a business process logic is changed. The task is always to come up with a compliant model, fulfilling the business goal. This implies that a process should be reorganized to assure that requirements are satisfied. Given that the set of activities required to construct a compliant process is available in the context, this task can be approached with techniques of automated planning [21].

The problem of automated planning can be described as follows. Given a system in an initial state it is required to come up with a sequence of actions, bringing the system to the goal state. The sought sequence of actions is called a plan. A system can be represented as a state-transition system which is a 3-tuple $\Sigma = (S, A, \gamma)$, where S is a finite set of states, A is a finite state of actions, and $\gamma : S \times A \rightarrow 2^S$ - a state transition function. A planning task for system $\Sigma = (S, A, \gamma)$, an initial state s_0, and a subset of goal states S_g is to find a sequence of actions $\langle a_1, a_2, \ldots, a_k \rangle$ corresponding to a sequence of transitions $(s_0, s_1, \ldots, s_k)$ such that $s_1 \in \gamma(s_0, a_1), s_2 \in \gamma(s_1, a_2), \ldots, s_k \in \gamma(s_{k-1}, a_k)$ and $s_k \in S_g$.

To formalize the resolution problem in terms of automated planning we need to explain what are $\Sigma = (S, A, \gamma)$, s_0, and S_g. The system Σ is a business environment, where a business process is executed and which evolves as the next activity completes. Hence, actions in the planning task are associated with instances of activities described by the business context, while system states—with the states of the environment where a process executes. Function γ defines transition rules in the planning domain. In the process domain the context defines the preconditions and effects of activities, which aligns with the transition function. A transition from the current state to the next state via application of an activity results in removing of all the effects defined by $post^-$ relation and adding the effects defined by $post^+$ relation. The current state reflects the effects of all the activities which have taken place. Initial state s_0 corresponds to the state of the environment before a certain activity of the process took place. Set S_g consists of the states in which the business goal of the process is fulfilled and a compliance rule is not violated. The states can be described in terms of first order logic (notice that a compliance rule can be described in first order logic as well). Finally, we argue that the resulting

plan corresponds to one instance of the business process. To retrieve a process model, all possible plans satisfying the initially stated goal should be considered. One possible approach enabling the construction of a model from several plans is process mining [23].

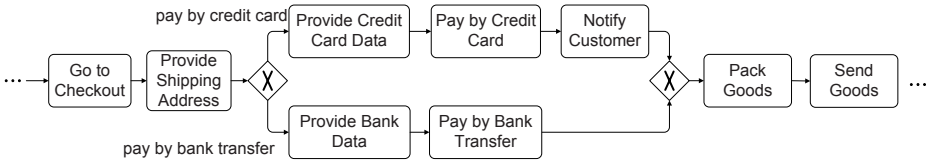
We propose to avoid construction of a business process from scratch. The preferable strategy is to identify a process fragment whose update is enough for achieving compliance. In [3] we have developed an approach enabling violation handling on the structural level. According to this approach, violations are classified into 4 categories and for each category an appropriate resolution technique is applied. In Section 4, we will demonstrate how automated planning techniques are employed in each case.

3 Motivating Example

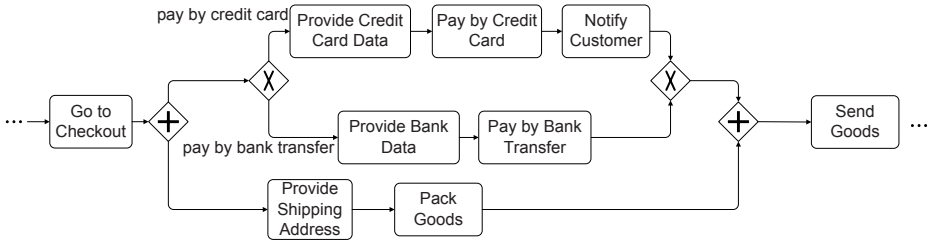
We introduce an example, to be used throughout the paper to illustrate the ideas. The example includes the resolution context and business process fragments. The resolution context is defined by the tuple $(N_{act}, A, T, type, con_{t \in T}, pre_{t \in T}, post_{t \in T})$. The set of activities N_{act} is formed by *Go to checkout*, *Notify customer*, *Pay by credit card*, *Pay by*

Table 1. Pre and post relations of the example resolution context

Activity	Precondition	Postcondition	
		Negative	Positive
Go to checkout	order [init]	order [init]	order [conf] payment method [card]
Go to checkout	order [init]	order [init]	order [conf] payment method [transfer]
Notify customer	order [conf] payment [received]	notification [init]	notification [sent]
Pay by credit card	payment method [card] card data [filled]	payment [init]	payment [received]
Pay by bank transfer	payment method [transfer] bank data [filled]	payment [init]	payment [received]
Provide bank data	payment method [transfer] bank data [unfilled]	bank data [unfilled]	bank data [filled]
Provide credit card data	payment method [card] card data [unfilled]	card data [unfilled]	card data [filled]
Prepare goods	order [conf]	goods [init]	goods [prepared]
Send goods	address [filled] payment [received] goods [prepared]	goods [prepared]	goods [sent]
Provide shipping address	order [conf]		address [filled]
Cancel order	order [init]	order [init]	order [canceled]
Archive order	order [conf]	order [conf]	order [archived]
	goods [sent]	order [conf]	order [archived]
	payment [received]	order [conf]	order [archived]
Archive order	order [canceled]	order [canceled]	order [archived]



(a) First process fragment



(b) Second process fragment

Fig. 1. Two process fragments consistent with the context

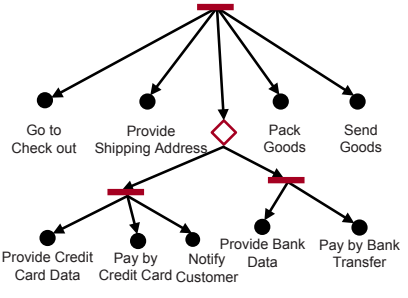


Fig. 2. The PST of the process model in Fig. 1(a)

bank transfer, *Provide bank data*, *Provide credit card data*, *Prepare goods*, *Send goods*, *Provide shipping address*, *Cancel order*, *Archive order*. In the example we consider one aspect—data flow. Hence, set T contains one element *data object*. The set of objects A is the set of data objects *address*, *bank data*, *card data*, *goods*, *order*, *notification*, *payment*, *payment method*. Subsequently, function $type$ relates each of the data objects to type *data object*. Table 1 captures $pre_{t \in T}$ and $post_{t \in T}$ relations. An activity may have more than one pre- or postcondition. For instance, activity *Archive order* expects either a confirmed order, received payment, and sent goods, or it expects the canceled order. In this way it is possible to express disjunctive preconditions. Activities *Pay by credit card* and *Pay by bank transfer* are the only contradicting activities.

Based on the above resolution context, the two process fragments shown in Fig. 1 represent two different ways of composing the activities, where the composition is consistent with the context. The tree representation of the process in Fig. 1(a) is shown in Fig. 3. In that figure, horizontal bars represent *sequence* blocks, diamond represents the choice block, and black circles represent the activities.

4 Resolving Violation

In this section we explain how compliance rule violations can be resolved. We base the resolution strategy on the violation patterns discussed in Section 2.2. The resolution technique to a large extent exploits automated planning.

Obviously, there are more than one possible way to resolve a certain violation [3]. However, not every resolution could be acceptable. For instance, one way of resolving a violation for a rule *A leads to B* would be by simply removing occurrences of A from the process model. Obviously, this is not acceptable as the resulting process might not be consistent with the context, i.e., it has broken dependencies. Thus, for each violation pattern, we discuss resolution strategies that guarantees both consistency and compliance, if possible.

Recalling the discussion about planning in Section 2.4, we use the function $findPlan(init, goal, condition, context)$ in our algorithms to encapsulate the call for an AI planner. The parameter $init$ describes the initial state for the planner. The $goal$ parameter determines which activity(ies) that have to be executed as goals for the planner. Moreover, the $condition$ parameter might be used to express extra constraints on the goal state of the planner. Finally, the $context$ parameter is the encoding of the resolution context. Resolution context is the domain knowledge used by the planner to find a plan.

Before we discuss violation resolution, we explain how the initial and goal states are calculated for the $findPlan$ function. Generally, for a rule *Source leads to Destination*; the $initial$ parameter reflects the execution of a set of activities from starting of the process up to and including the $source$ activity. $source$ is an occurrence of the Source activity in the process, i.e., $source \in N_{Source}$. We use the notion $source^{-}$ to reflect the execution history before $source$.

As discussed in Section 2.4, there might be more than one $initial$ state to check depending on the number of alternative branches. For instance, calling $findPlan($ $Pack\ Goods^{-}, Archive\ Order, true, context)$, regarding the process in Fig. 1(a), would constitute two initial states; reflecting the alternating branches. To succeed with resolving violation, a plan has to be found in each case.

4.1 Splitting Choice Violation

A violation of a compliance rule *A* `leads to` *B* is categorized as *splitting choice* if:

- a process model contains occurrences of activities A and B;
- there is a pair of nodes a and b, which are occurrences of A and B, respectively;
- b belongs to the path from a to a process end;
- there is a path from a to a process end, which does not contain occurrences of B.

Informally speaking, the model allows execution of A giving an option to skip activity B. The cause of splitting choice violation is the split node allocated on the path between occurrences of A and B.

Let us turn to the example process fragment in Fig. 1. The process fragment violates the compliance rule *Go to checkout* `leads to` *Notify customer*. *Go to checkout* activity is succeeded by the choice block. While one branch of the choice block contains activity *Notify Customer*, the other does not.

```
   input  : m—process structure tree
   input  : a, b ∈ N—the occurrences of activities A and B, respectively
   input  : c—resolution context
   output: m—updated process structure tree
 1 s = lca (a,b) ;
 2 if type(s) = loop then
 3     plan = findPlan (a⁻, b, exitCondition (s), c);
 4     if plan = ∅ then
 5         return null;
 6     insert plan into m exactly after the loop exit;
 7 else if type(s) = seq then
 8     x is the choice block containing b;
 9     forall branch is a branch of x with no b do
10         if branch has activities contradicting b then
11             remove branch from m ;
12         else
13             plan = findPlan (a⁻, b, condition (branch), c) ;
14             if plan = ∅ then
15                 remove branch from m;
16             else
17                 add plan to m merging it into branch;
18 return m;
```

Algorithm 1. Resolution of a *splitting choice* violation

A violation resolution implies that a process model is modified in such a way that activity B is always executed after a. We aim at introduction of local modifications to the model. Hence, we first identify the smallest fragment of a model, whose modification can be sufficient for the violation resolution. Afterwards, the fragment is modified in the way that execution of B is assured. Algorithm 1 provides a deeper insight to this approach. Initially, a block enclosing occurrences of activities A and B is sought (see line 1 in Algorithm 1). If the enclosing block is a loop, the algorithm has to assure that B is executed after the loop. Automated planning attempts to construct a suitable plan containing activity B. If the plan is constructed, it is inserted exactly after the loop block. Otherwise, the resolution cannot be performed. If the block is not a loop, we seek inside it for a choice block containing b on its branch (line 7). For each block branch missing B, we check if it has activities contradicting B, with respect to the resolution context C. If a contradiction exists, the branch is removed (line 11). In case of no contradictions, we use automated planning based on the information of the resolution context to find a path from a to b under the branch condition (see line 13). If no plan could be found, the branch is removed from the model. Otherwise, the found plan is merged to the branch to enforce compliance.

In the context of the example in Fig. 1(a) an occurrence of activity *Notify customer* is added to the branch where it was missing. The occurrence is added to the branch after activity *Pay by Bank Transfer*.

We argue that the proposed algorithm is correct, i.e., it resolves the violation of type splitting choice and delivers a consistent model free of contradictions.

Theorem 1. *If a process model contains a violation of type* splitting choice*, Algorithm 1 resolves this violation and delivers a model which is consistent with the context.*

Proof. To prove the theorem we have to show that:

- in the resulting process model the violation is resolved;
- the model does not have inconsistencies or contradictions.

Algorithm 1 localizes changes in the fragment s which is the least common parent of a and b. We focus the analysis on fragment s as well.

If fragment s is of type $loop$, a plan containing an occurrence of B is created and inserted exactly after the loop. This assures the violation resolution. The resulting plan is consistent with the process model, as this is one of the requirements to the planner.

If fragment s is a sequence, we identify a choice block within s, let it be x. Block x contains an occurrence of B on at least one of the branches. New occurrences of B are added to the branches where B is missing. If a branch has an activity contradicting B, the branch is removed. If a there is no contradicting activities, a plan containing B is designed by the planner and added to the branch. At this stage every branch in the choice contains B and, there is no way to skip B execution in the block. Thus, we have shown the first statement of the theorem.

Every branch which initially contained B is consistent and free of contradictions. Let us look at those branches where we add occurrences of B and their prerequisites. If such a branch has a contradiction or introduces an inconsistency, it is deleted. Removal of the branches does not lead to inconsistencies. Activities preceding a branch to be removed do not depend on activities of this branch. Activities succeeding the removed branch expect the effects of execution of at least one branch in the choice. While at least one branch of the original process is preserved, no inconsistencies can be caused by the branch removal. Thus, the resulting model is free of inconsistencies or contradictions. Thereby, we proved the second statement and completed the proof. □

4.2 Different Branches

A violation is of type *different branches* if a process model contains a and b (occurrences of activities A and B, respectively) and there is no path leading from a to b. A violation of this type takes place if the two activities are allocated on different branches of a block. One can notice that the violation occurs independent of a block type, i.e., in an OR, XOR, or AND block. However, the resolution strategy varies depending on the block type. Before we turn to a discussion of resolution strategies, let us illustrate different branches violation by an example. *Different branches* violation is shown in Fig. 1(b), where *Pay by credit card* and *Pack goods* activities are executed in parallel. Once the company policy requires to pack the goods after receiving the payment, the business process becomes non-compliant.

In case of an AND block, the resolution strategy aims at sequentializing A and B. To achieve the sequential execution of A and B, we move an occurrence of B from a block branch to the position exactly after the block. However, such a manipulation with an occurrence of B might introduce inconsistencies into the process model: there might be activities on the branch expecting B in the initial place. Hence, we move not only an occurrence of B, but the set of activities succeeding B on the branch and depending on B. An alternative strategy is to allocate a and move it, together with preceding activities on which a depends, exactly before the block. The preference to one of these strategies can be given basing on the number of activities to be moved.

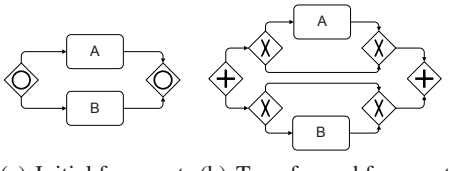
input : m—process structure tree
input : a, b $\in N$—the occurrences of activities A and B, respectively
input : c—resolution context
output: m—updated process structure tree

```
1  s = lca (a,b) ;
2  switch type(s) do
3      case AND
4          PRE_a is the set of all nodes that execute before a within the same thread;
5          POST_b is the set of all nodes that execute after b within the same thread;
6          if |PRE_a| <|POST_b| then
7              move PRE_a before the parallel block;
8          else
9              move POST_b after the parallel block;
10     case XOR
11         forall branch is a branch of x with a, but no b do
12             resolve Lack of activity violation;
13     case OR
14         restructure the model;
15         call to other resolution strategies;
16 return m;
```

Algorithm 2. Resolution of a *different branches* violation

The resolution of a different branches violation in a XOR block is reducible to the resolution of lack of activity violation. Indeed, the branch containing a misses an occurrence of B. Thus, the technique for lack of activity violation resolution is applicable for this branch (see Section 4.4).

Resolution of different branches violation in an OR block profits from the other resolution strategies. First, we replace an OR block with a combination of AND and XOR blocks, exhibiting the same behavior. The restructuring of the process relies on the approach introduced in [5]. Fig. 3 illustrates the solution for a trivial case of an OR block with two alternative branches. Once the OR block is replaced, other resolution strategies can be invoked.



(a) Initial fragment (b) Transformed fragment

Fig. 3. Example of overwriting mechanism

Returning to the running example in Fig. 1(b), the resolution algorithm moves activity *Pack Goods* from the lower branch of the parallel block to the position between the AND join and *Send Goods* activity.

Theorem 2. *If a process model contains a violation of type* different branches*, Algorithm 2 resolves this violation and delivers a model which is consistent with the context.*

Proof. Similar to the case described in Theorem 1, it is required to show that in the resulting process model a violation is resolved and the resulting model does not have inconsistencies or contradictions.

Let us consider the AND block, XOR block, and OR block one by one. In case of an AND block, an occurrence of B is moved to the position directly after the block. After this modification B is always executed after A, as B directly succeeds the parallel block containing A. The compliance requirement holds. The initial model is free of contradictions and inconsistencies. Moving the occurrence of B together with dependent activities does not introduce contradictions, not inconsistencies. This is true, since the performed sequentialization only restricts the initial model.

An OR block is reduced to a combination of XOR and AND blocks. A transition to XOR and AND blocks and the resolution strategies for these two blocks define the properties of OR block resolution. The transformation from an OR block to the combination of XOR and AND blocks does not introduce inconsistencies and contradictions, since the new construct exhibits the same behavior. We have already argued about the properties of an AND resolution. For the XOR case we employ the resolution of *lack of activity* violation. In Section 4.4 we will argue that lack of activity violation meets the stated requirements. Thus, for the OR and XOR blocks the desired properties hold. □

4.3 Inverse Order

A process model violates a compliance rule *A* `leads to` *B* if it contains a and b (occurrences of activities A and B, respectively) connected with a path, but this path leads from b to a. The inverse order violation can be illustrated by the process fragment in Fig. 1(a), where the company sends a notification with an order summary to a customer. Afterwards, the company contacts its logistics partner to pack and send goods. New business conditions might require the company to include the delivery information in the notification, i.e., first the goods should be packed. This requirement is captured in the rule *Pack goods* `leads to` *Notify customer* rule. This is an *inverse order* violation.

To resolve an *inverse order* violation, we propose to analyze the fragment from activity B to A. The main idea is to reorder A and B and achieve model compliance. Reordering of the activities A and B is feasible, if there are no dependencies of activity A on B. If a dependency exists, reordering introduces inconsistencies into a model. To answer if reordering is feasible, we attempt to construct a model fragment from the process start to the point when A is executed. To come up with this fragment, we construct a plan. In contrast to the initial model, activity B should not appear in this plan. The plan construction is approached with automatic planning techniques. The initial state of the planning task reflects the process state directly before activity B is executed. The goal state describes the process after activity A is executed. If the planner comes up with the plan, the reordering is feasible.

Once reordering turns out to be feasible, the designed plan must be complemented to assure execution of B. Again the planning task is carried out by the planner. The initial state of this new plan corresponds to the goal state of the previous step. The new goal state describes the process state directly after an execution of B in the initial process. The designed plan has to be inserted into the model. If after insertion of the plan, the model is free of contradictions and inconsistencies the resolution is completed. In the opposite case we say that the automatic resolution is not feasible.

In the example of *Pack goods* `leads to` *Notify customer* rule violation the resolution strategy moves the occurrence of *Notify Customer* activity to the position after *Pack Goods* activity.

Theorem 3. *If a process model contains a violation of type* inverse order*, the proposed resolution strategy resolves this violation and delivers a model which is consistent with the context.*

Proof. We have to show that in the resulting process model a violation is resolved and the resulting model does not have inconsistencies or contradictions.

If the planner succeeds with plan construction, the resulting plan contains B. As the plan is inserted into the process model exactly after A, *A* `leads to` *B* holds. The modifications to the model are limited to adding new fragments constructed by the planner. As the planner uses the context, the designed plans are free of contradictions and inconsistencies. Adding the plan into the model, we check if it has any contradictions with the rest of the model. If it is the case, the plan is not accepted. Hence, the resulting model (if produced) is free of contradictions and inconsistencies. □

4.4 Lack of Activity

A process model contains a violation of type *A* `leads to` *B* if it contains at least one occurrence of A and no occurrence of B. Consider a compliance rule *Go to checkout* `leads to` *Archive order*. Checking the process model fragment in Fig. 1(a) against this rule, we see that this rule is violated, since *Archive order* is missing in that fragment.

To resolve a violation of this type we introduce an occurrence of B into the process model exactly after an occurrence of A. If the process model does not contain activities contradicting to B, we construct a plan using *findPlan(A, B, true, context)*. The plan is merged into the process model directly after an occurrence of A. In case there is an activity contradicting to B, let it be C, the resolution requires extra actions. The actions depend on the relations between occurrences of A and C:

- occurrences of A and C are allocated on different branches of a choice block;
- occurrences of A and C are allocated on different branches of a parallel block;
- an occurrence of C is allocated before A;
- an occurrence of C is allocated after A.

In the first case, B can be added to the process model exactly after A. As the branch with an occurrence of A does not contain activities contradicting B, B can be introduced to this branch without any conflicts. In the latter three cases a process model contains occurrences of activities contradicting B. We propose to introduce an occurrence of B into the model in such a way that B and C appear on alternative branches. We first seek for a SESE fragment containing an occurrence of C and activities tightly coupled with C. Such a process fragment contains activities which are transitively dependent on C or on which only C transitively depends. The fragment is preceded by an activity, let it be pre, and succeeded by an activity—$post$. We aim at complementing the process model with a branch, alternative to the identified fragment with C and containing B. We can obtain such a sequence as a result of a planning task, requiring it to fit between pre and $post$ and containing B. Finally, we add the choice block with the two branches into the model: the branch with C and the branch with plan containing B.

Identification of the activities dependent on C is based on the analysis of the resolution context and can be found as the closure of all the activities transitively depending on C. Similarly, the activities on which only C transitively depends can be found. Once

input : m—process structure tree
input : a $\in N$—is the occurrences of A for which the violation has to be resolved
input : b $\in N$—a new occurrence of B to be added to the tree
input : c—resolution context
output: m—updated process structure tree

1 CON_b is the set of activities contradicting b based on c;
2 **if** $CON_b \neq \emptyset$ **then**
3 **forall** x $\in CON_b$ **do**
4 **if** *type(lca(*x,a*))* = *choice* **then**
5 **if** `findPlan` *(*a $^-$, b, `true`, c*)* $\neq \emptyset$ **then**
6 insert `findPlan` (a $^-$, b, `true`, c) after a in m;
7 **else**
8 **return** `null`;
9 **else**
10 Find pre, $post$;
11 **if** pre = `null` $\vee$ $post$ = `null` **then**
12 **return** `null`;
13 $FRAG_x$ contains activity x and its tightly coupled activities;
14 $FRAG_b$ contains activity b and the results of `findPlan` (pre $^-$, b, `true`, c) and `findPlan` (b $^-$, $post$, `true`, c);
15 insert $FRAG_x$ and $FRAG_b$ in a choice block between pre and $post$;
16 **else**
17 **if** `findPlan` *(*a $^-$, b, `true`, c*)* $\neq \emptyset$ **then**
18 insert `findPlan` (a $^-$, b, `true`, c) after a in m;
19 **else**
20 **return** `null`;
21 **return** m;

Algorithm 3. Resolution of a *lack of activity* violation

the model fragment is identified, it is possible to learn its first preceding activity—pre and the first succeeding—$post$.

As the result of the described model transformation, the violation type is no longer *lack of activity*. Instead, it changes either to *inverse order*, *splitting choice*, or *different branches* violation. Notice that a *lack of activity* violation can be reduced to *different branches* violation and vice versa. However, there is no mutual dependency between them, as we reduce these violations to not intersecting subcases of violations. Algorithm 3 summarizes the approach.

Returning to the example with a compliance rule *Go to checkout* `leads to` *Archive Order* and the process fragment in Fig. 1(a), according to the resolution strategy and the resolution context, activity *Archive Order* is added after goods are sent.

The resolution strategy for *lack of activity* violation reduces the violation to the previous three cases: *inverse order*, *splitting choice*, and *different branches*. Hence, its properties, i.e., model compliance, freedom of contradictions and inconsistencies originate from the properties of resolution methods for the named violation types. However, we have already shown that the resolutions for the three violations satisfy the stated requirement. Fig. 4 illustrates how a lack of activity violation is reduced to other violation types and is resolved.

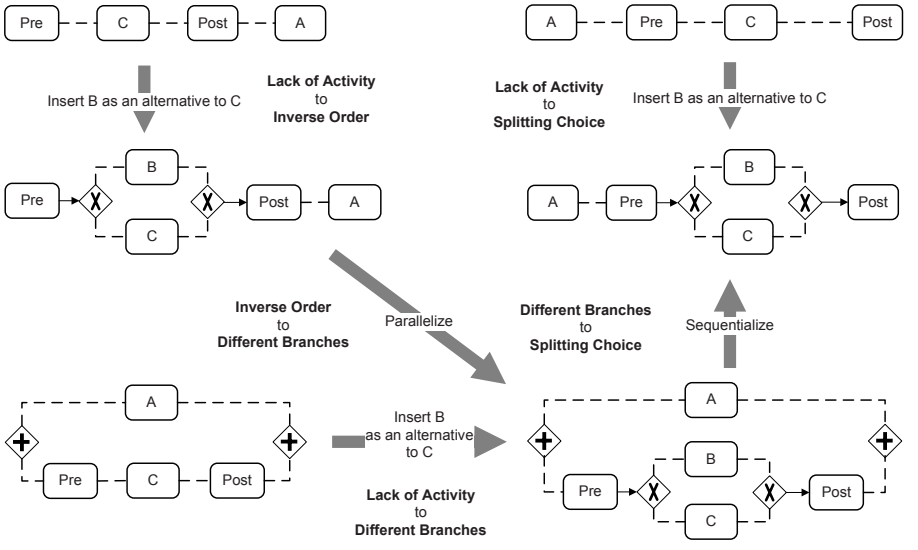


Fig. 4. Resolution of *lack of activity* violation in case of contradictions

4.5 The Overall Resolution Process

According to the discussion above, it might be the case that to resolve some violations is to first transform it from a violation type to another. For instance, in some cases of lack of activity violation, we transform it first to a splitting choice violation. Later on, we apply the algorithm of splitting choice to completely resolve the violation. Thus, the compliance violation resolution is of iterative nature. In each iteration, the violation type is recognized and the appropriate violation resolution algorithm is applied. This process is repeated until either no further violations are recognized or at one step the resolution algorithm either fails to resolve the violation or to transform it to another type. Algorithm 4 summarizes our approach to resolve compliance violations.

Utilizing the priori of pairs of activity occurrences, the identification of violation type is achieved in a polynomial time. The presented approach is prototypically implemented in our compliance management tool chain.

5 Related Work

An essential problem of compliance is identification of violations. Today a large body of research on compliance checking of business process models is published. Our primary interest is the work on execution order compliance checking. The research in this area can be divided into two directions: compliance by design and compliance checking of existing models. Compliance by design is to enforce process model compliance already at the stage of design. [8,9,13,19,20] show how this approach can be realized. A limitation of such approach is the need to recheck a process model once a rule is changed or newly introduced. Thus, it lacks the ability to resolve violations. The other branch of research employs model checking techniques to verify that existing process models satisfy the compliance rules. [2,6,15,27] consider this problem. Following this

```
   input  : m—process structure tree
   input  : A Leads to B
   input  : c—the resolution context
   output: m—violation free process structure tree
 1 violation=getViolationType (m,A,B);
 2 while violation ≠ none do
 3     switch violation do
 4         case Splitting Choice
 5             m = resolveSplittingChoice(m, a, b, c);
 6         case Different Branches
 7             m =resolveDifferentBranches (m,a,b,c);
 8         case Inverse Order
 9             m =resolveInverseOrder (m,a,b,c);
10         case Lack of Activity
11             m =resolveLackActivity (m,a,b,c);
12     if m =null then
13         return null;
14     violation=getViolationType (m,A,B);
15 return m;
```

Algorithm 4. Compliance violation resolution

approach the authors of [10,11] use business contracts as the source of compliance rules. To formalize compliance rules Formal Contract Language (FCL) is used. [22] separates the process modeling from control objectives; it also uses FCL to express control requirements over annotated process models. These approaches are capable of $verifying$ processes against compliance requirements. Again, the capability of resolving violations by means of changing the structure of the process through adding removing sets of activities is not addressed.

Resolution of violations can be driven by the severity of violations. In this sense an interesting method was introduced in [14] to measure the $degree$ of compliance. Once a compliance rule is specified, the approach tells the degree of compliance for a given process model on the scale from 0 to 1.

Recently, several requirements frameworks for business process compliance management have been proposed. In [16] the authors formulate requirements for compliance tools. The requirements address the issues of lifetime compliance. Among other requirements, the authors discuss compliance enforceability. In this context they perceive the violation resolution as a human task, i.e., only human experts are responsible for taking remedy actions when a violation is discovered.

It should be noticed that [7] discussed possible strategies for resolving compliance violations. However, the discussion was very high level.

In this paper, we demonstrated that automated resolution of violations is achievable. This is to be considered as a step forward in supported automated compliance checking.

Automated planning techniques [21] have been used for service composition, e.g., in [18,17] for realizing service oriented architecture SOA. Unlike our approach, compositions are made from scratch. I.e., the user needs behind the service composition define the goal state to be reached where they start from a $clean$ initial state. In our approach, we used the notion of cumulative effect to use planning to compose parts of the process that is related to the compliance rule. Thus, we developed more sophisticated techniques to construct the initial state for the planning problem.

6 Conclusions and Future Work

In this paper we have addressed the problem of compliance rule violation resolution. We focused on a special class of violations—execution order compliance rule violations. In order to cope with these violations we introduced a violation catalog, describing 4 violation types and their resolution methods. Once a violation is categorized according to the catalog, a proper resolution strategy is employed. The resolution strategies are based on automated planning techniques and an extensive description of a business domain—resolution context.

A violation resolution implies that a process model structure is changed. To control model structural modifications we employ the concept of SESE fragments. As a consequence, the approach is limited to process models, which are block-structured. To neglect this limitation advanced decomposition techniques, as described in [24], can be employed. This is the first direction of future work.

The proposed resolution strategy assumes that we know the nodes resulting in a violation. If each activity has a unique occurrence in the process model, identification of nodes causing a violation is straightforward. When activities have multiple occurrences, a mechanism for identification of occurrence pairs leading to a violation is required. A naive approach is to consider all combinations of node pairs. However, this problem is not trivial and is considered by us as another direction of future work.

The presented algorithms were dedicated to resolve violation to *leads to* rules. Adaptation of these algorithms to resolve violation to other rules is a future work.

Although the changes introduced by the resolution algorithms result in consistent and compliant processes models, an updated model might look unnatural for a human reader. For instance, sequentialization of two branches in case of different branches violation, leaves only one branch in the parallel block. Thus, a resulting process model needs refactoring to develop a naturally looking process model. Process refactoring techniques proposed in [26] can facilitate the problem solution.

References

1. Sarbanes-Oxley Act of 2002. Public Law 107-204 (116 Statute 745), United States Senate and House of Representatives in Congress (2002)
2. Awad, A., Decker, G., Weske, M.: Efficient Compliance Checking Using BPMN-Q and Temporal Logic. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 326–341. Springer, Heidelberg (2008)
3. Awad, A., Smirnov, S., Weske, M.: Towards Resolving Compliance Violations in Business Process Models. In: GRCIS, vol. 459. CEUR-WS.org (2009)
4. Awad, A., Weske, M.: Visualization of compliance violation in business process models. In: 5th Workshop on Business Process Intelligence BPI 2009. Springer, Heidelberg (to appear, 2009)
5. Decker, G., Kopp, O., Leymann, F., Pfitzner, K., Weske, M.: Modeling Service Choreographies Using BPMN and BPEL4Chor. In: Bellahsène, Z., Léonard, M. (eds.) CAiSE 2008. LNCS, vol. 5074, pp. 79–93. Springer, Heidelberg (2008)
6. Förster, A., Engels, G., Schattkowsky, T., Van Der Straeten, R.: Verification of Business Process Quality Constraints Based on VisualProcess Patterns. In: TASE, pp. 197–208. IEEE Computer Society, Los Alamitos (2007)
7. Ghose, A., Koliadis, G.: Auditing Business Process Compliance. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 169–180. Springer, Heidelberg (2007)

8. Goedertier, S., Vanthienen, J.: Compliant and Flexible Business Processes with Business Rules. In: BPMDS. CEUR Workshop Proceedings, vol. 236. CEUR-WS.org (2006)
9. Goedertier, S., Vanthienen, J.: Designing Compliant Business Processes from Obligations and Permissions. In: Eder, J., Dustdar, S. (eds.)BPD 2006. LNCS, vol. 4103, pp. 5–14. Springer, Heidelberg (2006)
10. Governatori, G., Milosevic, Z.: Dealing with Contract Violations: Formalism and Domain Specific Language. In: EDOC, pp. 46–57. IEEE Computer Society Press, Los Alamitos (2005)
11. Governatori, G., Milosevic, Z., Sadiq, S.: Compliance Checking between Business Processes and Business Contracts. In: EDOC, pp. 221–232. IEEE Computer Society Press, Los Alamitos (2006)
12. Hartman, T.: The Cost of Being Public in the Era of Sarbanes-Oxley. Foley&Lardner, Chicago, Ill (2006)
13. Lu, R., Sadiq, S., Governatori, G.: Compliance Aware Business Process Design. In: ter Hofstede, A.H.M., Benatallah, B., Paik, H.-Y. (eds.) BPM Workshops 2007. LNCS, vol. 4928, pp. 120–131. Springer, Heidelberg (2008)
14. Lu, R., Sadiq, S., Governatori, G.: Measurement of Compliance Distance in Business Processes. Inf. Sys. Manag. 25(4), 344–355 (2008)
15. Lui, Y., Müller, S., Xu, K.: A Static Compliance-checking Framework for Business Process Models. IBM Systems Journal 46(2), 335–362 (2007)
16. Ly, L.T., Göser, K., Rinderle-Ma, S., Dadam, P.: Compliance of Semantic Constraints – A Requirements Analysis for Process Management Systems. In: GRCIS, vol. 339. CEUR-WS.org (2008)
17. Marconi, A., Pistore, M., Traverso, P.: Automated Composition of Web Services: the ASTRO Approach. IEEE Data Eng. Bull. 31(3), 23–26 (2008)
18. Meyer, H., Weske, M.: Automated Service Composition Using Heuristic Search. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 81–96. Springer, Heidelberg (2006)
19. Milosevic, Z., Sadiq, S., Orlowska, M.: Translating Business Contract into Compliant Business Processes. In: EDOC, pp. 211–220. IEEE Computer Society Press, Los Alamitos (2006)
20. Namiri, K., Stojanovic, N.: Pattern-Based Design and Validation of Business Process Compliance. In: Meersman, R., Tari, Z. (eds.) OTM 2007, Part I. LNCS, vol. 4803, pp. 59–76. Springer, Heidelberg (2007)
21. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory & Practice. Morgan Kaufmann Publishers Inc., San Francisco (2004)
22. Sadiq, S., Governatori, G., Namiri, K.: Modeling Control Objectives for Business Process Compliance. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 149–164. Springer, Heidelberg (2007)
23. van der Aalst, W., Weijters, T., Maruster, L.: Workflow Mining: Discovering Process Models from Event Logs. IEEE Trans. on Knowl. and Data Eng. 16(9), 1128–1142 (2004)
24. Vanhatalo, J., Völzer, H., Koehler, J.: The Refined Process Structure Tree. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, Springer, Heidelberg (2008)
25. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 43–55. Springer, Heidelberg (2007)
26. Weber, B., Reichert, M.: Refactoring process models in large process repositories. In: Bellahsène, Z., Léonard, M. (eds.) CAiSE 2008. LNCS, vol. 5074, pp. 124–139. Springer, Heidelberg (2008)
27. Yu, J., Manh, T.P., Han, J., Jin, Y., Han, Y., Wang, J.: Pattern Based Property Specification and Verification for Service Composition. In: Aberer, K., Peng, Z., Rundensteiner, E.A., Zhang, Y., Li, X. (eds.) WISE 2006. LNCS, vol. 4255, pp. 156–168. Springer, Heidelberg (2006)