Efficient Full Compliance Checking of Concurrent Components for Business Process Models

Silvano Colombo Tosatto
Data61, CSIRO
silvano.colombotosatto@data61.csiro.au

Guido Governatori
Data61, CSIRO
guido.governatori@data61.csiro.au

Nick van Beest
Data61, CSIRO
nick.vanbeest@data61.csiro.au

Francesco Olivieri
Data61, CSIRO
francesco.olivieri@data61.csiro.au

Abstract

Business process compliance checking is an NP-complete problem, due to concurrency and different mutually exclusive execution paths. Although the complexity of real life process models usually allows for a brute force approach, environments with limited resources or computational power (like for instance blockchain environments) cannot rely on brute force approaches due to the computational complexity of the problem. In this paper, we present an approach to efficiently check a subclass of problems involving concurrent sub-processes. Our approach reduces the computational complexity of concurrent sub-processes from combinatorial to exponential. We prove the correctness of the approach, we experimentally validate the results and evaluate the scalability of the approach. We show that our approach is a significant improvement for highly concurrent processes and easily outperforms existing brute force approaches.

1 Introduction

One of the aspects related to legal reasoning concerns verifying whether a given behaviour complies with a set of given regulations. These so-called *Compliance Checking* procedures can be applied to sets of behaviours, like for instance a model describing the possible behaviours of an agent. The advantage of verifying models describing possible behaviours is that it ensures that all behaviour allowed by the model is proven to be compliant with a given set of regulations. This approach is also referred to as *Compliance by Design* [17], as it ensures that a model contains, by design, only or at least some compliant behaviours. Compliance by design is also known as *forward compliance*, referring to the techniques focused on preventing compliance breaches, which differ from *backward compliance*, as these techniques focus on identifying compliance breaches after they have happened.

Business Process Models are originally designed to formally represent the possible sequences of activities to be executed by an organisation to achieve a certain business goal. In addition to their utility in a legal setting, where they can be used to automatically verify compliance breaches, these models can potentially be used to represent collections of agent's plans and be automatically verified with respect to some given constraints (as as shown by Governatori and Rotolo [11]).

However, proving compliance by design of business process models is in general **NP**-complete, as shown by Colombo Tosatto et al. [6]. Accordingly, no polynomial solutions are possible for the general problem of proving compliance of business process models. Most of the current solutions for the problem, like for instance Regorous [12], adopt a brute force approach over the possible executions of a business process model to prove its compliance. Despite the high theoretical complexity of the problem, solutions like the aforementioned Regorous, still seem to offer practical solutions to some real life instances. Nevertheless, in a number of cases the size of the problem grows enough to not be solvable by brute force approaches, or in others, the environment may be providing only a limited amount of computational resources, such as for instance a blockchain. As a result, a more efficient use of these resources becomes desirable. Part of the complexity of the problem lies in the process model structure, where even compact structures can potentially represent an exponential number of possible executions. Within a business process model, certain sequences of activities can be mutually exclusive, while other activities are concurrent. Concurrency allows for a combinatorial number of possible execution orders of the activities involved, as it considers all possible interleaving of the activities unrestricted by explicit ordering constraints.

To address this issue, we propose an efficient algorithm to prove *full compliance* of business process models with respect to a set of given regulations, by verifying whether a counter-example exists. As such, we provide an approach to resolve compliance over concurrent paths, such that the computational complexity of verifying compliance is reduced from combinatorial to exponential. To keep the theoretical complexity of the solution tractable, we restrict the expressivity of the regulations being checked to literals and with no compensations for eventual violations. Furthermore, we restrict our approach to *structured* process models [26], as they allow to verify their soundness and correctness in polynomial time with respect to the size of the model.

The remainder of the paper is structured as follows: Section 2 discusses the related work. Section 3 introduces the classic regulatory compliance problem, and provides a high-level overview of the proposed solution. Section 4 describes the process models and the decomposition procedure. Section 5 describes the regulations and how Δ-constraints[1] are generated. Section 6 defines full compliance and how it can be proven through decomposed processes and Δ-constraints. Section 7 empirically evaluates the approach against a set of highly complex models. Finally, Section 8 concludes the paper.

2 Related Work

While the area of business process compliance received substantial research interest the past decade (Hashmi et al. [17] identify over 180 research papers between 2000 and 2015 specific to business process regulatory compliance), the study of the computational complexity properties (and solutions to reduce the search space) and whether the proposed techniques offer practical solutions has been largely neglected.

For example, Pulvermüller et al. [27] directly verify temporal logic based specifications on a process, without proper support for different branching options through gateways, and Awad et al. [2] utilise a reduction technique, which results in an incomplete model. Ramezani et al. [28] provide a set of generic compliance patterns, but do not offer the same expressivity as other approaches using defeasible logic or even temporal logic.

Other approaches introduce a large amount of overhead in the state space of the model encoding the process, which is often the result of ignoring the effect of encoding on the internal state machine of the model checker (see e.g. Latvala and Heljanko [23], Bianculli et al. [4], Kherbouche et al. [20], and Kheldoun et al. [19]). In particular, parallel branching constructs may cause a state space explosion, which only few approaches have successfully addressed. Some approaches, like the one proposed by Feja et al. [10] simply disregard parallel information entirely, such that full compliance cannot be ensured. Other approaches, such as Liu et al. [24], do interleave parallel branches correctly, but interleave to such an extent that concurrent executions are linearised entirely, resulting in a state explosion.

Another direction of research has focused on *conformance checking*, such as for instance van der Aalst et al. [30], where they analyse techniques to verify whether executions

[1]We discuss the reasoning behind the name used for the alternative representation of the regulation in Section 3.2.

actually belong to the allowed behaviour specified in the business process model. Conformance checking is an orthogonal discipline when compared to *compliance by design*, as the latter focuses on verifying properties of future executions of business process models, while the former focuses on verifying properties of existing executions compared to their corresponding process models. As such, an execution may be conformant (i.e. it matches an allowed execution specified in the model), but not compliant (i.e. it violates a regulation). Similarly, an execution can be non-conformant (i.e. its behaviour is not specified in the model), but it is compliant (i.e. it does not violate any regulation).

Although solutions based on temporal logic benefit from the optimisation and efficiency of modern model checkers (e.g. Awad et al. [3], Elgammal et al. [9], and Pesic et al. [25]), such approaches do not address the reduction of the search space. Moreover Governatori and Hashmi [13] show that given certain circumstances, like requiring compensatory obligations[2] and permissions determining whether an obligation is in force, do not allow temporal logic to soundly model regulatory requirements.

Some solutions have tried to address the complexity of the problem by reducing the required search space. Groefsema et al. [16], for example, propose a solution using Kripke structures to reduce the state explosion derived from concurrency components. However, this approach first creates a full state space, followed by a reduction. In contrast, the approach presented in this paper does not require a subsequent reduction to be feasible for model checking, as it directly allows for efficient compliance checking of concurrent process constructs.

3 Regulatory Compliance Problem and Our Approach

In this section, we first introduce the classic regulatory compliance problem. Subsequently, we introduce the idea behind the proposed approach in this paper. The formal details of both the regulatory compliance problem and the approach are discussed later in the paper.

3.1 The Regulatory Compliance Problem

The regulatory compliance problem is concerned with verifying whether a business process model is compliant with a given set of regulations. The problem contains two separate components: the business process model, describing a set of possible executions capable of achieving the business goal pursued by the model, and the set of regulations defining the constraints that are required to be fulfilled by each execution in the model.

[2]A type of obligation that become in force in response to another obligation being violated, and its fulfilment compensates the triggering violation.

Business Process Models

Business process models are formal representations capable of compactly representing multiple executions achieving a particular goal. In order to achieve their compactness, business process models use *coordinators* ([29]) to specify that some of the executions in the model have some parts in common, which the model needs to represent only once. These coordinators are capable of representing e.g. mutually exclusive or concurrent paths of executions. As such, they are based on the semantics of *exclusive* and *parallel* gateways as defined in *BPMN 2.0*, with the restriction of structuredness as described in Definition 2.

The coordinators allow to compactly represent multiple distinct executions within a single business process model. More precisely, a business process model can potentially contain a combinatorial number of executions with respect to the elements composing it. As a result, brute force approaches are potentially impractical as they require the generation and analysis of each possible execution contained within a model.

Regulations

The second component of a regulatory compliance problem concerns the regulations defining the regulatory requirements. These regulations determine which obligations are required to be fulfilled in each possible execution in the business process model.

Given an execution of a business process model and a regulation, the regulation defines a triggering condition, which (if satisfied by the execution) sets an obligation in force that is then required to be fulfilled. Additionally, the regulation also determines a deadline condition, expressing that when the obligation is in force, it must be fulfilled before or until the execution satisfies the deadline. This is illustrated graphically in Figure 1. We can consider an execution to be a sequence of activities each occurring at a distinct point in time. The interval between the point in time satisfying the triggering condition of a regulation and the point in time satisfying the corresponding deadline determines the in force interval of an obligation.

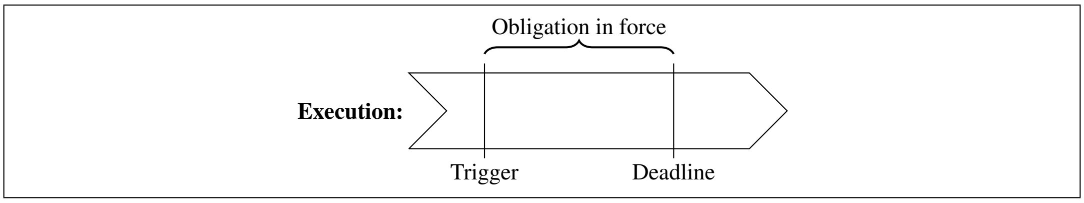


Figure 1: Obligation in force

We consider two types of obligations, *achievement* and *maintenance*, each of which having different properties [6]:

Achievement When this type of obligation is in force, the fulfilment condition specified by the regulation must be satisfied by the execution in at least one point in time before the deadline is satisfied. When this is the case, the obligation in force is considered to be satisfied, otherwise it is violated.

Maintenance When this type of obligation is in force, the fulfilment condition must be satisfied *continuously* until the deadline is satisfied. Again, if this is the case, the obligation in force is then satisfied, otherwise it is violated.

Types of Compliance

Given an execution and a regulation, if each in force interval determined by the obligation is satisfied by the regulation, then we can say that this particular execution complies with the regulation. Similarly, when considering a set of regulations instead of a single one, an execution is considered to comply with the set, if and only if it satisfied every in force interval of the obligations determined by the set of regulations.

However, when verifying whether a business process model is compliant with respect to a set of regulations, different types of compliance can be considered. A business process model can be either, *fully compliant*, *partially compliant*, or *not compliant* with respect to a set of regulations. This distinction between the different levels of business process compliance have been already introduced by Governatori and Rotolo [15].

Fully Compliant A business process model is considered fully compliant, if and only if each of its possible executions complies with the set of regulations.

Partially Compliant A business process model is considered partially compliant, if and only if there exists an execution of the business process model that complies with the set of regulations.

Not Compliant A business process model is considered not compliant, if and only if none of its possible execution complies with the set of regulations.

3.2 Approach Overview

We now provide an overview of the solution proposed in this paper. First, the solution focuses on proving whether a business process model is fully compliant with respect to a given set of regulations.

As we formally define in Section 4, mutual exclusion coordinators in a business process model are also referred to as *XOR Blocks*. Our proposed approach verifies whether

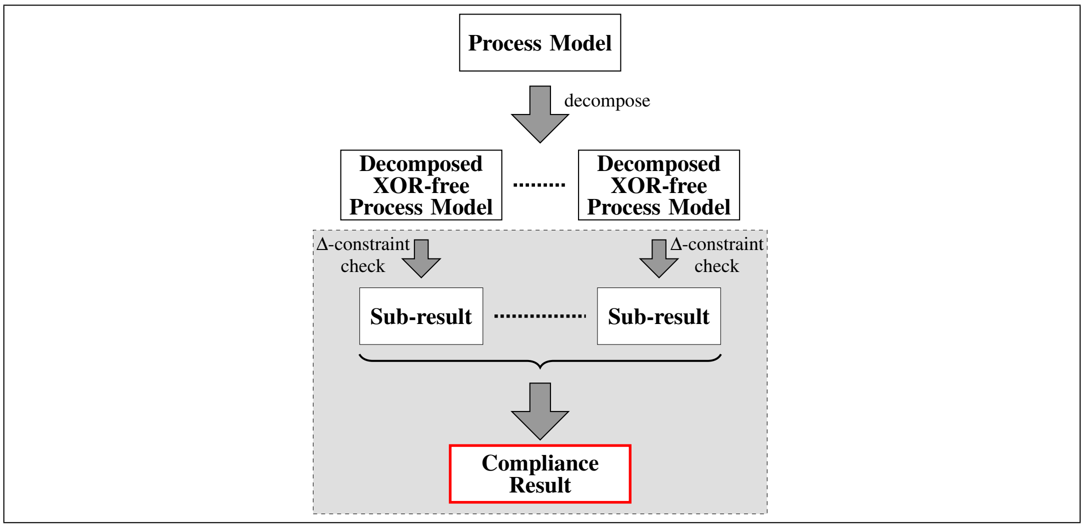


Figure 2: Approach Overview.

a business process model is compliant with a set of regulations as follows: first, it decomposes the full business process model into a set of *XOR-free Process Models*[3], which maintain the expressivity of the original process. That is, the set of executions available in the original process is the same as the sum of the executions of the set of XOR-free Process Models. Second, we represent the set of regulations as Δ-*constraints*, allowing us to verify whether an XOR-free Process Model violates a given regulation. This allows us to prove full compliance for the verified process in case a violation is not found. We adopt the name Δ-constraint to refer to the alternative representation of the regulations, as the new representation focuses on the differences between subsequent process states introduced by the execution of tasks (as described in Definition 8, and updated in accordance to the annotation of a task being executed as described in Definition 9), which can be understood as the Δ between such process states. Finally, the compliance results relative to the XOR-free Process Models are aggregated, in order to decide whether the original business process model is fully compliant with the given regulations. The procedure is graphically illustrated in Figure 2.

4 Business Process Models and Decomposition

In this section, we first introduce the formal syntax and semantics of business process models. Before introducing the decomposition procedure, we briefly discuss how the structural

[3]A set of process models not containing mutual exclusion coordinators.

components of a business process model contribute to the computational complexity of the problem. Finally, we define the decomposition procedure transforming a generic process model in a set of process models free of mutually exclusive choices, such that they can be handled by the solution proposed in this paper.

There exist multiple mainstream business process modelling notations, which include (among others) BPMN, EPC, and Petri nets. The mapping between these different formalisms has been extensively studied and described [31, 8, 22]. Although each of these graph-oriented formalisms allows to model structured business processes, they require a formal definition of the restriction to structured processes on top of the definition of the respective modelling notation itself. Therefore, we adopted a formal description that captures the structured nature of the business process by design, in order to provide a shorter and more intuitive notation throughout the remainder of this paper.

4.1 Business Process Models

We focus on a particular subclass of processes: *Structured Business Processes*, which are a class of processes similar to the structured workflows defined by Kiepuszewski et al. [21]. These processes are defined as a recursive nesting of their components, where each nesting structure is defined as a process block, as well as the process itself and its atomic components, the tasks.

The models used in the sub-problem are both structured and acyclic, such that each execution in the process model is guaranteed to terminate.

Definition 1 (Process Block)**.** *A process block B is a directed graph: the nodes are called* elements *and the directed edges are called* arcs*. The set of elements of a process block are identified by the function $V(B)$ and the set of arcs by the function $E(B)$. The set of elements is composed of tasks and coordinators. The coordinators are of 4 types:* and_split*,* and_join*,* xor_split *and* xor_join*. Each process block B has two distinguished nodes called the* initial *and* final *element. The initial element has no incoming arc from other elements in B and is denoted by $b(B)$. Similarly the final element has no outgoing arcs to other elements in B and is denoted by $f(B)$.*

A directed graph composing a process block is defined inductively as follows:

- *A single task constitutes a process block. The task is both initial and final element of the block.*
- *Let $B_1, \ldots, B_n$ be distinct process blocks with $n > 1$:*
  - $\mathsf{SEQ}(B_1, \ldots, B_n)$ *denotes the process block with node set $\bigcup_{i=0}^{n} V(B_i)$ and edge set $\bigcup_{i=0}^{n}(E(B_i) \cup \{(f(B_i), b(B_{i+1})) : 1 \leq i < n\})$.*
    The initial element of $\mathsf{SEQ}(B_1, \ldots, B_n)$ *is $b(B_1)$ and its final element is $f(B_n)$.*

- $\mathsf{XOR}(B_1,\dots,B_n)$ *denotes the block with vertex set* $\bigcup_{i=0}^{n} V(B_i) \cup \{\mathsf{xsplit},\mathsf{xjoin}\}$ *and edge set* $\bigcup_{i=0}^{n}(E(B_i) \cup \{(\mathsf{xsplit}, b(B_i)), (f(B_i), \mathsf{xjoin}) : 1 \leq i \leq n\})$ *where* xsplit *and* xjoin *respectively denote an* xor_split *coordinator and an* xor_join *coordinator, respectively. The initial element of* $\mathsf{XOR}(B_1,\dots,B_n)$ *is* xsplit *and its final element is* xjoin.
- $\mathsf{AND}(B_1,\dots,B_n)$ *denotes the block with vertex set* $\bigcup_{i=0}^{n} V(B_i) \cup \{\mathsf{asplit},\mathsf{ajoin}\}$ *and edge set* $\bigcup_{i=0}^{n}(E(B_i) \cup \{(\mathsf{asplit}, b(B_i)), (f(B_i), \mathsf{ajoin}) : 1 \leq i \leq n\})$ *where* asplit *and* ajoin *denote an* and_split *and an* and_join *coordinator, respectively. The initial element of* $\mathsf{AND}(B_1,\dots,B_n)$ *is* asplit *and its final element is* ajoin.

By enclosing a process block as defined in Definition 1 along with a start and end task in a sequence block, we obtain a *structured process model.*

Definition 2 (Structured Process Model). *A structured process model P is a directed graph composed of a process block B called the main process block. The process P is represented as a sequence block, as follows:* $\mathsf{SEQ}(\mathsf{start}, B, \mathsf{end})$, *where the vertex set of P is* $V(P) = V(B) \cup \{\mathsf{start};\mathsf{end}\}$ *and its edge set is* $E(P) = E(B) \cup \{(\mathsf{start},\ b(B)),\ (f(B),\ \mathsf{end})\}$. *The initial element of a structured process model is the pseudo-task* start *and its final element is the pseudo-task* end.

Example 1 (Structured Process Model). *Fig. 3 shows a structured business process containing four tasks labelled* t_1, t_2, t_3, t_4. *The structured process contains an* XOR *block delimited by the* xor_split *and the* xor_join. *The* XOR *block contains the tasks* t_1 *and* t_2. *The* XOR *block is itself nested inside an* AND *block with the task* t_3. *The* AND *block is preceded by the* start *and followed by task* t_4 *which in turn is followed by the* end *task.*

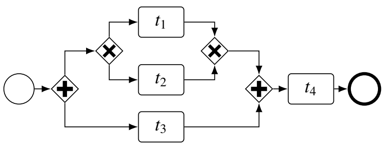


Figure 3: Example of a structured business process.

Considering the structured process in Figure 3 as a sequence block, it can be represented as follows:

$$P = \mathsf{SEQ}(\mathsf{start}, \mathsf{AND}(\mathsf{XOR}(t_1, t_2), t_3), t_4, \mathsf{end})$$

.

Note that for the process model P, $\mathsf{SEQ}(\mathsf{AND}(\mathsf{XOR}(t_1, t_2), t_3), t_4)$ *represents the main process block. The external sequence block of B is absorbed by the sequence block of process model itself, resulting in the final representation.*

Business Process Executions

In a structured process model, a valid execution identifies a path from the pseudo-task start to the pseudo-task end, and follows the semantics of the traversed coordinators. An execution is represented as a sequence of a subset of the tasks belonging to the process.

The possible executions allowed by a process model can be represented using partially ordered sets, where the ordering constraints represent the structure of the model. Thus a linear order of tasks following the partially ordered set ordering constraints represents a valid execution.

Definition 3 (Partially Ordered Set)**.** *A partially ordered set $\mathbb{P} = (\mathscr{S}, \prec_s)$ is a tuple where $\mathscr{S}$ is a set of elements and $\prec_s$ is a set of ordering relations between two elements of $\mathscr{S}$ such that $\prec_s \subseteq \mathscr{S} \times \mathscr{S}$ and for which* transitivity *and* antisymmetry[4] *hold.*

Let $\mathbb{P}_1 = (\mathscr{S}_1, \prec_{s_1})$ and $\mathbb{P}_2 = (\mathscr{S}_2, \prec_{s_2})$ be partially ordered sets, we define the following four operations:

- *Union: $\mathbb{P}_1 \cup_{\mathbb{P}} \mathbb{P}_2 = (\mathscr{S}_1 \cup \mathscr{S}_2, \prec_{s_1} \cup \prec_{s_2})$, where $\cup$ is the disjoint union.*
- *Concatenation: $\mathbb{P}_1 +_{\mathbb{P}} \mathbb{P}_2 = (\mathscr{S}_1 \cup \mathscr{S}_2, \prec_{s_1} \cup \prec_{s_2} \cup \{s_1 \prec s_2 | s_1 \in \mathscr{S}_1 \text{ and } s_2 \in \mathscr{S}_2\})$.*
- *Linear Extensions: $\mathscr{I}(\mathbb{P}_1) = \{(\mathscr{S}, \prec_s) | \mathscr{S} = \mathscr{S}_1, (\mathscr{S}, \prec_s)$ is a totally ordered sequence and $\prec_{s_1} \subseteq \prec_s\}$.*

The associative *property holds for Union, and Concatenation.*

A serialisation of a process block is a totally ordered sequence of a subset of the tasks. The sequence must follow the semantics of the coordinators contained in the block, and start with the block initial element and finish with its final element. Notice that the definition of serialisation is given as a byproduct of Definition 4.

Definition 4 (Process Block Serialisations)**.** *Given a process block B, the set of serialisations of B, written $\Sigma(B) = \{\varepsilon | \varepsilon$ is a serialisation of $B\}$. The function $\Sigma(B)$ is defined as follows:*

1. *If B is a task t, then $\Sigma(B) = \{(\{t\}, \emptyset)\}$*
2. *if B is a composite block with sub-blocks $B_1, \ldots, B_n$ let ε_i be the projection of ε on block B_i (obtained by ignoring all tasks which do not belong to B_i)*
   - *(a) If $B = \mathsf{SEQ}(B_1, \ldots, B_n)$, then $\Sigma(B) = \{\varepsilon_1 +_{\mathbb{P}} \cdots +_{\mathbb{P}} \varepsilon_n | \varepsilon_i \in \Sigma(B_i)\}$*
   - *(b) If $B = \mathsf{XOR}(B_1, \ldots, B_n)$, then $\Sigma(B) = \Sigma(B_1) \cup \cdots \cup \Sigma(B_n)$*

[4]*Antisymmetry*: if $a \prec_s b$ and $b \prec_s a$, then $a = b$.

(c) If $B = \mathsf{AND}(B_1,\dots,B_n)$*, then* $\Sigma(B) = \{\varepsilon | \varepsilon \in \mathscr{I}(\varepsilon_1 \cup_{\mathbb{P}} \cdots \cup_{\mathbb{P}} \varepsilon_n | \forall \varepsilon_i \in \Sigma(B_i))\}$.

A serialisation of a process model corresponds to one of its executions, hence the set of serialisations of a business process model corresponds to the set of possible executions of the model itself.

Definition 5 (Execution)**.** *Given a structured process P whose main process block is B, an execution of P corresponds to a serialisation of B including the pseudo-tasks* start *and* end.

$$\Sigma(P) = \{\mathbb{P}_{\mathsf{start}} +_{\mathbb{P}} \varepsilon +_{\mathbb{P}} \mathbb{P}_{\mathsf{end}} | \varepsilon \in \Sigma(B)\}$$

Example 2 (Execution)**.** *Consider a business process model P, like the one shown in Example 1:*

$$P = \mathsf{SEQ}(\mathsf{start}, \mathsf{AND}(\mathsf{XOR}(t_1,t_2),t_3),t_4,\mathsf{end})$$

The corresponding possible executions of P, $\Sigma(P)$*, are as follows:*

$\varepsilon_1 : \mathsf{start}, t_1, t_3, t_4, \mathsf{end},$

$\varepsilon_2 : \mathsf{start}, t_3, t_1, t_4, \mathsf{end},$

$\varepsilon_3 : \mathsf{start}, t_2, t_3, t_4, \mathsf{end},$

$\varepsilon_4 : \mathsf{start}, t_3, t_2, t_4, \mathsf{end}$

Annotations

The *state* of a process is represented using a set of literals. We assume that executing a task can alter the state of the process, represented using *annotations*, as described by Governatori et al. [14]. The state consists of sets of literals associated to the tasks, where a literal is either an atomic proposition or its negation. A task's annotation describes the changes in the process state when the associated task is executed.

Both the state of a process and the annotations of the tasks are represented by sets of literals, which are required to be consistent.

Definition 6 (Consistent literal set)**.** *A set of literals L is* consistent *if and only if it does not contain both l and* $\neg l$.

An annotated process is a process whose tasks are associated with consistent sets of literals.

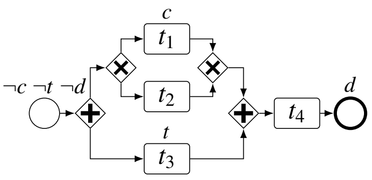


Figure 4: Example of an annotated business process.

Definition 7 (Annotated process)**.** *Let P be a structured process and let T be the set of tasks in P. An annotated process is a pair: (P, ann), where ann is a function associating a consistent set of literals $\mathsf{ann} : T \mapsto L$ to each task in T.*

Definition 8 (State)**.** *Let t_i be a task, and L is a consistent literal set. A state is a tuple $\sigma = (t_i, L)$, and represents the state holding after executing t_i.*

The state of a process is updated by each task's execution through an update operator. This operator is inspired by the AGM belief revision operator [1].

Definition 9 (Literal set update)**.** *Let $\bar{l}$ be the complementary literal as follows:*

- $\bar{l} = \neg\alpha$ *if* $l = \alpha$
- $\bar{l} = \alpha$ *if* $l = \neg\alpha$

Given two consistent sets of literals L_1 and L_2, the update of L_1 with L_2 (denoted by $L_1 \oplus L_2$) is a set of literals defined as follows:

$$L_1 \oplus L_2 = L_1 \setminus \{\bar{l} \mid l \in L_2\} \cup L_2$$

Finally, a *trace* represents the evolution of the state of a process during one of its executions.

Definition 10 (Trace)**.** *Given an annotated process (P, ann) and an execution sequence $\varepsilon = (t_1, \ldots, t_n)$ such that $\varepsilon \in \Sigma(P)$, a trace θ is a finite sequence of states: $(\sigma_1, \ldots, \sigma_n)$. Each state of $\sigma_i \in \theta$ contains a set of literals L_i capturing what holds after the execution of a task t_i. Each L_i is a set of literals such that:*

1. $L_1 = \mathsf{ann}(t_1)$;
2. $L_{i+1} = L_i \oplus \mathsf{ann}(t_{i+1})$, *for* $1 \leq i < n$.

We use $\Theta(B, \mathsf{ann})$ to denote the set of possible traces resulting from an annotated process block (B, ann), where B is a process block and ann is an annotation function.

Example 3 (Trace)**.** *Consider the annotated structured process in Figure 4, containing the following annotations:*

$\mathsf{start} : \{\neg c, \neg t, \neg d\}$

$t_1 : \{c\}$

$t_3 : \{t\}$

$\mathsf{end} : \{d\}$

The following traces correspond to the executions of P, as illustrated in Example 2, written $\Theta(P)$*:*

$\theta_1 : (\mathsf{start}; \neg c, \neg t, \neg d), (t_1; c, \neg t, \neg d), (t_3; c, t, \neg d), (t_4; c, t, \neg d), (\mathsf{end}; c, t, d),$

$\theta_2 : (\mathsf{start}; \neg c, \neg t, \neg d), (t_3; \neg c, t, \neg d), (t_1; c, t, \neg d), (t_4; c, t, \neg d), (\mathsf{end}; c, t, d),$

$\theta_3 : (\mathsf{start}; \neg c, \neg t, \neg d), (t_2; \neg c, \neg t, \neg d), (t_3; \neg c, t, \neg d), (t_4; \neg c, t, \neg d), (\mathsf{end}; \neg c, t, d),$

$\theta_4 : (\mathsf{start}; \neg c, \neg t, \neg d), (t_3; \neg c, t, \neg d), (t_2; \neg c, t, \neg d), (t_4; \neg c, t, \neg d), (\mathsf{end}; \neg c, t, d)$

4.2 On the Computational Complexity

Colombo Tosatto et al. [6] have shown that the problem of proving whether a business process model complies with a given set of regulations is an **NP**-complete problem, when structured business process models containing both concurrent components and mutually exclusive ones are used, and the regulations are expressed using literals. The structural components of business process models contribute to the computational complexity of the problem. In particular, two major contributors can be identified: XOR and AND process blocks, each of which will be discussed below.

XOR **Blocks**

XOR blocks contribute to the computational complexity of the problem by allowing the representation of multiple possible executions within a single business process model.

An XOR block allows us to represent possible branches in the executions within a model. The branching factor in an XOR block, represented by the sub-blocks contained, determines the amount of different possible executions obtainable by executing the block, as described in Definition 5.

A single XOR block does not substantially contribute to the computationally complexity of the problem, as it increases the amount of execution available within the model by the number of sub-blocks contained. However, nesting XOR blocks does increase the number of possible executions, as it is no longer polynomial with respect to the business process structure. Let n be the number of sub-blocks in an XOR block and l be the nesting level. Then the amount of possible executions in the worst case scenario, where each branch of an XOR block contains a nested XOR block with n branches, up to a nesting level l, is:

$$n^l$$

In general, when the branching factor of properly nested XOR blocks is not constant and not every branch necessarily nests another XOR block, then the number of possible executions deriving from this structure is calculated by counting the number of branches in the structure not containing a nested XOR block.

Example 4. *Considering the business process model illustrated in Figure 5. The model contains* 3 XOR *blocks nested within* 2 *levels, and each block contains* 2 *branches. Given the structure, we can then calculate the number of possible executions of the model, which is* $2^2 = 4$.

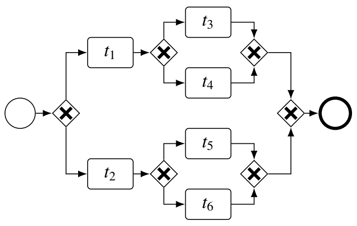


Figure 5: Executions in a model containing nested XOR blocks.

In constrast, when multiple XOR blocks are used sequentially, the amount of possible executions becomes exponential with respect to the number of XOR blocks. Let n be the number of sub blocks in an XOR block and k the number of XOR blocks. Then in the worst case scenario, where each block has the same branching factor, the number of possible executions in a business process model is:

$$n^k$$

Note that in the general case, where the number of branches in the different XOR blocks of a model is not the same, assuming that k is the number of XOR blocks in the model and n_i is the number of branches in the block i for some $1 \leq i \leq k$, the number of executions in the model is calculated as follows:

$$\sum_{i=1}^{k}(n_i)$$

Example 5. *Consider the business process model illustrated in Figure 6. The model contains* 2 XOR *blocks with* 2 *branches each. Given the structure, we can then calculate the number of possible executions of the model, which is* $2^2 = 4$.

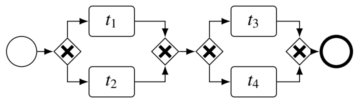


Figure 6: Executions in a model containing sequential XOR blocks.

Given the amount of possible executions with respect to the size of the model in the worst case scenario, a brute force analysis of each execution is theoretically intractable. In other words, it means that the time required to find a solution would exponentially increase with the size of the problem, hence practically making big enough problems unsolvable using brute force approaches.

AND **Blocks**

Similar to XOR blocks, AND blocks contribute to the computational complexity by allowing a compact representation of multiple different possible executions within a single business process model.

Contrary to XOR blocks, however, the positioning of AND blocks in the model does not strongly influence the amount of possible executions available in the business process model. Let k be the number of AND blocks, n the number of branches in an AND block, and m the length of the branches of the block in term of executable tasks. The amount of possible executions in a business process model is combinatorial with respect to the model structure:

$$\left(\frac{(n \times m)!}{(m!)^n}\right)^k$$

In the general case, where n_i is the number of branches of the block i for $1 \leq i \leq k$, and m_{i_j} is the length of the jth branch of block i for $1 \leq j \leq n_i$, the number of executions in a business process model with k AND blocks is calculated as follows:

$$\prod_{i=1}^{k}\left(\frac{(\sum_{j=1}^{n_i}(m_{i_j}))!}{\prod_{j=1}^{n_i}(m_{i_j}!)}\right)$$

Example 6. *Consider the business process model illustrated in Figure 7. The model contains a single* AND *block with* 3 *branches of size* 2 *each. Given the structure, we can then* *calculate the number of possible executions of the model, which is* $\frac{(2+2+2)!}{(2!)^3} = 90$. *However, if we increase the length of each branch by 1 (adding only 3 activities in total), the number of possible executions of the model is* $\frac{(3+3+3)!}{(3!)^3} = 1680$.

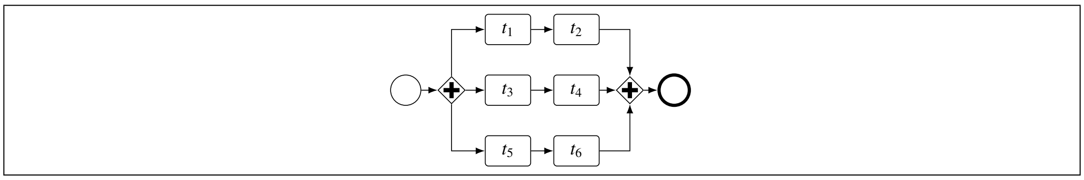


Figure 7: Executions in a model containing AND blocks.

Note that, as this brief analysis suggests, AND blocks contribute more heavily to the computational complexity of the problem than XOR blocks.

Complexity of Real-life Business Process Models

Consider for example the SAP R/3 collection of business process models, used by SAP to customize their R/3 ERP product as documented in [7]. As shown in Table 1, the structural complexity is reasonable with the most complex model having 86 activities and 6 AND splits. However, even when selecting the set of sound workflow models, the complexity in terms of total possible executions can grow as large as $1.76 \cdot 10^{10}$. This shows that the incentive for a more efficient algorithm is not just academic, but a necessity imposed by real-life model complexities.

Metric	**Min**	**Max**	**Mean**
Activities	3	86	15.98
XOR splits	0	6	0.64
Outdegree XOR	2	9	2.73
AND splits	0	6	1.14
Outdegree AND	2	18	3.14

Table 1: Statistics on SAP R/3 model complexity.

4.3 Process Model Decomposition.

A business process model can be decomposed by splitting each of the XOR blocks within the process representation in *n* different processes, where *n* corresponds to the branches in the XOR block and each of the new processes contains exactly one of the branches. This procedure is recursively applied until no XOR blocks are left.

The decomposition can potentially lead to an exponential number of decomposed processes, as can be inferred from the contribution of the XOR components to the computational complexity. However, the presence of AND blocks still allows a combinatorial amount of possible executions within the decomposed models. Therefore, a solution is required to prevent analysis of each possible execution to verify the regulatory compliance of the model.

Example 7 (Decomposition). *Consider the business process model described in Example 1:*

$$P = \mathsf{SEQ}(\mathsf{start}, \mathsf{AND}(\mathsf{XOR}(t_1, t_2), t_3), t_4, \mathsf{end})$$

Given that P contains a single XOR *block with 2 branches, application of the decomposition procedure leads to the following decomposed processes:*

1. $P' = \mathsf{SEQ}(\mathsf{start}, \mathsf{AND}(t_1, t_3), t_4, \mathsf{end})$
2. $P'' = \mathsf{SEQ}(\mathsf{start}, \mathsf{AND}(t_2, t_3), t_4, \mathsf{end})$

A process block serialisation (cf. Definition 4) is constructed through partially ordered sets operations depending on the type of process blocks, as described in Definition 3. Subsequently, a decomposed process can be represented as a partially ordered set by a recursive transformation closely following Definition 4, as reported in Definition 11 below:

Definition 11 (Decomposed Process as Partially Ordered Set). *A decomposed process P can be represented as a partially ordered set by applying the following recursive procedure, $\mathscr{P}(B)$, to each of its process blocks in (B):*

1. *If B is a task t, then* $\mathscr{P}(B) = \{(\{t\}, \emptyset)\}$
2. *if B is a composite block with subblocks* $B_1, \dots, B_n$*:*
   (a) *If* $B = \mathsf{SEQ}(B_1, \dots, B_n)$*, then* $\mathscr{P}(B) = \{\mathscr{P}(B_1) +_{\mathbb{P}} \cdots +_{\mathbb{P}} \mathscr{P}(B_n)\}$
   (b) *If* $B = \mathsf{AND}(B_1, \dots, B_n)$*, then* $\mathscr{P}(B) = \{\mathscr{P}(B_1) \cup_{\mathbb{P}} \cdots \cup_{\mathbb{P}} \mathscr{P}(B_n)\}$.

Example 8 (Decomposed Partially Ordered Sets). *Given the two decomposed processes from Example 7:*

1. $P' = \mathsf{SEQ}(\mathsf{start}, \mathsf{AND}(t_1, t_3), t_4, \mathsf{end})$
2. $P'' = \mathsf{SEQ}(\mathsf{start}, \mathsf{AND}(t_2, t_3), t_4, \mathsf{end})$

The corresponding partially ordered sets for the two decomposed processes are the following:

1. $\mathscr{P}(P') = (\{\mathsf{start}, t_1, t_3, t_4, \mathsf{end}\}, \{\mathsf{start} \prec t_1, \mathsf{start} \prec t_3, t_1 \prec t_4, t_3 \prec t_4, t_4 \prec \mathsf{end}\})$
2. $\mathscr{P}(P'') = (\{\mathsf{start}, t_2, t_3, t_4, \mathsf{end}\}, \{\mathsf{start} \prec t_2, \mathsf{start} \prec t_3, t_2 \prec t_4, t_3 \prec t_4, t_4 \prec \mathsf{end}\})$

Each decomposed process contains the possible executions given the same choices in the XOR blocks, and the execution set of a decomposed process is disjoint from any other decomposed process obtained. The union of the execution sets of the decomposed processes is exactly the execution set of the original process model.

5 Regulations and Δ-Constraints

In this section, we introduce the regulations that the process model must fulfil, along with an alternative representation of the regulations used by our solution which we refer to as Δ-constraints.

5.1 Regulations

The regulations are defined as conditional obligations. This kind of obligations consists of three conditions: a *trigger* defining when the obligation becomes in force, a *deadline* defining when its in force period terminates, and a *condition* defining the requirement in the in force period.

A conditional obligation can be either of the achievement or maintenance type, as represented in Definition 12 by $o \in \{a, m\}$ respectively. The notion of achievement and maintenance obligations are inspired by the notion of achievement and maintenance goals introduced by Cohen and Levesque [5], while the full semantics of such notions have been discussed by Hashmi et al. [18].

Definition 12 (Conditional Obligation)**.** *A local obligation ω is a tuple $\langle o, c, t, d \rangle$, where $o \in \{a, m\}$ represents the type of the obligation. The elements c, t and d are propositional literals in $\mathscr{L}$. c is the condition of the obligation, t is the trigger of the obligation and d is the deadline of the obligation.*

We use the notation $\omega = \mathscr{O}^o \langle c, t, d \rangle$ to represent a conditional obligation.

Definition 13. *Given an obligation $\mathscr{O}^o \langle c, t, d \rangle$, the annotation of* start *is assumed to contain the negation of each literal in the obligation tuple. The annotation of* end *is assumed to contain the literal referring to the deadline.*

Definition 13 provides an initial process state where none of the literals defining an obligation hold. In addition, it ensures that the in force interval of an obligation always terminates when the process execution ends. Note that obligations are evaluated one at a time, hence the annotation of start and end depends on the obligation being evaluated.

Limiting the Expressivity to Literals

Given our goal to reduce the computational complexity of solving sub-problems of verifying the compliance of business process models, we limit the expressivity of the obligations defining the requirements to simple propositional literals instead of propositional formulae. As we discuss later in Section 6, our solution manages to provide a more efficient solution for the sub-problem considered by avoiding to explicitly analyse each possible execution of a business process model, which would otherwise lead to a search state explosion.

The advantage of limiting the expressivity of the obligations by including only propositional literals allows to directly associate the interaction between the obligation's elements to tasks in the business process model. This implies that executing one of such tasks would immediately satisfy one of the elements of the obligation, like the *condition*, *trigger* or *deadline*.

Such direct association would not be possible if these elements in the obligations were to be expressed using propositional formulae. In that case, the satisfaction of an obligation's element can be potentially influenced by a combination of tasks being executed. For instance, assuming that the trigger of an obligation to be the formula $\alpha \land \beta$, its satisfaction can be achieved by executing two tasks, one introducing α in the process state, and another introducing β. Moreover, we would also need to track whether no other tasks are executed between those and removing such literals from the process state, which would not lead to the satisfaction of the formula when the second task would be executed. The complication brought by allowing formulae in the obligation's elements would require knowing the exact execution order of the tasks in order to determine whether an obligation is fulfilled. Furthermore, this would be required for each possible execution order of the business process model, which potentially leads to a intractability problem, as the number of executions of a business process model is in general combinatorial with respect to the number of elements composing the model.

Fulfilling Obligations

Before proceeding with the formal introduction of the different obligations, we first introduce two syntactical shorthands to keep the subsequent definitions more compact.

Definition 14 (Syntax Shorthands)**.** *To avoid cluttering, we adopt the following shorthands:*

- $\sigma \in \theta$ *such that* $\sigma \models l$ *is abbreviated as:* σ_l
- *A task-state pair appearing in a trace:* (t, σ) *such that* $l \in \mathsf{ann}(t)$*, is abbreviated as:* $\mathsf{contain}(l, \sigma)$

Note that an *in force* interval instance of an obligation, having l as trigger, is always started from a state σ, where $\mathsf{contain}(l, \sigma)$ is true. Therefore, multiple in force intervals of an obligation can co-exist at the same time. However, multiple in force intervals can be fulfilled by a single event happening in a trace, as shown in the following definitions.

An achievement obligation is fulfilled by a trace if the fulfilment condition holds at least in one of the trace states when the obligation is in force.

Definition 15 (Comply with Achievement). *Given an achievement obligation $\mathcal{O}^a\ \langle c,\, t,\, d\rangle$ and a trace θ, θ is compliant with $\mathcal{O}^a\langle c,t,d\rangle$ if and only if:* $\forall\sigma_t, \exists\sigma_c|\mathsf{contain}(t,\sigma_t)$ *and* $\sigma_t \preceq \sigma_c$ *and* $\neg\exists\sigma_d|\sigma_t \preceq \sigma_d \prec \sigma_c$*.*

A maintenance obligation, on the other hand, is fulfilled if the condition holds in each of the states where the obligation is in force.

Definition 16 (Comply with Maintenance). *Given a maintenance obligation $\mathcal{O}^m\ \langle c,\, t,\, d\rangle$ and a trace θ, θ is compliant with $\mathcal{O}^m\langle c,t,d\rangle$ if and only if:* $\forall\sigma_t, \exists\sigma_d|\mathsf{contain}(t,\sigma_t)$ *and* $\sigma_t \preceq \sigma_d$ *and* $\forall\sigma|\sigma_t \preceq \sigma \preceq \sigma_d, c \in \sigma$*.*

The two types of obligations considered in this paper allow to represent a variety of obligations existing in real world scenarios, like e.g requirements to achieve a certain condition before a deadline, or maintaining a condition for a period of time. We do not claim that the formalism adopted is sufficient to capture each possible requirement behaviour in real world scenarios, such as for example an obligation whose applicability condition is related to another obligation. In this paper, we do not consider these more complex behaviours, as the additional behavioural complexity would require to explicitly analyse each possible execution of a business process model in order to verify its compliance state.

5.2 On the Computational Complexity

Previously, we have discussed how the structural components of a business process model contribute to the computational complexity of the problem of proving regulatory compliance of business process models. In this subsection, we briefly discuss the impact of the regulatory requirements on the computational complexity of the problem.

Regulatory Complexity

The amount of expressivity allowed into describing the regulatory requirements to be verified directly influences the computational complexity of the problem. The use of logical formulae to represent the components of the regulatory requirements significantly influences the difficulty of finding a solution. As the components of the regulatory requirements need to be checked against the process state, the use of logical formulae to represent such components requires the exact execution history leading to a particular state, as any difference in the execution order can potentially lead to a different state and, hence, a differently evaluated formula.

However, relying on literals to represent the components of a regulations lifts the requirement of having to know the exact execution history. As a consequence, the verification can potentially be performed over the structure of the business process model, instead of the full list of possible executions.

In this paper, we adopt regulatory requirements restricted to be represented using literals instead of formulae, which allows to focus on the structural components contributing to the computational complexity of the problem. Despite the simplified regulatory requirements, however, Colombo Tosatto et al. [6] have shown that proving partial compliance[5] of a business process model against a set of regulatory requirements is still an **NP**-complete problem.

5.3 Translating the Obligations

Instead of checking all subsequent literals over each possible path in the process to ensure full compliance, we propose to verify whether a trace *violates* a given achievement obligation. The main advantage is that the conditions can then be verified directly on the process model, so that it is no longer required to generate and analyse all possible traces. The *failure condition* for achievement obligations is equivalent to the complement of Definition 15, as shown formally below:

Definition 17 (Achievement Failure). *Given an achievement obligation $\mathcal{O}^a\langle c,t,d\rangle$ and a trace θ, θ is not compliant with $\mathcal{O}^a\langle c,t,d\rangle$ if and only if:* $\exists\sigma_t,\ \sigma_d|\mathsf{contain}(t,\sigma_t)$ *and* $\sigma_t \preceq \sigma_d$ *and* $\neg\exists\sigma_c|\sigma_t \preceq \sigma_c \preceq \sigma_d$.

In order to compare the failure conditions and the business process model, we need to standardise the two representations. As such, we transform the failure conditions into so-called Δ-*constraints*, referring to the state update requirements ensuring that a given obligation is violated when such constraints are met. Therefore, Δ-constraints only require to prove the existence of a trace *failing* the fulfilment requirements, instead of proving that each possible trace fulfils them.

Translation for Achievement

The following definition translates Definition 17 into its Δ-constraints representation, describing the required order of state updates proving that a process model contains an execution violating the given obligation. For convenience, we use t_l to denote a task t such that $l \in \mathsf{ann}(t)$.

[5]Proving partial compliance requires to prove that a business process model contains at least one execution that is compliant with a set of regulatory requirements.

Definition 18 (Achievement Failure Δ-constraints)**.** *Given an achievement obligation $\mathcal{O}^a\langle c,t,d\rangle$ and an execution ε, ε is not compliant with $\mathcal{O}^a\langle c,t,d\rangle$, if and only if one of the following conditions is satisfied:*

1. *$\exists t_t$ such that:*

   $\exists t_{\neg c}, t_d | t_{\neg c} \preceq t_t \preceq t_d$ *and*

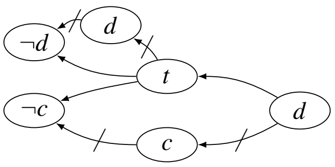
   $\neg\exists t_c | t_{\neg c} \preceq t_c \preceq t_d$ *and*

   $\exists t_{\neg d}, \neg\exists t_d | t_{\neg d} \preceq t_d \preceq t_t$

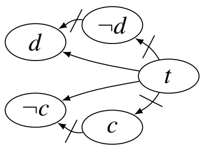
2. *$\exists t_t$ such that:*

   $\exists t_{\neg c}, \neg\exists t_c | t_{\neg c} \preceq t_c \preceq t_t$ *and*

   $\exists t_d, \neg\exists t_{\neg d} | t_d \preceq t_{\neg d} \preceq t_t$



$A\Delta1$ constraints



$A\Delta2$ constraints

Figure 8: Achievement constraints

Figure 8 provides a graphical representation of the Δ-constraints for achievement obligations. The nodes represent annotations in the tasks and the arrows represent the ordering relations that must be fulfilled by an execution of the process model to fulfil the Δ-constraint. A slashed arrow denotes the required absence of the respective element in the interval identified by the surrounding elements.

Lemma 1. *Given an achievement obligation, executing a task having the obligation's trigger annotated always results in the obligation being in force in the state after the task's execution, where the obligation can be potentially fulfilled or violated.*

Proof. **Sketch**

Either the execution changes the previous state where t was not holding to one where it is, or t had been holding already.

In the first case, the execution brings a new in force period for the obligation.

In the second case, either the obligation is in force and not fulfilled in the previous state, or it becomes fulfilled in the previous state. In both cases, the execution of the task brings the obligation in force again and requires to be fulfilled. □

Theorem 1. *Given an execution ε, represented as a sequence of tasks, if ε satisfies either of the Achievement Failure Δ-Constraints (Definition 18), then ε violates the obligation related to the Achievement Failure Δ-Constraints.*

Proof. **Soundness**:

First case: Achievement Failure Δ-Constraint 1

1. From the hypothesis, and Definition 18, it follows that ε satisfies the following conditions:

   (a) $\exists t_t | \exists t_{\neg c}, t_d | t_{\neg c} \preceq t_t \preceq t_d$

   (b) $\exists t_t | \neg \exists t_c | t_{\neg c} \preceq t_c \preceq t_d$

   (c) $\exists t_t | \exists t_{\neg d}, \neg \exists t_d | t_{\neg d} \preceq t_d \preceq t_t$

2. From 1.(a), and Lemma 1, it follows that: t holds and c does not, in the state holding after the execution of the task t_t.

3. From 2., and 1.(b), it follows that: there is no state included between the state after executing the task t_t and one where d starts holding, where c holds.

4. From 3. and Definition 17, it follows that ε would violate the obligation related to the Achievement Failure Δ-Constraints.

Second case: Achievement Failure Δ-Constraint 2

1. From the hypothesis, and Definition 18, it follows that ε satisfies the following conditions:

   (a) $\exists t_t | \exists t_{\neg c}, \neg \exists t_c | t_{\neg c} \preceq t_c \preceq t_t$

   (b) $\exists t_t | \exists t_d, \neg \exists t_{\neg d} | t_d \preceq t_{\neg d} \preceq t_t$

2. From 1.(a), and Lemma 1, it follows that t holds and c does not, in the state holding after the execution of the task t_t.

3. From 1.(b), it follows that t holds and d holds, in the state holding after the execution of the task t_t.

4. From 2. and 3., it follows that after executing the task t_t, t and d hold and c does not hold.

5. From 4. and Definition 17, it follows that ε would violate the obligation related to the Achievement Failure Δ-Constraints.

Completeness:

1. Given a trace violating a given achievement obligation.
2. From 1. and Definition 17 it follows that $\exists \sigma_t, \sigma_d | \mathsf{contain}(t, \sigma_t)$ and $\sigma_t \preceq \sigma_d$ and $\neg \exists \sigma_c | \sigma_t \preceq \sigma_c \preceq \sigma_d$.
3. From 2., it follows that a task with t annotated is executed.
4. From 2., it follows that a task with d annotated is executed.
5. From 2., it follows that $\neg c$ holds and no task with c annotated is executed between the one with t and the one with d.
6. Following from 3., 4., and 5. two cases are possible:

   $t_d \preceq t_t$ this case is covered by the second set of conditions in Definition 18.

   $t_t \preceq t_d$ this case covered by the first set of conditions in Definition 18.
7. Thus, all cases are covered and a violating trace is always identified by the Achievement Failure Δ-constraints. □

Translation for Maintenance

The translation is also applied to maintenance obligations. Definition 19 describes the failure condition for maintenance obligations, which is the complement of Definition 16. Definition 20 describes the corresponding Δ-constraints.

Definition 19 (Maintenance Failure)**.** *Given a maintenance obligation $\mathcal{O}^m\langle c,t,d\rangle$ and a trace θ, θ is not compliant with $\mathcal{O}^m\langle c,t,d\rangle$ if and only if:*

$$\exists \sigma_t \forall \sigma_d | \mathsf{contain}(t, \sigma_t) \textit{ and } \sigma_t \preceq \sigma_d \textit{ and } \exists \sigma_{\neg c} | \sigma_t \preceq \sigma_{\neg c} \preceq \sigma_d$$

Definition 20 (Maintenance Failure Δ-constraints)**.** *Given an achievement obligation $\mathcal{O}^m\langle c,t,d\rangle$ and an execution ε, ε is not compliant with $\mathcal{O}^m\langle c,t,d\rangle$ if and only if one of the following conditions is satisfied:*

1. *$\exists t_t$ such that:*

   $\exists t_{\neg c}, \neg \exists t_c | t_{\neg c} \preceq t_c \preceq t_t$
2. *$\exists t_t$ such that:*

   $\exists t_c, \neg \exists t_{\neg c} | t_c \preceq t_{\neg c} \preceq t_t$ *and*

   $\forall t_d (\exists t_{\neg c} | t_t \preceq t_{\neg c} \prec t_d)$

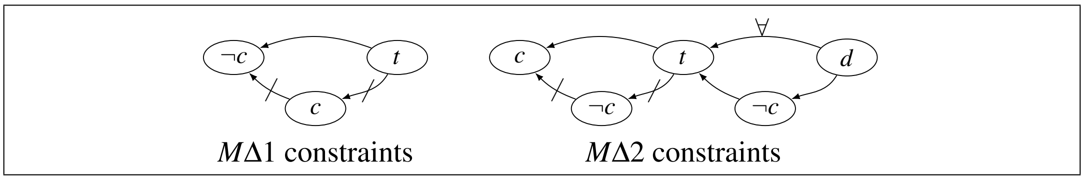


Figure 9: Maintenance constraints

Theorem 2. *Given an execution ε, represented as a sequence of tasks, if ε satisfies either of the Maintenance Failure Δ-Constraints (Definition 20), then ε violates the obligation related to the Maintenance Failure Δ-Constraints.*

Proof. **Soundness**:

First case: Maintenance Failure Δ-Constraint 1

1. From the hypothesis, and Definition 20, it follows that ε satisfies the following condition:

   (a) $\exists t_t | \exists t_{\neg c}, \neg \exists t_c | t_{\neg c} \preceq t_c \preceq t_t$

2. From 1.(a), and Lemma 1, it follows that: t holds and c does not, after the execution of the task t_t annotated.

3. From 2. and Definition 19, it follows that ε would violate the obligation related to the Maintenance Failure Δ-Constraints.

Second case: Maintenance Failure Δ-Constraint 2

1. From the hypothesis, and Definition 20, it follows that ε satisfies the following condition:

   (a) $\exists t_t | \exists t_c, \neg \exists t_{\neg c} | t_c \preceq t_{\neg c} \preceq t_t$ and
   (b) $\exists t_t | \forall t_d (\exists t_{\neg c} | t_t \preceq t_{\neg c} \prec t_d)$

2. From 1.(a), and Lemma 1, it follows that t holds and c holds, in the state holding after executing t_t.

3. From 1.(b), and Lemma 1, it follows that after executing the t_t, t it is always the case that in the following states c stops holding before d starts holding.

4. From 2. it follows that after the execution of the task t_t, c holds due to the constraint preventing a task having $\neg c$ annotated to be executed and cancelling c from the process state, which is already holding due to the execution of a task with c annotated.

5. From 3. it follows before the execution of any task having d in its annotation, a task having $\neg c$ annotated is executed, leading to the removal of c from the process state.

6. From 4. and 5. and Definition 19, it follows that ε would violate the obligation related to the Maintenance Failure Δ-Constraints.

Completeness:

1. Given a trace violating a given maintenance obligation.

2. From 1. and Definition 19 it follows that $\exists\sigma_t\ \forall\sigma_d \mid \mathsf{contain}(t,\sigma)$ and $\sigma_t \preceq \sigma_d$ and $\exists\sigma_{\neg c}|\sigma_t \preceq \sigma_{\neg c} \preceq \sigma_d$.

3. From 2., it follows that a task with t annotated is executed.

4. From 2., it follows that a task with $\neg c$ annotated is executed.

5. Following from 3., and 4. two cases are possible:

   $t_{\neg c} \preceq t_t$ case covered by the first set of conditions in Definition 20.

   $t_t \preceq t_{\neg c}$ case covered by the second set of conditions in Definition 20.

6. Thus, all cases are covered and a violating trace is always identified by the Maintenance Failure Δ-constraints.

□

Verifying Δ-constraints

Verifying whether a business process model satisfies a Δ-constraint instead of the original regulation is equivalent to looking for a counter-example falsifying whether a model is fully compliance. Thus, if such an example cannot be found in any possible execution, then the business process model is proven to be fully compliant. Note that every possible execution is implicitly checked by analysing the decomposed business processes as partially ordered sets, as discussed in Section 4.3.

Translation Complexity

Translating a given obligation in the corresponding set of Δ-constraints can be done in constant time, since depending on the type of obligations, the Δ-constraints need to be instantiated with the parameters of the obligation.

6 Verifying Full Compliance

In this section, we show how full compliance is verified for a process model with respect to a given regulation, by proving that the Δ-constraints are consistent with the set of partially ordered set representations of a decomposed process.

In addition, we argue that extending the procedure to prove full compliance against a *set of* regulations only requires to iterate the process over each obligation in the set, thus increasing the complexity of proving compliance for a single regulation by a polynomial factor.

6.1 Full Compliance

A trace is compliant with a set of obligations if it fulfils all obligations belonging to the set. Note that according to Definitions 15 and 16 an obligation that is never activated by a trace is considered to be fulfilled by such a trace.

Definition 21 (Set Fulfilment)**.** *Given a trace θ and a set of obligations $\mathbb{O} = \{\omega_1, \ldots, \omega_n\}$, $\theta \vdash \mathbb{O}$, iff:*

$$\forall \omega_i \in \mathbb{O}, (\theta \vdash \omega_i)$$

Otherwise $\theta \nvdash \mathbb{O}$.

Finally, a process model is said to be fully compliant with a set of obligations, if and only if each of its executions fulfils each of the obligations belonging to the given set.

Definition 22 (Process Full Compliance)**.** *Given a process (P, ann) and a set of obligations $\mathbb{O}$.*

- ***Full Compliance:*** $(P, \mathsf{ann}) \vdash^{F} \mathbb{O}$
  iff $\forall \theta \in \Theta(P, \mathsf{ann}), \theta \vdash \mathbb{O}$.

6.2 Δ-constraints Verification

Given a partially ordered set representation of a decomposed business process model and the Δ-constraints representation of a given obligation, we illustrate how the constraints can be verified in the partially ordered sets, signifying that the original business process model contains at least one execution failing the original obligation.

Relevant Tasks

The first step of the verification consists of populating the sets of *relevant tasks*. Each set of relevant tasks contains the tasks having annotated a parameter of the obligation being checked. For instance, the set of relevant tasks for d contains every task in the business process model having d annotated.

From Definition 18 and Definition 20, it follows that viable task sets are required for: t, c, $\neg c$, d, and $\neg d$. These viable task sets are respectively represented as follows: $\mathbb{T}$, $\mathbb{C}$, $anti\mathbb{C}$, $\mathbb{D}$, $anti\mathbb{D}$. It follows from Definition 13 that the end task always belongs to $\mathbb{D}$.

Algorithm

Intuitively, to prove whether a process model is not fully compliant with a given regulation, we have to show that there exists a task in the process model having t annotated, and that it is possible to find instantiations of the relevant tasks for the existential quantifiers[6] in the Δ-constraints satisfying their ordering conditions.

```
input : Relevant tasks: $\mathbb{T}$, $anti\mathbb{C}$, $\mathbb{C}$, $\mathbb{D}$, $anti\mathbb{D}$ and a partially ordered set representation of a decomposed business
        process model $\mathscr{P}(P')$
output: Whether a partially ordered representation $P$ contains an execution violating $\omega$ by a trigger of $t$
1  for $t_t \in \mathbb{T}$ do
2      for $t_{\neg c} \in anti\mathbb{C}$ do
3          for $t_d \in \mathbb{D}$ do
4              if $t_{\neg c} \preceq t_t \preceq t_d$ compatible with $\mathscr{P}(P')$ then
5                  good = true;
6                  for $t_c \in \mathbb{C}$ do
7                      if $t_{\neg c} \preceq t_c \preceq t_d$ compatible with $\mathscr{P}(P')$ then
8                          good = false;
                       end
                   end
9                  if good then
10                     for $t_{\neg d} \in anti\mathbb{D}$ do
11                         if $t_{\neg d} \preceq t_t$ then
12                             good = true;
13                             for $t2_d \in \mathbb{D}$ do
14                                 if $t_{\neg d} \preceq t2_d \preceq t_t$ compatible with $\mathscr{P}(P')$ then
15                                     good = false;
                                   end
                               end
16                             if good then
17                                 return true;
                               end
                           end
                       end
                   end
               end
           end
       end
   end
18 return false;
```

Algorithm 1: Achievement Δ-constraint 1 ($A\Delta 1$)

Algorithm 1 illustrates how to verify whether a partially ordered set representation of a

[6]In Definition 20, the universal quantifier can be understood as it is referring to the earliest happening element quantified.

decomposed business process model satisfies the first Δ-constraint for achievement obligations. Note that checking whether a given ordering of tasks is compatible with the ordering of a partially ordered set, appearing in Algorithm 1 in lines 4, 7, and 14, can be done in polynomial time.

Complexity of Algorithm 1

Let $\mathbb{X}$ be the number of tasks in P, and assuming the worst case scenario, where the cardinality of each relevant set is $\mathbb{X}$. The computational complexity of Algorithm 1 is the following:

$$\mathbf{O}(|\mathbb{X}|^2 \times (|\mathbb{X}| + |\mathbb{X}|^2))$$

Thus the computational complexity of Algorithm 1 is polynomial in time with respect to the size of the process model. Algorithms to verify the other Δ-constraints closely resemble Algorithm 1[7], hence we do not explicitly illustrate and discuss them in this paper. Moreover, as Achievement Failure Δ-Constraint 1 is the one containing more ordering conditions (as shown in Definition 18 and Definition 20), the computational complexity of the other algorithms not explicitly discussed is at most as high as Algorithm 1.

6.3 Proving Full Compliance

Algorithm 2 shows how the algorithms verifying whether the Δ-constraints are satisfied in a decomposed business process model can be used to prove whether a business process model is fully compliant with a given obligation.

```
input  : An obligation $\omega$, a set of decomposed processes $\mathbb{P}$ and its viable task sets with respect to the obligation
output: Whether the process model corresponding to $\mathbb{P}$ is fully compliant with $\omega$

foreach decomposed process $P' \in \mathbb{P}$ do
    if $\omega$ is achievement then
        if $A\Delta 1(\mathbb{T}, anti\mathbb{C}, \mathbb{C}, \mathbb{D}, anti\mathbb{D}, \mathscr{P}(P'))$ then return false;
        if $A\Delta 2(\mathbb{T}, anti\mathbb{C}, \mathbb{C}, \mathbb{D}, anti\mathbb{D}, \mathscr{P}(P'))$ then return false;
    else
        if $M\Delta 1(\mathbb{T}, anti\mathbb{C}, \mathbb{C}, \mathbb{D}, anti\mathbb{D}, \mathscr{P}(P'))$ then return false;
        if $M\Delta 2(\mathbb{T}, anti\mathbb{C}, \mathbb{C}, \mathbb{D}, anti\mathbb{D}, \mathscr{P}(P'))$ then return false;
    end
end
return true;
```

Algorithm 2: Proving Full Compliance

[7]Other Δ-constraints checking algorithms would be structurally equivalent to Algorithm 1, with the only difference that the instantiations of the relevant tasks would be done on the ordering conditions of the other Δ-constraints.

Complexity of Algorithm 2

Let $\mathbb{P}$ be the set of the decomposed representations of the original process. The computational complexity of proving whether the process is compliant with a given regulation is the following:

$$\mathbf{O}(|\mathbb{P}| \times (\mathbf{O}(A\Delta 1) + \mathbf{O}(A\Delta 2) + \mathbf{O}(M\Delta 1) + \mathbf{O}(M\Delta 2)))$$

As $\mathbb{P}$ can be potentially exponential in size with respect to the size of the original process, we cannot claim that the complexity of Algorithm 2 is polynomial. However, as the computational complexity of any brute force approach to solve regulatory compliance is combinatorial with respect to the size of the problem (see Example 6), the proposed solution represents a more efficient approach as its computational complexity is exponential in time with respect to the size of the problem.

Illustrating the Verification

Examples 9 and 10 show how the Δ-constraints allow to identify fully compliant processes by analysing their executions. However, note that the thorough analysis of the executions is given only for illustration purposes, reminding that the proposed approach in the paper verifies the Δ-constraints directly on the partially ordered sets, as show by Algorithm 1.

Example 9 (Non-compliance)**.** *Consider the annotated business process depicted in Figure 3 and its corresponding partially ordered sets of Example 8. P' denotes a concurrent execution of t_1 and t_3 after* start*, followed by t_4.*

P' allows two valid executions: ε_1: start,t_1,t_3,t_4,end *and ε_2:* start,t_3,t_1,t_4,end*. We substitute the task's labels with their annotations, making it easier to observe whether the Δ-constraints are fulfilled.*
ε_1: $(\neg c, \neg t, \neg d)$, (c), (t), $()$, (d)
ε_2: $(\neg c, \neg t, \neg d)$, (t), (c), $()$, (d)

Similarly, $\mathbb{P}''$ denotes a concurrent execution of t_2 and t_3 after start*, followed by t_4. As such, $\mathbb{P}''$ allows two valid executions:*
ε_3: $(\neg c, \neg t, \neg d)$, $()$, (t), $()$, (d)
ε_4: $(\neg c, \neg t, \neg d)$, (t), $()$, $()$, (d)

Given an obligation $\mathcal{O}^a\langle c,t,d\rangle$, applying the achievement patterns to ε_1, it is easy to see that c exists before d. As such, Achievement Failure Δ-Constraint 1 fails, as it requires the absence of c before d. Equivalently, there exists no d before t and Achievement Failure Δ-Constraint 2 fails immediately as well. Similarly for ε_2.

For ε_3, there exists no d before t, as in ε_1 and ε_2. From start *$\neg c$ holds, continues to hold through t_2, and still holds when t occurs in t_3. As t_1 is not part of the trace, c does not occur before d. Therefore, ε_3 fulfils the Achievement Failure Δ-Constraint 1 pattern and is, as a* *result, not compliant. This is similar for ε_4. Consequently, the process of Figure 3 is not fully compliant.*

Example 10 (Full compliance)**.** *Consider the following annotated business process model:*

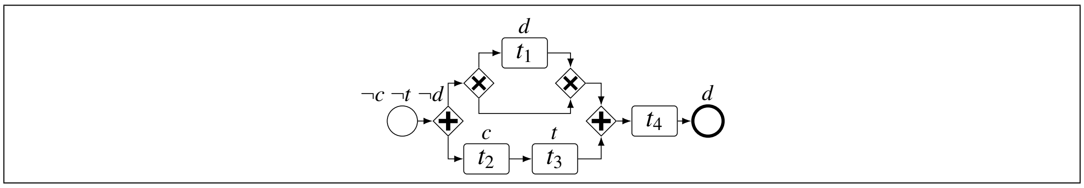


Figure 10: Example of a compliant process

This process can be decomposed as follows:

1. $P' = \mathsf{SEQ}(\mathsf{start}, \mathsf{AND}(t_1, \mathsf{SEQ}(t_2, t_3)), t_4, \mathsf{end})$

2. $P'' = \mathsf{SEQ}(\mathsf{start}, \mathsf{AND}(\emptyset, \mathsf{SEQ}(t_2, t_3)), t_4, \mathsf{end})$

The corresponding partially ordered sets for the two decomposed processes are the following:

1. $\mathbb{P}' = (\{\mathsf{start}, t_1, t_2, t_3, t_4, \mathsf{end}\}, \{\mathsf{start} \prec t_1, \mathsf{start} \prec t_2, t_1 \prec t_4, t_2 \prec t_3, t_3 \prec t_4, t_4 \prec \mathsf{end}\})$

2. $\mathbb{P}'' = (\{\mathsf{start}, t_2, t_3, t_4, \mathsf{end}\}, \{\mathsf{start} \prec t_2, t_2 \prec t_3, t_3 \prec t_4, t_4 \prec \mathsf{end}\})$

$\mathbb{P}'$ *allows three valid executions*[8]*:*
ε_1*:* $(\neg c, \neg t, \neg d)$, (d), (c), (t), $()$, (d)
ε_2*:* $(\neg c, \neg t, \neg d)$, (c), (d), (t), $()$, (d)
ε_3*:* $(\neg c, \neg t, \neg d)$, (c), (t), (d), $()$, (d)
$\mathbb{P}''$ *allows only one valid execution:*
ε_4*:* $(\neg c, \neg t, \neg d)$, (c), (t), $()$, (d).

All four executions have c before t. Given two obligations: $\mathcal{O}^a\langle c, t, d\rangle$ and $\mathcal{O}^m\langle c, t, d\rangle$, the respective Δ-constraints Achievement Failure Δ-Constraint 1, Achievement Failure Δ-Constraint 2 and Maintenance Failure Δ-Constraint 1 fail, as they require the absence of c before t. Maintenance Failure Δ-Constraint 2 does have c before t, but also requires $\neg c$ between t and d for every d and fails, therefore, as well for all executions. As none of the patterns apply to the process, we can conclude that the process is fully compliant.

	Nesting							
	Level 1	Level 2	Level 3	Executions	Dec.	Time	Check T.	Total Time
1	$1 \cdot \text{AND}_{2\times5}$	–	–	252	1	0.08 ms	0.04 ms	0.12 ms
2	$1 \cdot \text{AND}_{4\times5}$	–	–	1.17E+10	1	0.08 ms	0.05 ms	0.12 ms
3	$2 \cdot \text{AND}_{2\times5}$	–	–	63 504	1	0.07 ms	0.04 ms	0.11 ms
4	$2 \cdot \text{AND}_{4\times5}$	–	–	1.38E+20	1	0.08 ms	0.20 ms	0.29 ms
5	$1 \cdot \text{AND}_{2\times5}$	$1 \cdot \text{XOR}_{2\times5}$	–	194 480	4	0.06 ms	0.30 ms	0.36 ms
6	$1 \cdot \text{AND}_{4\times5}$	$1 \cdot \text{XOR}_{4\times5}$	–	3.43E+20	256	3.13 ms	2.21 ms	5.34 ms
7	$1 \cdot \text{AND}_{2\times5}$	$1 \cdot \text{AND}_{2\times5}$	–	2.55E+12	1	0.05 ms	0.14 ms	0.19 ms
8	$1 \cdot \text{AND}_{4\times5}$	$1 \cdot \text{AND}_{4\times5}$	–	1.27E+95	1	0.42 ms	2.85 ms	3.27 ms
9	$1 \cdot \text{AND}_{2\times5}$	$1 \cdot \text{XOR}_{2\times5}$	$1 \cdot \text{AND}_{2\times5}$	2.30E+15	4	0.18 ms	0.85 ms	1.02 ms
10	$1 \cdot \text{AND}_{4\times5}$	$1 \cdot \text{XOR}_{4\times5}$	$1 \cdot \text{AND}_{4\times5}$	1.11E+107	256	11.88 ms	120.50 ms	132.38 ms

Table 2: Evaluation models and performance.

7 Evaluation

We implemented the proposed method as a standalone Java tool. We tested our approach over a set of synthetic process models of increasing complexity, up to the point where the amount of concurrency is well beyond realistic business scenarios. The models consist of a set of nested process constructs, which are either a structured AND-block or structured XOR-block with *n* branches of *m* activities long. Each of the synthetic models is randomly annotated, which ensures that one every three tasks in the model is annotated with a randomly selected set of literals. Figure 11 shows the basic structure of the models.

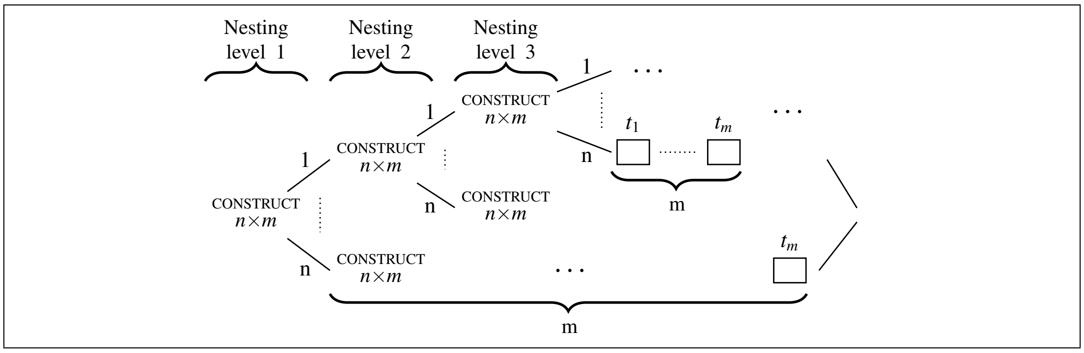


Figure 11: Synthetic process structure.

All tests were performed on a computer equipped with a quad core Intel® Core™ i7-7700HQ CPU @ 3.80GHz, 16GB RAM, running Ubuntu 16.10 and Java 1.8.0_131. To eliminate load times, each test was executed five times, where the average time of three executions was recorded while removing the fastest and the slowest.

[8]We use again the tasks' annotation instead of their labels to clearly show whether a Δ-constraint is satisfied.

The results are shown in Table 2. The first three columns of the table contain details about the process structure. The columns under *Nesting* contain information on the process structure, by depicting for each nesting level the number of blocks, their type, the number of branches, and the length of each branch. As such, the column *Level 2* describes the blocks nested in each branch of each block described in *Level 1*. Similarly, *Level 3* contains information of the blocks nested in each branch of each block in *Level 2*. The fourth column, *Executions*, describes how many different executions are hypothetically possible in the given model when linearising the concurrent paths to indicate the theoretical complexity of the models when adopting a brute force approach. The simplest model (1) has 252 possible executions, while the most complex model (10) has 1.11E+107 possible executions.

The column *Dec.* contains the number of decomposed processes generated from the original process and *Time* contains the time required for their generation. Finally, *Check T.* contains the time required to solve all decomposed processes, and *Total Time* contains the total time required by the procedure to obtain a result concerning the compliance of the process. That is, it records the total time required for decomposition and evaluation of the decomposed processes.

Existing approaches, such as Regorous [12], use brute force, thereby evaluating all possible executions. Regorous is able to solve the first process and third process from Table 2 in 23 seconds and 47 seconds, respectively. For the other processes, we stopped Regorous after 10min, without being able to decide on the solution within the given time.

8 Conclusion

In this paper, we proposed an approach capable of efficiently verifying whether process models comprising concurrency are fully compliant with a set of obligations. Some of the key contributions of the proposed approach are the introduction of Δ-constraints, an alternative representation of the obligations used to specify the compliance requirements, and the ability to verify whether a business process model is fully compliant directly analysing its structure, without explicitly generating its executions. Compared to other approaches trying to solve the compliance problem through *brute force* or using *heuristics*, our proposed approach reduces the overall computational complexity of solving a sub-problem of the compliance problem by using a divide an conquer approach, while still steering clear from approximate solutions.

Although theoretically exponential in complexity (due to exclusive paths), we have empirically shown that the combined approach is capable of solving highly complex processes that are otherwise infeasible using existing brute force approaches. Even processes with more than 250 possible paths and 1.11E+107 possible executions were checked within 132ms.

However, the demonstrated performance gain does come at a tradeoff, as *brute force based* approaches are capable of solving more expressive instances of the problem than our approach, as we allow only literals and not formulae. While this limitation prevents us from compliance checking with full regulatory specifications, it can be successfully used for many aspects of structural compliance (i.e. conditions about the tasks appearing and their mutual relationships). Despite the structural limitations over the process model, we show how our solution can be combined with additional procedures in order to solve more generic problems.

As future work, we plan to improve the current solution in order to be able to resolve some of the current limitations. We reckon that a possibility to improve the current approach is to investigate how the Δ-constraints introduced in the solution can be generalised, and potentially reused in a more modular fashion to create efficient solutions for more generic sub-problems of business process compliance.

Acknowledgments

This research is supported by the science and industry endowment fund.

References

[1] Carlos E. Alchourrón, Peter Gärdenfors, and David Makinson. On the logic of theory change: Partial meet contraction and revision functions. *J. of Symbolic Logic*, 50(2):510–530, 1985.

[2] Ahmed Awad, Gero Decker, and Mathias Weske. Efficient compliance checking using bpmn-q and temporal logic. In *BPM 2008*, pages 326–341. Springer, 2008.

[3] Ahmed Awad, Matthias Weidlich, and Mathias Weske. Visually specifying compliance rules and explaining their violations for business processes. *J. Vis. Lang. Comput.*, 22(1):30–55, 2011.

[4] Domenico Bianculli, Carlo Ghezzi, and Paola Spoletini. A model checking approach to verify bpel4ws workflows. In *Service-Oriented Computing and Applications, 2007. SOCA'07. IEEE International Conference on*, pages 13–20, 2007.

[5] Philip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artif. Intell.*, 42(2-3):213–261, 1990.

[6] Silvano Colombo Tosatto, Guido Governatori, and Pierre Kelsen. Business process regulatory compliance is hard. *IEEE Transactions on Services Computing*, 8(6):958–970, 2015.

[7] Thomas Curran, Gerhard Keller, and Andrew Ladd. *SAP R/3 Business Blueprint: Understanding the Business Process Reference Model.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[8] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in bpmn. *Information and Software Technology*, 50(12):1281–1294, 2008.

[9] Amal Elgammal, Oktay Turetken, Willem-Jan van den Heuvel, and Mike Papazoglou. Formalizing and Applying Compliance Patterns for Business Process Compliance. *Software & Systems Modeling*, pages 1–28, 2014.

[10] Sven Feja, Andreas Speck, and Elke Pulvermüller. Business process verification. In *GI Jahrestagung*, pages 4037–4051, 2009.

[11] G. Governatori and A. Rotolo. How do agents comply with norms? In *2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*, volume 3, pages 488–491, Sept 2009.

[12] Guido Governatori. The Regorous approach to process compliance. In *2015 IEEE 19th International EDOC Workshop*, pages 33–40. IEEE Press, 2015.

[13] Guido Governatori and Mustafa Hashmi. No time for compliance. In Sylvain HallÃl' and Wolfgang Mayer, editors, *2015 IEEE 19th Enterprise Distibuted Object Computing Conference*, pages 9–18. IEEE, 2015.

[14] Guido Governatori, Jörg Hoffmann, Shazia Wasim Sadiq, and Ingo Weber. Detecting regulatory compliance for business process models through semantic annotations. In *Business Process Management Workshops*, volume 17 of *LNBIP*, pages 5–17. Springer, 2008.

[15] Guido Governatori and Antonino Rotolo. An algorithm for business process compliance. In Enrico Francesconi, Giovanni Sartor, and Daniela Tiscornia, editors, *The Twenty-First Annual Conference on Legal Knowledge and Information Systems*, volume 189 of *Frontieres in Artificial Intelligence and Applications*, pages 186–191. IOS Press, 2008.

[16] Heerko Groefsema, Nick R T P van Beest, and Marco Aiello. A formal model for compliance verification of service compositions. *IEEE Transactions on Services Computing*, 11:466 – 479, 2018.

[17] Mustafa Hashmi, Guido Governatori, Ho-Pun Lam, and Moe Thandar Wynn. Are we done with business process compliance: State-of-the-art and challenges ahead. *Know. and Inf. Sys.*, 2018.

[18] Mustafa Hashmi, Guido Governatori, and Moe Thandar Wynn. Normative requirements for regulatory compliance: An abstract formal framework. *Information Systems Frontiers*, 18(3):429–455, 2016.

[19] Ahmed Kheldoun, Kamel Barkaoui, and Malika Ioualalen. Specification and verification of complex business processes - a high-level petri net-based approach. In *Business Process Management*, volume 9253 of *LNCS*, pages 55–71. Springer International Publishing, 2015.

[20] Oussama M. Kherbouche, Adeel Ahmad, and Henri Basson. Using model checking to control the structural errors in bpmn models. In *Research Challenges in Information Science (RCIS), 2013 IEEE Seventh International Conference on*, pages 1–12, 2013.

[21] Bartek Kiepuszewski, Arthur H. M. ter Hofstede, and Christoph Bussler. On structured workflow modelling. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, pages 431–445, London, UK, UK, 2000. Springer-Verlag.

[22] Marcello La Rosa, Hajo A Reijers, Wil M P Van Der Aalst, Remco M Dijkman, Jan Mendling, Marlon Dumas, and Luciano García-Bañuelos. Apromore: An advanced process model repository. *Expert Systems with Applications*, 38(6):7029–7040, 2011.

[23] Timo Latvala and Keijo Heljanko. Coping with strong fairness. *Fundamenta Informaticae*, 43(1-4):175–193, 2000.

[24] Ying Liu, Samuel Müller, and Ke Xu. A static compliance-checking framework for business process models. *IBM Systems Journal*, 46:335–361, 2007.

[25] M. Pesic, H. Schonenberg, and Wil van der Aalst. DECLARE: Full Support for Loosely-Structured Processes. In *Procedings of 11th IEEE International Conference on Enterprise Distributed Object Computing (EDOC'07)*, pages 287–287, 2007.

[26] Artem Polyvyanyy, Luciano García-Bañuelos, and Marlon Dumas. Structuring acyclic process models. In *International Conference on Business Process Management*, pages 276–293. Springer, 2010.

[27] Elke Pulvermüller, Sven Feja, and Andreas Speck. Developer-friendly verification of process-based systems. *Knowl.-Based Syst.*, 23(7):667–676, 2010.

[28] Elham Ramezani, Dirk Fahland, and Wil MP van der Aalst. Supporting domain experts to select and configure precise compliance rules. In *International Conference on Business Process Management*, pages 498–512. Springer, 2013.

[29] Shazia Sadiq, Guido Governatori, and Kioumars Namiri. Modeling control objectives for business process compliance. In *International conference on business process management*, pages 149–164. Springer, 2007.

[30] van der Aalst Wil, Adriansyah Arya, and van Dongen Boudewijn. Replaying history on process models for conformance checking and performance analysis. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(2):182–192, 2012.

[31] Boudewijn F van Dongen, Wil M P Van der Aalst, and Henricus M W Verbeek. Verification of epcs: Using reduction rules and petri nets. In *International Conference on Advanced Information Systems Engineering*, pages 372–386. Springer, 2005.

Received 27 September 2018