Algorithms for Basic Compliance Problems

Silvano Colombo Tosatto[a] Marwane El Kharbili[a] Guido Governatori[b]
Pierre Kelsen[a] Qin Ma[a] Leendert van der Torre[a]

[a] *University of Luxembourg, Luxembourg*
[b] *NICTA, Australia*

Abstract

The present paper focus on the problems to verify compliance for global achievement and maintenance obligations. We first introduce the elements needed to identify and study this fragment of compliance, such as processes and obligations. Afterwards we define the procedures and the algorithms to efficiently deal with this fragment of the compliance problem. Both algorithms proposed in the paper belongs to the **P** complexity class.

1 Introduction

Compliance initiatives are becoming more and more important in enterprises with the increase of the number of regulatory frameworks explicitly requiring businesses to show compliance with them. Most compliance solutions are ad hoc solutions and typically are time consuming to implement and to maintain. A classification of compliance in both preventive and detective activities is proposed in [10]. Auditing is a typical example of a detective activity. Preventive solutions, on the other hand, consider the activities to be done to achieve business objectives and their interactions with and the impact on them of the obligations and prohibitions imposed on a business by a normative framework. The proposal in [7] advances a compliance-by-design methodology. The methodology is based on the use of business process models to describe the activities of an enterprise and to couple them with formal specifications of the regulatory frameworks regulating the business. Business process models describe the activities to be done, and the order in which the task can be executed. Several approaches to handle compliance and to formalize normative requirements, based on different logical formalisms have been proposed (see for example [5, 9, 3]).

The aim of this paper is not to propose yet another formalism for business process compliance, but to offer two efficient algorithms to deal with its most basic problems. In doing so, it allows scholars to reuse these basic algorithms in more complex frameworks thanks to their low computational complexity. In [2] it is shown that, in general, even for 'well behaved' classes of processes (i.e., structured processes), checking whether a process is compliant with a (formalised) legislation, is computationally hard. In the rest of paper we show how to use the abstract framework, to identify classes of compliance problems for which efficient solutions are possible.

The paper is structured as follows: Section 2 defines the compliance problem by introducing the definitions of processes and obligations. Section 3 describes the algorithms and their complexity. Section 4 concludes the paper.

2 Background

The scope of this paper is to decide wether a process is compliant with a given global obligation. In this section we formally describe processes and global obligations.

2.1 Processes

A process models a collection of methods to perform an activity. For instance an activity can be preparing coffee, a process modeling such activity would comprehend many ways to prepare coffee, such as using

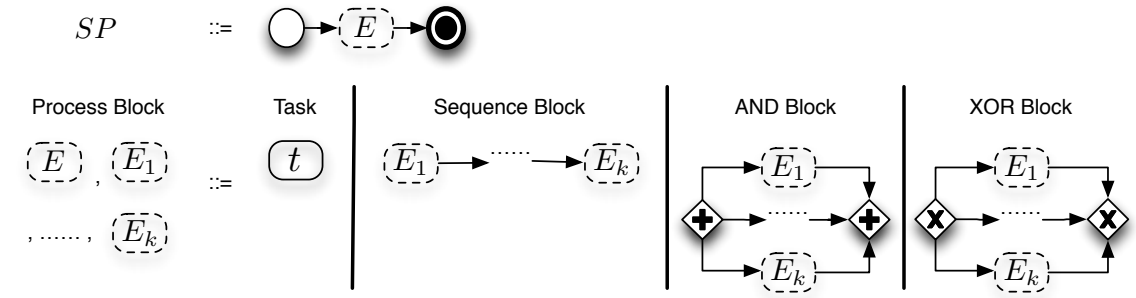
lyophilized coffee, brewing it, etc.

A process is composed of tasks and coordinators. Tasks represents the actions that can be done during the execution of a process. For instance considering the process that models how to prepare coffee, an action can be to heat the water. Coordinators are used to define the valid executions of a process. For instance coordinators can define that a certain task has to be done before another one or which tasks are mutually exclusive. Arrows connecting the elements of a process identify a general order in which the elements can be executed.

To represent a process we use a fragment of BPMN[1]. The fragment considered uses only AND and XOR coordinators in addition to start and end. The AND coordinator is used to coordinate tasks which can be concurrently executed. The XOR is used to define which tasks are mutually exclusive. AND and XOR coordinators consist in blocks of tasks within the process which are enclosed between two coordinators of the same type.

We consider only processes which do not contain cycles and are structured. A process is structured if it consists of hierarchically nested blocks as depicted by the following classification.

Definition 1 (Process) *A structured business process P is a business process generated by the following grammar given in the format of an graphical extension of BNF (with the vertical lines indicating alternative for the right hand side):*



The coordinator ◯ is called start and the coordinator ◉ is called end. The coordinator ✚ is called ANDsplit in case of multiple outgoing arrows and ANDjoin in case of multiple incoming arrows. A pair of ANDsplit and ANDjoin coordinators groups a set of sub-blocks indicating a logical relationship to activate all the sub-blocks concurrently. Finally, the coordinator ✖ is called XORsplit in case of multiple outgoing arrows and XORjoin in case of multiple incoming arrows. A pair of XORsplit and XORjoin coordinators groups a set of sub-blocks indicating a logical relationship to activate exactly one of the sub-blocks, chosen arbitrarily.

We assume that all the tasks in a structured business process carry a distinct identity that constitutes a key part of the label of a task. Therefore, a task (t) can directly be referenced by its label t. Similarly, (process) block identities are also distinct hence a block (E) can directly be referenced by its label E. As a consequence, for simplicity, we also allow a textual way to reference the graphical representation of structured business processes.

Example 1 *In Fig. 1 we provide an example of a process containing four tasks labeled $t_1, \ldots, t_4$. Within the process it is shown an* XOR *block containing in different branches the tasks t_1 and t_2. The* XOR *block is nested within an* AND *block, forming one of its branches and task t_3 forming the other one. The* AND *block is preceded by the* start *coordinator and followed by task t_4 which in turn is followed by the* end *coordinator.*

Given a process modeling an activity, an execution of such a process represents one way to perform it. An execution is a valid serialization of a subset of tasks composing the process. A serialization is considered valid if it starts from the start coordinator and terminates at the end. In addition a valid serialization has to comply with the semantics of the coordinators and the connections between the tasks.

A process is defined as $P = \text{start}\ E\ \text{end}$. An execution of P is equivalent to executing the block E within start and end. Thus we will provide the formal semantics for executing blocks which can be used for process execution as well.

[1] Business Process Model Notation, Version 2.0, `http://www.omg.org/spec/BPMN/2.0`

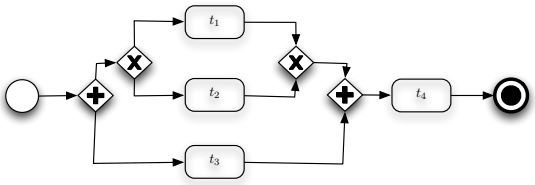


Figure 1: Example of a process

Definition 2 (Block Execution) *A process block E can be serialized into a set of finite sequences of tasks , written $\Sigma(E)$, defined by the following structural recursion. We call each sequence in $\Sigma(E)$ an execution of E, ranged over by ϵ.*

1. $E = t$: $\Sigma(E) = \{(t)\}$;
2. $E = \mathsf{SEQ}(E_1, \ldots, E_k)$: $\Sigma(E) = \{\epsilon_1; \ldots; \epsilon_k \mid \epsilon_1 \in \Sigma(E_1), \ldots, \epsilon_k \in \Sigma(E_k)\}$, *where* ; *stands for sequence concatenation.*
3. $E = \mathsf{XOR}(E_1, \ldots, E_k)$: $\Sigma(E) = \Sigma(E_1) \cup \ldots \cup \Sigma(E_k)$;
4. $E = \mathsf{AND}(E_1, \ldots, E_k)$: $\Sigma(E) = \{(t_1, \ldots, t_n)\}$ *such that*
   (a) $\forall i, 1 \leq i \leq k, \exists \epsilon_i \in \Sigma(E_i)$ *such that* $\{t_1, \ldots, t_n\} = \bigcup_{1 \leq i \leq k} \mathsf{Tasks}(\epsilon_i)$
   (b) $\forall E_i \in E = \mathsf{AND}(E_1, \ldots, E_k), t_h, t_j \in E_i | t_h < t_j \rightarrow \forall \epsilon \in \Sigma(E), t_h > t_j$.

   Namely $\Sigma(E)$ is the set of sequences each of which merges a sequence of $\Sigma(E_1)$, ..., and of $\Sigma(E_k)$. Merging a set of sequences gives rise to a sequence that includes all the elements of the operand sequences. Moreover, the ordering in the result sequence should be compatible with the ordering in the operand sequences.

In an arbitrary process and its possible executions. If the process is conform with Definition 1, then a task belonging to the process appears in at least one of its executions. This means that each task contained in a process has the possibility to be executed as stated in the following lemma.

Lemma 1 (Block Execution) *Given a process block E and a task t in E, $\exists \epsilon \in \Sigma(E)$ such that $t \in \epsilon$.*

Example 2 *Taking into account the process in Fig. 1 as $P =$ start E end. We have that $\Sigma(E) = \{\epsilon_1, \epsilon_2, \epsilon_3, \epsilon_4\}$ where $\epsilon_1 = (t_1, t_3, t_4), \epsilon_2 = (t_2, t_3, t_4), \epsilon_3 = (t_3, t_1, t_4)$ and $\epsilon_4 = (t_3, t_2, t_4)$. $\Sigma(E)$ contains the four possible executions of the process P. An execution not contained in $\Sigma(E)$, like $\epsilon_5 = (t_3, t_4, t_1)$, is not a valid execution of P. In this particular case one of the reasons why ϵ_5 is not a valid execution is because after t_4 the task t_1 is executed, which is not possible because t_1 belongs to an* XOR *block nested in an* AND *block that precedes the task t_4 in a sequence block.*

The state of the process changes while executing the tasks. We represent the state of a process as an incomplete consistent set of literals. Given a language, a set is called incomplete if there exists a literal of the language such that neither literal nor its complement belongs to the set.

Definition 3 (Consistent Literal Set) *Given a literal l, let $\tilde{l}$ be its complement. A set of literals L is* consistent *if and only if it does not contain l and $\tilde{l}$ at the same time for every literal $l \in L$.*

Example 3 *In a language of literals containing $\{\alpha, \beta, \gamma\}$, the following states: $L_1 = \{\alpha, \tilde{\beta}\}, L_2 = \{\tilde{\alpha}, \tilde{\beta}, \gamma\}, L_3 = \{\alpha, \tilde{\alpha}, \beta\}$. L_1 is an incomplete state because it does not contain either γ or its complement. L_1 is also consistent because it does not contain a literal and its complement. L_2 is a complete state because it contains all the literals or their complement belonging to the alphabet and L_3 in inconsistent because it contains both α and $\tilde{\alpha}$.*

Executing a task can change the current state of the process. Such changes depend on a consistent set of literals associated to the task being executed. We refer to a task with an associated set of literals as *annotated task*. The set of literals of an annotated task indicates the postconditions that have to hold after the task is executed. A process containing annotated tasks is called an *annotated process*.

Definition 4 (Annotated Process) *An annotated process is a pair: (P, ann), where P is a process and $\mathsf{ann} : T \rightarrow 2^{\mathcal{L}}$ is a function from the set T of P to consistent sets of literals of a language $\mathcal{L}$.*

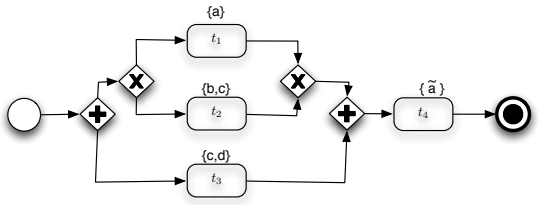


Figure 2: Example of an annotated process

Example 4 *Fig. 2 resumes the previous example shown in Fig. 1 by including annotations for its tasks. We can see that after executing task t_1, the literal a has to hold in the successive state. Annotations are not limited to single literals: tasks t_2 and t_3 are both annotated by multiple literals.*

The execution state of a process has to be kept consistent. This means that after the execution of an annotated task, the literals in the set associated to such task must hold in the state but the state has to be kept consistent. To allow such behavior, before updating the current state we remove from it the literals which could cause inconsistencies with the ones introduced by the task execution. After this step the state can be updated by including the literals in the set of the annotated task. Being the set of literals introduced consistent by definition, the result is still a consistent set.

Definition 5 (Literal set update) *Given two consistent sets of literals L_1 and L_2, the update of L_1 with L_2 is a set of literals defined as follows:*

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

Example 5 *This example shows how the state of a process is updated after executing a task. Given three sets of literals: $L_1 = \{a, b\}, L_2 = \{a, b, c\}$ and $L_3 = \{\tilde{c}\}$. In case of $L_1 \oplus L_3$ the result is the set $\{a, b, \tilde{c}\}$ which represent the union of L_1 and L_3. Differently if we consider $L_2 \oplus L_3$ the result is again $\{a, b, \tilde{c}\}$ but this time the result in not equivalent to $L_2 \cup L_3$ because L_3 contains $\tilde{c}$ that is the complement of one of the literals in L_2. The literal c is discarded from L_2 before joining it with L_3 so that the result is a consistent set. We can notice that $\oplus$ is not commutative because in the case $L_3 \oplus L_2$ the result would be $\{a, b, c\}$.*

During one of its possible executions, a process typically goes through several states. Each of these states can be associated to the execution of one of the annotated tasks belonging to the execution. We call a *trace* such sequence of states and tasks.

Definition 6 (Trace) *The trace θ corresponding to an execution $\epsilon = (t_1, \ldots, t_k)$ of an annotated process (P, ann) is a finite sequence of pairs of the form $((t_1, L_1), \ldots, (t_k, L_k))$, where $L_1, \ldots, L_k$ are sets of literals such that:*

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

We write $\Theta((P, \mathsf{ann}))$ to denote the set of traces of an annotated process, and let θ range over it.

Lemma 2 (Trace) *Traces and execution of a process are in one to one correspondence.*

Example 6 *This example shows the traces of the annotated process (P, ann) illustrated in Fig. 2. In the following table we show for each execution of P the corresponding trace. Each trace is represented as a sequence of pairs where every pair represents the task executed and the state holding after its execution.*

execution	*trace*
(t_1, t_3, t_4)	$((t_1, \{a\}), (t_3, \{a, c, d\}), (t_4, \{\tilde{a}, c, d\}))$
(t_2, t_3, t_4)	$((t_2, \{b, c\}), (t_3, \{b, c, d\}), (t_4, \{\tilde{a}, b, c, d\}))$
(t_3, t_1, t_4)	$((t_3, \{c, d\}), (t_1, \{a, c, d\}), (t_4, \{\tilde{a}, c, d\}))$
(t_3, t_2, t_4)	$((t_3, \{c, d\}), (t_2, \{b, c, d\}), (t_4.\{\tilde{a}, b, c, d\}))$

2.2 Obligations

A trace is said to be compliant if it respects a given *global obligation*. We use a subset of Process Compliance Logic (PCL) [6] to specify the global obligations.

A global obligation is an obligation that holds from the start till the end of a process. There are two different types of global obligations: achievement and maintenance.

Definition 7 (Global Obligations) *Given a literal l as the condition of a global obligation $\mathcal{O}$, we represent the two types of obligations as follows:*

$$\begin{array}{rcll} \mathcal{O} & ::= & O^a(l) & \textit{Achievement Obligation} \\ & | & O^m(l) & \textit{Maintenance Obligation} \end{array}$$

The condition of an achievement obligation, has to be verified in at least one state of the trace between the start and the end of a trace. An achievement obligation is violated if no state before the end of the trace satisfies the condition.

For maintenance obligations, every state between the start and the end of a trace has to fulfill the condition. A maintenance obligation is violated as soon as a state does not verify the condition.

Definition 8 (Global Obligation Fulfillment) *Given a global obligation $\mathcal{O}$ and a trace $\theta = ((t_1, L_1), \ldots, (t_k, L_k))$, θ fulfills $\mathcal{O}$ ($\theta \vdash \mathcal{O}$) iff:*

- $\mathcal{O} = O^a(l)$ $\theta \vdash O^a(l)$ *iff* $\exists L_i, 1 \leq i \leq k | l \in L_i$.
- $\mathcal{O} = O^m(l)$ $\theta \vdash O^m(l)$ *iff* $\forall L_i, 1 \leq i \leq k | l \in L_i$.

2.3 Process Compliance

Checking if a process is compliant with an obligation can return three different results. A process is fully compliant if every trace of the process is compliant with the obligation. A process is partially compliant if there exists a trace compliant with the obligation. If none of the traces of a process are compliant with the obligation, then the process is not compliant.

Definition 9 (Process Compliance) *Given an annotated process (P, ann) and a global obligation $\mathcal{O}$*

- ***Full compliance*** $(P, \mathsf{ann}) \vdash^F \mathcal{O}$ *iff* $\forall \theta \in \Theta((P, \mathsf{ann})), \theta \vdash \mathcal{O}$.
- ***Partial compliance*** $(P, \mathsf{ann}) \vdash^P \mathcal{O}$ *iff* $\exists \theta \in \Theta((P, \mathsf{ann})), \theta \vdash \mathcal{O}$.
- ***Not compliant*** $(P, \mathsf{ann}) \nvdash \mathcal{O}$ *iff* $\nexists \theta \in \Theta((P, \mathsf{ann})), \theta \vdash \mathcal{O}$.

3 Algorithms and Complexity

In this section we present the algorithms to verify the compliance of a process with respect to a global obligation. We design two algorithms, one for each type of global obligation.

3.1 Algorithm for Global Achievement Obligations

The algorithm for achievement obligations uses the function *Task Removal*. This function is used to remove a set of tasks from a process. By removing some tasks, the executions that contain that task are no longer allowed. In some cases by removing one or more tasks from a block it is possible that no executions remain available, if this is the case the function does not return a process block but $\perp$.

Definition 10 (Task Removal) *Given a process $P = \mathsf{start}\ E\ \mathsf{end}$ and a set of tasks T,* task removal *$R(E, T)$ returns either a new process block E' or $\perp$ as follows:*

1. $E = t$*: if* $t \in T$ *then return* $\perp$ *else return* E*;*
2. $E = \mathsf{SEQ}(E_1, \ldots, E_k)$*:*
   if $\exists i, 1 \leq i \leq k$ *such that* $R(E_i, T) = \perp$ ***then*** *return* $\perp$*,*
   else *return* $\mathsf{SEQ}(R(E_1, T), \ldots, R(E_k, T))$*;*
3. $E = \mathsf{XOR}(E_1, \ldots, E_k)$*:*
   if $\forall i, 1 \leq i \leq k$*,* $R(E_i, T) = \perp$ ***then*** *return* $\perp$*,*
   else if $\exists! i, 1 \leq i \leq k$ *such that* $R(E_i, T) \neq \perp$ ***then*** *return* $R(E_i, T)$*,*
   else *return* $\mathsf{XOR}(R(E_{m_1}, T), \ldots, R(E_{m_n}, T))$ *for all* $\forall E_{m_j} \in \{E_1, \ldots, E_k\} | R(E_{m_j}, T) \neq \perp$*;*

4. $E = \mathsf{AND}(E_1, \ldots, E_k)$:
   if $\exists i, 1 \leq i \leq k$ *such that* $R(E_i, T) = \bot$ ***then*** *return* $\bot$,
   else *return* $\mathsf{AND}(R(E_1, T), \ldots, R(E_k, T))$.

Lemma 3 (Task Removal) *Given a process block E and a set of tasks T in this block,*

1. $R(E, T) = \bot$ *iff* $\forall \epsilon_i \in \Sigma(E)$, $\epsilon_i \cap T \neq \emptyset$;
2. *otherwise,* $R(E, T) = E'$ *where:*
   (a) $\Sigma(E') \subseteq \Sigma(E)$;
   (b) $\forall \epsilon_i \in \Sigma(E)$, $\epsilon_i \cap T = \emptyset$ *iff* $\epsilon_i \in \Sigma(E')$, *and* $\forall \epsilon_i \in \Sigma(E')$, $\epsilon_i \cap T = \emptyset$.

In other words: E' contains exactly the traces of E that do not have tasks in T.

Algorithm 1 *Given an annotated process (P, ann) and a global achievement obligation $O^a(l)$, this algorithm returns whether (P, ann) is compliant with $O^a(l)$.*

Suppose $P = \mathsf{start}\ E\ \mathsf{end}$.*;*
if $\forall t$ *in* $E, l \notin \mathsf{ann}(t)$ ***then***
  return $(P, \mathsf{ann}) \not\vdash O^a(l)$*;*
else
  if $R(E, \{t \mid t \text{ is a task in } E \text{ and } l \in \mathsf{ann}(t)\}) = \bot$ ***then***
    return $(P, \mathsf{ann}) \vdash^F O^a(l)$*;*
  else
    return $(P, \mathsf{ann}) \vdash^P O^a(l)$*;*
  end if
end if

Due to the nature of an achievement obligation $O^a(l)$, it is satisfied when a task whose annotation contains the condition l of the obligation is executed. By removing all the tasks containing l in their annotations, the remaining executions are the ones which do not fulfill the obligation. If there are no possible executions remaining, this means that every execution has to go through at least a task having l annotated, thus the process is fully compliant with the obligation. In case there are no tasks having l in their annotation, then no execution can fulfill the obligation and the process is not compliant. At last if some tasks are removed and some possible executions remain, then the process is partially compliant.

Complexity of Algorithm 1: Assuming that the size of each annotation is $\mathbf{O}(1)$, i.e. independent of the number of tasks. The time of Algorithm 1 is dominated by the time for the task removal algorithm which is linear in the number of tasks of the process.

3.2 Algorithm for Global Maintenance Obligations

We first introduce the notion of *first tasks*, which are the set of tasks of a process that can be scheduled at the beginning of an execution.

Definition 11 (First Task(s)) *Given a process block E $\mathsf{First}(E)$, returns a set of tasks as follows:*

- $E = t$: $\{t\}$;
- $E = \mathsf{SEQ}(E_1, \ldots, E_k)$ *where* $k \geq 2$: $\mathsf{First}(E_1)$;
- $E = \mathsf{AND}(E_1, \ldots, E_k)$ *where* $k \geq 2$: $\bigcup_{i=1}^{k} \mathsf{First}(E_i)$;
- $E = \mathsf{XOR}(E_1, \ldots, E_k)$ *where* $k \geq 2$: $\bigcup_{i=1}^{k} \mathsf{First}(E_i)$.

Given a block E and a task $t \in \mathsf{First}(E)$, let X denote the set of executions in E that have t as the first task. The function *Task Rooting* returns a subset of the executions contained in X. In Lemma 4 we provide a sketch of a proof showing that the approximation considered by *Task Rooting* does not affect the result of checking compliance for maintenance obligations.

Definition 12 (Task Rooting) *Given a process block E and a task $t \in \mathsf{First}(E)$,* task rooting $F(E, t)$ *returns a new process block as follows:*

1. $E = t$: *return* E;
2. $E = \mathsf{SEQ}(E_1, \ldots, E_k)$: *return* $\mathsf{SEQ}(F(E_1, t), E_2, \ldots, E_k)$;
3. $E = \mathsf{XOR}(E_1, \ldots, E_k)$: *return* $F(E_p, t)$ *where* $E_p \in \{E_1, \ldots, E_k\}$ *and* $t \in E_p$;

4. $E = \mathsf{AND}(E_1, \ldots, E_k)$*: return* $\mathsf{SEQ}(F(E_p, t), \mathsf{AND}(E_{i_1}, \ldots, E_{i_{k-1}}))$*, where* $\{i_1, \ldots, i_{k-1}, p\} = \{1, \ldots, k\}$ *and* $t \in E_p$.

Lemma 4 (Task Rooting) *Let E be a block, t be a task such that $t \in \mathsf{First}(E)$ and, X be the set of executions of E that start with t. We denote with $\theta \in \Theta_X$ a trace associated to an $\epsilon \in X$ according to the annotation an* ann*:*

$$((\mathsf{start}, (F(E,t), \mathsf{ann}), \mathsf{end}) \vdash^{F} O^m(l)) \Leftrightarrow (\forall \theta \in \Theta_X, \theta \vdash O^m(l))$$
$$((\mathsf{start}, (F(E,t), \mathsf{ann})\mathsf{end}) \vdash^{P} O^m(l)) \Leftrightarrow (\exists \theta \in \Theta_X, \theta \vdash O^m(l))$$
$$((\mathsf{start}, (F(E,t), \mathsf{ann})\mathsf{end}) \nvdash O^m(l)) \Leftrightarrow (\forall \theta \in \Theta_X, \theta \nvdash O^m(l))$$

Proof 1 *Given a process P, we know that X contains all executions of P starting with an arbitrary task t. The function* task rooting *returns an approximation of the set X only in the case where t belongs to an* AND *block. The executions that are contained in the process block returned by* task rooting *are the one which have as a prefix the brach of the* AND *block starting with t. The executions lost by* task rooting *are the ones where some tasks from other branches in the* AND *block are interleaved with the ones belonging to the branch containing t. We can focus on the serialization of the* AND *block because it is where some executions are lost.*

We distinguish now two cases: $l \notin \mathsf{ann}(t)$ and $l \in \mathsf{ann}(t)$. In the first case both the executions in X and the ones given by task rooting *are not compliant according to Definition 8. In the second case we have to analyze the remainder tasks in the* AND *block.*

In case none of the remainder tasks annotates $\tilde{l}$, then we can safely that in both cases the AND *block is fully compliant with the maintenance obligation. In case where some of the tasks contain in their annotation $\tilde{l}$, we have to analyze two cases: the first where such tasks are not avoidable, i.e. these tasks are not within an* XOR *block, which means that l would stop holding due to the execution of one of these tasks, thus the* AND *block would be not compliant in this case both in X and* task rooting*, because at least one task containing $\tilde{l}$ had to be executed. In case such tasks are avoidable, thus both for X and* task rooting *exists at leas an execution which is compliant with the maintenance obligation, making the* AND *block partially compliant with the obligation.*

It is not necessary to analyze the part of the executions following the AND *block because no more approximations are involved, thus after having shown that the approximation on the* AND *block does not alter the result, we can say that the result obtainable by checking a maintenance obligation after applying* task rooting*, would be the same as checking it on the set X.*

Algorithm 2 *Given an annotated process (P, ann) and an atomic maintenance obligation $O^m(l)$, this algorithm returns whether (P, ann) is compliant with $O^m(l)$.*

1: *Suppose* $P = \mathsf{start}\ E\ \mathsf{end}$*;*
2: $res = (P, \mathsf{ann}) \nvdash O^m(l)$*;*
3: $T_F = \mathsf{First}(E)$*;*
4: $T_{\tilde{l}} = \{t \in P : \tilde{l} \in \mathsf{ann}(t)\}$*;*
5: ***if*** $\forall t$ *in* $T_F, l \in \mathsf{ann}(t)$ *and* $T_{\tilde{l}} == \emptyset$ ***then***
6: ***return*** $(P, \mathsf{ann}) \vdash^{F} O^m(l)$*;*
7: ***else***
8: ***for each*** $t \in T_F$ *such that* $l \in \mathsf{ann}(t)$ ***do***
9: ***if*** $R(F(E,t), T_{\tilde{l}}) \neq \perp$ ***then***
10: ***return*** $(P, \mathsf{ann}) \vdash^{P} O^a(l)$*;*
11: ***end if***
12: ***end for each***
13: ***return*** $(P, \mathsf{ann}) \nvdash O^m(l)$*;*
14: ***end if***

Algorithm 2 identifies the set of the tasks that can appear as first in the possible executions of the process in analysis. From such set the algorithm identifies which executions has the possibility to be compliant by starting with a task having l annotated. For each execution that can be compliant, *Task Removal* is used to verify that they don't contain a task with $\tilde{l}$ annotated.

Complexity of Algorithm 2: Both R and F can be computed in time $O(n)$ where n is the number of tasks in E. Thus the call to R on line 9 can also be computed in time $O(n)$. Assuming each annotation has size $O(1)$ we then see that the overall complexity is $O(n^2)$.

4 Conclusion

Business process compliance received increased attention in the field of business process modeling in the past few years. The majority of approaches propose some logics for compliance (e.g., deontic logic [5], linear temporal logic [11], clause based logic/logic programming [3, 4], extensions of BPMN languages [1]). However, to the best of our knowledge, this is a first systematic investigation on the complelxity of business process compliance. [6] provides a linear time algorithm to check whether a single trace is compliant, and [8] gives approximate solutions in linear time. [2] shows that the problem of checking whether a process is compliant or not is computationally infeasible. In this paper we have identified some tractable sub-problems and their solutions.

As future work we plan to identify further problems and provide solutions by integrating our algorithms in an *abstract framework* or designing new ones if needed.

References

[1] Ahmed Awad, Gero Decker, and Mathias Weske. Efficient compliance checking using bpmn-q and temporal logic. In Marlon Dumas, Manfred Reichert, and Ming-Chien Shan, editors, *BPM*, volume 5240 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2008.

[2] Silvano Colombo Tosatto, Guido Governatori, Pierre Kelsen, and Leendert van der Torre. Business process compliance is hard. Technical report, NICTA, 2012.

[3] Aditya Ghose and George Koliadis. Auditing business process compliance. In Bernd J. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *ICSOC*, volume 4749 of *Lecture Notes in Computer Science*, pages 169–180. Springer, 2007.

[4] Guido Governatori, Jörg Hoffmann, Shazia Wasim Sadiq, and Ingo Weber. Detecting regulatory compliance for business process models through semantic annotations. In Danilo Ardagna, Massimo Mecella, and Jian Yang, editors, *Business Process Management Workshops*, volume 17 of *Lecture Notes in Business Information Processing*, pages 5–17. Springer, 2008.

[5] Guido Governatori, Zoran Milosevic, and Shazia Sadiq. Compliance checking between business processes and business contracts. In Patrick C. K. Hung, editor, *10th International Enterprise Distributed Object Computing Conference (EDOC 2006)*, pages 221–232. IEEE, 2006.

[6] Guido Governatori and Antonino Rotolo. Norm compliance in business process modeling. In *Proceedings of the 4th International Web Rule Symposium: Research Based and Industry Focused (RuleML 2010)*, volume 6403 of *LNCS*, pages 194–209. Springer, 2010.

[7] Guido Governatori and Shazia Sadiq. The journey to business process compliance. In Jorge Cardoso and Wil van der Aalst, editors, *Handbook of Research on BPM*, chapter 20, pages 426–454. IGI Global, 2009.

[8] Jörg Hoffmann, Ingo Weber, and Guido Governatori. On compliance checking for clausal constraints in annotated process models. *Information Systems Frontiers*, 14(2):155–177, 2012.

[9] Dumitru Roman and Michael Kifer. Reasoning about the behaviour of semantic web services with concurrent transaction logic. In *VLDB*, pages 627–638, 2007.

[10] Shazia Sadiq and Guido Governatori. Managing regulatory compliance in business processes. In Jan van Brocke and Michael Rosemann, editors, *Handbook of Business Process Management*, volume 2, chapter 8, pages 157–173. Springer, Berlin, 2010.

[11] Wil M. P. van der Aalst, Maja Pesic, and Helen Schonenberg. Declarative workflows: Balancing between flexibility and support. *Computer Science - R&D*, 23(2):99–113, 2009.