Towards an Abstract Framework for Compliance

Silvano Colombo Tosatto*‡, Guido Governatori† and Pierre Kelsen*
*University of Luxembourg, Luxembourg
{silvano.colombotosatto, pierre.kelsen}@uni.lu
†NICTA, Australia
{guido.governatori}@nicta.com.au
‡Universitá di Torino, Italy

***Abstract*—The present paper aims at providing an abstract framework to define the regulatory compliance problem. In particular we show how the framework can be used to solve the problem of deciding whether a structured process is compliant with a single regulation, which is composed of a primary obligation and a chain of compensations.**

***Index Terms*—Compliance, Normative Reasoning, Contrary to Duty Obligations.**

I. 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. As a consequence IT support for compliance activities within enterprises is growing. Compliance is the set of initiatives in an organization to ensure that the core business activities and procedures are in agreement with specific normative frameworks. Most compliance solutions are ad hoc solutions and typically, implementation and maintenance are time consuming [1]. Sadiq and Governatori propose in [2] a classification of compliance activities into preventive and detective. Auditing is a typical example of a detective activity, where logs of already executed business activities are examined to discover if there were non-compliance issues, and based on the analysis of samples recommendations on measures to prevent reoccurrence of compliance breaches are proposed. 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 or regulation.

Preventive solutions become crucial in areas where compliance breaches may lead to critical failures of a system, or when the recovering from such breaches is costly. Banking is a typical example of such areas, where transactions that do not follow the enforced regulations have to be avoided.

The implementation of a particular regulation is conducted as part of a compliance initiative and involves the production of so-called compliance artifacts. These artifacts are used to represent compliance requirements and check them against the enterprise information systems. Usually, regulatory frameworks explicitly require compliance initiatives to provide proof of compliance. Governatori and Sadiq [3] propose a compliance-by-design methodology to address the compliance problem. 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 task (activities) to be done, and the order in which the task can be executed. In [4] it is argued that this is not enough to ensure that a process is compliant, because normative frameworks often specify requirements on other aspects (e.g., data, resources, timeframes, relations between different pieces of data and resources). The solution to obviate this problem is to extend business processes with semantic annotations. Tasks in a process are associated with sets of annotations providing information not typically available in a business process model. Several approaches to handle compliance and to formalize normative requirements, based on different logical formalisms have been proposed, see for example ([5], [6], [7], [8], [9]). Kharbili [10] gives a comparative analysis of a collection of solutions to business process compliance management.

From the above point of view, compliance can be understood as an additional correctness criterion for business processes. Verification of a business processes is a very well studied area (see for example the seminal paper by van der Aalst [11]). However, compliance adds complexity to the verification task: the structural correctness of structured processes can be verified in linear time [12], but [13] shows that, even for structured processes, checking whether a process is compliant with a (formalised) legislation, is computationally hard (i.e., checking whether there is at least one possible way to execute the process without violating the regulations is an NP-complete problem, and checking whether every execution is compliant is NP-hard). Accordingly, there is a need to identify heuristics of classes of tractable problems.

The aim of this paper is not to propose yet another formalism for business process compliance, but to offer an abstract framework capable to verifying whether a business process is compliant with a given regulation. A regulation consists of a primary obligation to which a chain of compensations can be associated. Compensations are obligations that have to be fulfilled in case the primary obligation is not. Such representation of a regulation follows the concept of contrary to duties [14] [15].

To achieve this goal we propose an abstract formalism for business process compliance. The advantages of an abstract formalism are that the framework highlights the crucial aspects

of compliance without worrying about the details of a specific formalism, i.e. the reasoning behind the obligations and how the business processes are constructed. An additional benefit of the abstract framework is that it allows scholars to classify the approaches to compliance based on the features identified by the abstract formalism. In this way one can study general properties of business process compliance.

The paper is structured as follows: Section 2 defines the abstract framework for compliance by introducing the definitions of processes and regulations. Section 3 describes a way to check compliance using the abstract framework for compliance. Section 4 concludes the paper.

II. Business Process Regulatory Compliance

In this section we first introduce the business process regulatory compliance problem and its elements: the business process model and the regulations.

A. Business Process

We consider a particular class of processes: the structured processes. Such a class of processes is limited in its expressivity because it does not allow cycles and its components have to be properly nested. An advantage of using structured processes is that their correctness can be verified in polynomial time as shown in [16] by Kiepuszewski et al. for structured workflow models.

Polyvyanyy et al. [17] report that 406 out of 604 of the reference models in the SAP collection are structured, and 19 of the remaining ones can be transformed into equivalent structured processes. Thus, there is evidence that structured processes still cover a substantial part of processes deployed in practice (about 60%).

We define our processes in a similar way as the workflows defined by Kiepuszewski et al.

Definition 1 (Process Block): Let $\mathcal{T}$ be the set of all tasks. A process block B is inductively defined as follows:

- $\forall t \in \mathcal{T}, t$ is a process block.
- Let $B_1, \ldots, B_n$ be process blocks:
  - A sequence block: $\mathsf{SEQ}(B_1, \ldots, B_n)$ is a process block.
  - An XOR block: $\mathsf{XOR}(B_1, \ldots, B_n)$ is a process block.
  - An AND block: $\mathsf{AND}(B_1, \ldots, B_n)$ is a process block.

A process block can be defined also as a directed acyclic graph: the nodes are called elements and the edges are called transitions.

Definition 2 (Process Block as a Directed Graph): A process block B can be represented as a directed acyclic graph $G = (V_B, E_B)$, where $V_B \subseteq \mathcal{T} \cup \mathcal{C}$, where $\mathcal{C}$ is a set of coordinators which are of one of the four types: xor_split, xor_join, and_split, and_join; and $E_B \subseteq V_B \times V_B$ the set of edges. The coordinators of type xor_split and xor_join (resp. and_split and and_join) are used to enclose XOR blocks (resp. AND blocks) in the graphical representation of a process block. Each process block B has an initial node and a final node denoted respectively by b_B and f_B.

V_B, E_B, b_B and f_B of a given process block B are defined inductively as follows:

- For a block containing a single task, $B = t$: $V_B = \{t\}$, $E_B = \emptyset$, $b_B = t$ and $f_B = t$
- Let $B_1, \ldots, B_n$ be process blocks:
  - For a sequence block $B = \mathsf{SEQ}(B_1, \ldots, B_n)$: $V_B = \bigcup_{i=1}^{n} V_{B_i}$ and $E_B = \bigcup_{i=1}^{n} E_{B_i} \cup \bigcup_{j=1}^{n-1} \{(f(B_j), b(B_{j+1}))\}$. $b_B = b_{B_1}$ and $f_B = f_{B_n}$.
  - For an XOR block $B = \mathsf{XOR}(B_1, \ldots, B_n)$: $V_B = \bigcup_{i=1}^{n} V_{B_i} \cup \{\mathsf{xsplit}, \mathsf{xjoin}\}$, where xsplit and xjoin are respectively of type xor_split and xor_join. $E_B = \bigcup_{i=1}^{n} (E_{B_i} \cup \{(\mathsf{xor\_split}, b_{B_i}), (f_{B_i}, \mathsf{xor\_join})\})$. $b_B = \mathsf{xsplit}$ and $f_{B_{XOR}} = \mathsf{xjoin}$.
  - For an AND block $B = \mathsf{AND}(B_1, \ldots, B_n)$: $V_B = \bigcup_{i=1}^{n} V_{B_i} \cup \{\mathsf{asplit}, \mathsf{ajoin}\}$, where asplit and ajoin are respectively of type and_split and and_join. $E_B = \bigcup_{i=1}^{n} (E(B_i) \cup \{(\mathsf{asplit}, b_{B_i}), (f_{B_i}, \mathsf{ajoin})\})$. $b_B = \mathsf{asplit}$ and $f_B = \mathsf{ajoin}$.

In the remainder of the paper we will mainly use Definition 1. However we also introduced Definition 2 and we will use it to graphically represent the business processes.

Definition 3 (Structured Process Model): A structured process P is a sequence block $\mathsf{SEQ}(\mathsf{start}, B, \mathsf{end})$, where start and end are two pseudo tasks, $\{\mathsf{start}, \mathsf{end}\} \cap \mathcal{T} = \emptyset$. Those pseudo-tasks are used to identify the beginning of a structured process model and its ending.

Structured process models can be graphically represented using *Business Process Model and Notation 2.0*[1]. The symbol ◯ is used to represent the start and ◉ to represent the end. The and_split and and_join coordinators are represented both by ✚. The and_split is identified by a single incoming transition and multiple outgoing transitions. The opposite is true for the and_join, which is identified by multiple incoming transitions and a single outgoing transition. In the same way, the operator ✖ identifies both xor_split and xor_join coordinators.

Example 1: Fig. 1 shows a structured process containing four tasks labelled $t_1, \ldots, t_4$. 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.

Considering the structured process in Fig. 1 as a sequence block, we can represent it as follows:

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

Structured processes exclude processes containing badly nested blocks (Fig. 2.(a)) and processes with loops (Fig. 2.(b)).

An execution of a structured process is a sequence of a subset of the tasks belonging to the process. A valid execution identifies a path from the start to the end of the process and follows the semantics of the coordinators and the transitions that are traversed.

[1] http://www.omg.org/spec/BPMN/2.0

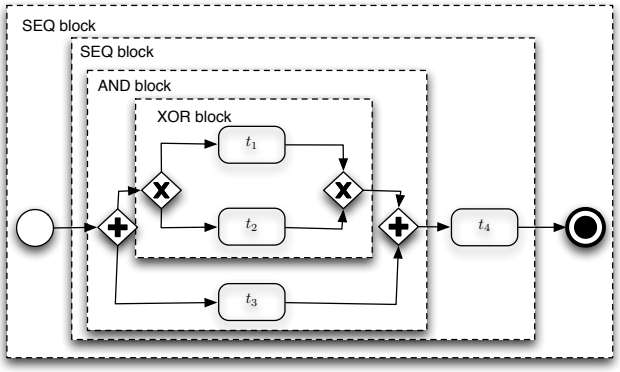

Fig. 1. A graphical representation of a structured process

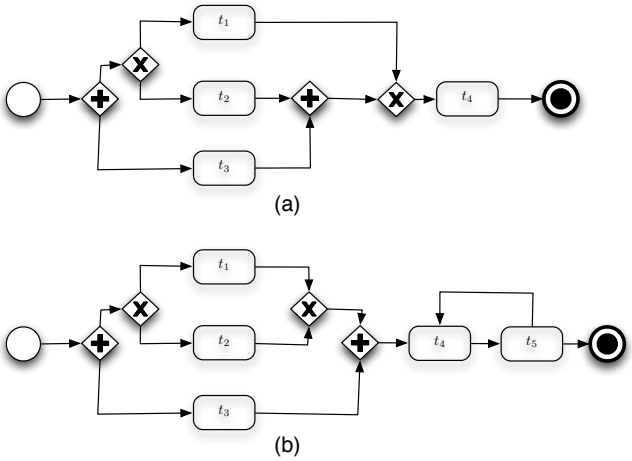

Fig. 2. Examples of non-structured processes

Definition 4 (Execution): Given a process block B, an execution of B, written ϵ, is a totally ordered set $(T_\epsilon, \prec_\epsilon)$, where $T_\epsilon \subseteq \mathcal{T}$. An execution ϵ is constructed from B as follows:

1) If $B = t$, for some $t \in \mathcal{T}$, then $\epsilon = (\{t\}, \emptyset)$.
2) If $B = \mathsf{SEQ}(B_1, \ldots, B_n)$, let $\epsilon_1, \ldots, \epsilon_n$ be executions of $B_1, \ldots, B_n$, then ϵ is the ordered sum of $\epsilon_1, \ldots, \epsilon_n$, written $\epsilon_1 + \cdots + \epsilon_n$.
3) If $B = \mathsf{XOR}(B_1, \ldots, B_n)$, then $\epsilon = \epsilon_i$ where ϵ_i is an execution of B_i for some $1 \leq i \leq n$.
4) If $B = \mathsf{AND}(B_1, \ldots, B_n)$, let $\epsilon_1, \ldots, \epsilon_n$ be executions of $B_1, \ldots, B_n$, then ϵ is a linear extension of $\bigcup_{i=1}^n \epsilon_i = (\bigcup_{i=1}^n T_{\epsilon_i}, \bigcup_{i=1}^n \prec_{\epsilon_i})$, i.e. a total ordering for all the tasks in the B_i's that is compatible with the ordering constraints in each of the ϵ_i.

We represent totally ordered sets as sequences and denote the set of possible executions of a process block B as $\Sigma(B)$.

If we consider structured processes to be process blocks as well, Definition 4 can be used to define the executions of either a whole process or just a part of it. Given a structured process P, each execution $\epsilon \in \Sigma(P)$ has the peculiarity that the pseudo task start is the minimal element and end is the maximal element.

If a structured process conforms with Definition 3, then a task belonging to such a structured process appears in at least one of its executions. This means that each task contained in a process has the possibility of being executed. In general, if a task belongs to a process block, there exists at least one execution of this block containing the task, as stated in the following lemma.

Lemma 1 (Execution): Given a process block B and a task t, if $t \in B$, then $\exists \epsilon \in \Sigma(B)$ such that $t \in \epsilon$.

***Proof** (Execution):*

Prove by structural induction on the process block B. ■

Example 2: Let us denote the structured process shown in Fig. 1 by P. We have that $\Sigma(P) = \{\epsilon_1, \epsilon_2, \epsilon_3, \epsilon_4\}$ where $\epsilon_1 = (\mathsf{start}, t_1, t_3, t_4, \mathsf{end}), \epsilon_2 = (\mathsf{start}, t_2, t_3, t_4, \mathsf{end}), \epsilon_3 = (\mathsf{start}, t_3, t_1, t_4, \mathsf{end})$ and $\epsilon_4 = (\mathsf{start}, t_3, t_2, t_4, \mathsf{end})$. Any other execution such as $\epsilon_5 = (\mathsf{start}, t_3, t_4, t_1, \mathsf{end})$, is not a valid execution of P. In this case ϵ_5 is not a valid execution of P because t_1 is executed after task t_4, which according to Definition 4 is not possible because t_1 appears before t_4 in the sequence block to which they belong.

The state of a process can evolve during the execution of the process. An annotated process is a process whose tasks are associated with consistent sets of literals. These sets of literals are called *annotations* [18] and indicate what has to hold after a task is executed.

Definition 5 (Consistent literal set): Let $\mathcal{L}$ be the set of all literals. A set of literals $L \subseteq \mathcal{L}$ is consistent if $\forall l \in \mathcal{L}, l \in L$ implies $\tilde{l} \notin L$, where $\tilde{l}$ denotes the negation of l[2].

We define an update operator (inspired by AGM belief revision [19]) to revise a set of literals with respect to another.

Definition 6 (Literal set update): 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:

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

Definition 7 (Annotation Function): An annotation function ann is a total function associating to each task in $\mathcal{T}$ a consistent set of literals, $\mathsf{ann} : \mathcal{T} \to 2^{\mathcal{L}}$. However, in the extensional representation of the function, i.e., defining ann by listing the pairs, only the meaningful ones are shown. This means that for all tasks t such that $\mathsf{ann}(t)$ does not appear in any of the listed pairs, then the associated set of literals is $\emptyset$.

An annotation function ann can be used to annotate a process block (B, ann) (according to Definition 3 a structured process P is also a process block).

Example 3: We can annotate the process P in Fig. 1 with the following function:

$$\mathsf{ann} = \{(t_1, \{a\}), (t_2, \{b, c\}), (t_3, \{c, d\}), (t_4, \{\neg a\})\}$$

The same can be visually represented as shown in Fig. 3.

[2]Let α be an atomic proposition. If $l = \alpha$, then $\tilde{l} = \neg\alpha$ and if $l = \neg\alpha$, then $\tilde{l} = \alpha$. This follows the double negation elimination rule of propositional logic.

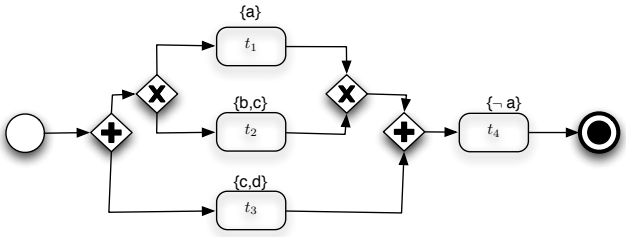

Fig. 3. An annotated process

Definition 8 (State): Given a task t and a consistent set of literals L, a state is a tuple $\sigma = (t, L)$ where L represents the set of literals holding after the execution of t.

A *trace* represents the change of states during an execution. It is composed of a sequence of states.

Definition 9 (Trace): Given a process block (B, ann) as context and one execution of B: $\epsilon = (t_1, \ldots, t_n)$, 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. 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 traces of a process block B given an annotation function ann.

Example 4: Table I shows the traces of the annotated process (P, ann) illustrated in Fig. 3. The first column contains the possible executions of P. The second column contains the corresponding traces.

B. Regulations

A regulation R is composed of a primary obligation followed by a chain of compensations $R = \mathbf{O} \otimes \lightning$, where $\mathbf{O}$ represents the primary obligation and $\lightning$ the chain of compensations. A chain of compensations consists of a sequence of secondary obligations $\lightning = \Omega_1 \otimes \cdots \otimes \Omega_n$. If the primary obligation is not fulfilled, then the first secondary obligation of the chain is enforced. The same applies when a secondary obligation of the chain is enforced but not fulfilled. The difference is that in such case the next secondary obligation of the chain is enforced.

We specify the primary and secondary obligations using part of Process Compliance Logic (PCL) [20].

Each obligation has a lifeline and a deadline. These elements define the activation period of the obligation. Notice that the lifelines and deadlines considered in the present paper are not related to temporal values, but to states in a trace fulfilling the condition given specified in the lifelines or the deadlines.

Once triggered by its lifeline, an obligation becomes active. If an obligation is already active, further triggers of its lifeline have no effect. Similarly when its deadline is triggered, an obligation is deactivated. Because we consider only traces containing a finite number of states, we assume that the last state of a trace (the one containing the pseudo task end) triggers every deadline.

We identify two types of obligations: achievement and maintenance. An achievement obligation has to verify a condition in at least one state within the activation period. A maintenance obligation has to verify a condition in each state within the activation period.

Definition 10 (Primary Obligation): Let l_c, l_b and l_d be literals. A primary obligation $\mathbf{O}$ is a triple $\mathbf{O} = \langle \mathcal{O}, l_b, l_d \rangle$ where l_b is the lifeline condition, l_d is the deadline condition and $\mathcal{O}$ is one of the following types where l_c is their fulfillment condition:

$$\begin{array}{rcll} \mathcal{O} & ::= & O^a(l_c) & \text{achievement} \\ & | & O^m(l_c) & \text{maintenance} \end{array}$$

Literals are satisfied in a state if and only if the literal set of the state contains them. Lifelines and deadlines activate or deactivate an obligation in the states that satisfy them. The activation period of an obligation is identified by all the states between the state where the lifeline is satisfied, excluded, and the state where the deadline is satisfied, included.

Achievement obligations prematurely terminate their activation period in the state where they are fulfilled. In contrast, for maintenance obligations, their activation period is prematurely terminated when a state does not fulfill the condition of the obligation.

Definition 11 (Fulfillment): Let $\sigma \models l$ iff $l \in L$, where L is the literal set of the state; and given a sequence of states $(\sigma_1, \ldots, \sigma_n)$, let $\sigma_i \prec \sigma_j$ iff $i < j$ (the same using respectively $\preceq$ and $\leq$). We also define the following exceptions: the state $\sigma_{\mathsf{end}} = (\mathsf{end}, L)$ containing the pseudo task end always satisfies $\sigma_{\mathsf{end}} \models l_d$ and $\sigma_{\mathsf{end}} \not\models l_b$ where l_b is a lifeline and l_d a deadline.

Given an obligation $\mathbf{O} = \langle \mathcal{O}, l_b, l_d \rangle$ and a trace θ, θ fulfills $\mathbf{O}$, written $\theta \vdash \mathbf{O}$, iff:

- $\mathcal{O} = O^a(l_c)$: $\theta \vdash \langle O^a(l)_c, l_b, l_d \rangle$ iff: $\forall \sigma_i \in \theta$ it holds that $\sigma_i \models l_b$ then $\exists \sigma_j \in \theta$ such that $\sigma_j \models l_c$ and $\sigma_j \succ \sigma_i$, and $\neg \exists \sigma_h \in \theta$ such that $\sigma_h \models l_d$ and $\sigma_i \prec \sigma_h \prec \sigma_j$.
- $\mathcal{O} = O^m(l_c)$: $\theta \vdash \langle O^m(l)_c, l_b, l_d \rangle$ iff: $\forall \sigma_i \in \theta$ it holds that $\sigma_i \models l_b$ then $\exists \sigma_h \in \theta$ such that $\sigma_h \models l_d$ and $\forall \sigma_j \in \theta$ such that $\sigma_j \models l_c$ and $\sigma_i \prec \sigma_j \preceq \sigma_h$.

Otherwise θ does not fulfill $\mathbf{O}$, written $\theta \nvdash \mathbf{O}$.

An alternative way of representing the activation period of an obligation is by using a finite state automaton. Fig. 4.(a) shows the automaton modeling the activation period of an achievement obligation. Fig. 4.(b) represents the automaton modeling the activation period of a maintenance obligation. We can notice that in both cases, an obligation becomes active only if is inactive and a state triggering the lifeline is found. Finding such state while the obligation is already active has no impact on the activation period of the obligation.

The activation period terminates when the obligation is fulfilled or when it is not possible to fulfill it in the successive states. The two automata are consistent with the semantics of Definition 11. Notice that in Fig. 4.(a), the two conditions on the transition from *Active* to *Not Fulfilled* have to be considered in conjunction (The same for the transition from *Active* to

$\Sigma(P)$	$\Theta(P, \mathsf{ann})$
$(\mathsf{start}, t_1, t_3, t_4, \mathsf{end})$	$((\mathsf{start}, \{\emptyset\}), (t_1, \{a\}), (t_3, \{a, c, d\}), (t_4, \{\neg a, c, d\}), (\mathsf{end}, \{\neg a, c, d\}))$
$(\mathsf{start}, t_2, t_3, t_4, \mathsf{end})$	$((\mathsf{start}, \{\emptyset\}), (t_2, \{b, c\}), (t_3, \{b, c, d\}), (t_4, \{\neg a, b, c, d\}), (\mathsf{end}, \{\neg a, b, c, d\}))$
$(\mathsf{start}, t_3, t_1, t_4, \mathsf{end})$	$((\mathsf{start}, \{\emptyset\}), (t_3, \{c, d\}), (t_1, \{a, c, d\}), (t_4, \{\neg a, c, d\}), (\mathsf{end}, \{\neg a, c, d\}))$
$(\mathsf{start}, t_3, t_2, t_4, \mathsf{end})$	$((\mathsf{start}, \{\emptyset\}), (t_3, \{c, d\}), (t_2, \{b, c, d\}), (t_4, \{\neg a, b, c, d\}), (\mathsf{end}, \{\neg a, b, c, d\}))$

TABLE I
PROCESS TRACES OF THE ANNOTATED PROCESS IN FIG. 3

Fulfilled in Fig. 4.(b)). Once the obligation is either fulfilled or not, the state of the automaton is brought back to *Inactive* thanks to the ϵ transitions. This represents that an obligation can be activated multiple times by a trace.

Given a trace and an obligation, the automaton in Fig. 4 are used to determine whether an obligation is active or inactive with respect to the states of the given trace. For this reason we avoid to represent any final state in the automaton since they are not meaningful.

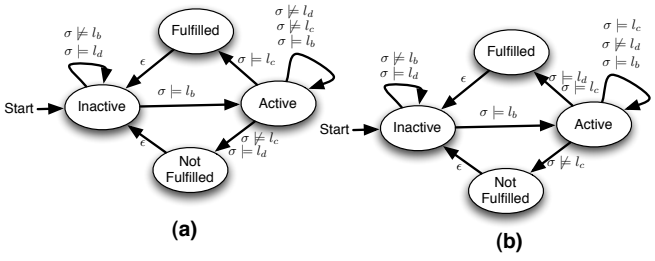

Fig. 4. Activation Periods using Finite State Automaton

Example 5 (Achievement): In a game of chess, moving a piece is an achievement obligation triggered by the opponent move. The deadline of such obligation can be considered as the time allowed for the player to make his move. Thus the player has to make his move before the allotted time expires.

Example 6 (Maintenance): While accessing secure data there is the obligation to have the proper credentials for the whole access period. The lifeline is when the access starts and the deadline when the access ends. For the whole time period of the access the credential must be retained.

1) Primary Obligation Compliance: Given a primary obligation, an annotated process can be fully compliant, partially compliant or not compliant with such primary obligation. An annotated process is fully compliant if every trace fulfills the obligation. It is partially compliant, if at least one trace fulfills the obligation. If none of the traces fulfill the obligation, then the annotated process is not compliant.

Definition 12 (Process Primary Obligation Compliance): Given an annotated process (P, ann) and an obligation $\mathbf{O}$.

- **Full Compliance** $(P, \mathsf{ann}) \overset{\mathrm{F}}{\vdash} \mathbf{O}$: if $\forall \theta \in \Theta(P, \mathsf{ann}), \theta \vdash \mathbf{O}$.
- **Partial Compliance** $(P, \mathsf{ann}) \overset{\mathrm{P}}{\vdash} \mathbf{O}$: if $\exists \theta \in \Theta(P, \mathsf{ann}), \theta \vdash \mathbf{O}$.
- **Not Compliant** $(P, \mathsf{ann}) \nvdash \mathbf{O}$: if $\neg\exists \theta \in \Theta(P, \mathsf{ann}), \theta \vdash \mathbf{O}$.

Full compliance implies partial compliance. This means that if a process is fully compliant with an obligation, then such process is also partially compliant with the same obligation.

2) Compensation Chains: Obligations are often described as soft constraints due to the possibility that they can be violated. Thus, as happens in the real world, it is useful to consider what should be done in case an obligation is violated.

Compensations are used to define the behavior that should be adopted in the case an obligation is violated. A compensation is associated to an obligation and it defines a secondary obligation that is activated when the one preceding it is violated. Secondary obligations are a particular type of obligations whose lifeline is the violation of the obligation they try to compensate.

Definition 13 (Violation): Given an obligation $\mathbf{O} = \langle \mathcal{O}, l_b, l_d \rangle$ and a trace θ. If $\theta \nvdash \mathbf{O}$, then $\exists \sigma_i \in \theta$ where $\mathbf{O}$ is violated. We use $\mathcal{V}_{\mathbf{O}}$ to indicate the set of states in θ that violate $\mathbf{O}$. $\mathcal{V}_{\mathbf{O}}$ is defined as follows:

In case of $\mathcal{O} = O^a(l_c)$, a state in $\mathcal{V}_{\mathcal{O}}$ written σ_h satisfies the following:

- $\sigma_h \models l_d$
- $\exists \sigma_i$ such that $\sigma_i \prec \sigma_h$ and $\sigma_i \models l_b$
- $\neg\exists \sigma_k$ such that $\sigma_i \prec \sigma_k \prec \sigma_h$ and $\sigma_k \models l_d$
- $\neg\exists \sigma_j$ such that $\sigma_i \prec \sigma_j \preceq \sigma_h$ and $\sigma_j \models l_c$

In case of $\mathcal{O} = O^m(l_c)$, a state in $\mathcal{V}_{\mathcal{O}}$ written σ_j satisfies the following:

- $\sigma_j \not\models l_c$
- $\exists \sigma_i$ such that $\sigma_i \prec \sigma_j$ and $\sigma_i \models l_b$
- $\neg\exists \sigma_h$ such that $\sigma_h \models l_d$ and $\sigma_i \prec \sigma_h \preceq \sigma_j$

Definition 14 (Compensation): A compensation, written Ω, is a tuple $\langle \mathcal{O}, l_d \rangle$ where $\mathcal{O}$ can be either $O^a(l_c)$ or $O^m(l_c)$, and l_d represents the deadline condition.

Because compensations are also obligations, it can also be that they are violated. It is possible then to associate a compensation to another compensation. This way of assigning compensations can create a sequence of compensations, where a compensation is activated when the previous is violated. We call such sequence of secondary obligations a chain of compensations.

Definition 15 (Compensation Chain): Given a primary obligation $\mathbf{O}$, $\lightning$ is the chain of compensations for $\mathbf{O}$, written $\mathbf{O} \otimes \lightning$. A chain of compensation is a sequence of compensations $\Omega_1 \otimes \cdots \otimes \Omega_n$ where each Ω_i aims to compensate the violations of the previous one in the chain.

In general we can assume that each $\mathbf{O}$ has a compensation chain associated, however this chain can be empty if $\mathbf{O}$ does not allow compensations.

Definition 16 (Compensation Lifeline): Given an obligation with a chain of compensations associated $\mathbf{O} \otimes \lightning$, where $\lightning$ is sequence of one or more secondary obligations $\Omega_1 \otimes \cdots \otimes \Omega_n$,

let $l_{\mathcal{V}_y}$ be a special literal where y refers to the previous obligation in the chain.

The first compensation of $\lightning$ is Ω_1 and can be rewritten as an obligation: $\mathbf{O}_{\Omega_1} = \langle \mathcal{O}, l_{V_{\mathbf{O}}}, l_d \rangle$. Each other compensation in the chain Ω_m can be rewritten as an obligation $\mathbf{O}_{\Omega_m} = \langle \mathcal{O}, l_{\mathcal{V}_{\Omega_{m-1}}}, l_d \rangle$.

Given a state $\sigma_i \in \theta$, $\sigma_i \models l_{\mathcal{V}_y}$ iff $\sigma_{i+1} \in \mathcal{V}_y$ as explained in Definition 13.

In Definition 16, the activation period of a compensation includes the state triggering it (the state violating the previous obligation of the chain), differently from the activation period of a primary obligation. We decided to adopt this approach to be able to handle situations where a task violating an obligation also fulfills the compensation associated with it.

Example 7: A classic example featuring these conditions is the *Gentle Murder*, introduced in [14], where the primary obligation is represented by the prohibition "don't kill" and the compensation by "if you kill, then do it gently". Without including the violating state into the activation period of the compensation, we would not be able to capture this type of compensations which relies on executing an action which indeed violates the obligation, but in such a way that it does represent an exception.

Definition 17 (Compensation Fulfillment): Given a trace θ and an obligation to which is associated a chain of compensations $\mathbf{O} \otimes \lightning$, where $\lightning = \Omega_1 \otimes \cdots \otimes \Omega_n$:

- **Fulfilling the Obligation** $\theta \vdash^{\text{O}} \mathbf{O} \otimes \lightning$: iff $\theta \vdash \mathbf{O}$
- **Fulfilling a Compensation** $\theta \vdash^{\text{C}} \mathbf{O} \otimes \lightning$: iff $\exists \Omega_i \in \lightning$, $\theta \vdash \mathbf{O}_{\Omega_i}$
- **Not Fulfilling** $\theta \nvdash \mathbf{O} \otimes \lightning$: iff $\neg\exists \Omega_i \in \lightning$, $\theta \vdash \mathbf{O}_{\Omega_i}$

From Definition 11 we know that an obligation whose lifeline is never fulfilled is considered to be fulfilled. Thus from this observation we can state that *fulfilling the obligation* implies *fulfilling a compensation*, because if the primary obligation is fulfilled, which is the case for *fulfilling the obligation*, then none of the secondary obligations are activated and each of them is considered to be fulfilled.

Definition 18 (Process Compensation Compliance): Given an annotated process (P, ann) and a primary obligation with a chain of compensations associated $\mathbf{O} \otimes \lightning$:

- **Full Obligation Compliance** $(P, \mathsf{ann}) \vdash^{\text{FO}} \mathbf{O} \otimes \lightning$: if $\forall \theta \in \Theta(P, \mathsf{ann}), \theta \vdash^{\text{O}} \mathbf{O} \otimes \lightning$.
- **Full Compensation Compliance** $(P, \mathsf{ann}) \vdash^{\text{FC}} \mathbf{O} \otimes \lightning$: if $\forall \theta \in \Theta(P, \mathsf{ann}), \theta \vdash^{\text{C}} \mathbf{O} \otimes \lightning$.
- **Partial Obligation Compliance** $(P, \mathsf{ann}) \vdash^{\text{PO}} \mathbf{O} \otimes \lightning$: if $\exists \theta \in \Theta(P, \mathsf{ann}), \theta \vdash^{\text{O}} \mathbf{O} \otimes \lightning$.
- **Partial Compensation Compliance** $(P, \mathsf{ann}) \vdash^{\text{PC}} \mathbf{O} \otimes \lightning$: if $\exists \theta \in \Theta(P, \mathsf{ann}), \theta \vdash^{\text{C}} \mathbf{O} \otimes \lightning$.
- **Not Compliant** $(P, \mathsf{ann}) \nvdash \mathbf{O} \otimes \lightning$: if $\neg\exists \theta \in \Theta(P, \mathsf{ann}), \theta \vdash^{\text{C}} \mathbf{O} \otimes \lightning$.

We can see from Definition 18 that full obligation compliance implies both full compensation compliance and partial obligation compliance. Both full compensation compliance and partial obligation compliance imply partial compensation compliance.

When a structured process is *partially obligation compliant* with a regulation, it can be also be the case that it is *fully compensation compliant*. However this is not always the case, otherwise *Partial Obligation Compliance* would have implied *Full Compensation Compliance*. Because of this, the abstract framework proposed in the following section explicitly distinguishes whether the structured process is only *partially obligation compliant* or also *fully compensation compliant*.

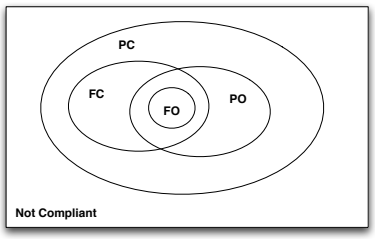

Fig. 5. Relations between the compliance results (Definition 18)

We represent the relations among the possible compliance results in Fig. 5 using *Venn Diagrams*. The set containing the partial compensation compliant results (PC) is disjoint from the set of not compliant results. The other sets are subsets of PC and the intersection of full compensation compliance (FC) and partial obligation compliance (PO) includes the set of full obligation compliant results (FO).

III. Checking Compliance

In this section we propose some algorithms and procedures to verify compliance using our framework. Given a structured process and a regulation, the proposed solution determines whether the structured process is compliant with the given regulation. We use as an algorithm a more detailed sequence of instructions. We define the procedures as interfaces, describing the properties of the output given the input.

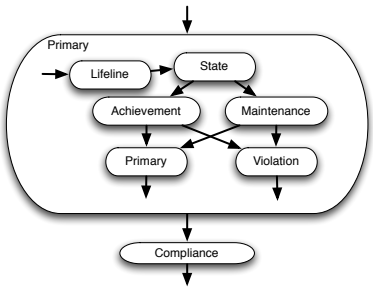

Fig. 6. I/O relations of the Procedures and Algorithms of the verification

In Fig. 6 we show the relations between the components. The input is processed by *Primary* and then *Compliance*. It can be noticed that within *Primary* other components process the input, and the whole *Primary* itself is recursive. For the sake

of simplicity we avoided to show the details inside *Violation* since it contains the same structure as *Primary*.

We define two new operators: the *executive union* $\cup_\epsilon$ and the *executive intersection* $\cap_\epsilon$. Both operations are binary. The executive union, written $B_1 \cup_\epsilon B_2 = B_3$, is defined as follows: $\Sigma(B_1) \cup \Sigma(B_2) = \Sigma(B_3)$. The executive intersection, written $B_1 \cap_\epsilon B_2 = B_3$, is defined as follows: $\Sigma(B_1) \cap \Sigma(B_2) = \Sigma(B_3)$. We also introduce an *empty block*, written $B_\emptyset$, and $\Sigma(B_\emptyset) = \emptyset$. When $\Sigma(B^{'}) \subseteq \Sigma(B)$, then $B^{'}$ is a sub-process block of B.

Algorithm 1:

$$Main(P, \mathsf{ann}, R)$$

1: $(B_{\mathbf{O}}, B_{\lightning}, B_{not}) = Primary((B, \emptyset), \mathbf{O}, B, B, B_\emptyset)$
2: **return** $Compliance(B, B_{\mathbf{O}}, B_{\lightning}, B_{not})$

The input is a structured process $P = \mathsf{SEQ}(\mathsf{start}, B, \mathsf{end})$ and a regulation $R = \mathbf{O} \otimes \lightning$. The regulation is composed of a primary obligation with a chain of compensations associated $\mathbf{O} \otimes \lightning$. The primary obligation is expressed as $\mathbf{O} = \langle \mathcal{O}, l_b, l_d \rangle$ and the chain of compensations $\lightning = \Omega_1 \otimes \cdots \otimes \Omega_n$. Each compensation in the chain is expressed as $\Omega = \langle \mathcal{O}, l_d \rangle$. We handle $\lightning$ as a vector, meaning that $\lightning[i]$ returns the i-th element in the chain. In case the index given to the vector is greater than the length of the chain or in case the chain is empty, $null$ is returned.

The verification procedure uses the algorithm *Primary*, which has the task to verify for which subsets of the possible traces the primary obligation is fulfilled. When a subset of traces does not fulfill the primary obligation, the algorithm takes care to pass such subset to the algorithm in charge to verify if the possible compensations are fulfilled in such subset. The result of *Primary* is later passed to *Compliance* which interprets it and decides the compliance class of P with respect to R.

Before describing *Primary* we introduce the four procedures it relies on: *Lifeline*, *Starting State*, *Achievement* and *Maintenance*.

Procedure 1 (Lifeline):

$$\mathcal{B}_{Life} = Life(B, l_b, \mathcal{X})$$

Output:

1) $\forall (B', t) \in \mathcal{B}_{Life}, l_b \in \mathsf{ann}(t)$
2) $\forall (B', t) \in \mathcal{B}_{Life}, \forall \epsilon \in \Sigma(B'), t \in \epsilon$ and $\neg \exists t_x \in \epsilon | t_x \in \mathcal{X}$ and $t_x \succ t$
3) $\forall (B', t) \in \mathcal{B}_{Life}, \forall \epsilon \in \Sigma(B'), \forall t_x \in \mathcal{X}, \neg \exists t_j \in \epsilon$ such that $l_b \in \mathsf{ann}(t_j)$ and $t_x \preceq t_j \prec t$

The procedure *Lifeline* takes as input a process block B, whose structure is consistent with Definition 1, a literal l_b and a set of tasks $\mathcal{X}$. The set $\mathcal{X}$ can contain a single task or can be empty. If $\mathcal{X}$ is not empty, the task contained represent up to which task of B the compliance has already been checked, otherwise it means that no part B has been evaluated yet.

The procedure returns a set of tuples $\mathcal{B}_{Life}$, each tuple belonging to the set is composed of a process block $B^{'}$ and a task t. Each B' is a sub-process block of B allowing only executions containing the task t. Each tuple represents a set of traces which always trigger the activation period of the obligation being analyzed in the state containing the task t. Independently from the task in B chosen to be t in a tuple of $\mathcal{B}_{Life}$, there exists at least an execution of B containing this task (Lemma 1).

The set $\mathcal{X}$ is used to avoid *livelocks*, it can contain a task indicating till where the fulfillment of an obligation has already been verified. Thus it avoids that the procedure *Lifeline* returns a tuple in the output which has already been considered.

Procedure 2 (Starting State):

$$((B_1, t), (B_2, t), (B_3, t), (B_4, t)) = State((B, t), l_c, l_d)$$

Output:

1) $\bigcup_{i=1}^{4} \Sigma(B_i) = \Sigma(B)$
2) $\forall \theta \in \Theta(B_i, \mathsf{ann}), \exists \sigma \in \theta | t \in \sigma$ and $cond$

B_1: $cond = \sigma \models l_c$ and $\sigma \models l_d$
B_2: $cond = \sigma \models l_c$ and $\sigma \not\models l_d$
B_3: $cond = \sigma \not\models l_c$ and $\sigma \models l_d$
B_4; $cond = \sigma \not\models l_c$ and $\sigma \not\models l_d$

The procedure *Starting State* takes as input a tuple (B, t), where B is a process block and t a task, plus two literals: l_c and l_d.

The output of this procedure is composed of four tuples (B_i, t). The union of the serializations of all the blocks in the tuples in the output has to contain the same executions as B. Each possible execution of B belongs to exactly one of the B_i of the output.

Each tuple (B_i, t) ensures a condition for the state containing t, for all of its possible traces. The condition ensured is different from all B_i.

The procedure *Starting State* divides a process block into sub-blocks according to which of two given literals are contained in the state containing t. Such division is necessary because the fulfillment of an obligation can depend on the state holding at the beginning of its activation period. Moreover, when verifying if a violation can be fulfilled or not, we need to consider only the traces which have brought about to the violation, hence the division according to the state holding at the beginning of the activation period.

Procedure 3 (Achievement):

$$\mathcal{B}_c, \mathcal{B}_v = A((B, t), l_c, l_d)$$

Output:

1) $\forall (B_c, \mathcal{X}) \in \mathcal{B}_c, \forall (B_v, v) \in \mathcal{B}_v, \bigcup \Sigma(B_c) \cup \bigcup \Sigma(B_v) = \Sigma(B)$ and $\bigcup \Sigma(B_c) \cap \bigcup \Sigma(B_v) = \emptyset$ and $\bigcap \Sigma(B_c) = \emptyset$ and $\bigcap \Sigma(B_v) = \emptyset$
2) $\forall (B_c, \mathcal{X}) \in \mathcal{B}_c, \forall \theta \in \Theta(B_c, \mathsf{ann}), \exists \sigma_k \in \theta | t \in \sigma_k, \exists \sigma_i \in \theta | \sigma_i \models l_c$ and $\neg \exists \sigma_j \in \theta | \sigma_j \models l_d$ and $\sigma_k \prec \sigma_j \prec \sigma_i$
3) $\forall (B_c, \mathcal{X}) \in \mathcal{B}_c, \forall \theta \in \Theta(B_c, \mathsf{ann}), \exists \sigma_i \in \theta | t \in \sigma_i, \exists \sigma_x \in \theta | t_x \in \sigma_x$ and $\sigma_x \models l_c$, $\neg \exists \sigma_j | \sigma_j \models l_c$ and $\sigma_i \prec \sigma_j \prec \sigma_x$
4) $\forall (B_c, \mathcal{X}) \in \mathcal{B}_c, \forall \theta \in \Theta(B_c, \mathsf{ann}), \exists \sigma_i \in \theta | t \in \sigma_i, \neg \exists \sigma_j \in \theta | \sigma_j \models l_c, \sigma_i \prec \sigma_j$ or $\exists \sigma_k \in \theta | \sigma_k \models l_d$ and $\sigma_i \prec \sigma_k \prec \sigma_j$
5) $\forall (B_v, v) \in \mathcal{B}_v, \forall \theta \in \Theta(B_v, \mathsf{ann}), \exists \sigma_i \in \theta | t \in \sigma_i, \exists \sigma_j \in \theta | v \in \sigma_j$, (either $\sigma_j \models l_d$ or $\neg \exists \sigma_h \in \theta | \sigma_h \succ \sigma_j$) and $\neg \exists \sigma_k \in \theta | \sigma_k \models l_d$ and $\sigma_i \prec \sigma_k \prec \sigma_j$

The procedure *Achievement* takes as input a tuple (B, t) containing a process block B and a task t, plus two literals: l_c and l_d. Each execution of B contains the task t.

The output of this procedure consists of two sets: $\mathcal{B}_c, \mathcal{B}_v$. The set $\mathcal{B}_c$ is composed of tuples: $(B_c, \mathcal{X})$, where B_c is a sub-process block of B and $\mathcal{X}$ a set containing a single task t_x. The set $\mathcal{B}_v$ is composed of tuples (B_v, v), where B_c is a sup-process block of B and v is a task.

Each element of $\mathcal{B}_c$ is a tuple $(B_c, \mathcal{X})$, the block B_c contains only traces where an achievement obligation $\langle O^a(l_c), l_b, l_d\rangle$ is satisfied in the activation period triggered by the state containing t. The set $\mathcal{X}$ contains a single task t_x, which belongs to the first state where the obligation is fulfilled in the activation period triggered by the state containing t. The task t_x also represents until which point the compliance of a block has already been checked.

Each element of $\mathcal{B}_v$ is a tuple (B_v, v), the block B_v allows only traces where an achievement obligation $\langle O^a(l_c), l_b, l_d\rangle$ is satisfied in the activation period triggered by the state containing t. The task v identifies where the obligation has been violated, namely the state terminating the activation period triggered by the state containing t. The task v also indicates from where a possible compensation has to be verified.

Procedure 4 ($Maintenance$):

$$\mathcal{B}_c, \mathcal{B}_v = M((B, t), l_c, l_d)$$

Output:

1) $\forall (B_c, \mathcal{X}) \in \mathcal{B}_c, \forall (B_v, v) \in \mathcal{B}_v, \bigcup \Sigma(B_c) \cup \bigcup \Sigma(B_v) = \Sigma(B)$ and $\bigcup \Sigma(B_c) \cap \bigcup \Sigma(B_v) = \emptyset$ and $\bigcap \Sigma(B_c) = \emptyset$ and $\bigcap \Sigma(B_v) = \emptyset$
2) $\forall (B_c, \mathcal{X}) \in \mathcal{B}_c, \forall \theta \in \Theta(B_c, \mathsf{ann}), \exists \sigma_i \in \theta | t \in \sigma_i$, either $(\exists \sigma_j \in \theta | \sigma_j \models l_d$ and $\sigma_j \succ \sigma_i$, $\forall \sigma_k \in \theta | \sigma_i \prec \sigma_k \prec \sigma_j, \sigma_k \models l_c)$ or $(\forall \sigma_k \in \theta | \sigma_i \prec \sigma_k, \sigma_k \models l_c)$
3) $\forall (B_c, \mathcal{X}) \in \mathcal{B}_c, \forall \theta \in \Theta(B_c, \mathsf{ann}), \exists \sigma_i \in \theta | t \in \sigma_i$, $\exists \sigma_j \in \theta | x \in \sigma_j$ and (either $\sigma_j \models l_d$ or $\neg \exists \sigma_h \in \theta | \sigma_h \succ \sigma_j$), and $\sigma_j > \sigma_i$, $\neg \exists \sigma_k \in \theta | \sigma_k \models l_d$ and $\sigma_i \prec \sigma_k \prec \sigma_j$
4) $\forall (B_v, v) \in \mathcal{B}_v, \forall \theta \in \Theta(B_v, \mathsf{ann}), \exists \sigma_i) \in \theta | t \in \sigma_i$, $\exists \sigma_j \in \theta | \sigma_j \not\models l_c$ and $\sigma_j \succ \sigma_i$, $\neg \exists \sigma_k \in \theta | \sigma_k \models l_d$ and $\sigma_i \prec \sigma_k \prec \sigma_j$
5) $\forall (B_v, v) \in \mathcal{B}_v, \forall \theta \in \Theta(B_v, \mathsf{ann}), \exists \sigma_i \in \theta | t \in \sigma_i, \exists \sigma_j \in \theta | v \in \sigma_j, \sigma_j \not\models l_c$ and $\sigma_j \succ \sigma_i$, $\neg \exists \sigma_k \in \theta | \sigma_k \not\models l_c$ and $\sigma_i \prec \sigma_k \prec \sigma_j$

The input and the output of the procedure *Maintenance* contain the same elements as the procedure *Achievement*. The output consists in two sets containing the same kind of tuples of the two sets constituting the output of the procedure *Achievement*. Each execution of B must appear in the set of possible executions of only one of the blocks within the tuples of $\mathcal{B}_c$ or the tuples of $\mathcal{B}_v$.

Each element of $\mathcal{B}_c$ is a tuple $(B_c, \mathcal{X})$ where B_c is a sub-process block of B and $\mathcal{X}$ is a set containing a single task x. The block B_c allows only traces where a maintenance obligation $\langle O^m(l_c), l_b, l_d\rangle$ is satisfied in the activation period triggered by the state containing t. The set $\mathcal{X}$ contains a single task t_x, which belongs to the state deactivating the activation period of the obligation. As per Procedure 3, the task t_x also represents until which point the compliance of a block has already been checked.

Each element of $\mathcal{B}_v$ is a tuple (B_v, v), where B_v is a sub-process block ob B and v a task belonging to each execution of B_v. Each B_v contains only traces where a maintenance obligation $\langle O^m(l_c), l_b, l_d\rangle$ is not fulfilled in the activation period triggered by the state containing t. In each tuple, the task v identifies where the maintenance obligation is violated, namely in the state of the trace containing it. It also indicates from where a possible compensation has to be verified.

Algorithm 2 (Primary):

$$(B_{\mathfrak{O}}{}', B_{\lightning}{}', B_{not}{}') = Primary((B, \mathcal{X}), \mathfrak{O}, B_{\mathfrak{O}}, B_{\lightning}, B_{not})$$

1: $\mathcal{B}_{life} = Life(B, l_b, \mathcal{X})$
2: **for all** $(B_{life}, t) : \mathcal{B}_{life}$ **do**
3: $((B_1, t), (B_2, t), (B_3, t), (B_4, t)) = State((B_{life}, t), l_c, l_d)$
4: **for** $1 \leq i \leq 4$ **do**
5: **if** $\mathcal{O} = O^a(l_c)$ **then**
6: $\mathcal{B}_c, \mathcal{B}_v = A((B_i, t), l_c, l_d)$
7: **else**
8: $\mathcal{B}_c, \mathcal{B}_v = M((B_i, t), l_c, l_d)$
9: **end if**
10: $\forall (B_c, \mathcal{X}) \in \mathcal{B}_c$: $B_{\mathfrak{O}} = B_{\mathfrak{O}} \cap_\epsilon (\bigcup_\epsilon B_c)$
11: **for all**$(B_c, \mathcal{X}) : \mathcal{B}_c$ **do**
12: $(B_{\mathfrak{O}}{}', B_{\lightning}{}', B_{not}{}') =$ $Primary((B, \mathcal{X}), \mathfrak{O}, B_{\mathfrak{O}}, B_{\lightning}, B_{not})$
13: $B_{\mathfrak{O}} = B_{\mathfrak{O}} \cap_\epsilon B_{\mathfrak{O}}{}'$
14: $B_{\lightning} = B_{\lightning} \cap_\epsilon B_{\lightning}{}'$
15: $B_{not} = B_{not} \cup_\epsilon B_{not}{}'$
16: **end for all**
17: **for all**$(B_v, v) : \mathcal{B}_v$ **do**
18: $(B_{\mathfrak{O}}{}'', B_{\lightning}{}'', B_{not}{}'') =$ $Violation((B_v, v), \mathfrak{O}, \lightning[1], B_{\mathfrak{O}}, B_{\lightning}, B_{not})$
19: $B_{\mathfrak{O}} = B_{\mathfrak{O}} \cap_\epsilon B_{\mathfrak{O}}{}''$
20: $B_{\lightning} = B_{\lightning} \cap_\epsilon B_{\lightning}{}''$
21: $B_{not} = B_{not} \cup_\epsilon B_{not}{}''$
22: **end for all**
23: **end for**
24: **end for all**
25: **return** $(B_{\mathfrak{O}}, B_{\lightning}, B_{not})$

The algorithm *Primary* takes as input a process block B, a primary obligation $\mathfrak{O}$ and three process blocks: $B_{\mathfrak{O}}, B_{\lightning}, B_{not}$. The output of the algorithm is composed of three process blocks: $B_{\mathfrak{O}}{}', B_{\lightning}{}', B_{not}{}'$ which respectively contains the traces compliant with the primary obligation, the traces compliant with the compensations and the traces not compliant with the regulation.

As per *Primary*, before describing the algorithm *Violation* we introduce the procedure *Violation Lifeline* it relies on.

Procedure 5 (Violation Lifeline):

$$\mathcal{B}_{life} = VLife(B, v)$$

Output:

1) $\forall (B', t) \in \mathcal{B}_{life}, \forall \epsilon \in \Sigma(B'), \exists v \in \epsilon, \exists t \in \epsilon | t \prec v$ and $\neg \exists k \in \epsilon | t \prec k \prec v$

The procedure *Violation Lifeline* takes as input a process block B and a task v belonging to B. Each execution of B contains v.

According to Definition 16, the state where a violation occurs has to be included in the activation period of the compensation. Similarly to the procedure *Lifeline*, the procedure *Violation Lifeline* returns a set of tuples $(B^{'}, t)$ where t is a task which has the possibility to be executed just before v and $B^{'}$ always contains in its executions the sequence (t, v). By building the tuples in this way, it is possible to reuse the achievement and maintenance procedure already defined to verify the primary obligation. Because these procedures consider t as the state triggering the activation period and as a result v always belongs to the first state of the activation period.

Algorithm 3 (Violation):

$(B_{\mathfrak{O}}{}', B_{\lightning}{}', B_{not}{}') = Violation((B_v, v), \mathfrak{O}, \lightning[i], B_{\mathfrak{O}}, B_{\lightning}, B_{not})$

1: **if** $\lightning[i] = null$ **then**
2: $B_{not} = B_{not} \cup_{\epsilon} B_v$
3: **else**
4: $\mathcal{B}_{life} = VLife(B_v, v)$
5: **for all**$(B_{life}, t) : \mathcal{B}_{life}$ **do**
6: $((B_1, t), (B_2, t), (B_3, t), (B_4, t))=$ $State((B_{life}, t), l_c, l_d)$
7: **for** $1 \leq i \leq 4$ **do**
8: **if** $\mathcal{O} = O^a(l_c)$ **then**
9: $\mathcal{B}_c, \mathcal{B}_v = A((B_i, t), l_c, l_d)$
10: **else**
11: $\mathcal{B}_c, \mathcal{B}_v = M((B_i, t), l_c, l_d)$
12: **end if**
13: **for all**$(B_c, \mathcal{X}) : \mathcal{B}_c$ **do**
14: $(B_{\mathfrak{O}}{}', B_{\lightning}{}', B_{not}{}') =$ $Primary((B, \mathcal{X}), \mathfrak{O}, B_{\mathfrak{O}}, B_{\lightning}, B_{not})$
15: $B_{\mathfrak{O}} = B_{\mathfrak{O}} \cap_{\epsilon} B_{\mathfrak{O}}{}'$
16: $B_{\lightning} = B_{\lightning} \cap_{\epsilon} B_{\lightning}{}'$
17: $B_{not} = B_{not} \cup_{\epsilon} B_{not}{}'$
18: **end for all**
19: **for all**$(B_v, v) : \mathcal{B}_v$ **do**
20: $(B_{\mathfrak{O}}{}'', B_{\lightning}{}'', B_{not}{}'') =$ $Violation((B_v, v), \mathfrak{O}, \lightning[i+1], B_{\mathfrak{O}}, B_{\lightning}, B_{not})$
21: $B_{\mathfrak{O}} = B_{\mathfrak{O}} \cap_{\epsilon} B_{\mathfrak{O}}{}''$
22: $B_{\lightning} = B_{\lightning} \cap_{\epsilon} B_{\lightning}{}''$
23: $B_{not} = B_{not} \cup_{\epsilon} B_{not}{}''$
24: **end for all**
25: **end for**
26: **end for all**
27: **end if**
28: **return** $(B_{\mathfrak{O}}, B_{\lightning}, B_{not})$

The algorithm *Violation* takes as input a tuple (B_v, v) where B_v is a block and v is a task, a primary obligation $\mathfrak{O}$, a compensation Ω_i (the i^{th} element of the vector $\lightning[i]$) and three blocks: $B_{\mathfrak{O}}, B_{\lightning}, B_{not}$. The output of the algorithm is a tuple composed of three blocks, where each of the elements respectively contains the traces compliant with the primary obligation, the traces compliant with the compensations and the traces not compliant with the regulation.

Algorithm 4 (Compliance):

$$Compliance(B, B_{\mathfrak{O}}, B_{\lightning}, B_{not})$$

The algorithm *Compliance* takes as input four process blocks: B the original process block of P, $B_{\mathfrak{O}}$ contains all the traces of P compliant with the primary obligation of R, $B_{\lightning}$ contains all the traces compliant with the compensations of R and B_{not} contains all the traces not compliant with R.

The algorithm returns a compliance result for the structured process with respect to the regulation given as input to the procedure as follows:

1: **if** $B_{\mathfrak{O}} = B$ **then**
2: **return** $(P, \mathsf{ann}) \vdash^{\text{FO}} R$
3: **else**
4: **if** $B_{\mathfrak{O}} = \emptyset$ **then**
5: **if** $B_{\lightning} = B$ **then**
6: **return** $(P, \mathsf{ann}) \vdash^{\text{FC}} R$
7: **else**
8: **if** $B_{\lightning} = \emptyset$ **then**
9: **return** $(P, \mathsf{ann}) \nvdash R$
10: **else**
11: **return** $(P, \mathsf{ann}) \vdash^{\text{PC}} R$
12: **end if**
13: **end if**
14: **else**
15: **if** $B_{\lightning} = B$ **then**
16: **return** $(P, \mathsf{ann}) \vdash^{\text{PO}} R$ and $(P, \mathsf{ann}) \vdash^{\text{FC}} R$
17: **else**
18: **return** $(P, \mathsf{ann}) \vdash^{\text{PO}} R$
19: **end if**
20: **end if**
21: **end if**

The procedure allows to determine the compliance class of a process with respect to a regulation. In case a process needs to be restructured to avoid violations, the input of the algorithm *Compliance* can be used to identify the problematic executions.

A. Complexity

The number of traces contained in a structured process can be exponential in the size of the process itself. Our verification procedure avoids to analyze the problem trace by trace (*brute force* approach) when possible. The procedure deals with sub-blocks of the process containing traces with similar properties. However, in the case where all the traces contained in a process do not share similar properties (worst case), then the procedure solves the problem analyzing each of the traces separately.

The details of Algorithms 1 and 2 allow to notice the existence of various branching points inside the procedure. These branching points occur when different traces need a different type of analysis. One of the branching points of the procedure is identified by the procedure *State*, which divides a process block into four sub-process blocks depending whether the fulfillment condition and the deadline condition hold at the beginning of the activation period of an obligation. By pointing out these differences, we can define the procedure to verify achievement and maintenance obligations in a similar way as the ones described in [21] which have the advantage to be computable in polynomial time.

IV. Conclusion and Related Work

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 [22], clause based logic/logic programming [8], [18], extensions of BPMN languages [23]).

To the best of our knowledge this is the first approach to propose an abstract framework capable of capturing the general features of the business process regulatory compliance problem. In the current paper we do not argue about the complexity of the solution proposed, apart from a brief discussion at the end of Section 3, however the problem tackled in this paper intersects the one which is proven to be NP-complete [13] and extends it with the concept of compensations.

Other works like [20] and [24] provide solutions in linear time, however the first verifies the compliance of a trace, instead of a process, and the second provides an approximate solution.

Linear Temporal Logic and model checking are very powerful techniques for the verification of different type of systems, and it can be used for the verification of business processes and some aspects of compliance [22]. However, the complexity of linear temporal logic is NP-complete for the language including the F (sometimes in the future) operator and PSPACE-complete for the extensions with F, X (next), U(until) operators. In addition, while it is tempting to represent the deontic operators (obligations) in temporal logic, temporal logic is not fully appropriate for the task [25], since it is not able to capture in a natural way the different nuances required by normative reasoning, for example, the representation on norms that can be violated but for which compensation are possible.

As future work we plan to extend the current framework, allowing it to capture more general business process regulatory compliance problems. As a consequence we aim to show how some of other existing approaches can be considered instances of the abstract framework proposed, allowing in this way to compare them. Finally, we also aim to use the abstract framework proposed to systematically analyze the complexity of the problem itself.

ACKNOWLEDGMENT

- NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.
- Silvano Colombo Tosatto is supported by the National Research Fund, Luxembourg.

REFERENCES

[1] M. El Kharbili, S. Stein, I. Markovic, and E. Pulvermueller, "Towards a framework for semantic business process compliance management," in *Proceedings of the GRCIS workshop.*, ser. CEUR Workshop Proceedings, S. Shazia, I. Marta, and z. M. Michael, Eds., Montpellier, France, June 17th 2008, pp. 1–15.

[2] S. Sadiq and G. Governatori, "Managing regulatory compliance in business processes," in *Handbook of Business Process Management*, J. van Brocke and M. Rosemann, Eds. Berlin: Springer, 2010, vol. 2, ch. 8, pp. 157–173.

[3] G. Governatori and S. Sadiq, "The journey to business process compliance," in *Handbook of Research on BPM*, J. Cardoso and W. van der Aalst, Eds. IGI Global, 2009, ch. 20, pp. 426–454.

[4] S. Sadiq, G. Governatori, and K. Naimiri, "Modelling of control objectives for business process compliance," in *BPM 2007*, ser. Lecture Notes in Computer Science, G. Alonso, P. Dadam, and M. Rosemann, Eds., no. 4714. Berlin: Springer, 2007, pp. 149–164.

[5] G. Governatori, Z. Milosevic, and S. Sadiq, "Compliance checking between business processes and business contracts," in *10th International Enterprise Distributed Object Computing Conference (EDOC 2006)*, P. C. K. Hung, Ed. IEEE, 2006, pp. 221–232.

[6] S. Goedertier and J. Vanthienen, "Designing compliant business processes with obligations and permissions," in *Business Process Management (BPM) Workshops*, 2006, pp. 5–14.

[7] D. Roman and M. Kifer, "Reasoning about the behaviour of semantic web services with concurrent transaction logic," in *VLDB*, 2007, pp. 627–638.

[8] A. Ghose and G. Koliadis, "Auditing business process compliance," in *ICSOC*, ser. Lecture Notes in Computer Science, B. J. Krämer, K.-J. Lin, and P. Narasimhan, Eds., vol. 4749. Springer, 2007, pp. 169–180.

[9] A. Awad, R. Goré, J. Thomson, and M. Weidlic, "An iterative approach for business process template synthesis from compliance rules," in *CAISE 2011*. Springer, 2011.

[10] M. Kharbili, "Business process regulatory compliance management solution frameworks: A comparative evaluation," in *Asia-Pacific Conference on Conceptual Modelling (APCCM 2012)*, ser. CRPIT, A. Ghose and F. Ferrarotti, Eds., vol. 130. Melbourne, Australia: ACS, 2012, pp. 23–32. [Online]. Available: http://crpit.com/confpapers/CRPITV130Kharbili.pdf

[11] W. M. P. van der Aalst, "The Application of Petri Nets to Workflow Management," *Journal of Circuits Systems and Computers*, vol. 8, no. 1, pp. 21–66, 1998.

[12] ——, "Formalization and verification of event-driven process chains," *Information and Software technology*, vol. 41, no. 10, pp. 639–650, 1999.

[13] S. Colombo Tosatto, G. Governatori, P. Kelsen, and L. van der Torre, "Business process compliance is hard," NICTA, Tech. Rep., 2012.

[14] H. Prakken and M. Sergot, "Dyadic deontic logic and contrary-to-duty obligations," 1997.

[15] A. Jones and J. Carmo, "Deontic logic and contrary-to-duties," in *Handbook of Philosophical Logic*, D. Gabbay and F. Guenthner, Eds. Kluwer Academic Publishers, 2002, pp. 265–343.

[16] B. Kiepuszewski, A. H. M. t. Hofstede, and C. Bussler, "On structured workflow modelling," in *Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, ser. CAiSE '00. London, UK, UK: Springer-Verlag, 2000, pp. 431–445. [Online]. Available: http://dl.acm.org/citation.cfm?id=646088.679917

[17] A. Polyvyanyy, L. García-Bañuelos, and M. Dumas, "Structuring acyclic process models," *Inf. Syst.*, vol. 37, no. 6, pp. 518–538, Sep. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.is.2011.10.005

[18] G. Governatori, J. Hoffmann, S. W. Sadiq, and I. Weber, "Detecting regulatory compliance for business process models through semantic annotations," in *Business Process Management Workshops*, ser. Lecture Notes in Business Information Processing, D. Ardagna, M. Mecella, and J. Yang, Eds., vol. 17. Springer, 2008, pp. 5–17.

[19] C. E. Alchourrón, P. Gärdenfors, and D. Makinson, "On the logic of theory change: Partial meet contraction and revision functions," *Journal of Symbolic Logic*, vol. 50, no. 2, pp. 510–530, 1985.

[20] G. Governatori and A. Rotolo, "Norm compliance in business process modeling," in *Proceedings of the 4th International Web Rule Symposium: Research Based and Industry Focused (RuleML 2010)*, ser. LNCS, vol. 6403. Springer, 2010, pp. 194–209.

[21] S. Colombo Tosatto, M. El Kharbili, G. Governatori, P. Kelsen, Q. Ma, and L. van der Torre, "Algorithms for basic compliance problems," in *Benelux conference on Artificial Intelligence*, Array, Ed., 2012, pp. 67–74.

[22] W. M. P. van der Aalst, M. Pesic, and H. Schonenberg, "Declarative workflows: Balancing between flexibility and support," *Computer Science - R&D*, vol. 23, no. 2, pp. 99–113, 2009.

[23] A. Awad, G. Decker, and M. Weske, "Efficient compliance checking using bpmn-q and temporal logic," in *BPM*, ser. Lecture Notes in Computer Science, M. Dumas, M. Reichert, and M.-C. Shan, Eds., vol. 5240. Springer, 2008, pp. 326–341.

[24] J. Hoffmann, I. Weber, and G. Governatori, "On compliance checking for clausal constraints in annotated process models," *Information Systems Frontiers*, vol. 14, no. 2, pp. 155–177, 2012.

[25] R. H. Thomason, "Deontic logic as founded on tense logic," in *New studies in deontic logic: norms, actions, and the foundations of ethics*, R. Hilpinen, Ed. Dordrecht, Holland: D. Reidel Publishing Company, 1981, pp. 165–176.