Norm Compliance in Business Process Modeling

Guido Governatori[1] and Antonino Rotolo[2]

[1] NICTA Queensland Research Lab., Australia, guido.governatori@nicta.com.au
[2] CIRSFID, University of Bologna, Italy, antonino.rotolo@unibo.it

Abstract. We investigate the concept of norm compliance in business process modeling. In particular we propose an extension of Formal Contract Logic (FCL), a combination of defeasible logic and a logic of violation, with a richer deontic language capable of capture many different facets of normative requirements. The resulting logic, called Process Compliance Logic (PCL), is able to capture both semantic compliance and structural compliance. This paper focuses on structural compliance, that is we show how PCL can capture obligations concerning the structure of a business process.

1 Introduction

Recent works in business process modeling focus on the concept of norm compliance (see the literature in Section 6). Norm compliance is aimed at ensuring that business processes are in accordance with a prescribed set of norms. More specifically by norm compliance we understand a relationship between two sets of specifications describing the alignment of formal specifications for business processes and formal specifications relevant law and regulations. In other terms compliance is the certification that a process is executed correctly does not result in a breach of the rules governing it. Compliance requirements may stem from legislation and regulatory bodies, standards and codes of practice, and business partner contracts. However, some research issues are still underdeveloped. We focus here on three of them, which are related to the three sources of complexities.

A *first source of complexities* resides in the fact that norms often regulate processes by specifying obligatory actions to be taken in case of breaches of some of the norms, actions which can vary from penalties to the termination of an interaction itself. Obligations in force after some other obligations have been violated correspond to contrary-to-duty obligations (CTDs) [1]. Among them, we have the reparative obligations, which are meant to 'repair' or 'compensate' violations of primary obligations [2]. These constructions identify situations that are not ideal but still acceptable. The ability to deal with violations is an essential requirement for processes where some failures can occur, but they do not necessarily mean that the whole process has to fail. However, these constructions can give rise to very complex rule dependencies, because we can have that the violation of a single rule can activate other (reparative) rules, which, in case of their violation, refer to other rules, and so forth [3].

A *second source of complexities* depends on the fact that processes may be regulated by different types of obligations (see Section 2). We may have obligations requiring (1) to be always fulfilled during the execution of the entire process or of some subpaths of

M. Dean at al. (eds)
RuleML 2010, The 4th International Web Rule Symposium
LNCS 6403, pp. 194–209
© Springer 2010.
The original publication is available at www.springerlink.com.

it, (2) that a certain condition must occur at least once before the execution of a certain task *A* of the process and such that the obligations may, or may not, persist after *A* if they are not complied with, (3) that something is done in a single task [4]. These types of obligation make things more complex when we deal with the compliance of a process with respect to chains of reparative obligations. For example, if the primary obligation is persistent and states to pay before task A, and the secondary (reparative) obligation is to pay a fine in the task B successive to A, the process is compliant not only when we pay before A, but also when we do not meet this deadline, pay later and pay the fine at B. If the secondary obligation rather requires to be always fulfilled for all tasks successive to A, compliance conditions will change.

The *third source of complexities* arises from different types of conditions we have for business processes. We can have normative requirements about the artifacts of a business process, over the activities (tasks) to be performed and over the order on which they are executed, as well as their combinations.

Most of the approaches to business process compliance address only one of these aspects. We propose an approach able to capture compliance requirements through a generic requirements modeling framework, and subsequently facilitate the propagation of these requirements into business process models and enterprise applications, thus achieving *compliance by design*. To achieve this objective we show how to use the language and the algorithm we have proposed in [5] to capture normative conditions on the tasks of a process.

Ensuring automated detection and/or enforcement of compliance requires in this paper to address the following related research tasks. *First*, we have to define in Section 3 a language to represent, and reason about, chains of reparative obligations of the types discussed in Section 2. *Second*, we need a mechanism for normalising a system of norms, namely, identify formal loopholes, deadlocks and inconsistencies in it, and to make hidden conditions explicit; without this, we do not have any guarantee that a given process is compliant, because we do not know if all relevant norms have been considered (Section 3). *Third*, we have to specify a suitable language for business process modeling able to automate and optimise business procedures and to embed normative constraints (Section 4).

2 Normative Constraints: Violations and Types of Obligation

We can distinguish *achievement obligations* from *maintenance obligations* [4]. For an *achievement obligation*, a certain condition must occur at least once before a deadline:

Example 1. Customers must pay before the delivery of the good, after receiving the invoice.

The deadline (before the delivery of the good)—which of course meaningfully applies if the customer is informed about the the maximum timespan within which the good can be delivered—refers to an obligation triggered by receipt of the invoice: such an obligation is persistent. After that the customer is obliged to pay. The obligation terminates only when it is complied with. Note that the obligation persists after the deadline, until it is achieved. But we may have cases where achievement obligations do not persist after the deadline:

Example 2. Once the submissions to RuleML 2010 are made available to RuleML-2010 PC members, the reviewers must send their reports before the notifications are delivered to the authors

Indeed, the obligation to deliver a review does not persist after the deadline, since after the review result has been notified to the authors, the paper has been accepted or rejected on the basis of the other reports delivered in time.

For *maintenance obligations*, a certain condition must obtain during all instants before the deadline:

Example 3. After opening a bank account, customers must keep a positive balance until bank charges are taken out.

By definition, maintenance obligations do not persist after the deadline. In Example 3, the deadline only signals that the obligation is terminated. A violation occurs when the obliged state does not obtain at some point before the deadline.

Finally, *puctual obligations* only apply to single tasks or instants:

Example 4. When banks proceed with any wire transfer, they must transmit a message, via SWIFT, to the receiving bank requesting that the payment is made according to the instructions given.

Punctual obligations apply only to single instants or tasks; mathematically they can be thought as either maintenance obligations or achievement obligations in force in time intervals where the endpoints are equal. Typically punctual obligations must occur at the same time of their triggering conditions, as shown in the above example.

Many norms can be associated with an explicit sanction. Consider

Example 5. Customers must pay before the delivery of the good, after receiving the invoice. Otherwise, an additional fine must be paid.

Example 6. After opening a bank account, customers must keep a positive balance until bank charges are taken out. Otherwise, their account is blocked.

An explicit sanction is often implemented through a separate obligation, which is triggered by a detected violation. Thus, further deadlines can be introduced to enforce the sanctions, leading to a chain of obligations. For instance, the payment of a fine mentioned in Example 5 could be due before the execution of a subsequent task.

We can also distinguish *preemptive obligations* from *non-preemptive obligations*. Suppose that, in Example 1, the price is 200$, and the customer, by mistake, transferred an amount of 200$ to the bank account of the seller before the date of the invoice. In this case, the early transfer may count as a payment and the customer could claim that her obligation to pay the seller is already fulfilled. This is an example of *preemptive* obligation. *Non-preemptive obligations* do not work as above. Consider this example:

Example 7. Executors and administrators of a decedent's estate will be required to give notice to each beneficiary named in the Will within 60 days after the date X of an order admitting a will to probate has been signed.

If an executor gives a notice to the beneficiaries before X, she will have to resend the notification after that. Note that the distinction between preemptive and non-premptive obligations applies only to achievement obligations, while it does not make sense with the maintenance and punctual ones.

What happens if the above types of obligations are combined into chains of reparative obligations? The expression of violation conditions and the reparations is an important requirement for designing subsequent processes to minimise or deal with such violations and also to determine the compliance of a process with the relevant norms. The violation expression consists of the primary obligation, its violation conditions, an obligation generated upon the violation condition occurs, and this can recursively be iterated, until the final condition is reached. We introduced in [3,6] the non-boolean connective $\otimes$: a formula like $OA \otimes OB$ means that A is obligatory, but if the obligation OA is not fulfilled (i.e., when $\neg A$ is the case), then the obligation OB is activated and becomes in force until it is satisfied or violated. However, the violation condition of an obligation varies depending on the types of obligations used. In the next section, we will extend the approach of [3,6] to cover these cases.

3 Process Compliance Language (PCL)

We now provide a formal account of the ideas presented above. Our formalism, called Process Compliance Language (PCL), is a combination of Defeasible Logic (DL) [7] and a deontic logic of violations [6]. PCL significantly extends the logic of [3] with types of obligations discussed in Section 2 and preserves the linear complexity of DL.

PCL formal language consists of a numerable set of propositional letters $p, q, r, \ldots$, intended to represent the state variables and the tasks of a process. Formulas are constructed using the negation $\neg$, the non-boolean connective $\otimes$ (for the reparative operator), and the deontic operators O^x_y, for obligation (where y can be empty). Based on the discussion in Section 2 we have three main classes of deontic operators: punctual obligations (O^p), maintenance obligations (O^m) and achievement obligations (O^a); achievement obligations in turn can be classified based on two orthogonal distinctions: persistent ($O^{a,\pi}$) vs non-persistent ($O^{a,\tau}$), and preemptive ($O^{a,x}_{pr}$) vs non-preemptive ($O^{a,x}_{n-pr}$).

The formulas of PCL are constructed in two steps according to the following formation rules: (i) every propositional letter is a literal; (ii) the negation of a literal is a literal; (iii) if X is a deontic operator and l is a literal then Xl and $\neg Xl$ are deontic literals.

After we have defined the notions of literal and deontic literal we can use the following set of formation rules to introduce $\otimes$-expressions, i.e., the formulas used to encode chains of obligations and violations: (a) every deontic literal is an $\otimes$-expression; (b) if $Xl_1, \ldots, Xl_n$ are deontic literals, then $Xl_1 \otimes \ldots \otimes Xl_n$ is an $\otimes$-expression.

The connective $\otimes$ permits combining primary and reparative obligations into unique regulations. The meaning of an expression like $O^{a,\pi}_{pr} A \otimes O^p B \otimes O^m C$ is that the primary provision is an achievement, persistent, preemptive obligation to do A, but if A is not done, then we have a punctual obligation to do B. If B fails to be realised, then we obtain a maintenance obligation to do C. Thus B is the reparation of the violation of the obligation $O^{a,\pi}_{pr} A$. Similarly C is the reparation of the obligation $O^p B$, which is in force when the violation of A occurs.

Each norm is represented by a rule in PCL like $r : A_1, \ldots, A_n \Rightarrow C$, where r is the id of the norm, $A_1, \ldots, A_n$ is the set of the premises of the rule, and C is the conclusion of the rule. Each A_i is either a literal or a deontic literal and C is an $\otimes$-expression.

PCL is also equipped with another type of rules, called defeaters (marked with arrow $\rightsquigarrow$) and a superiority relation (a binary relation) over the rule set.

In DL, the superiority relation ($\prec$) determines the relative strength of two rules, and it is used when rules have potentially conflicting conclusions. For example, given the rules $r_1 : a \Rightarrow O^m b \otimes O^{a,\pi}_{n-pr} c$ and $r_2 : d \Rightarrow \neg O^{a,\pi}_{pr} c$, $r_1 \prec r_2$ means that rule r_1 prevails over rule r_2 in situations where both fire and they are in conflict.

Defeaters play a peculiar role, as they cannot lead to any conclusion but are used to defeat some rules by producing evidence to the contrary. Thus, defeaters are suitable to model the termination of the persistence of obligations [8]. Consider Example 5:

$$\text{inv}_{init} : \text{invoice} \Rightarrow O^{a,\pi}_{pr}\text{pay} \otimes O^{p}\text{pay\_fine} \qquad \text{inv}_{term} : \text{pay} \rightsquigarrow \neg O^{a,\pi}_{pr}\text{pay}$$

Here, compliance is the only condition that terminates the obligation to pay: if not complied with, the obligation in fact persists beyond the deadline (we have still to pay), so failing to meet the deadline is used to signal a violation and trigger a sanction.

Normal Forms We introduce transformations of a PCL representation of a normative system to produce a normal form of the same (NPCL). The purpose of a normal form is to "clean up" the PCL representation of a normative system, to identify formal properties, e.g., loopholes, inconsistencies, ..., and to make hidden conditions explicit. We first describe a mechanism, based on [6], to derive new conditions by merging together existing normative clauses. Then, we examine the problem of redundancies, and we give a condition to identify and remove redundancies from the formal normative specification. Finally, we discuss how to solve possible conflicts between deontic provisions.

Merging Norms One of the features of the logic of violations is to take two rules, or norms, and merge them into a new clause.

Consider a norm like (Γ and Δ are sets of premises) $\Gamma \Rightarrow O^m A$. If we have that the violation of $O^m A$ is part of the premises of another norm, for example, $\Delta, \neg A \Rightarrow O^p C$, then the latter must be a good candidate as reparative obligation of the former:

$$\frac{\Gamma \Rightarrow O^m A \qquad \Delta, \neg A \Rightarrow O^p C}{\Gamma, \Delta \Rightarrow O^m A \otimes O^p C}$$

This reads as follows: given two policies such that one is a conditional obligation ($\Gamma \Rightarrow O^m A$) and the antecedent of second contains the negation of the propositional content of the consequent of the first ($\Delta, \neg A \Rightarrow O^p C$), then the latter is a reparative obligation of the former. Their interplay makes them two related norms so that they cannot be viewed anymore as independent. Therefore we can combine them to obtain an expression (i.e., $\Gamma, \Delta \Rightarrow O^m A \otimes O^p C$) that exhibits the *explicit reparative obligation* of the second norm with respect to the first.

Let X, Y, Z be deontic operators. The following is the general rule for merging norms based on [6,2]:

$$\frac{\Gamma \Rightarrow Xa \otimes (\bigotimes_{i=1}^{n} Yb_i) \otimes Zc \qquad \Delta, \neg b_1, \ldots, \neg b_n \Rightarrow Zd}{\Gamma, \Delta \Rightarrow Xa \otimes (\bigotimes_{i=1}^{n} Yb_i) \otimes Zd} \tag{1}$$

Removing Redundancies It is possible to combine rules in slightly different ways, and in some cases the meaning of the rules resulting from such operations is already covered by other rules. In other cases the rules resulting from the merging operation are generalisations of the rules used to produce them, consequently, the original rules are no longer needed in the specifications. To deal with this issue we introduce the notion of subsumption between rules. A rule subsumes a second rule when the behaviour of the second rule is implied by the first rule. For example, let us consider the rules

$$r : \mathit{Invoice} \Rightarrow O^{a,\pi}_{pr}\mathit{Pay7Days} \otimes O^{p}\mathit{PayInterest} \qquad r' : \mathit{Invoice}, \neg \mathit{Pay7Days} \Rightarrow O^{a,\pi}_{n-pr}\mathit{PayInterest}.$$

The first rule says that after the seller sends the invoice the buyer has the achievement, persistent and preemptive obligation to pay within one week, otherwise immediately after the violation the buyer has to pay the principal plus the interest. Thus we have the primary obligation $O^{a,\pi}_{pr}\mathit{Pay7Days}$, whose violation is repaired by the secondary obligation $O^{p}\mathit{PayInterest}$. According to the second rule, given the same set of circumstances *Invoice* and $\neg$*Pay7Days* we have the achievement, persistent and non-preemptive obligation $O^{a,\pi}_{n-pr}\mathit{PayInterest}$. However, (a) the primary obligation of r' obtains when we have a violation of the primary obligation of r; (b) after the obligation $O^{a,\pi}_{pr}\mathit{Pay7Days}$ is violated, complying with the secondary obligation $O^{p}\mathit{PayInterest}$ of r entails complying with the primary obligation $O^{a,\pi}_{n-pr}\mathit{PayInterest}$ of r' (but not vice versa); (c) hence, r is more general than r', and so the latter can be discarded.

In what follows, Definition 4 characterizes subsumption (which refers to Definitions 1, 2, and 3 to establish when the compliance conditions for an $\otimes$-expression cover the compliance conditions of another $\otimes$-expression).

Definition 1. *Let* $X, Y \in \{O^{a,\pi}_{pr}, O^{a,\pi}_{n-pr}, O^{a,\tau}_{pr}, O^{a,\tau}_{n-pr}, O^{m}, O^{p}\}$. *Then,* $Y \sqsubseteq X$ *iff*

(i) *if* $Y = O^{a,\pi}_{pr}$, *then* $X \in \{O^{a,\pi}_{pr}, O^{a,\pi}_{n-pr}, O^{a,\tau}_{pr}, O^{a,\tau}_{n-pr}, O^{m}, O^{p}\}$;
(ii) *if* $Y = O^{a,\pi}_{n-pr}$, *then* $X \in \{O^{a,\pi}_{n-pr}, O^{a,\tau}_{n-pr}, O^{m}, O^{p}\}$;
(iii) *if* $Y = O^{a,\tau}_{pr}$, *then* $X \in \{O^{a,\pi}_{pr}, O^{a,\pi}_{n-pr}, O^{a,\tau}_{pr}, O^{a,\tau}_{n-pr}, O^{m}, O^{p}\}$;
(iv) *if* $Y = O^{a,\tau}_{n-pr}$, *then* $X \in \{O^{a,\pi}_{n-pr}, O^{a,\tau}_{n-pr}, O^{m}, O^{p}\}$;
(v) *if* $Y = O^{m}$, *then* $X = O^{m}$;
(vi) *if* $Y = O^{p}$, *then* $X \in \{O^{p}, O^{m}\}$.

Definition 2. *Let* Xa *be a deontic literal and* Y *any deontic operator. If* $X = \neg Y$, X *is a* negative operator*; if* $X = Y$, *it is a* positive operator.

Definition 3. *Let* $A = \bigotimes_{i=1}^{m} Xa_i$ *and* $B = \bigotimes_{i=1}^{n} Yb_i$ *be two* $\otimes$*-expressions. Then,* A deontically includes B *iff* $m = n$, *and for each* Xa_i, Yb_i *(1)* $a_i = b_i$, *and (2) if* X *and* Y *are positive operators, then* $Y \sqsubseteq X$.

Definition 4. *Let* $r_1 : \Gamma \Rightarrow A \otimes B \otimes C$ *and* $r_2 : \Delta \Rightarrow D$ *be two rules, where* $A = \bigotimes_{i=1}^{m} Xa_i$, $B = \bigotimes_{i=1}^{n} Yb_i$ *and* $C = \bigotimes_{i=1}^{p} Zc_i$. *Then* r_1 subsumes r_2 *iff*

1. $\Gamma = \Delta$ *and* A *deontically includes* D*; or*
2. $\Gamma \cup \{\neg a_1, \ldots, \neg a_m\} = \Delta$ *and* B *deontically includes* D*; or*
3. $\Gamma \cup \{\neg b_1, \ldots, \neg b_n\} = \Delta$ *and* $A \otimes \bigotimes_{i=0}^{k \leq p} c_i$ *deontically includes* D.

Consider, e.g., the obligation $B = O^{a,\tau}_{n-pr}b$. If another obligation A is equal to B, compliance conditions for both are trivially the same. If A is either $O^{a,\pi}_{n-pr}b$, $O^{m}b$, or $O^{p}b$, A deontically includes B, because, if both are in force, the compliance of A implies the compliance of B. However, notice that if A is a preemptive achievement obligation, we have no guarantee that its compliance supports the compliance of B: indeed, b could have been obtained before A and B were in force, which is enough for fulfilling only A.

Solving Conflicts Conflicts often arise in normative systems. However, we have to determine whether we have genuine conflicts between $\otimes$-expressions or whether such $\otimes$-expressions admit states where all can be complied with. Suppose that $A = O^{p}a \otimes O^{m}b$ and $B = O^{a,\pi}_{pr}\neg a \otimes O^{m}\neg b$ are in force. The secondary obligations of A and B are in contradiction but their primary obligations do not necessarily lead to a joint non-compliance: if it is now forbidden to pay, and it is obligatory to pay by tomorrow, I can comply with both obligations by simply paying tomorrow.

Therefore, we have first to identify what $\otimes$-expressions do conflict with one another. First of all, let us define when two single obligations are in conflict:

Definition 5. *Let l, Xl, and Y be a literal, a deontic literal, and a positive operator, respectively. The complement $\sim l$ is $\neg p$ if $l = p$, and p if $l = \neg p$. The complement $\sim Xl$ is defined as follows:*

- *If $Xl = Yl$, $\sim Xl = \{Zp | Z$ is positive, $p = \sim l$, either $Z \sqsubseteq Y$ or $Y \sqsubseteq Z\} \cup \{\neg Zq | Z = Y, q = l\}$;*
- *If $Xl = \neg Yp$, $\sim Xl = \{Zq | Z$ is positive, $Z = Y, q = l\}$.*

Definition 6 states under what conditions two $\otimes$-expressions are in conflict.

Definition 6. *Let $A = \bigotimes_{i=1}^{m} Xa_i$ be an $\otimes$-expression. Then, $\sim A = \{B = \bigotimes_{i=1}^{n} Yb_i | m = n, \forall Xa_i, Yb_i : Xa_i = \sim Yb_i\}$.*

Given a theory consisting of a set of rules R, a set S of facts (literals and deontic literals), and a superiority relation, we can use the inference mechanism of Defeasible Logic to compute, in time linear to the size of the theory, the set of its conclusions. This implies to solve genuine conflicts by resorting to the superiority relation over the rules. Once we have defined when two $\otimes$-expressions are in conflict (Definition 6), we can simply use the same reasoning mechanism described in [2].

Normalisation Process The PCL normal form of a normative system provides a representation of normative specifications in a format that can be used to check the compliance of a process. This consists of the following steps:

1. Starting from a formal representation of the explicit clauses of a set of normative specifications we generate all the implicit conditions that can be derived from the normative system by applying the merging mechanism of PCL.
2. We can clean the resulting representation by throwing away all redundant rules according to the notion of subsumption.
3. Finally we detect and solve normative conflicts.

In general the process at step 2 must be done several times in the appropriate order as described above. The normal form of a set of rules in PCL is the fixed-point of the above constructions. A normative system contains only finitely many rules and each rule has finitely many elements. Notice that the operation on which the construction is defined is monotonic [6], so by set theory results the fixed-point exists and is unique.

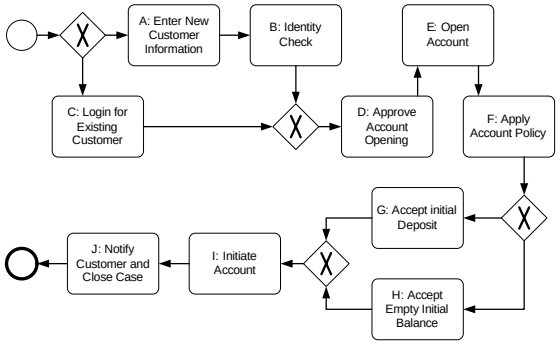
4 Process Modeling

A business process model (BPM) describes the tasks to be executed (and the order in which they are executed) to fulfill some objectives of a business. A language for BPM usually has two main elements: tasks and connectors. Tasks correspond to activities to be performed by actors and connectors describe the relationships between tasks: a minimal set of connectors consists of sequence (a task is performed after another task), parallel –AND-split and AND-join– (tasks are to be executed in parallel), and choice –(X)OR-split and (X)OR-join– (at least (most) one task in a set of task must be executed).

Execution Semantics The execution semantics of the control flow aspect of a BPM is defined using token-passing mechanisms, as in Petri Nets. The definitions used here extend the execution semantics of [9] with semantic annotations in the form of effects and their meaning.

A process model is seen as a graph with nodes of various types –a single start and end node, task nodes, XOR split/join nodes, and parallel split/join nodes– and directed edges (expressing sequentiality in execution). The number of incoming (outgoing) edges are restricted as follows: start node 0 (1), end node 1 (0), task node 1 (1), split node 1 ($>$1), and join node $>$1 (1). The location of all tokens, referred to as a *marking*, manifests the state of a process execution. An execution of the process starts with a token on the outgoing edge of the start node and no other tokens in the process, and ends with one token on the incoming edge of the end node and no tokens elsewhere. Task nodes are executed when a token on the incoming link is consumed and a token on the outgoing link is produced. The execution of an XOR (Parallel) split node consumes the token on its incoming edge and produces a token on one (all) of its outgoing edges, whereas an XOR (Parallel) join node consumes a token on one (all) of its incoming edges and produces a token on its outgoing edge.

Annotation of Processes The starting point of [5] was the methodology proposed by [10] where the task of a process are annotated with the (i) the artifacts or effects of executing and (ii) the rules describing the obligations for the process, where the rules are expressed in PCL. As for the semantic annotations, the vocabulary is presented as a set of predicates P. There is a set of process variables (x and y in Fig. 1), over which logical statements can be made, in the form of literals involving these variables. The task nodes can be annotated using *effects* which are conjunctions of literals using the process variables. If executed, a task changes the state of the world according to its effect: every literal mentioned by the effect is true in the resulting world; if a literal l was true before, and is not contradicted by the effect, then it is still true. We assume that effects in parallel tasks do not contradict each other.



Task	Semantic Annotation
A	$newCustomer(x)$
B	$checkIdentity(x)$
C	$checkIdentity(x)$, $recordIdentity(x)$
D	$accountApproved(x)$
E	$owner(x,y)$, $account(y)$
F	$accountType(y,type)$
G	$positiveBalance(y)$
H	$\neg positiveBalance(y)$
I	$accountActive(y)$
J	$notify(x,y)$

Fig. 1. Example account opening process in private banking, and task annotations

An example of the rules for the process in Figure 1 is "All new customers must be scanned against provided databases for identity checks" (this rule is taken from the Australian *Anti-Money Laundering and Counter-Terrorism Financing Act 2006*)

$$r_1 : newCustomer(x) \Rightarrow O^{a,\tau}_{pr}\, checkIdentity(x)$$

The predicate $newCustomer(x)$ is such that if x is a new customer, we have the obligation to check the data against provided databases. The resulting obligation is non-persistent, i.e., the identity check must be made immediately after we discover that x is a new customer. In addition the obligation is preemptive: if for some reasons the check was already previously performed there is no need to perform it again.

Compliance Checking Our aim in the compliance checking is to figure out (a) which obligations will definitely appear when executing the process, and (b) which of those obligations may not be fulfilled. PCL constraint expressions for a normative system define a behavioural and state space which can be used to analyse how well different behaviour execution paths of a process comply with the PCL constraints. In [5] we have shown how to adapt the algorithm to check compliance proposed in [3] to take into account the rich ontology of norm types we have discussed in the previous sections. The introduction of the types of obligations allows us to model not only semantic compliance (compliance of the effects of the tasks against a regulation) but also structural compliance, that is, for example, to check the order in which the tasks in a process are executed, and whether two tasks can be executed it the same process.

To check compliance we use the following procedure (for the details see [5]):

Step 1 We traverse the graph describing the process and we identify the sets of effects (sets of literals) for all the tasks (nodes) in the process according to the execution semantics outlined in Section 4.

Step 2 For each task we use the set of effects for that particular task to determine the obligations triggered by the execution of the task. This means that effects of a task are used as a set of facts, and we compute the conclusions of the defeasible theory resulting from the effects and the PCL rules annotating the process. In the same

way we accumulate effects, we also accumulate (undischarged) obligations from one task in the process to the task following it in the process.

Step 3 For each task we compare the effects of the tasks and the obligations accumulated up to the task. If an obligation is fulfilled by a task, we discharge the obligation, if it is violated we signal this violation. Finally if an obligation is not fulfilled nor violated, we keep the obligation in the stack of obligations and propagate the obligation to the successive tasks.

5 From Processes to Rules

The aim of this section is twofold. First we want to show that PCL can be used to express conditions on order of the tasks, the structure of the process, including thus common process control flow patterns, as well as other complex conditions about relationships among tasks in a process. In this way, we can use the same language to express the conditions about the effects or artifacts of a process as well as its tasks and we can combine the two to obtain a more expressive formalism able to capture complex compliance requirements. Second, resorting to the same language to express control flows and compliance requirements allows one to use an appropriate rule engine for multiple functions; in particular, we can check the compliance of a process at design-time, and monitoring compliance at run-time. Actually, we can push this one step forward, as the process can be executed directly by the rule engine, thus the monitoring of compliance coincides with the execution of the process. The advantage of this approach is that a business analyst can continue to model a process in familiar standard graphical languages (e.g., BPMN, EPC, Petri-Nets, YAWL, ...), and integrate it with the compliance requirements, and then the combination of the two is executed directly by one engine (the rule engine). This minimises risks of "lost in translation" issues that occur when both the graphical model and the compliance model have to be translated into an execution language for the (common) execution of the two. The use of executable specifications, as in PCL where the rules can be executed directly by a rule engine like SPINdle [11], greatly reduces these risks. On the other hand the mapping of control flow patterns and other complex constraints offers the opportunity for a fully declarative language for business process modeling. In the remaining of this section we illustrate this idea and we show how to capture the most common and basic control flow patterns. Notice that the technique used does not relay on any specific business process language.

To capture control flows and other complex relationships among the tasks in a process we extend the language of PCL with a set of propositional letters to denote the tasks; in what follows we will use $t, t_1, t_2, \ldots$ to refer to them, and these propositional letters correspond to the names/ids of the tasks in a process. For the execution of the process, these names can correspond to calls to the procedures that implement the tasks. In addition, for the representation of OR-split, we need to introduce auxiliary propositional letters corresponding to structural nodes in a process model (i.e., connectors).

Sequence A sequence means that tasks are executed one after the other. The standard execution pattern for a sequence operator in process language is that one task is executed immediately after another. Thus the sequence connection in Figure 2 between tasks t_1

and t_2 is that task t_2 is executed after task t_1. The relationship between the two task can be modeled by the rule

$$t_1 \Rightarrow O^p t_2$$

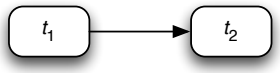


Fig. 2. Sequence operator

After task t_1 has been executed, the literal t_1 triggers the rule that puts the punctual obligation $O^p t_2$ in the stack of obligations to be fulfilled at the next step. Thus, the failure to perform task t_2 in the step following the step in which t_1 completed results in a violation and thus we have a non-compliant execution trace.

After The pattern after, modeled by the rule schema

$$t_1 \Rightarrow O^{a,\pi}_{n-pr} t_2,$$

is a variant of sequence. The idea is that after task t_1 we have the obligation to achieve task t_2, but not necessarily in the step immediately after the step in which t_1 has been executed. It is worth noting that in this case we have to use a non-preemptive obligation to avoid that an execution of t_2 before t_1 fulfils this obligation. Compare this with the co-occurrence condition below.

Parallel tasks: AND-split, AND-join An AND-split starts several sub-processes to be executed in parallel. The condition encoding this pattern is modeled by a set of rules,

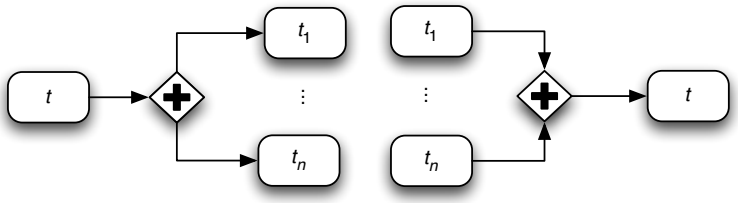


Fig. 3. AND-split and AND-join

$$t \Rightarrow O^p t_1 \quad \ldots \quad t \Rightarrow O^p t_n$$

all of which have the same antecedent, the task t whose completed execution triggers the split. The conclusions of such rules are punctual obligations for the tasks $t_1, \ldots, t_n$ starting the sub-processes to be executed in parallel. This means that the tasks $t_1, \ldots, t_n$ are inserted in the stack of obligations to be executed in the step after task t.

Similarly, an AND-join requires the synchronisation of a number of sub-processes before proceeding to the next task. Accordingly, an AND-join is captured by the rule

$$t_1, \ldots, t_n \Rightarrow O^p t$$

This rule needs all the antecedents to hold to fire, and to conclude the punctual obligation $O^p t$. Hence, all last tasks $t_1, \ldots, t_n$ of the sub-processes to be synchronised have to be completed before we move to the task after the merge of the sub-processes.

Choice: (X)OR split, OR join An OR-split is intended to capture sub-processes where one has a choice on how to continue a process. For the representation of an OR-split

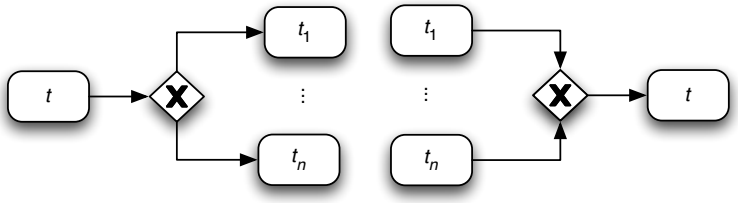


Fig. 4. (X)OR-split and OR-join

pattern in PCL, we have to use the auxiliary propositional letter. For each OR-split connector in a diagram we establish a one-to-one mapping between the connector and the auxiliary propositional letter. Then the set of rules required to model this pattern is

$$t \Rightarrow O^p(ORsplitID) \qquad t_1 \Rightarrow ORsplitID \quad \ldots \quad t_n \Rightarrow ORsplitID$$

The first rule on the left side tells us that the completion of task t trigger the obligation to fulfill the obligation for *ORsplitID*, where *ORsplitID* is the propositional letter of the corresponding OR-connector, and the obligation is in the stack of obligations to be fulfilled in the step immediately after the step where we have t. The other rules do not generate normative conclusion, but just factual conclusions. Thus, the meaning of the first rule is that the completion of task t_1 (which we assume to be the first task in one of the outgoing sub-processes after the OR-split) fulfills the obligation, or in other terms that *ORsplitID* holds.

For an XOR, in addition we need, the rules

$$t_i \Rightarrow \neg t_j \quad i \neq j, 1 \leq i, j \leq n$$

which state that if t_i holds then t_j does not hold, thus it is not possible to have a situation where both t_i and t_j hold. As a consequence, only one of the alternative sub-processes can be executed. This method requires to generate n^2 additional rules for each XOR-split. An alternative encoding of XOR-split, in particular when a default choice is present, is to use a rule with reparative deontic conclusions, thus

$$t \Rightarrow O^p t_1 \otimes O^p t_2 \otimes \cdots \otimes O^p t_n$$

According to the rule above, the best option after t is t_1, but if t_1 is not performed, then the second best option is t_2 and so on. Thus, the above rule determines a total order on the preferences of the alternative choices in an XOR-split. In addition it is possible to combine the above two techniques, so we can have a rule like $t \Rightarrow O^p t_i \otimes O^p ORsplitID$. This gives a default choice over t_i but no preferences over the others sub-processes:

$$t_1 \Rightarrow O^p t \quad \ldots \quad t_n \Rightarrow O^p t$$

Absence The absence is the condition that establishes that one task cannot be anymore scheduled in the process if another task already happened in the process. This condition can be represented by the rule

$$t_1 \Rightarrow O^m \neg t_2$$

that uses a maintenance prohibition (i.e., $O\neg$) stating that the task t_2 cannot happen after the execution of task t_1.

Co-occurrence This pattern is designed to check that two tasks, let us say t_i and t_j, occur in the process. This can expressed as follows: if task t_i happens in the process, this should also include task t_j. The idea is similar to the after pattern; the difference is that in after the second task (for the sake of argument, t_j) should occur in a step successive to that including t_i. For the co-occurrence pattern, this restriction is lifted so task t_j can appear anywhere in the process. To express this we use a non-preemptive obligation. Accordingly, the pattern is modeled by

$$t_i \Rightarrow O^{a,\pi}_{pr} t_j \qquad\qquad t_j \Rightarrow O^{a,\pi}_{pr} t_i$$

The first rule on the left says that t_j must occur when t_i occurs, and the second that t_i must occur when t_j does. Thus, depending on the situation, one can use either one of the two rules or both. In case we have only the first rule, an execution trace is non compliant when we have t_i but not t_j, but non-compliance does not occur when we only have t_j (similarly, for the second rule). If both rules are in force, then a trace is compliant if either both tasks are in the trace or none is.

Conditional Occurence With the previous patters we have examined situations where if one task is included in an execution trace so do other tasks. With this pattern we consider a subtle difference: we consider the case where one task has to be included if another one has to be included as well. This pattern is described by the rule

$$O^x t_1 \Rightarrow O^x t_2$$

The difference with the other patterns is that in the antecedent we have an obligation instead of a factual premise. Most of the considerations regarding the co-occurrence pattern apply to this patter as well; but there is one difference. Suppose that the rule fires, thus we have the obligation of performing task t_1. The obligation to perform task t_2 still exists even if for some reasons task t_1 is not done (for example, let us say there is a situation where it is possible not to execute t_1 provided some compensatory actions are taken).

In Between and Discharge The aim of this pattern is to model the condition that one task must be executed after another one but before a third one, for example, that task t_j is executed between tasks t_i and t_j. In PCL this can be expressed as

$$t_i \Rightarrow O^{a,\pi}_{n-tr} t_j \qquad\qquad t_k \rightsquigarrow \neg O^{a,\pi} t_j \qquad\qquad t_k \Rightarrow \neg t_j$$

The first rule on the left side is just the rule for the after pattern. The second rule in the middle terminates the obligation to achieve t_j when t_k is performed, in addition the

performance of t_k signals that t_j has not been executed. Thus if task t_j is not executed in between the other two task, we have an unfulfilled obligation resulting in a non-compliant situation. Please compare the idea of this pattern with the discussion about deadlines and whether obligations persist after the deadlines.

Loops, Hooks and Loop Termination Most BPM notations allow us to represent loops (reoccurring sub-processes). PCL is able to represent loops as well, with rules like

$$t_i \Rightarrow O^{a,\tau}_{n-pr} t_i$$

or more in general with rules such as

$$t_i \Rightarrow O^{x} t_j$$

where t_j is in the dependence graph of t_i.

To avoid infinite loops, a loop termination condition can be expressed by a rule

$$p \rightsquigarrow \neg O^{x} t_i$$

where t_i is a task involved in a loop (we avoid the discussion about fairness conditions for p and fairness conditions for loop termination for tasks inside OR-split blocks inside loop blocks).

An interesting rule is

$$t_1 \Rightarrow O^{m} t_2$$

This rule requires task t_2 to be execute in every step following a step where task t_1 successfully completed; the obligation generated by the rule is a maintenance condition. The intuition is that t_2 is a hook task, that is a task that must be executed every time the business process activates another task.

6 Summary and Related Work

Given two tasks t_1, t_2 of a process we can use the types of obligations defined in Section 2 to describe relationships between these two tasks (the types of obligations provide a comprehensive classification of the possible obligations). In particular we have seen that some of them give rise to natural and common control flow patterns in business processes, in particular, even if we limit ourselves to basic relationships, we can express patterns like those in Table 1.

$t_i \Rightarrow O^{p} t_j$	sequence
$t_i \Rightarrow O^{a,\pi}_{n-pr} t_j$	after
$t_i \Rightarrow O^{a,\pi}_{pr} t_j$	co-occurrence
$t_i \Rightarrow O^{m} t_j$	process hook

Table 1. Flow patterns

In addition we can represent many more patterns including those that are difficult to express in standard BPM languages, for example, conditions using tasks from different branches of a process (e.g., in an OR-block), and we can mix information about tasks (task literals) and data conditions. Thus, it seems to us that PCL offers a rich, compact and holistic framework for business process compliance

in such a way as we also can use a rule engine for PCL as a process engine. To understand the full extent of the proposed approach we plan a comprehensive comparison with control flow patterns [12], data patterns [13], and the declarative patterns of [14].

A number of works have been devoted to compliance in control modelling. [15] presents the logical language PENELOPE, that provides the ability to verify temporal constraints arising from compliance requirements on effected business processes. [16] develops a method to check compliance between object lifecycles that provide reference models for data artifacts e.g. insurance claims and business process models. [17] provides temporal rule patterns for regulatory policies, although the objective of this work is to facilitate event monitoring rather than the usage of the patterns for support of design time activities. Furthermore, [18] presented an architecture for supporting Sarbanes-Oxley Internal Controls, which include functions such as workflow modeling, active enforcement, workflow auditing, as well as anomaly detection. [19] studies the performance of business contract based on their formal representation. [20] seeks to provide support for assessing the correctness of business contracts represented formally through a set of commitments. The reasoning is based on value of various states of commitment as perceived by cooperative agents. Also, there have been recently some efforts towards support for process modelling against compliance requirements. [10] proposes an approach based on control tags to visualize internal controls on process models. [21] takes a similar approach of annotating and checking process models against compliance rules, although the visual rule language (BPSL) does not directly address the deontic notions providing compliance requirements.

Many works proposed declarative languages to model business processes. [14,22] used a language based on linear temporal logic to model processes to check conformance by symbolic model checking, [23] show how to use Concurrent Transaction Logic to represent the structure of of workflows, while [24] advance a prolog-like language for the same scope. The use of logic and rule based languages to describe business processes is not new. However, most works are restricted to limited patters of tasks, and almost no work uses the same for data (artifact) requirements, nor it address deontic concerns and is able to handle violations and possible compensations for violations.

Acknowledgement

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 and Queensland Government.

References

1. Carmo, J., Jones, A.: Deontic logic and contrary to duties. In Gabbay, D., Guenther, F., eds.: Handbook of Philosophical Logic, 2nd Edition. Kluwer (2002) 265–343
2. Governatori, G.: Representing business contracts in RuleML. International Journal of Cooperative Information Systems **14** (2005) 181–216
3. Governatori, G., Rotolo, A.: An algorithm for business process compliance. In Sartor, G., ed.: Jurix 2008, IOS Press (2008) 186–191

4. Governatori, G., Hulstijn, J., Riveret, R., Rotolo, A.: Characterising deadlines in temporal modal defeasible logic. In Orgun, M.A., Thornton, J., eds.: Australian AI. LNCS 4830, Berlin, Springer (2007) 486–496
5. Governatori, G., Rotolo, A.: A conceptually rich model of business process compliance. In Link, S., Ghose, A., eds.: APCCM 2010. CRPIT, ACS (2010)
6. Governatori, G., Rotolo, A.: Logic of violations: A Gentzen system for reasoning with contrary-to-duty obligations. Australasian Journal of Logic **4** (2006) 193–215
7. Antoniou, G., Billington, D., Governatori, G., Maher, M.J.: Representation results for defeasible logic. ACM Transactions on Computational Logic **2** (2001) 255–287
8. Governatori, G., Rotolo, A.: Changing legal systems: Legal abrogations and annulments in defeasible logic. The Logic Journal of IGPL **18** (2010) 157–194
9. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and more focused control-flow analysis for business process models through sese decomposition. In Krämer, B.J., Lin, K.J., Narasimhan, P., eds.: ICSOC. LNCS 4749, Springer (2007) 43–55
10. Sadiq, S.W., Governatori, G., Namiri, K.: Modeling control objectives for business process compliance. [25] 149–164
11. Lam, H.P., Governatori, G.: The making of SPINdle. In Governatori, G., Hall, J., Paschke, A., eds.: RuleML 2009. LNCS 5858, Berlin, Springer (2009) 315–322
12. van der Aalst, W.M., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and Parallel Databases **14** (2003) 5–51
13. Russell, N., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Workflow data patterns: Identification, representation and tool support. In Delcambre, L.M.L., Kop, C., Mayr, H.C., Mylopoulos, J., Pastor, O., eds.: ER. LNCS 3716, Springer (2005) 353–368
14. Pesic, M., van der Aalst, W.M.P.: A declarative approach for flexible business processes management. [26] 169–180
15. Goedertier, S., Vanthienen, J.: Designing compliant business processes with obligations and permissions. [26] 5–14
16. Küster, J.M., Ryndina, K., Gall, H.: Generation of business process models for object life cycle compliance. [25] 165–181
17. Giblin, C., Müller, S., Pfitzmann, B.: From regulatory policies to event monitoring rules: Towards model driven compliance automation. Technical report, IBM Zurich Lab. (2006)
18. Agrawal, R., Johnson, C.M., Kiernan, J., Leymann, F.: Taming compliance with Sarbanes-Oxley internal controls using database technology. In Liu, L., Reuter, A., Whang, K.Y., Zhang, J., eds.: ICDE, IEEE Computer Society (2006) 92
19. Farrell, A.D.H., Sergot, M.J., Sallé, M., Bartolini, C.: Using the event calculus for tracking the normative state of contracts. International Journal of Cooperative Information Systems **14** (2005) 99–129
20. Desai, N., Narendra, N.C., Singh, M.P.: Checking correctness of business contracts via commitments. In: Proc. AAMAS 2008. (2008) 787–794
21. Liu, Y., Müller, S., Xu, K.: A static compliance-checking framework for business process models. IBM Systems Journal **46** (2007) 335–362
22. Rozinat, A., van Der Aalst, W.M.: Conformance checking of processes based on monitoring real behavior. Information Systems **33** (2008) 64–95
23. Roman, D., Kifer, M.: Reasoning about the behaviour of semantic web services with concurrent transaction logic. In: VLDB. (2007) 627–638
24. Gregory, S., Paschali, M.: A prolog-based language for workflow programming. In Murphy, A.L., Vitek, J., eds.: COORDINATION. LNCS 4467, Springer (2007) 56–75
25. Alonso, G., Dadam, P., Rosemann, M., eds.: BPM 2007. LNCS 4714, Springer (2007)
26. Eder, J., Dustdar, S., eds.: Business Process Management Workshops. LNCS 4103, Springer (2006)