Compliant business processes with exclusive choices from agent specification

Francesco Olivieri[†], Matteo Cristani[†], and Guido Governatori[•⋆]

[†]Department of Computer Science, University of Verona, Italy
[•]NICTA, Queensland Research Laboratory, Australia

Abstract. In this paper we analyse the problem of synthesising compliant business processes from rules-based declarative specifications for agents. In particular, we consider the approach by [1,2] and we propose computationally efficient algorithms to combine plans extracted from the deliberation of an agent to generate the corresponding business processes with exclusive choice patterns.

1 Introduction

The standard architectures for cognitive agents (e.g., the BDI architecture) distinguish three phases: the deliberation phase where agents deliberate what are their goals; then in the plan selection phase, the agent selects, based on the outcome of the first phase, which plan to actuate from her plan library; finally, in the last phase, the agent executes the selected plan. [1,2], in addition to the inclusion of norms, propose a different approach where an agent is defined by a set of rules describing the environment where the agent is situated, the capabilities of the agent (the actions or tasks the agent can perform, the conditions under which the agent can perform them, and the effects they generate), the aims of the agent, and the norms the agent is subject to. Given a set of facts describing a situation, the agent deliberates using its rule base to determine whether a particular outcome is attainable without violating the relevant norms. Given the information in the rule base, the deliberation contains information about the tasks to be performed (and the relative order in which they have to be executed) to reach the objective. Thus, the deliberation effectively generates a plan. [3,4] propose an efficient algorithm to extract such plans and to visualise them in form of business process models.

A business process model is a compact (graphical) representation of a set of activities and the order in which they have to be executed to reach a business objective. The tasks or activities are connected by control flow connectors. In particular, the basic control flow connectors are: sequence (task t' directly follows task t), AND split or parallel split (all tasks in all the branches are executed), exclusive choice or X-OR split (only the tasks in one of the branches are executed), AND join (all tasks in the incoming branches must have been executed), and X-OR join (one branch must have been executed). A business process model can be seen as a set of sequences of tasks, which corresponds to a set of plans. One of the advantages of business process models being that they provide an intuitive (graphical) representation that can be understood by the stakeholders

⋆ NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

of the business process, and that they can be executed by workflow engines. While rules (and, in general, declarative specifications) can provide powerful and flexible representation languages, and the individual rules can be easily understood by stakeholders and domain experts (including rules modelling norms), it can be daunting to understand what is going on with large knowledge bases. To mitigate this problem, [3,4] propose to translate a case (set of facts) into a business process model (i.e., a model guaranteed to comply with the applicable norms and fulfilling the predefined objective).

The approach in [3,4] has the limitation that it synthesises business processes without exclusive choices. This is due to the following two factors: the logic underlying the declarative agent specifications (Defeasible Logic, a sceptical non-monotonic logic) and the use of a consistent set of facts as input for the case. However, nothing prevents us from considering different (and incompatible) sets of facts as input. For each set, we can generate the resulting business process and we could join them using each business processes as a branch in a global X-OR block. However, this would defy the purpose of aiming at compact representations. In this paper, we address the problem of how to generate business process models with exclusive choice patterns starting from a set of cases (each represented by a set of facts encoding, for example, standard configurations and most common choices) and a set of declarative specifications for an agent.

2 Logic

The aim of this section is to give some basic notions of the logic presented in a series of works [1,2] and to make the reader acquainted with the intuitions behind such a logic apparatus while avoiding instead the technicalities.

DL is typically sceptical, meaning that it allows rules for opposite conclusions. In the situation where rules for opposite literals are activated (may *fire*), the logic does not produce any inconsistency but it simply does not draw any conclusion unless a preference (or superiority) relation states that one rule prevails over the other. A *defeasible theory* D is defined as a structure $(F,R,>)$, where (i) F is a set denoting simple pieces of information that are considered to be always true (e.g., a fact is that "Sylvester is a cat", formally $cat(Sylvester)$), (ii) R contains two finite sets of rules: defeasible rules and defeaters, (iii) $> \subseteq R \times R$ is a binary relation. A rule is an expression $r : A(r) \hookrightarrow C(r)$, and consists of: (i) a unique name r, (ii) the antecedent $A(r)$ which is a finite set of (modal) literals, (iii) an *arrow* $\hookrightarrow \in \{\Rightarrow, \rightsquigarrow\}$ denoting, respectively, defeasible rules and defeaters, and (iv) its consequent $C(r)$ which is a single literal (or a chain of modal literals, see below). A *defeasible* rule can be defeated by contrary evidence; for example the rule representing "cats typically eat birds" is "$cat(X) \Rightarrow eat\_birds(X)$", means that if something is a cat, then we may conclude that it eats birds, unless there is evidence proving otherwise. *Defeaters* are special rules whose only purpose is to defeat defeasible rules by producing contrary evidence. The *superiority relation* $>$ is used to define when one rule may override the conclusion of another one, e.g., given the rules "$r : cat(X) \Rightarrow eat\_birds(X)$" and "$s : domestic\_cat(X) \Rightarrow \neg eat\_birds(X)$", if we state that "$s > r$", then Sylvester does not to eat birds.

A defeasible conclusion is a tagged literal and can have one of the following form: (1) $+\partial q$ which means that q is defeasibly provable in D, and (2) $-\partial q$ which means

that q is refuted, or not defeasibly provable in D. The idea of $+\partial$ derivation being that a given literal q is defeasibly provable if either it is definitely provable, or we argue by using the defeasible part of the theory. In the latter case, $\sim q$ must be not definitely provable, and there must exist an applicable strict or defeasible rule for q. Finally, every attack on such a rule is either discarded, or defeated by a stronger rule supporting q.

The logic of [1,2] is equipped with modal rules and operators to capture the obligations an agent has to comply with and the goal-like mental attitudes of the agent. However, such components are not needed to synthesis the plans an agent commits to, and, for the purpose of this paper they can be considered as simple conditions literals.

Given a defeasible theory D, we define the set of positive and negative conclusions as the *extension* of the theory. The positive extension is noted by $E^+(D)$, the negative one by $E^-(D)$. More formally the positive and negative extensions are defined as follows: $E^+(D) = \{p|D \vdash +\partial p\}$ and $E^-(D) = \{p|D \vdash -\partial p\}$.

Observation. *An extension is characteristic of a specific set of facts: fixed the rules and the superiority relation, two different sets of facts may lead to two distinct extensions.*

Exclusive choice patterns Here we describe three possible variants to effectively model exclusive choice patterns in defeasible logic.

When we consider only a single literal t, a *choice pattern* is a set of rules proving t. Typically, such rules have distinct sets of antecedents. On the contrary, an *exclusive choice pattern* must not consider a single literal, but a set of distinct literals, $t_1, \ldots, t_n$. These literals must share, to a certain extent, the same set*s* of antecedents (activation elements). What do we mean by *to a certain extent*? Recall that literals in our theory are conceptually divided between condition- and task-literals. Hence, $t_1, \ldots, t_n$ must have a common set of activation task-literals, while differing on the activation condition-literals. For instance, in the following scenario

$$r_1 : a, b, c_1 \Rightarrow t_1 \qquad\qquad r_2 : a, b, c_2 \Rightarrow t_2,$$

t_1 and t_2 share the set of common activation task-literals (a and b), while the distinctive activation condition-literals are c_1 for t_1, and c_2 for t_2. Hereafter, condition-literals are denoted by cs with a subscript notation.

The key distinction between a choice and an exclusive choice is that, in the former, every alternative can be executed at the same time while, in the exclusive choice, once an alternative is executed, none of the others can. Thus, the logic must exhibit structures to prevent the execution of all the other alternatives once a choice is made. This can be easily handled by the use of defeater rules.

Variant 1 sees a preferred task (say t_1) while all the other alternatives are of equal importance with respect to one another. Formally,

$$\begin{aligned} r_1 &: \Delta \Rightarrow t_1 \\ r_i &: \Delta, \Gamma_i \Rightarrow t_i \quad \text{for } 1 < j \leq n \\ d_{1j} &: t_1 \rightsquigarrow \sim t_j \quad \text{for } 1 < j \leq n \\ d_{ij} &: t_i \rightsquigarrow \sim t_j \quad \text{for } 1 \leq i, j \leq n,\ i \neq j. \end{aligned}$$

Δ the set of common activation task-literals; Γ_i denotes the set of condition-literals distinctive for task-literal t_i. Task t_1 is the default choice: as such, no activation conditions are needed. Defeaters d_{1i} and d_{ij} ensure that once an alternative is chosen, no other can.

Variant 2 considers none of the tasks involved to be preferred to the others.

$$r_i : \Delta, \Gamma_i \Rightarrow t_i \qquad d_{c_{ij}} : t_i \rightsquigarrow \sim t_j \quad \text{for } 1 \leq i, j \leq n,\ i \neq j.$$

A different form of this second variant excludes sets Γ_is from playing a role in the choice of which branch to run. We think that such a variation is conceptually weak given that there should always be some way to discriminate why running an alternative instead of another. Nonetheless, in the following algorithms, it will be trivially calculated.
Variant 3. There are situations where a branch of the exclusive choice pattern does not actually perform any action, but it is just used to skip the run to the end of the X-OR join gate (we can see it like an empty branch).

$$\begin{array}{rl} r_1 : \Delta, \Gamma_1 \Rightarrow t_1 & \\ r_i : \Delta, \Gamma_i \Rightarrow t_i & \text{for } 1 < i \leq n \\ r_{i1} : t_i \Rightarrow t_1 & \text{for } 1 < i \leq n \\ d_{1j} : \Gamma_1 \rightsquigarrow \sim t_i & \text{for } 1 < i \leq n \\ d_{ij} : t_i \rightsquigarrow \sim t_j & \text{for } 1 < i, j \leq n \text{ and } i \neq j. \end{array}$$

Here, the execution of t_1 prevents the execution of any other t_i.

3 Algorithms

In [3,4], we presented methodologies to compute a process graph starting by a set of declarative specifications being able to describe: (i) the system's environment, (ii) the active norms and (iii) a set of goals (called outcomes) the system aims to achieve. The execution of that step was possible upon previous computations on a series of algorithms which efficiently compute which actions the system is meant to perform in order (i) to achieve a set of goals (ii) while it does not to violate certain norms (or to compensate all the violated ones) [1,2]. The input of [1,2]'s algorithms was (1) the underlying modal logic (of the type of the one presented in Section 2), and (2) an assignment to the set of facts. The output was an extension. We recall that, fixed the rules, distinct set of facts (typically) generate different extensions. Given a set of facts, some rules are activated to produce certain effects and these effects may, in turn, activate other rules which produce other effects (and so on). Therefore, the positive extension represents all the *active* literals. This means that if the literal stands for a condition, such a condition is fulfilled (for instance, the norm is not violated or compensated); if the literal stands for a task, that task will be executed in the corresponding process.

The algorithms of [3,4] started from a single extension. The focus, and novelty, of the present research being that such an assignment to the set of facts may well not be the only one resulting in a compliant situation. This is typical in business practices as we argued above where, instead of a single case, we consider multiple (possibly incompatible) cases corresponding to scenarios the agent has to deal with. Before showing the algorithmic results for the exclusive choice pattern computation, we briefly describe of how [3,4]'s algorithms work. The idea of those algorithms is to start from a set of *proved* goals and from each one of them to navigate backwards the derivation tree by considering only those literals in the positive extension. This procedure is then recursively iterated on each of those literals. This schema naturally captures the three main

features of a process graph: (1) *sequence*, given a literal p, if a belongs to a certain $A(r)$ such that $C(r) = p$, then in the graph there is an edge linking node A to node P; (2) *parallel execution*, if also literal b is in $A(r)$ then it is natural that nodes A and B are linked to P by and AND JOIN gate; (3) *choice*, many rules proving p represent different alternatives to obtain p and we link them to P through an OR JOIN gate. Once this backwards phase ended, the process graph is synthesised by recognising *co-occurrence* patterns and by removing condition-literals, substituted by labelled edges.

X-OR Pattens algorithms The algorithms we describe hereafter recognise exclusive choice patterns. Algorithm 1 (X-OR) has been designed to be a procedure invoked by the main algorithm of [3,4] after the condition-task elimination has taken place. Let us understand the basic principles behind it by helping us with the following examples.

Example 1. Let D be the theory, with empty superiority relation, such that

$$r_1 : a, c_1 \Rightarrow t_1 \quad r_2 : a, c_2 \Rightarrow t_2 \quad r_3 : t_1 \Rightarrow b \quad r_4 : t_2 \Rightarrow b \quad d_{12} : t_1 \rightsquigarrow \sim t_2 \quad d_{21} : t_2 \rightsquigarrow \sim t_1.$$

Here, if we have the two distinct assignments $F' = \{a, c_1\}$ and $F'' = \{a, c_2\}$, then task-literals t_1 and t_2 are in an exclusive choice pattern. Indeed, (i) t_1 is proved only when using F' (and symmetrically for t_2 with F''), (ii) t_1 and t_2 share the activation task a, (iii) c_1 (resp. c_2) is the unique activation condition for t_1 (resp. t_2), and (iii) the activation of one task defeats the activation of the other through the presence of defeaters d_{12} and d_{21}. Finally, both t_1 and t_2 derive b. We can thus close the pattern by linking T_1 and T_2 to an X-OR JOIN gate-node, which in turn is followed by B.

Exclusive choice variants described in Section 2 are useful to define some theoretical properties and to give the reader an understanding of which differences distinguish one variant from another. Their common, focal point being that the tasks in the exclusive choice share a set of activation task-literals while other sets of condition-literals are characteristic of which choice-branch to run. Given that Algorithm 1 X-OR's execution begins after the condition nodes have been removed from the graph, to gather for such activation requirements is not trivial, as following Example 2 points out.

Example 2. Let D be the theory, with empty superiority relation, such that

$$\begin{array}{llll} r_1 : b, c_3 \Rightarrow t_1 & r_2 : d \Rightarrow t_1 & r_3 : a \Rightarrow c_1 & \\ r_4 : c_1 \Rightarrow c_3 & r_5 : t_1 \Rightarrow e & r_6 : a, b, d, c_2 \Rightarrow t_2 & \\ r_7 : t_2 \Rightarrow e & r_8 : e \Rightarrow f & d_{12} : t_1 \rightsquigarrow \sim t_2 & d_{21} : t_2 \rightsquigarrow \sim t_1. \end{array}$$

Given two assignments to the set of facts (e.g., $F' = \{a, b, d, c_3\}$ and $F'' = \{a, b, d, c_2\}$), are t_1 and t_2 in an exclusive choice pattern? Yes, they are. Apparently, the immediately previous activation task-literals of t_1 are b and d only (resp. due to r_1 and r_2), while for t_2 are a, b and d. What about a for t_1? a is the antecedent of r_3 for proving c_1 which, in turn, is used by r_4 to prove c_3. Recall that, once the process graph is made, synthesis Phase 2 sees nodes C_1 and C_3 being removed from the graph and substituted by a labelled arc connecting A *directly* to AND-J_{r_1}. Thus, task-node A can be seen as an activation task of T_1. Therefore, activation tasks of t_1 are the same of t_2 even if, in case of t_1, they come from four distinct rules, while in case of t_2 from the single r_5. Is that nonetheless correct? Again, the answer is yes, provided that, for every each t_i in the exclusive choice pattern, all its activation literals are derived in the same extension.

Before the detailed description of the algorithms, we introduce the two last preliminary notions. A task dependency graph is essentially a dependency graph where we consider task-literals only. The *task dependency graph* of D, $TDG(D)$, is the directed graph defined as follows: the set of vertices is $V_{TDG(D)} = \{t | t \text{ is a task-literal in } D\}$. The set of arcs is $E_{TDG(D)} = \{(a,b) | \exists r \in R_{sd}[b] : a \in A(r) \text{ and } a,b \in V_{TDG(D)}$; or $\exists r_1, \ldots, r_n \in R_s d : a \in A(r_1), b = C(r_n), C(r_j) \notin V_{TDG(D)}\ j < n$, and $C(r_i) \in A(r_{i+1})\}$. Given the task dependency graph $TDG(D)$ and a task-literal l in it, define *Reachability*(l) as the set of nodes reachable from a l. Formally *Reachability*$(l) = \{\mathrm{M} \in V_{TDG(D)} | \exists \mathrm{M}_1, \ldots \mathrm{M}_n :$ $(\mathrm{L}, \mathrm{M}_1), (\mathrm{M}_i, \mathrm{M}_{i+1}) \in E_{TDG(D)}$ and $\mathrm{M} = \mathrm{M}_n\}$. Notice that computing the task dependency graph and the set of reachable nodes is quadratic in the number of vertices.

Algorithm 1 X-OR

1: Compute the task-dependency graph TDG
2: $alreadyXed \leftarrow \emptyset$, $conditionsLabels \leftarrow \emptyset$, $XJoin \leftarrow \emptyset$
3: **for** $\mathrm{T} \in V \setminus alreadyXed$. $\exists i, j.\ t \in E_i^+(D)$ and $t \notin E_j^+(D)$ with $i, j \leq m, i \neq j$ **do**
4: $backwardTasks \leftarrow$ BACKWARDTASKSPROJ(T)
5: $XORtasks \leftarrow$ IFX-OR(T, i, $backwardTasks$)
6: **if** $XORtasks \neq \emptyset$ **then**
7: $XJoin(\mathrm{T}_l) \leftarrow$ Reachability(t_l)
8: $closure \leftarrow \bigcap_{1 \leq l \leq n} XJoin(\mathrm{T}_l)$
9: $closure \leftarrow closure \setminus \{\mathrm{P} \in closure |$ P depends on Q $\in closure$ or P is not a task-node$\}$
10: $\mathrm{T}_{final} \leftarrow XORtasks \cap closure$
11: $V \leftarrow V \cup \{\text{XOR-S}_{\mathrm{T}_1,\ldots,\mathrm{T}_n}, \text{XOR-J}_{\mathrm{T}_1,\ldots,\mathrm{T}_n}\}$ with $\mathrm{T}_1, \ldots, \mathrm{T}_n \in XORtasks$
12: **for** $\mathrm{T}_l \in XORtasks \setminus \{\mathrm{T}_{final}\}$ **do**
13: $E \leftarrow E \cup \{(\mathrm{L}, \text{XOR-S}_{t_1,\ldots,t_n}) | \mathrm{L} \in in_V(\mathrm{T}_i)\} \setminus \{(\mathrm{L}, \mathrm{T}_i)\}$
14: $E \leftarrow E \cup \{e = (\text{XOR-S}_{\mathrm{T}_1,\ldots,\mathrm{T}_n}, \mathrm{T}_i)\}$
15: $label(e) \leftarrow conditionLabels(\mathrm{T}_i)$
16: **if** $closure \neq \emptyset$ **then**
17: **if** $\mathrm{T}_{final} \neq$ null **then**
18: $E \leftarrow E \cup \{(\text{XOR-J}_{\mathrm{T}_1,\ldots,\mathrm{T}_n}, \mathrm{T}_{final})\} \cup \{(\mathrm{L}, \text{XOR-J}_{\mathrm{T}_1,\ldots,\mathrm{T}_n}) |\ \exists 1 \leq j \leq n.\ \mathrm{L} \in in_V(\mathrm{T}_{final}) \cap XJoin(\mathrm{T}_j)\} \setminus \{(\mathrm{L}, \mathrm{T}_{final}) | \mathrm{L} \in in_V(\mathrm{T}_{final}) \cap XJoin\}$
19: **else**
20: **for** $\mathrm{P} \in closure$ **do**
21: $E \leftarrow E \cup \{(\mathrm{L}, \text{XOR-J}_{\mathrm{T}_1,\ldots,\mathrm{T}_n}) | \exists 1 \leq j \leq n.\ \mathrm{L} \in in_V(\mathrm{P}) \cap XJoin(\mathrm{T}_j)\} \setminus \{(\mathrm{L}, \mathrm{P}) | \mathrm{L} \in in_V(\mathrm{P}) \cap XJoin\} \cup \{(\text{XOR-J}_{\mathrm{T}_1,\ldots,\mathrm{T}_n}, \mathrm{P})\}$
22: $alreadyXed \leftarrow alreadyXed \cup XORtasks$

Algorithm 1 (X-OR) is the main procedure, which invokes its subroutines Algorithm 2 (BACKWARDTASKSPROJ) and Algorithm 3 (IFX-OR). Algorithm 1 is conceptually divided in two phases. During the first phase of the main algorithm, it collects the tasks appearing in an exclusive choice pattern. The second phase starts if an X-OR has been found and manages the graph operations needed to insert the X-OR SPLIT and X-OR JOIN into the process graph.

In the first phase, for every task-node, Algorithm 1 gathers (a) the activation tasks and (b) the activation conditions. It stores the tasks in the set *backwardTasks* and the conditions in *conditionLabels*. *conditionLabels* is an array of arrays and will be afterwards used to label edges: for each task that will be present in the X-OR pattern, there is a set where to store all the activation conditions.

Steps (a) and (b) are performed by Algorithm 2: it stores all nodes with an outgoing edge towards the node under examination in the set *premises* (T in the algorithm).

Algorithm 2 BACKWARDTASKSPROJ

1: **procedure** BACKWARDTASKSPROJ(node T)
2: $bwTasks \leftarrow \emptyset$, $premises \leftarrow in_V(\mathrm{T})$, $pastPrem \leftarrow \mathrm{T}$
3: **while** $premises \neq \emptyset$ **do**
4: Let P be the first element of *premises*
5: **switch** (Node type of P)
6: **case** P is a task node**:**
7: $conditionLabels(\mathrm{P}) \leftarrow conditionLabels(\mathrm{P}) \cup labels(e)$ for each $e \in \{e' \in E \mid e' = (\mathrm{P},\mathrm{L})$ and $\mathrm{L} \in pastPrem\}$
8: $premises \leftarrow premises \setminus \{\mathrm{P}\}$
9: $bwTasks \leftarrow bwTasks \cup \{\mathrm{P}\}$
10: **case** P is and AND-J_r node**:**
11: $conditionLabels(\mathrm{P}) \leftarrow conditionLabels(\mathrm{P}) \cup \{c$ is condition-literal $\mid c \in A(r)\} \cup labels(e)$ for each $e \in \{e' \in E \mid e' = (\mathrm{P},\mathrm{L})$ and $\mathrm{L} \in pastPrem\}$
12: $premises \leftarrow premises \cup in_V(\mathrm{P}) \setminus \{\mathrm{P}\}$
13: **case** P is and OR-J_P node**:**
14: $conditionLabels(\mathrm{P}) \leftarrow conditionLabels(\mathrm{T}) \cup labels(e)$ for each $e \in \{e' \in E \mid e' = (\mathrm{P},\mathrm{L})$ and $\mathrm{L} \in pastPrem\}$
15: $premises \leftarrow premises \cup in_V(\mathrm{P}) \setminus \{\mathrm{P}\}$
16: **end switch**
17: $pastPrem \leftarrow pastPrem \cup \{\mathrm{P}\}$
18: **return** *bwTasks*

The information about such nodes is provided by $in_V(\mathrm{T})$. We say that, for any node X, $in_V(\mathrm{X}) = \{\mathrm{Y} \in V \mid (\mathrm{Y},\mathrm{X}) \in E\}$ denotes the set of all nodes reaching X. All elements stored in *premises* are now analysed. For every of such element P, if P is a task node, then we do not need to further analyse P's predecessors: we simply save it in *bwTasks* and update $conditionLabels(\mathrm{T})$. We update $conditionLabels(\mathrm{T})$ by adding to it the label of each edge e connecting P to a node previously analysed (that is, a node in *pastPrem*). The edge labelling process recursively collects conditions occurring between tasks, as illustrated in Example 2. In case P is an AND-J_r or an OR-J_N node, we update *premises* with P's immediate predecessors ($in_V(\mathrm{P})$). In case P is an OR-J node, we update *conditionLabels* as the previous case, while in case is an AND-J_r node, we also add to *conditionLabels* all the condition-literals present in $A(r)$. Notice that subscript r uniquely identifies the corresponding AND-J node.

Once Algorithm 2 has done collecting such information, the execution returns to Algorithm 1 which invokes Algorithm 3 to discover whether an exclusive choice pattern actually exists. To do so, Algorithm 3 searches for all eligible task nodes. A node T′ is eligible if (i) task-literal t' is in (at least) a positive extension E_j^+ "not used" by any other task in the X-OR pattern. For instance, let us assume we are considering an X-OR pattern between T_1, T_2 and T_3, and we know that there are five extensions. If t_1 is in E_1^+, while t_2 is in E_2^+ and E_3^+, then t_3 must have been proved in E_4^+ or E_5^+.

If T′ is eligible, Algorithm 2 is invoked to compute whether the activation tasks of T′ are the same of T's. If that is the case, Algorithm 3 lastly inspects whether a suitable defeaters' structure exists. The execution now returns to Algorithm 1: if an exclusive choice pattern has been found, the algorithm passes to the second phase and performs the operations described hereafter, otherwise it proceeds in controlling the next candidate node. The computation of tasks in *XORtasks* gives knowledge of where to insert the X-OR SPLIT gate-node (X-OR-$\mathrm{S}_{\mathrm{T}_1,\ldots,\mathrm{T}_n}$ in notation, where T_1–T_n are the

nodes in the exclusive choice pattern). We now need to understand where to insert the X-OR JOIN gate-node (X-OR-$J_{T_1,...,T_n}$ in notation). For each task T_l in *XORtasks* we store in *XJoin* (a set of sets) the task-nodes reached by T_l. If the intersection of such T_ls (*closure*) is not empty, the exclusive choice pattern is well structured; otherwise the declarative specifications were poorly written and, consequently, no X-OR-J can be inserted into the process graph.

Algorithm 3 IFX-OR

```
1: procedure IFX-OR(task T, index i, set TbwTasks)
2:     Xtasks ← {T}, extensions ← {i}
3:     for T' ∈ V \ alreadyXed \ Xtasks s.t. ∃j ∉ extensions. t' ∈ E_j^+(D) do
4:         TjTasks ← BACKWARDTASKSPROJ(T', j)
5:         if TjTasks = TbwTasks then Xtasks ← Xtasks ∪ {T'}, extensions ← extensions ∪ {j}
6:     supp ← ∅
7:     if ∃!T_i ∈ Xtasks s.t. conditionLabels = ∅ then
8:         for T_j ∈ Xtasks with j ≠ i do
9:             if ∃d ∈ R_dft[∼t_j].A(d) = {t_i} then
10:                for T_k ∈ Xtasks with k ≠ i, j do
11:                    if ∃d ∈ R_dft[∼t_j].A(d) = {t_k} then supp ← supp ∪ {T_j}
12:    if (supp ≠ ∅) then return supp
13:    supp ← ∅
14:    for T_j ∈ Xtasks do
15:        for T_k ∈ Xtasks with k ≠ j do
16:            if ∃d ∈ R_dft[∼t_j].A(d) = {t_k} then
17:                if ∃d ∈ R_dft[∼t_k].A(d) = {t_j} then supp ← supp ∪ {T_j}
18:    return supp
```

When *closure* is not empty, we need to remove from it all the gate-nodes along with those nodes that depend on nodes in *closure*. The operations described above serve exactly to this purpose: to eliminate nodes like F from *closure*. Finally, if one of the task in *closure* belongs to *XORtasks* as well, we have an instance of Variant 3. We first identify such a task-node (T_{final} in notation), and then we link the X-OR-$J_{T_1,...,T_n}$ to it (Lines 17–18). In both circumstances, edges from a node in *X join* towards a node in *closure* are erased and substituted to proper connect them to the new inserted X-OR-$S_{T_1,...,T_n}$ node (resp. Lines 18 and 21).

The structure of the algorithms and the operations used in the algorithms indicate that the complexity of the problem investigated in this paper remains polynomial. A thorough analysis is left for future work.

4 Conclusion and related work

In this paper we addressed the theoretical issue of how to synthesise compliant business process models incorporating exclusive choice patterns from declarative agent specifications. We proposed computationally efficient algorithms to merge alternative plans into a single business process model. The suitability of the approach to model real life applications is left for future work.

Our approach departs from the standard BDI architecture and agent programming languages implementing it [5,6], and extensions with norms in several respects [7].

While in the above mentioned approaches the agent has to select predefined plans from a library, we propose that the agent generates on the fly a set of plans to meet the objectives without violating the norms. [8] present *norm-aware agents*; a norm-aware agent can deliberate on its goals, norms, and sanctions before deciding which plan to select and execute. In this respect, our agents are norm-aware. [9] provide an account of goals by integrating BDI failure mechanisms with HTN planning techniques. HTN planing is notoriously undecidable even if no variables are allowed, or PSPACE-hard if some restrictions are given. The main feature of their CAN$^{\mathscr{A}}$ is that, if a plan fails, alternative plans are tried. Compared to theirs, our framework has the advantage that we generate all the possible plans at design time. [10] "force" the notion of obligation within the STRIPS framework for agent planning. Their framework is lacking in at least two aspects if compared to ours: (i) they cannot specify the motivational aspects of BDI agents, (ii) their framework cannot generate alternative plans or process graph as we do.

References

1. Governatori, G., Olivieri, F., Rotolo, A., Scannapieco, S., Cristani, M.: Picking up the best goal - an analytical study in defeasible logic. In Morgenstern, L., Stefaneas, P.S., Lévy, F., Wyner, A., Paschke, A., eds.: RuleML 2013. Volume 8035 of Lecture Notes in Computer Science., Springer (2013) 99–113
2. Governatori, G., Olivieri, F., Scannapieco, S., Rotolo, A., Cristani, M.: The rationale behind the concept of goal. Theory and Practice of Logic Programming (in Press)
3. Olivieri, F., Governatori, G., Scannapieco, S., Cristani, M.: Compliant business process design by declarative specifications. In Boella, G., Elkind, E., Savarimuthu, B.T.R., Dignum, F., Purvis, M.K., eds.: PRIMA 2013. Volume 8291 of Lecture Notes in Computer Science., Springer (2013) 213–228
4. Olivieri, F.: Compliance by design: Synthesis of business processes by declarative specifications. PhD thesis, Griffith University and University of Verona (2015)
5. Dastani, M.: 2APL: a practical agent programming language. Autonomous Agents and Multi-Agent Systems **16**(3) (2008) 214–248
6. Bordini, R.H., Hübner, J.F.: BDI agent programming in agentspeak using *Jason* (tutorial paper). In Toni, F., Torroni, P., eds.: CLIMA VI. Volume 3900 of Lecture Notes in Computer Science., Springer (2005) 143–164
7. Andrighetto, G., Governatori, G., Noriega, P., van der Torre, L.W.N., eds.: Normative Multi-Agent Systems. Leibniz-Zentrum fuer Informatik
8. Alechina, N., Dastani, M., Logan, B.: Programming norm-aware agents. In van der Hoek, W., Padgham, L., Conitzer, V., Winikoff, M., eds.: AAMAS, IFAAMAS (2012) 1057–1064
9. Sardiña, S., Padgham, L.: A BDI agent programming language with failure handling, declarative goals, and planning. Autonomous Agents and Multi-Agent Systems **23**(1) (2011) 18–70
10. Panagiotidi, S., Vázquez-Salceda, J.: Towards practical normative agents: A framework and an implementation for norm-aware planning. In Cranefield, S., van Riemsdijk, M.B., Vázquez-Salceda, J., Noriega, P., eds.: COIN@AAMAS&WI-IAT. Lecture Notes in Computer Science, Springer (2012) 93–109