# Semi-automated Modelling of Optimized BPMN Processes

Yliès Falcone, Gwen Salaün and Ahang Zuo
Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG 38000 Grenoble, France

***Abstract*—In this paper, we propose a semi-automated approach for helping non-experts in BPMN to model business processes using this notation. We focus on a timed version of BPMN where tasks are associated with a range indicating the minimum and maximum duration needed to execute that task. In a first step, the user defines the tasks involved in the process and possibly gives a partial order between tasks. A first algorithm then generates an abstract graph, which serves as a simplified version of the process being specified. Given such an abstract graph, a second algorithm computes the minimum and maximum time for executing the whole graph. The user can rely on this information for refining the graph. For each version of the graph, these minimum/maximum execution times are computed. Once the user is satisfied with a specific abstract graph, we propose a third algorithm to synthesize the BPMN process from that graph.**

## I. Introduction

Business process modelling and optimization is a strategic activity in organizations because of its potential to increase profit margins and reduce operating costs. The Business Process Model and Notation (BPMN) [1] is a graphical modeling language for specifying business processes using a workflow-based notation. BPMN is a rich notation and modelling processes requires expertise. Process mining and discovery techniques are helpful to automatically infer processes from execution logs, but they do not provide a solution to make users more comfortable with BPMN. Moreover, beyond modelling, optimization is a tricky issue, which should be taken into account as soon as possible during the development process. This allows one to build efficient processes, and thus to reduce the process execution time and its associated cost.

In this paper, we propose a semi-automated approach for supporting users in the modelling of business processes with BPMN. The main idea is to start with a rough model of the process-to-be and refine it by introducing further details in the process step by step. The notation used for modelling processes is an abstract directed graph. In this work, we focus on a timed version of BPMN where tasks are associated with a range indicating the minimum and maximum duration needed to execute that task. We can thus compute the minimum and maximum execution time of the process under construction. This information serves as a guide to the user for refining the process, particularly by reducing this execution time and thus the associated costs. When the user is satisfied with a specific version of a graph and its corresponding execution times, the final BPMN process is automatically generated.

Figure 1 overviews our approach. As a first step, we expect from the user that (s)he defines the set of tasks involved in the process. Each task comes with a range of minimum and maximum durations. This means that the duration for executing that task is within the given range. For instance, to fill in some forms for applying for some grant, it takes between 1 and 5 days. Given a set of tasks and a partial order between some of these tasks, we define a first algorithm for automatically generating a first abstract version of the process. The notation used for this description is a notion of abstract graph. We use an abstract graph for modelling purposes because it avoids introducing gateways and possible complex combination of gateways that are necessary for expressing looping behaviour for example. Given such an abstract graph, we define a second algorithm, which computes the minimum and maximum times for executing the whole graph. This information is particularly interesting for optimization purposes and the user can rely on this information for refining the abstract graph. Refinement is a design method, which aims at revising the current graph by adding new nodes or merging existing ones. Adding a new node is necessary for expressing a choice for instance to go to one or another node. Merging existing nodes is useful for adding more parallelism to the process when several tasks can be executed at the same time. When the user is satisfied with a specific abstract graph, (s)he can decide to automatically generate the corresponding BPMN process using a third algorithm. The abstract graph model and the BPMN process are equivalent.

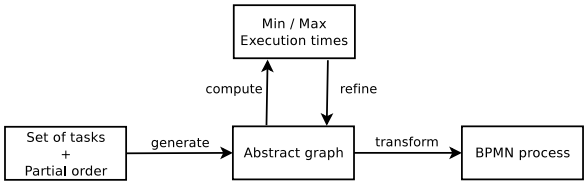


Figure 1. Approach overview

The organization of the rest of the paper is as follows. Section II introduces the subset of the BPMN notation used in this work. Section III defines several models that are central to our approach. Section IV presents several algorithms for supporting the abstract modelling of BPMN processes step by step guided by their minimum and maximum execution time. Section V ends this paper with some concluding remarks.

## II. BPMN with Time

This section introduces the subset of BPMN used in this paper, which focuses on behavioural aspects (start/end events,

tasks, flows, gateways) and time (task duration). A BPMN process is a directed graph with nodes as vertices and sequence flows as directed edges. A node is a start or end event, a task, or a gateway. Start and end events are used to initialize and terminate processes, respectively. A task represents an atomic activity, and has exactly one incoming and one outgoing flow. A gateway is used to control the split patterns (i.e., flow divergence) and merge patterns (i.e., flow convergence) of execution in a process. In this paper, a process is considered to have exactly one start event and at least one end event. The two main gateways available in BPMN are called exclusive and parallel gateways. An exclusive gateway chooses one out of a set of mutually exclusive alternative incoming or outgoing branches. A parallel gateway creates concurrent flows for all its outgoing branches or synchronizes concurrent flows for all its incoming branches. Looping behaviours and unbalanced structure of the process (no strict correspondence between split and merge gateways) are supported in this work. Data to express variables and conditions are not supported, but they can be added to the process at the end of the approach, once we have generated the BPMN skeleton.

In addition to these classic BPMN constructs, time can be associated to tasks and flows. In this work, a duration can be added to a flow. When a flow has a duration $d$ greater than zero, it means that the destination node is triggered after $d$ units of time. If the duration is zero, that node is immediately triggered. As far as tasks are concerned, each task also defines a range of durations corresponding to the minimum and maximum duration it takes to execute that task. Therefore, once a task is triggered, it executes for a duration included in the range of durations defined for that task. Once the task completes, its outgoing flow is triggered.

Figure 2 summarizes the syntax of BPMN presented in this section, including examples of the timing constructs. The informal semantics of BPMN is described in official documents [1], [2] and some attempts have been made to formalize it (e.g., [3]–[7]).

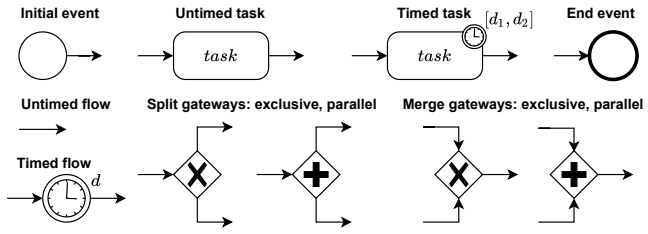


Figure 2. BPMN syntax with time features

## III. Models

In this section, we define several notions and models which are central to our approach for supporting the modelling of BPMN processes. We formally define the notion of task (a.k.a. activity).

*Definition 1 (Task):* A task is a tuple $(Id, Desc, (D_{\min}, D_{\max}))$ where $Id$ is the task identifier (typically a string), $Desc$ is a textual description of the task, and $(D_{\min}, D_{\max})$, such that $D_{\min} \leq D_{\max}$, is a pair of real numbers denoting the interval between time bounds $D_{\min}$ and $D_{\max}$ for executing that task.

We now define a constrained form of partial order between tasks. The partial order consists of a set of tasks $\mathbb{T}$ (all tasks involved in a process) and a relation $O$ between tasks. When two tasks $T \in \mathbb{T}$ and $T' \in \mathbb{T}$ are related (i.e., $(T, T') \in O$), it means that $T$ must execute before $T'$ in the final process. In this case, we say that $T'$ is a successor of $T$. In the following, we use the set of successors of a task $T$, noted $\mathrm{succ}_O(T)$ and defined as $\{T' \in \mathbb{T} \mid (T, T') \in O\}$.

Moreover, relation $O$ between tasks should respect the two following constraints:

- $O$ should be irreflexive, that is no task is related to itself.
- For any task $T'$, if $T'$ is a common successor to two tasks $T$ and $T''$, then $T$ and $T''$ have the same successors. For example, the relation $\{(T_1, T_3), (T_2, T_3), (T_1, T_4), (T_2, T_4)\}$ respects the contraint, because $T_1$ and $T_2$ have the same successors, namely $T_3$ and $T_4$. In contrast, $\{(T_1, T_3), (T_2, T_3), (T_1, T_4)\}$ is not accepted because $T_1$ and $T_2$ do not have the same successors.

Such constraints ensure that it is always possible to generate an abstract graph or business process from a partial order respecting these constraints.

*Definition 2 (Partial order over tasks):* A partial order over tasks is a pair $(\mathbb{T}, O)$ where $\mathbb{T}$ is a set of tasks and $O \subseteq \mathbb{T} \times \mathbb{T}$ is a relation over tasks such that:

$$\begin{aligned} &\forall T \in \mathbb{T} : \\ &\quad (T, T) \notin O \\ &\wedge \quad \big(|O \cap (\mathbb{T} \times \{T\})| \leq 1 \\ &\qquad \vee \forall T_1, T_2 \in \mathbb{T} : (T_1, T) \in O \wedge (T_2, T) \in O \\ &\qquad\qquad \implies \mathrm{succ}_O(T_1) = \mathrm{succ}_O(T_2)\big) \end{aligned}$$

We now define the notion of abstract graph which is used as an abstract representation of a BPMN process. It consists of a set of nodes and a set of directed edges. A node is defined as a set of tasks and a set of graphs. The graph is thus a hierarchical structure. Abstract graphs are simpler than BPMN processes because they do not rely on nodes and gateways, which can result in intricate structures when they are nested or when they express specific behaviours such as loops or unbalanced structures.

*Definition 3 (Abstract graph):* An abstract graph is a (hierarchical) directed graph $(AN, AE)$ where $AN$ is a set of nodes and $AE$ is a set of directed edges connecting these nodes. A node is defined as a pair $(Ts, Gs)$ where $Ts$ is a set of tasks and $Gs$ is a set of abstract graphs.

Note that the above modeling implies that we associate nodes with a parallel execution semantics. This means that all the tasks and graphs in a node execute in parallel. A choice in the graph can be expressed by several edges outgoing from the same node. Looping behaviour can be expressed using an edge going back to an already visited node. We will see in Section IV how such an abstract graph can be automatically

generated from a partial order, and how the maximum and minimum execution times can be automatically computed for such an abstract graph.

Now we formally define BPMN graphs, following the description of BPMN as presented in Section II. We will see in Section IV how a BPMN graph can be automatically generated from an abstract graph.

*Definition 4 (BPMN graph):* A BPMN graph is a directed graph $(BN, BE)$ where $BN$ is a set of BPMN nodes and $BE$ is a set of directed edges connecting these nodes. A BPMN node is either a start node, a task node, a split node, a merge node, or an end node. Split and merge nodes also come with a type, which is either exclusive or parallel.

## IV. From Partial Order to BPMN

In this section, we detail the three algorithms supporting our approach. The first one generates an initial abstract graph from a set of tasks and a partial order between these tasks (section IV.A). The second algorithm computes the maximum and minimum execution times given an abstract graph (section IV.B). The third algorithm transforms an abstract graph into a BPMN model (section IV.D). In addition, we also define how an abstract graph can be iteratively revised using refinement techniques (section IV.C).

### A. Partial Order to Initial Graph

We present the algorithm which takes a set of tasks $T$ and a set of pairs of tasks $O$ as input, and generates as output the initial abstract graph $AG$ consisting of a set of nodes $AN$ and a set of directed edges $AE$. The algorithm first creates a directed flat graph from the partial order. Then, it finds the nodes (tasks) in the directed graph through which all paths pass. It partitions the directed graph according to the locations of these nodes (tasks). Finally, it recursively processes these subgraphs and returns a hierarchical abstract graph.

Before the algorithm starts, the partial order provided as input is checked to be sure it meets the constraints described in the previous section. Only when such constraint is satisfied the algorithm starts to execute.

Algorithm 1 first initialises the two sets in the output abstract graph $AG = (AN, AE)$: $AN$ is the set of nodes of the abstract graph and $AE$ is the set of directed edges of the abstract graph (line 1). Then, the algorithm transforms the input partial order into a directed flat graph $G$, where the set of nodes consists of the individual tasks (one task per node, without subgraphs), and the set of edges is a set of relations between tasks in $O$. A flat graph may contain several start nodes. Next, the algorithm calls the recursive function $\mathrm{processGraph}$ to partition the graph into different subgraphs. To do so, function $\mathrm{processGraph}$ (lines 5 to 24) first uses function $\mathrm{findSharedTasks}$ to compute the set of so-called shared tasks, which are the common tasks that occur in any path from each start node to the final nodes in a directed graph. The set of shared tasks is stored in $sharedTasks$. This function distinguishes two cases depending on whether $sharedTasks$ is empty or not:

**Algorithm 1** Partial Order to Initial Graph (POToGraph)

**Input:** Partial Order $(\mathbb{T}, O)$:
**Output:** Abstract Graph $AG = (AN, AE)$

```
1:  AN ← ∅, AE ← ∅
2:  G ← (T, O)
3:  AN, AE ← processGraph(G)
4:  return AG = (AN, AE)
5:  function processGraph(Graph)
6:      Nodes ← ∅, Edges ← ∅, subGraphs ← ∅
7:      sharedTasks ← findSharedTasks(Graph)
8:      if sharedTasks ≠ ∅ then
9:          subGraphs ← splitGraph(Graph, sharedTasks)
10:         for each graph in subGraphs do
11:             Nodes ← Nodes ∪ createNode(graph)
12:             processGraph(graph)
13:         end for
14:     else
15:         node ← getNode(Graph)
16:         node.Ts ← getIsoNodes(Graph)
17:         subGraphs ← filterGraph(Graph)
18:         for each graph in subGraphs do
19:             node.Gs ← node.Gs ∪ processGraph(graph)
20:         end for
21:     end if
22:     Edges ← Edges ∪ Edge(Nodes)
23:     return Nodes, Edges
24: end function
```

- If $sharedTasks$ is not empty, the algorithm iterates over $sharedTasks$ and calls function $\mathrm{splitGraph}$, which splits the graph into subgraphs. Each shared task appears alone in a specific node. Function $\mathrm{splitGraph}$ works as follows: All tasks before the first shared task are gathered in a subgraph. The first shared task is isolated in a specific node. The tasks between the two shared tasks are gathered within another node, and so on. For each subgraph, a node is created and then function $\mathrm{processGraph}$ is called recursively on the graph contained in this new node.
- If $sharedTasks$ is empty, the algorithm first uses function $\mathrm{getNode}$ to get the node that contains this graph, and then operates on this node (line 15). That node, according to its definition, consists of a pair $Ts$ and $Gs$. Function $\mathrm{getIsoNodes}$ detects all isolated nodes, which are not connected to any node in the graph, and stores them in the set of tasks $Ts$ of the node $node$ ($node.Ts$, line 16). The algorithm then uses function $\mathrm{filterGraph}$, which is used to remove these isolated nodes from the node, and returns all remaining subgraphs (line 17). Finally, the algorithm calls function $\mathrm{processGraph}$ recursively on these subgraphs and stores them in the set of abstract graphs $Gs$ of the node ($node.Gs$, lines 18 to 20).

At the end of function $\mathrm{processGraph}$, function $\mathrm{Edge}$ traverses the set of nodes and adds a directed edge to the set of nodes between adjacent nodes in the set of edges (line 22). When the call to $\mathrm{processGraph}$ ends, the algorithm finally

returns the abstract graph $AG$.

This algorithm traverses all tasks, and for each task, it traverses its subsequent tasks. Therefore, the time complexity of Algorithm 1 is $O(n^2)$, where $n$ is the total number of tasks.

*Example 1:* Let us illustrate this step of our approach with an employee recruitment process. This process focuses on the different tasks to be carried out once the employee has successfully passed the interview. The employee has to complete some paperwork. (S)He has to see the doctor for a medical check-up. If the employee needs a visa, (s)he should also apply for a working visa. At some point, (s)he should submit all documents. If these documents are not satisfactory, the company may ask for them again. If everything is fine, all documents are accepted as are. All provided documents are properly archived once validated. The employee is also added to the personnel database and Human Resources (HR) anticipate wage payment while an assistant prepares the welcome kit (office, badge, keys, etc.).

According to this short description of the expected process, the user (someone from the HR staff for example) first needs to define the corresponding tasks and gives an approximate duration for each task as follows:

- $T_1$: "Fill-in form", [1 day - 2 days]
- $T_2$: "Medical check-up", [1 day - 5 days]
- $T_3$: "Visa application", [7 days - 14 days]
- $T_4$: "Submit documents", [1 day - 2 days]
- $T_5$: "Documents accepted", [1 day - 2 days]
- $T_6$: "Documents rejected", [1 day - 2 days]
- $T_7$: "Archive all documents", [1 day - 3 days]
- $T_8$: "Update personnel database", [1 day - 2 days]
- $T_9$: "Anticipate wages", [3 days - 10 days]
- $T_{10}$: "Prepare welcome kit", [3 days - 5 days]

The user can then define an order between some of these tasks. In the case of this example, the following ordering constraints are defined by the user:

- submitting documents can only appear after filling forms, medical check-up, and visa application $\rightsquigarrow (T_1, T_4)$, $(T_2, T_4)$, and $(T_3, T_4)$;
- documents are accepted or rejected once they have been submitted $\rightsquigarrow (T_4, T_5)$ and $(T_4, T_6)$;
- archiving documents, updating database, anticipating wages, and preparing welcome kit can appear only after validation of the documents $\rightsquigarrow (T_5, T_7)$, $(T_5, T_8)$, $(T_5, T_9)$, and $(T_5, T_{10})$;
- updating personnel database should be executed before anticipating wages and preparing welcome kit $\rightsquigarrow (T_8, T_9)$ and $(T_8, T_{10})$.

When Algorithm 1 takes this data (set of tasks and set of pairs) as input, it returns the abstract graph given in Figure 3. Algorithm 1 works as follows. First, it detects that tasks $T_1$, $T_2$, and $T_3$, have as common successor task $T_4$. Then $T_4$ is the shared task, which splits the entire directed graph into two parts. Therefore, the algorithm creates a node to store $T_1$, $T_2$, and $T_3$ and a second node to store $T_4$. Since $T_4$ has two successor tasks $T_5$ and $T_6$, the algorithm then creates a new node to store them. Since $T_5$ has some successor tasks, and $T_6$ does not have any successor task, the algorithm creates a subgraph in this new node, and moves $T_5$ to this subgraph. The rest of the algorithm execution extends this subgraph to integrate the remaining tasks and finally returns the graph given in Figure 3.

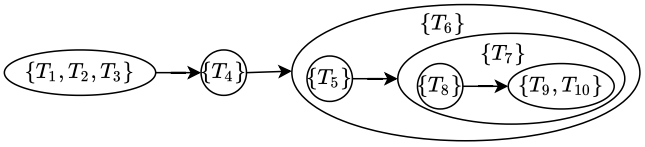


Figure 3. Initial abstract graph generated from partial order

### B. Minimum/Maximum Time computation

In this section, we describe an algorithm that takes as input an abstract model and a value of $k$ ($k \in \mathbb{N}^+$), which is the maximum number of times that each node of the abstract graph can be traversed in the presence of cycles. The algorithm then computes the maximum and minimum execution times for this abstract graph. The algorithm begins from the start node of the abstract graph and relies on a depth-first search/traversal. The algorithm completes when reaching the final node(s) of the abstract graph and then computes the maximum and minimum execution times.

**Algorithm 2** Min/Max Time Computation (CompTimes)

**Input:** Abstract Graph $AG = (AN, AE), k$
**Output:** $T_{\min}, T_{\max}$

```
1:  T_max ← 0, T_min ← ∞
2:  start_node ← startNode(AG)
3:  if succNodes(start_node) is empty then
4:     T_min ← minTimeNode(start_node)
5:     T_max ← maxTimeNode(start_node)
6:     return T_min, T_max
7:  else
8:     for each succ_node in succNodes(start_node) do
9:        incCounter(succ_node)
10:       if getCounter(succ_node) ≤ k then
11:          mini, maxi ← CompTimes(succ_node, k)
12:          T_min ← min(T_min, mini)
13:          T_max ← max(T_max, maxi)
14:       end if
15:    end for
16:    T_min ← minTimeNode(start_node) + T_min
17:    T_max ← maxTimeNode(start_node) + T_max
18:    return T_min, T_max
19: end if
```

In Algorithm 2, $T_{\min}$ and $T_{\max}$ are first initialized to infinite and zero, respectively (line 1). The algorithm uses function startNode to obtain the initial node of the graph. Then, it detects whether the start node of the abstract graph has successor nodes. If it does not, then $T_{\min}$ and $T_{\max}$ of the abstract graph

are respectively the minimum and maximum execution time of that node (lines 3 to 6). Function minTimeNode computes the minimum execution time of a node. Similarly, function maxTimeNode computes the maximum execution time of a node. Note that functions minTimeNode and maxTimeNode are in charge of computing minimum and maximum times of subgraphs by calling again function compTimes. If the start node has successor nodes, the algorithm traverses all its successor nodes. The algorithm uses the node counters to avoid infinite computations when there are cycles in the graph. Each node is equipped with a counter, which is initialised to zero. When traversing a node, its counter is incremented (incCounter, line 9). Then, we check whether the counter value for this node does not exceed $k$ (line 10). The computation of all subsequent nodes relies on a recursive call (line 11). According to the depth-first search algorithm, the algorithm explores all the paths from the initial node to the final node(s) of the graph. Once these final node(s) are reached, it goes backwards and computes the minimum and maximum execution times. At the end, $T_{\min}/T_{\max}$ correspond to the minimum/maximum execution time of the initial node for that graph plus the minimum/maximum execution time of all its subsequent nodes (lines 16 and 17).

It is worth noting that when we calculate the minimum execution time of a node, because the elements (tasks, subgraphs) of the node are executed in parallel, it is equal to the element that takes the longest execution time.

As far as computational complexity is concerned, all nodes need to be traversed once, including the nodes appearing in subgraphs, so the complexity of this algorithm is $O(n)$ where $n$ is the total number of nodes.

*Example 2:* We recall that the tasks introduced in Example 1 have the following ranges of durations: $T_1 : [1, 2]$, $T_2 : [1, 5]$, $T_3 : [7, 14]$, $T_4 : [1, 2]$, $T_5 : [1, 2]$, $T_6 : [1, 2]$, $T_7 : [1, 3]$, $T_8 : [1, 2]$, $T_9 : [3, 10]$, and $T_{10} : [3, 5]$. Algorithm 2 takes the graph built for this example given in Figure 3 as input and returns 13 as minimum execution time and 30 as maximum execution time. This abstract graph contains three nodes. The algorithm first computes the minimum/maximum execution time of the nodes. These times for the first node are 7 and 14, 1 and 2 for the second node, and 5 and 14 for the last node. Since the nodes in the graph are sequentially executed, the execution times of these three nodes are summed to compute the final result.

### C. Abstract Graph to Abstract Graph

In this section, we describe how the user can refine an abstract graph into another one. This part of the method cannot be automated because only the user knows the goal of the process-to-be and should decide to make some specific changes to reach that goal. In this section, we explain the possible changes that can be applied on a given abstract graph. Note that this refinement step is guided by the minimum and maximum execution times computed by Algorithm 2. Indeed, these changes aim at revising the abstract model to get closer to the final one and to obtain better results in terms of execution time of the whole process.

The possible changes to transform an abstract graph can be described with three possible operations:

- addition of a node to the graph;
- suppression of an existing node in the graph;
- move of a task or subgraph from one node to another node.

By combining these three operations, one can achieve more complicated updates. For instance, to merge two nodes, the user should first move the contents of one node to the other and then suppress the empty node. To transform a node with several tasks to a choice, the user should first create two new nodes, and move parts of the contents of the existing node to these new nodes. It is worth mentioning that beyond creation/suppression of nodes, the user can also add and remove edges.

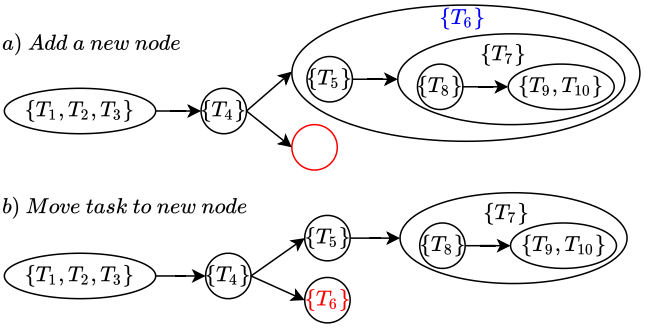


Figure 4. Example of the refinement for the running example

*Example 3:* Consider the abstract graph returned by Algorithm 1 shown in Figure 3. For this abstract graph, the refinement process consists of two steps shown in Figure 4. Tasks $T_5$ and $T_6$ (documents accepted or rejected) should appear in two different nodes. To do so, in a first step, we add a new node in the graph. In a second step, we move the task $T_6$ to this new node. We then recompute the execution times of this new abstract graph. Its minimum execution time is 9 and its maximum execution time is 30. After the refinement step, the maximum time is the same, but the minimum time has improved (going from 13 to 9).

### D. Abstract Graph to BPMN

In this section, we present the algorithm that takes the abstract graph as input, and returns the corresponding BPMN model. The main idea of this algorithm is to detect specific structures in the abstract graph and map them to BPMN constructs. For instance, a node with several outgoing edges is transformed into an exclusive gateway in BPMN. The algorithm works by transforming one node after the other, and finally returns the corresponding BPMN process.

Algorithm 3 traverses the set of nodes in the abstract graph (line 1), and adds start and end nodes only for the first call

**Algorithm 3** Abstract Graph to BPMN (AbsToBPMN)

**Input:** Abstract Graph $AG = (AN, AE)$
**Output:** BPMN Graph $BG = (BN, BE)$

```
1: for each node in AN do
2:   if isNotSubGraph(AG) then
3:     if inDegree(node) == 0 then
4:       startBG(node)
5:     end if
6:   end if
7:   if outDegree(node) == 0 then
8:     endBG(node)
9:   end if
10:  if outDegree(node) ≥ 2 then
11:    excluSplitBG(node)
12:  end if
13:  if inDegree(node) ≥ 2 then
14:    excluJoinBG(node)
15:  end if
16:  if elements(node) == 1 then
17:    seqBG(node)
18:  else
19:    parallBG(node)
20:  end if
21: end for
22: return BG
```

of function AbsToBPMN. This function is also called on subgraphs but in that case, there is no addition of start and end nodes. So if this is the main graph, which is not a subgraph (line 2) and the number of incoming edges (inDegree) for a node is equal to zero, function startBG adds a start event before this node and creates an edge between them (lines 3 to 5). If the number of outgoing edges (outDegree) for a node is equal to zero, function endBG adds an end event after this node and creates an edge between them (lines 7 to 9). When the outDegree value of a node is greater than 2, function excluSplitBG adds a split exclusive gateway between this node and the subsequent nodes (lines 10 to 12). When the algorithm adds a new gateway, the corresponding function also adds related directed edges between nodes or elements (tasks, subgraphs) and gateways. Similarly, when the inDegree value of a node is greater than 2, function excluJoinBG adds a join exclusive gateway between its predecessor nodes (lines 13 to 15). The last part of the algorithm deals with the internals of a node. There are two cases. When there is only one element in the node, it can only be a task, then function seqBG generates a BPMN task, which is executed sequentially with its predecessor node (line 17). When the node contains multiple elements (tasks or graphs), function parallBG inserts a parallel gateway before (split) and after (join) the elements of this node (line 19). This function also generates a BPMN task for each task appearing in that node, and calls again function AbsToBPMN for each subgraph appearing in that node. All required BPMN flows/edges are also inserted by function parallBG.

This algorithm needs to traverse all the nodes in the graph. Therefore, the time complexity of the algorithm is $O(n)$, where $n$ is the total number of nodes in the graph.

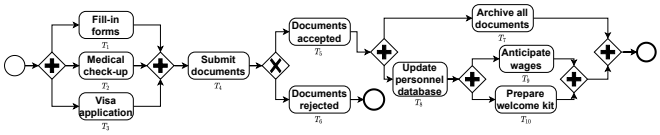


Figure 5. Example of generated BPMN process

*Example 4:* Let us use the abstract graph given in Figure 4 (bottom) for illustration purposes. As a result, the algorithm generates the BPMN process shown in Figure 5. We can see that the first node with three tasks ($T_1$, $T_2$, $T_3$) transforms to a split and a join parallel gateway. The algorithm also generates a split exclusive gateway right after $T_4$ because the corresponding node has two outgoing edges. After $T_5$, there is again several parallel gateways because there are multiple tasks within the same node.

## V. Concluding Remarks

To facilitate the understanding of BPMN notation and the semantics of its underlying control-flow graph, we define a semi-automated approach that supports end users when modeling BPMN. We consider a subset of BPMN consisting of the usual nodes and gateways extended with time durations for flows and tasks. Our solution relies on a simple notation for describing an abstract version of a process called abstract graph. By defining the set of tasks and a partial order between some of them, a first algorithm generates an abstract graph. The user can then successively refine this abstract graph. Refinement is guided by the minimum and maximum execution times needed for executing the whole process. Once the user is satisfied, a last algorithm transforms this graph into a BPMN process. The main perspective of this work is to complete the implementation of the tool supporting this approach.

**Acknowledgements.** This work was supported by the Région Auvergne-Rhône-Alpes within the "*Pack Ambition Recherche*" programme.

## References

[1] ISO/IEC, "International Standard 19510, Information technology – Business Process Model and Notation," 2013.
[2] *Business Process Model and Notation (BPMN) – Version 2.0*, OMG, january 2011.
[3] R. Dijkman, M. Dumas, and C. Ouyang, "Semantics and Analysis of Business Process Models in BPMN," *Information and Software Technology*, vol. 50, no. 12, pp. 1281–1294, 2008.
[4] P. Wong and J. Gibbons, "A Process Semantics for BPMN," in *Proc. of ICFEM'08*, ser. LNCS, vol. 5256. Springer, 2008, pp. 355–374.
[5] L. E. M. Morales, M. I. Capel, and M. A. Pérez, "Conceptual Framework for Business Processes Compositional Verification," *Information and Software Technology*, vol. 54, no. 2, pp. 149–161, 2012.
[6] F. Durán, C. Rocha, and G. Salaün, "Stochastic Analysis of BPMN with Time in Rewriting Logic," *Sci. Comput. Program.*, vol. 168, pp. 1–17, 2018.
[7] A. Krishna, P. Poizat, and G. Salaün, "Checking Business Process Evolution," *Sci. Comput. Program.*, vol. 170, pp. 1–26, 2019.