

# Automated Generation of BPMN Processes from Textual Requirements

Quentin Nivon  Gwen Salaün   
quentin.nivon@inria.fr\*, gwen.salaun@inria.fr

Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, F-38000 Grenoble France

**Abstract.** Modelling and designing business processes has become a crucial activity for companies in the last 20 years. As a consequence, multiple workflow modelling notations were proposed. Business Process Modelling Notation (BPMN) is one of them and is now considered as the *de facto* standard for process modelling. The BPMN notation offers a rich syntax that requires a certain level of expertise before being able to write correct and well-structured processes corresponding to some expected requirements. The BPMN modelling phase can thus be tedious and error-prone if carried out by non-experts. The main goal of the approach presented in this paper is to help users modelling BPMN processes. To do so, the approach takes as input the requirements of the user in a textual format informally describing the tasks and their ordering constraints, and generates as output a BPMN process satisfying them. The solution has been implemented as a tool that was applied on a large number of examples for evaluation purposes.

## 1 Introduction

Developing and controlling business processes has become a major activity for companies, since this is crucial in order to improve productivity and reduce costs. A first and mandatory step in that direction is the modelling and the design of business processes. Several workflow-based notations have emerged for processes, one of them being the Business Process Modelling Notation (BPMN) [9]. BPMN has become the standard notation for modelling business processes, and many graphical tools have been developed for supporting BPMN modelling.

The BPMN syntax is quite rich and offers a large number of constructs. There is therefore a need to learn the BPMN notation in order to be able to write syntactically and semantically correct processes. Modelling business processes with BPMN thus remains a tedious and error-prone task for non-experts. Moreover, the existing modelling tools allow a lot of freedom in the design of the processes, and do not systematically provide integrated solutions for asserting their correctness.

The approach proposed in this paper aims at helping users during the BPMN modelling phase. To do so, it takes as input a textual description of the process. From these textual requirements, the approach automatically generates the corresponding BPMN process. This solution is useful for non-experts since it provides a way to specify BPMN processes without mastering the intricacies of the notation. It is also helpful for experts

because it simplifies the modelling step by generating BPMN processes automatically, thus avoiding the burden of graphically writing the entire workflow step by step.

More precisely, the textual description is converted into its corresponding BPMN process in three main steps. In the first step, the capabilities of GPT-3.5<sup>1</sup> are used to extract the tasks and their ordering constraints from the description written by the user. To provide interpretable results, GPT-3.5 was fine-tuned in order to make it capable of converting the textual ordering constraints into an internal format similar to regular expressions [11]. Each expression returned by GPT-3.5 is then transformed into its corresponding abstract syntax tree [13]. In a second step, several operations are applied on the generated abstract syntax trees in order to gather the task ordering constraints that they contain into a single abstract syntax tree. Finally, the resulting abstract syntax tree is translated into a BPMN process and returned to the user. The whole approach is fully automated by a tool that was applied on multiple descriptions written by several users (novices or experts) to measure the quality of the generated results.

Section 2 introduces several notions on which the approach relies. Section 3 provides a description of the different steps necessary to transform text to BPMN. Section 4 describes the tool support and some experimental results used to validate the approach. Section 5 compares the solution to related work, and Section 6 concludes this paper.

## 2 Preliminaries

### 2.1 BPMN

BPMN 2.0 (BPMN, as a shorthand, in the rest of this paper) was published as an ISO/IEC standard in 2013 and is nowadays extensively used for modelling and developing business processes. This paper focuses on activity diagrams including the BPMN constructs related to control-flow modelling and behavioural aspects.

Specifically, the node types event, task, and gateway, and the edge type sequence flow are considered. Start and end events are used, respectively, to initialize and terminate processes. A task represents an atomic activity that has exactly one incoming and one outgoing flow. A sequence flow connects two nodes executed one after the other in a specific execution order. Gateways are used to control the divergence and convergence of the execution flow. In this work, the two main kinds of gateways used in activity diagrams are considered, namely, exclusive and parallel gateways. Gateways with one incoming branch and multiple outgoing branches are called splits, e.g., split parallel gateway. Gateways with one outgoing branch and multiple incoming branches are called merges, e.g., merge parallel gateway. A parallel gateway creates concurrent flows for all its outgoing branches or synchronizes concurrent flows for all its incoming branches. An exclusive gateway chooses one out of a set of mutually exclusive alternative incoming or outgoing branches. Such gateways can also be used to represent repetitive behaviours (i.e., loops).

---

<sup>1</sup> <https://www.openai.com/>

## 2.2 Textual Requirements

In this work, the user must provide as input a textual representation of a business process that is going to be generated. This description is informal, in the sense that no prerequisite is required to be able to write a valid description. In this description, the user describes in natural language the tasks that the process should contain, along with their ordering constraints. For instance, if two tasks must be executed one after the other in the final process, this should be stated in the description. To improve the results, the user is advised to name the tasks that should appear in the process. However, this is not mandatory for the approach to work.

*Running example.* The following description, in which the tasks that should appear in the resulting process have been named, describes the opening of a bank account: “First, the banker either *CreateProfile* (*CP*) for the user, or, if it is not needed, he *RetrieveCustomerProfile* (*RCP*) which triggers the system to perform the *AnalyseCustomerProfile* (*ACP*) task. Then, the user executes the task *ReceiveSupportDocuments* (*RSD*) so that the system can start *UpdateInfoRecords* (*UID*) and perform a *BackgroundVerification* (*BV*). If the verification finds missing or incorrect information, the system *RequestAdditionalInfo* (*RAI*) to the user, who has to *ReceiveSupportDocuments* (*RSD*) again. Otherwise, the process ends with *CreateAccount* (*CA*).” As the reader can see, the specification is rather informal, except that names are given to the tasks that should appear in the process. Also, the specification can be written in various styles, with or without context around the important information. It is worth noting that the acronyms written between parenthesis are not mandatory, and are just presented here to shorten the size of the future examples.

## 2.3 Task Ordering Constraints

To generate a BPMN process from a textual description, one of the intermediate steps consists in extracting tasks ordering constraints or dependencies from the text. Several works, such as [6], make use of sequential constraints written as couples to represent ordering constraints between tasks. However, such constraints are limited in the sense that they only allow one to represent sequence and parallelism. In this work, not only sequential and parallel constraints are supported, but also exclusive choices, and looping behaviours. Although allowing a greater expressiveness, these new constraints require a more powerful language than couples of tasks. The language chosen in this approach to capture the supported subset of the BPMN syntax can be seen as a variant of the language of regular expressions. This language defines several usual operators, such as the ‘|’ operator which symbolises an exclusive choice, the ‘&’ operator which symbolises the parallelism (i.e., two elements that can be executed simultaneously), or the ‘<’ operator which symbolises the sequential dependency (i.e., an element must be executed before another one). This language also has a loop operator ‘\*’ that encloses element that can be repeated. Moreover, as in BPMN, loops are split into two parts. The first part, that is necessarily executed, is represented inside a loop using the ‘+’ operator. It corresponds to the part of a BPMN loop that is located between the exclusive merge gateway and the exclusive split gateway. The second part, that is optionally executed, is represented inside a loop using the ‘?’ operator. It corresponds

to the part of a BPMN loop that is located between the exclusive split gateway and the exclusive merge gateway. It is worth noting that these two loop operators are used internally but can not appear in the generated expressions. Finally, the operator ‘,’ is used to list elements that are constrained to each other. Expressions written in this language must obey the rules of the following Backus-Naur Form (BNF) grammar:

$$\begin{aligned}\langle E \rangle &::= t \mid (\langle E \rangle) \mid \langle E_1 \rangle \langle op \rangle \langle E_2 \rangle \mid (\langle E_1 \rangle)^* \\ \langle op \rangle &::= ' \mid '& \mid '&' \mid '<' \mid ',',\end{aligned}$$

where  $t$  is a terminal symbol representing the name of a task. This language has priority between operators ( $' > '& > '< > ',' > '*'$ ) and is right-associative. It is worth noting that this language suffices to capture the subset of the BPMN syntax supported in this work.

*Example.* Let us consider the textual description proposed in the example of Section 2.2. By analysing it, one can generate three expressions capturing all the tasks ordering constraints that it contains: (i)  $(RCP < ACP) \mid CP$ , (ii)  $(RCP, ACP, CP) < (RSD < (UIR, BV))$  and (iii)  $(UIR, BV) < ((RAI < RSD) \mid CA)$ . As an illustration, expression (i) means that either  $RCP$  should be executed before  $ACP$ , or  $CP$  is executed alone.

## 2.4 Abstract Syntax Tree

Due to the properties of the language presented above (priority of operators, right associativity, ...), abstract syntax trees (ASTs) were chosen to represent and manipulate the expressions written in this language. Indeed, abstract syntax trees are suitable to represent hierarchical priorities between operators while allowing powerful recursive computations. An abstract syntax tree is a regular tree representing the abstract syntactic structure of a text written in a formal language.

**Definition 1.** *(Tree) A tree is a set of nodes  $S_N$  and edges  $S_E$  where  $S_E \subseteq S_N \times S_N$ . An edge between two nodes is represented as  $n \rightarrow n' \in S_E$ . A tree has exactly one root node that has no incoming transition, i.e.,  $\exists!n \in S_N$  such that  $\forall n_p \in S_N, \nexists e_p = n_p \rightarrow n \in S_E$ . The other nodes have exactly one predecessor, i.e.,  $\forall n_i \in S_N \setminus \{n\}, \exists!n_{i-1}, n_{i-1} \neq n_i$ , such that  $n_{i-1} \rightarrow n_i \in S_E$ . Finally, each node can have 0, 1, or several successor nodes.*

*Example.* Figure 1 presents the ASTs generated from the expressions corresponding to the running example. As the reader can see, the priority between the operators of the expressions is represented by the hierarchy of nodes in the corresponding AST. For instance, in Figure 1(a), nodes  $RCP$  and  $ACP$  are below a ‘<’ node meaning that they are both in sequence. Moreover, their position (i.e.,  $RCP$  on the left and  $ACP$  on the right) is important as it indicates the direction of the sequential dependency (i.e., if  $RCP$  is executed before  $ACP$  or the opposite). Finally, the ‘<’ node is below the ‘|’ node, meaning that there is a mutual exclusion between both  $RCP$  and  $ACP$  and the node  $CP$ .

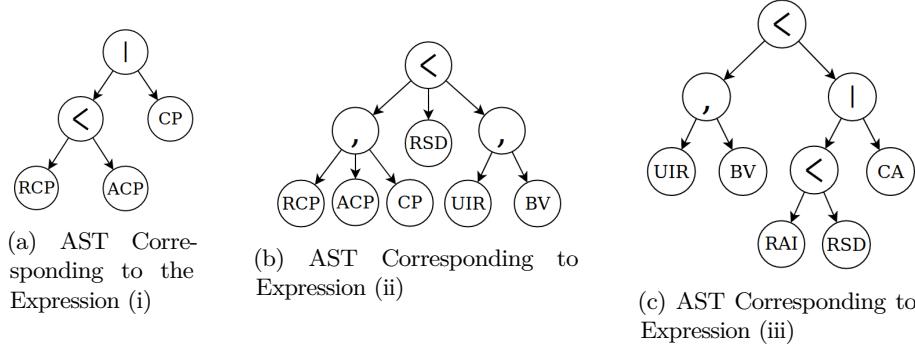


Fig. 1: ASTs Corresponding to the Expressions of Section 2.3

## 2.5 GPT

Inferring ordering constraints from a textual description is not an easy task, as it requires complex mechanisms to understand the structure of the text, extract its components (i.e., the tasks), and discover the multiple relationships connecting them. In this work, Large Language Models (LLMs) are used to perform this analysis, and more precisely, the GPT model. GPT, which stands for Generative Pre-trained Transformer, is an open-access generative model developed by OpenAI, and freely accessible through the well-known website ChatGPT. GPT, and more precisely its “3.5-turbo-0125” version, is used in this work for its natural language processing capabilities, that are helpful to produce expressions corresponding to the language presented in Section 2.3.

## 3 Core of the Approach

The approach proposed in this paper starts by submitting the textual requirements of the user to a fine-tuned version of GPT and asks it to convert them into expressions corresponding to the language defined in Section 2.3. Each resulting expression is then parsed and mapped to its corresponding AST. When multiple expressions are returned by GPT, and thus multiple ASTs are generated, it is not straightforward to compute the corresponding BPMN process. This is due to the fact that the generated BPMN process should contain all the information scattered among the generated ASTs. To merge this information, the solution proposed in this paper consists in analysing the generated ASTs in order to produce a single AST containing all the information of the original ones. This AST can then be converted to its equivalent BPMN process. Figure 2 illustrates these multiple steps, which are detailed in the rest of this section, except for the parsing and mapping of the expressions to their corresponding ASTs, which is straightforward.

### 3.1 Fine-tuned GPT 3.5

The standard version of the GPT 3.5 model has no knowledge about the expected output language defined in Section 2.3. Thus, it is not straightforward for it to generate

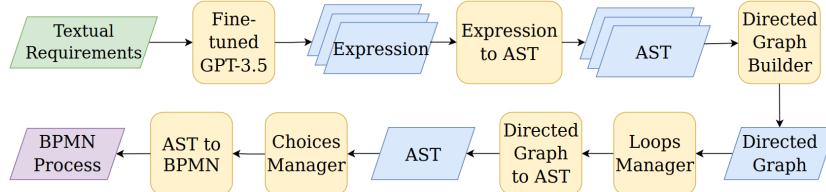


Fig. 2: Overview of the Approach

expressions corresponding to this language. One of the opportunities of GPT is to allow fine-tuning in order to increase its capabilities in precise fields. Roughly speaking, fine-tuning is an approach consisting in improving a model by training it on new data. Often, fine-tuning becomes a tedious task as adjusting hyper-parameters and providing sufficient data can be rather complex and time-consuming. Hopefully, GPT proposes intuitive and easy-to-follow fine-tuning options, which do not require large amount of data to get started. This phase, which can be repeated at any time to improve the quality of the results, has for now been performed on four hundred examples. It is worth noting that the fine-tuning was performed on the version 3.5 of GPT, as more recent models (such as GPT-4, GPT-4-o or GPT-4-turbo) are not yet available for fine-tuning. The training examples provided to GPT consist of three elements. The first one is a system prompt, which describes the expected behaviour of GPT (i.e., the fact that GPT should extract task ordering constraints from the textual requirements given as input) and the shape that its output should take. The second one is a user prompt usually corresponding to the question asked by the user (i.e., the textual requirements here). The last one is an assistant prompt, corresponding to the answer that GPT should provide. For the system prompt and the assistant prompt, the expected output is a set of expressions corresponding to the language defined in Section 2.3. Once training and validation data are given to GPT, the fine-tuning process starts automatically. When the fine-tuning finishes, it outputs a new version of the model that can be used by its owner. After this fine-tuning phase, the generated model was able to transform textual requirements into expressions. It is worth noting that, for a single textual description, GPT may generate several expressions to represent all the task ordering constraints that it found. It is for instance the case for the description “A before C, B before C, B before D” that can not be represented with a single expression. It requires at least two expressions, such as  $(A, B) < C$  and  $B < D$ . Indeed, a single expression, such as  $(A, B) < (C, D)$  would add the unnecessary constraint  $A < D$ .

### 3.2 Directed Graph Construction

To gather the information scattered in the generated ASTs, the first step consists in building the skeleton of the process-to-be. This is done by extracting the sequential constraints stored in the ASTs, in order to build a unique directed graph comprising all of them. This extraction is performed by a classical depth-first search algorithm, which traverses each original AST and creates couples of tasks ( $T_1, T_2$ ) for each

node of the AST containing a ‘<’ operator. Once this algorithm terminates, the generated couples are analysed by another algorithm in charge of generating a directed (possibly cyclic) graph corresponding to the extracted sequential constraints. The generated graph is then transitively reduced using classical algorithms [1].

*Example.* Let us consider the ASTs shown in Figure 1. The first algorithm extracts the following couples from them:  $(RCP, ACP)$ ,  $(RCP, RSD)$ ,  $(ACP, RSD)$ ,  $(CP, RSD)$ ,  $(RSD, UIR)$ ,  $(RSD, BV)$ ,  $(BV, RAI)$ ,  $(BV, CA)$ ,  $(UIR, RAI)$ ,  $(UIR, CA)$ ,  $(RAI, RSD)$ . From these couples, the second algorithm generates the directed cyclic graph given in Figure 3. It is worth noting that some sequential constraints, such as  $(RCP, RSD)$ , have been suppressed by the transitive reduction algorithm.

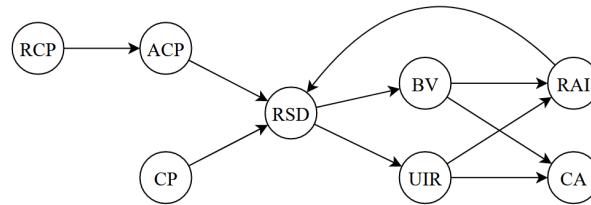


Fig. 3: Directed Cyclic Graph Corresponding to the ASTs in Figure 1

### 3.3 Loops Management

The directed graph generated in the previous step may be cyclic, which indicates loops in the process-to-be. As such structures generate complexity in the graph, they are removed from the graph (i.e., the graph is made acyclic), and all the information required to recreate them is stored in internal structures. In BPMN, a loop can be represented using four elements: its entry nodes, its exit nodes, its mandatory paths, and its optional paths. The entry nodes are the first reachable nodes of the graph belonging to the loop. The exit nodes are the first reachable nodes of the loop having at least one child node not belonging to the loop. Finally, the mandatory (resp. optional) paths are all the paths starting with the entry (resp. exit) nodes and ending with an exit (resp. entry) node. The computations of these four elements is presented below.

- (i) To identify the entry node(s) of the loop, the graph is traversed in a depth-first way. During the traversal, each node is analysed to detect whether it belongs to a loop or not. If that is the case, it is marked as entry point and the exploration stops for the current branch.
- (ii) Similarly to step (i), the exit nodes are computed using a depth-first traversal of the graph. For each node belonging to the loop, the algorithm checks whether it has at least one child node that does not belong to the loop. If that is the case, the current node is marked as exit point and the exploration stops for the current branch.

- (iii) Similarly to the previous steps, a depth-first traversal of the graph between the entry (resp. exit) node(s) and the exit (resp. entry) node(s) suffices to compute all the mandatory (resp. optional) paths.

Once these elements are computed, the graph is made acyclic by removing all the incoming transitions of the entry node(s) of the loop coming from nodes belonging to the loop.

*Example.* Figure 4 depicts the four steps presented above. Figure 4(a) presents the result of the computation of the entry nodes on the graph in Figure 3. *RSD* is the first reachable node of the graph belonging to a loop. Thus, it is tagged as entry node of the loop. Figure 4(b) shows the result of the computation of the exit nodes on the graph of Figure 3. Both *BV* and *UIR* have a child node that does not belong to the loop (*CA*), and both are at equal distance of the entry node. Thus, they are both tagged as exit nodes of the loop. Figure 4(c) describes the result of the computation of the mandatory paths of the loop, which are all the paths starting from the entry node and ending with an exit node. In this case, there are two paths: (*RSD, BV*) and (*RSD, UIR*). Figure 4(d) illustrates the result of the computation of the optional paths of the loop, which are all the paths starting from an exit node and ending with the entry node. For the given example there is only one path: (*RAI, RSD*). Finally, the graph is made acyclic by removing the transition between *RAI* and *RSD*.

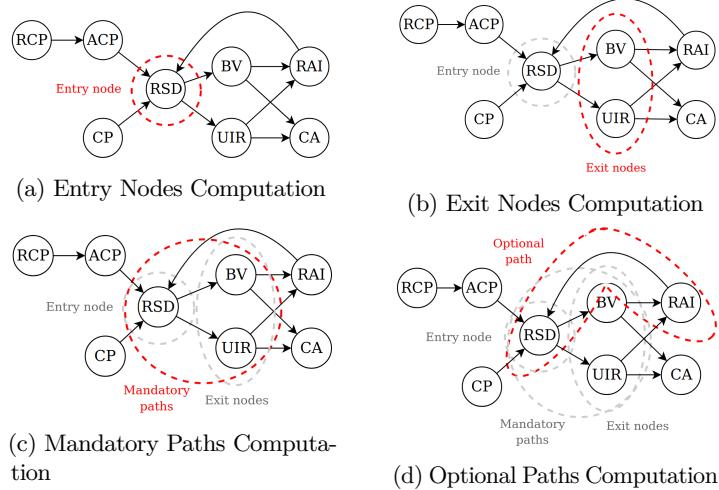


Fig. 4: Graph Loops Management

### 3.4 Directed Graph to AST

The next step consists in converting the directed graph into its corresponding AST. To do so, the graph is traversed in a depth-first way. During this traversal, each task

of the graph is examined to know whether it has already been added to the AST or not. If not, all the ancestor (resp. successor) nodes of this task in the graph are retrieved. Among them, only the closest ones already added to the AST are kept. This computation returns the nodes of the AST after (resp. before) which the current task should be placed. They are called the *left-bounding nodes* (resp. *right-bounding nodes*). It is worth noting that the AST may already contain nodes between the left-bounding nodes and the right-bounding nodes of the task to insert. As these nodes are not bounding nodes, they are not constrained with regards to the task to insert. Consequently, the task to insert will be put in parallel of them in the AST. If the task is the entry node of a loop, the loop is entirely generated and added to the AST at the position where the current task should have been inserted.

*Example.* Let us consider the AST shown in Figure 5(a). This AST already contains tasks *RCP*, *ACP*, *RSD*, *BV*, *UIR*, *RAI* and *CA*. The loop containing *RSD*, *BV*, *UIR* and *RAI* has been replaced by a ‘...’ node for brevity. The next task to add is *CP*. By analysing the graph in Figure 3, one can see that *CP* has no ancestor and five successors: *RSD*, *BV*, *UIR*, *RAI* and *CA*. As it has no ancestor, it consequently has no left-bounding node. On the other hand, the analysis of the AST shows that the closest successor already in it is *RSD*. Thus, the right-bounding node of *CP* is *RSD*. As *RSD* belongs to a loop to which *CP* does not belong, the right-bounding node of *CP* becomes the root node of the loop, i.e., the ‘\*’ node. This means that *CP* must be put on the left of this ‘\*’ node. As one can see, there are already two nodes on the left of the ‘\*’ node, *RCP* and *ACP*. These nodes are not constrained to *CP* (otherwise they would be bounding nodes), and will thus be put in parallel of it. To preserve the sequential constraint between them, they will also be placed under a new ‘<’ node. The result of this insertion is shown in Figure 5(b).

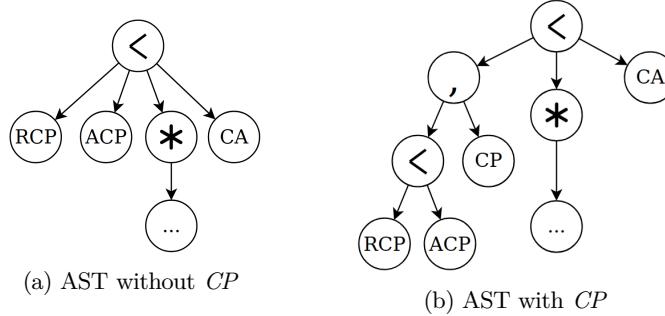


Fig. 5: Illustration of the AST Generation Process

### 3.5 Choices Management

The AST generated in the previous step is almost complete. The only constraints that have not been managed yet are the eventual exclusive choices between the tasks.

For each exclusive choice between two tasks, the AST undergoes a modification depending on its structure. The three possible modifications are listed below.

- (i) The two tasks do not yet belong to the tree. In this case, the two tasks are put below a ‘|’ node which is put in parallel of the whole tree.
- (ii) One of the two tasks already belongs to the tree. In this case, the two tasks are put below a ‘|’ node, which is inserted at the position of the task already belonging to the tree, thus replacing it.
- (iii) Both tasks already belong to the tree. In this case, they have a least common ancestor that should be a ‘,’ node (otherwise they are already constrained and can consequently not be mutually exclusive). This ‘,’ node is thus simply replaced by a ‘|’ node to represent the exclusive choice between the two tasks.

*Example.* Figure 6(b) shows the result of adding the exclusive choice constraints to the AST of Figure 6(a). As the reader can see, either *RCP* and *ACP* can be chosen or *CP*, as required by the original textual description. The original constraints also state that after *UIR* and *BV*, the process should contain an exclusive choice executing either *CA* or *RAI* followed by *RSD*. This constraint, although not visible in the AST, is implicitly present in it. Indeed, at the end of the mandatory part of a loop, one can either execute the optional part that goes back to the beginning, or leave the loop. Here, after *UIR* and *BV*, one can either perform *RAI* and restart the loop with *RSD*, or leave it and execute *CA*. Thus, one has to make a “choice” between *RAI* followed by *RSD* and *CA*.

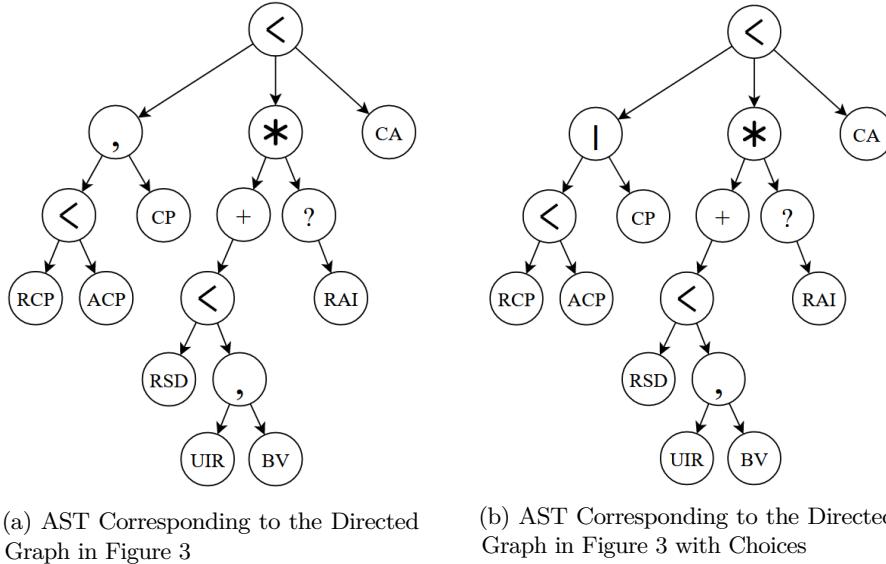


Fig. 6: Choices Management

Table 1: Transformation Patterns from AST to BPMN

Pattern	AST	BPMN
(1) Sequential Pattern		
(2) Parallel Pattern		
(3) Choice Pattern		
(4) Loop Pattern		

### 3.6 AST to BPMN

At this step of the approach, the AST resulting from the former steps is complete, meaning that it contains all the information of the original constraints stated by GPT. Thus, it is ready to be converted into its equivalent BPMN process. To do so, the transformation patterns presented in Table 1 are applied recursively on the nodes of the AST in a bottom-up fashion. Concretely, the deepest nodes of the AST (i.e., the task nodes) are the first being generated. Then, these tasks are connected together by applying the appropriate pattern. This generates a BPMN sub-process that will be connected to other sub-processes by recursively applying the patterns on their parent nodes up to the root.

*Example.* The BPMN process in Figure 7 shows the result of the transformation of the AST presented in Figure 6(b). This process was obtained by applying first the sequential pattern to tasks *RCP* and *ACP*, then the choice pattern to task *CP* and the freshly generated sub-process containing *RCP* and *ACP* in sequence, and so on until reaching the root of the AST. It is worth noting that the ordering constraints of the AST are respected, as are the textual requirements presented in the example of Section 2.3.

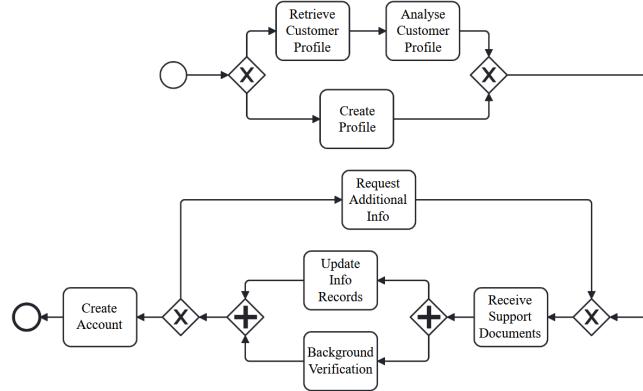


Fig. 7: BPMN Process Generated from the AST in Figure 6(b)

## 4 Tool and Experiments

This approach has been entirely implemented and validated by a tool written in Java. To facilitate its usage, the Java code has been embedded in the backend of a web server which is freely available online<sup>2</sup>. The implementation details are given in Figure 8. The user writes his textual description on the web application that is developed in HTML, CSS, JavaScript and makes use of JQuery, Ajax and BootStrap. The description is then transmitted to the backend written in NodeJS, which asks the Java program to send description to GPT. The expressions returned by GPT are transformed into their corresponding ASTs, which are eventually converted into the resulting BPMN process. This process is finally rendered by bpmn.io<sup>3</sup>, and displayed in the web application.

This approach was tested and validated on 200 descriptions from various sources. 25% come from the literature (PET dataset [3], proceedings, ...). The remaining 75% were handcrafted by 9 users (5 experts and 4 novices) who experimented the tool. All these examples contain tasks which were named beforehand. Experiments were also conducted on raw descriptions (i.e., without names for tasks) and showed a 24% loss of accuracy. This is mainly due to the fact that GPT often misses tasks in the description, or does not detect implicit loops. It is also worth noting that, for these 200 examples, GPT returned syntactically correct expressions with regards to the grammar defined in Section 2.3. The central part of these experiments consisted in comparing the tool proposed in this approach to other tools coming from the literature, and to LLMs directly. To the best of our knowledge, only one tool is recent enough and available online: ProMoAI [14]. Our tool was also compared to the LLMs Gemini [4] and GPT-4-turbo [5]. ProMoAI was used as is, while the two LLMs were given the simple instruction “*I want you to generate the BPMN process corresponding to the description provided between curly brackets: {Lorem ipsum dolor sit amet,*

<sup>2</sup> <https://quentinnivon.github.io/pages/givup.html>

<sup>3</sup> <https://bpmn.io/>

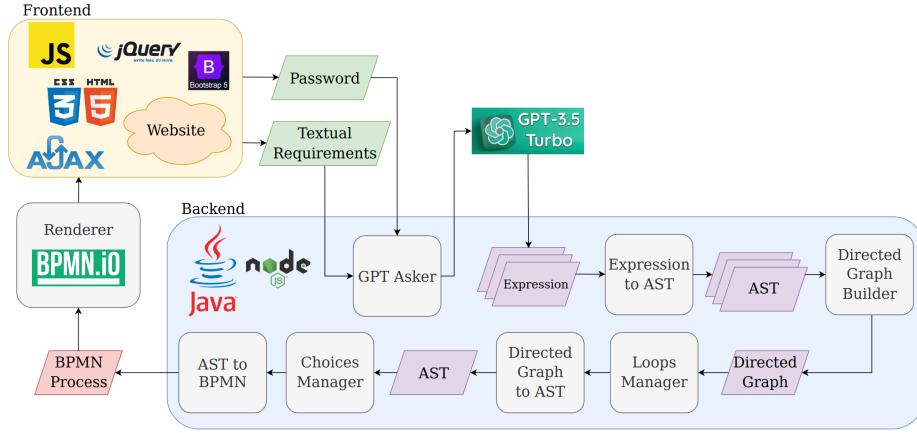


Fig. 8: Overview of the Toolchain

*consectetur adipiscing elit.}”. These experiments aimed at assessing both the accuracy of the result and the time taken by the tools to generate the BPMN process.*

The results, presented in Table 2, are split into three different groups. The first group, labelled with a tick, represents the processes that were considered as valid by the two experts who analysed the results, called reviewing experts. Here, the notion of validity relies on the correspondence between the expected process and the generated one. In other words, a process is considered as valid if it corresponds exactly to the expectations of the reviewing experts, and thus to the textual requirements. The second group, labelled with a question mark, represents the processes that were considered as ambiguous by the reviewing experts. Such processes are considered as ambiguous because, according to their textual description, one may generate several valid processes. As a choice has to be done, one of these possible processes is generated, which may not correspond to the expectations of the expert. For this reason, they belong to the group of ambiguous processes. For instance, a simple sentence such as “*I want A and B and C*” does not state how *A*, *B*, and *C* are related to each others. Thus, putting them in sequence, in parallel or partially in sequence and in parallel remains correct with regards to the description. Similarly, a sentence such as “*I want A before B or C before D*” can be interpreted as a choice between *A before B* and *C before D*, or as a sequence executing first *A*, then *B or C*, and finally *D*. For this reason, such processes have been separated from the others, but remain considered as valid. The third and last group, labelled with a cross, represents the processes that were considered as invalid by the reviewing experts. Such processes are at least partially non-compliant with the textual description. It is for instance the case when a non-ambiguous constraint is missing (e.g., two tasks are not put in sequence although they should be), or erroneous (e.g., two tasks are put in an exclusive choice instead of one after the other). Invalid processes are generated when GPT is not able to extract a constraint stated textually, or when it misinterprets it.

The results of these experiments, provided in Table 2, showed that our tool obtains the best results both in terms of generation quality (with 78.5% of well-formed processes) and execution time (with an average execution time of 4.07s). Without much surprise, the execution time of this approach grows as the textual description grows, and as the number of generated expressions increases. Regarding the 13.5% incorrect processes, a deeper analysis shows that they are usually close to the expected process, with very few missing or erroneous constraints. Rather surprisingly, GPT-4-turbo obtained very good results, especially with regards to its low failure percentage. It also greatly outperformed Gemini which obtained the worst results of these experiments. However, the results obtained by the LLMs must be handled with care as they are very probably overrated. Indeed, to the best of our knowledge and experiments, LLMs are, for now, not capable of generating directly the XML code of a BPMN process correctly. For this reason, the LLMs were asked to generate a textual representation of the BPMN process, which was then visually analysed by the reviewing experts and used to compute the score of these models. As generating the exact XML code adds an additional difficulty layer for the LLM, it is likely that the results shown in Table 2 would be lower.

## 5 Related Work

In [10], the authors propose a solution for modelling BPMN processes from natural language. To do so, they design a Domain-Specific Language (DSL) along with its corresponding grammar to manage textually the dependencies that may exist between the elements of the process-to-be. Once the user has written a specification that is compliant with the proposed grammar, the DSL parser extracts traces from it. These traces are then transformed into a BPMN model with the help of a process mining algorithm. The goal of our approach is similar, in the sense that it also aims at generating a BPMN process from a textual input. The main difference between their approach and ours is that we allow textual descriptions written in natural language while they require a description that is compliant with their grammar, and must be learned before use.

In [16], the authors give insights of how LLMs could be used within the Business Process Management lifecycle. For each step, namely *identification*, *discovery*, *analysis*, *redesign*, *implementation* and *monitoring*, they synthesise how LLMs could be used jointly with human interactions to improve the quality of the result or lower the time needed to perform the step. The work presented in [15] presents a theoretical approach aiming at extracting a business process from a natural language specification

Table 2: Experimental Results

Tool/Model	✓	?	✗	Avg. Exec. Time (s)
Our tool	<b>78.5%</b>	8%	13.5%	<b>4.07</b>
ProMoAI	50%	8.7%	41.2%	24.7
Gemini	32.2%	8.1%	59.7%	8.32
GPT-4-turbo	66.6%	21.1%	<b>12.2%</b>	19.2

with the help of LLMs. In this work, the authors propose to use sentence level and text level analysis to infer activities and dependencies from a specification in natural language. In both works, the focus is made on partial extraction of data (activities, actors, dependencies) that requires a human intervention in a second phase in order to obtain a business process. In our case, we propose to automatically generate a complete BPMN process from a textual description, without any human intervention.

In [12], the authors perform a series of question/answer exchanges with a chatbot enhanced with natural language processing capabilities. The goal of these iterations is to create or improve process models. In particular here, the authors state that a fully integrated conversational modelling toolchain would include task extraction, logic extraction, BPMN layout creation and BPMN refinement capabilities, but they restrict their focus on the extraction of tasks. The authors also provide several metrics about the quality of the results returned by several well-known LLMs. In [2], the authors propose a technique to transform a natural language specification into some formatted output that is understandable by a computer. To do so, they make use of in-context learning, that gives the opportunity of guiding the dialog towards the desired output. In both papers, the authors end up with pieces of processes (tasks, participants, ...), which have to be put together manually, while our approach automatically provides a BPMN process in which the information extracted from the specification is represented.

In [8], the authors make use of subject-verb-object techniques to extract tasks and participants from the specification, and search for gateway-related keywords to extract information about gateways. From this information, they build an internal spreadsheet format corresponding to the execution sequence of the process, which is finally converted to BPMN. This approach requires partially formatted text in the sense that non-verbal sentences, or sentences without keywords (i.e., “if” for choices) are not likely to be recognised. Also, only sequential composition and choices seem to be supported. In our approach, the input text does not need to be formatted, and parallelism and loops are supported. In [7], the authors make use of a DSL to pre-train a LLM on the BPMN semantics. By doing so, they are able to extract tasks and relationships between them from natural language descriptions. However, the supported syntax is restricted to exclusive and parallel gateways with two paths and loops are not considered. In our approach, such elements are supported.

## 6 Concluding Remarks

The main goal of the approach proposed in this paper is to automatically convert a textual description into its corresponding BPMN process. To do so, tasks and constraints are extracted from the textual requirements and represented as ASTs. These ASTs are then analysed, modified and recomposed through several steps in order to obtain a single AST containing all the information of the original textual description. Finally, this AST is converted to its corresponding BPMN process and shown to the user. The approach has been implemented, tested and validated by both experts of the BPMN community and novice users.

This work offers several axes of improvements. The main one being to use the GPT-4 model (and/or derivatives) instead of GPT-3.5, which will be done as soon as this model becomes available for fine-tuning. Using a more recent model is very likely to improve significantly the quality of the generated expressions, and thus the generated process. Similarly, increasing the fine-tuning training set would enlarge the proportion of natural language understood by the model, thus limiting its mistakes and increasing the quality of the resulting process. Another possibility could be to extend this work with model checking driven by a temporal logic property written in text.

## References

1. A. V. Aho, M. R. Garey, and J. D. Ullman. The Transitive Reduction of a Directed Graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.
2. P. Bellan, M. Dragoni, and C. Ghidini. Extracting Business Process Entities and Relations from Text Using Pre-trained Language Models and In-Context Learning. In *Proc. EDOC'22*, pages 182–199. Springer International Publishing, 2022.
3. P. Bellan, H. van der Aa, M. Dragoni, C. Ghidini, and S. P. Ponzetto. PET: An Annotated Dataset for Process Extraction from Natural Language Text Tasks. volume 460 of *Lecture Notes in Business Information Processing*, pages 315–321. Springer, 2022.
4. G. T. et al. Gemini: A Family of Highly Capable Multimodal Models, 2024.
5. O. et al. GPT-4 Technical Report, 2024.
6. Y. Falcone, G. Salaün, and A. Zuo. Semi-automated Modelling of Optimized BPMN Processes. In *Proc. of SCC'21*, pages 425–430. IEEE, 2021.
7. H.-G. e. a. Fill. Conceptual Modeling and Large Language Models: Impressions From First Experiments With ChatGPT. In *Proc. EMISAJ'23*, pages 1–15, 2023.
8. K. Honkisz, K. Kluza, and P. Wiśniewski. A Concept for Generating Business Process Models from Natural Language Description. In *Proc. KSEM'18*, pages 91–103, 2018.
9. ISO/IEC. International Standard 19510, Information technology – Business Process Model and Notation. 2013.
10. A. e. a. Ivanchikj. From Text to Visual BPMN Process Models: Design and Evaluation. In *Proc. of MODELS'20*, page 229–239. Association for Computing Machinery, 2020.
11. S. C. Kleene. Representation of Events in Nerve Nets and Finite Automata. *Automata Studies*, pages 3–41, 1951.
12. N. Klievtsova, J.-V. Benzin, T. Kampik, J. Mangler, and S. Rinderle-Ma. Conversational Process Modelling: State of the Art, Applications, and Implications in Practice. In *Proc. BPM'23*, pages 319–336. Springer Nature Switzerland, 2023.
13. D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, 1969.
14. H. Kourani, A. Berti, D. Schuster, and W. M. P. van der Aalst. ProMoAI: Process Modeling with Generative AI, 2024.
15. K. Sintoris and K. Vergidis. Extracting Business Process Models Using Natural Language Processing (NLP) Techniques. In *Proc. KSEM'17*, pages 135–139, 2017.
16. M. Vidgof, S. Bachhofner, and J. Mendling. Large Language Models for Business Process Management: Opportunities and Challenges, 2023.