Semi-Automated Refactoring of BPMN Processes

Quentin Nivon, Gwen Salaün

Semi-Automated Refactoring of BPMN Processes

Quentin Nivon, Gwen Salaün
Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, F-38000 Grenoble France
quentin.nivon@inria.fr*, gwen.salaun@inria.fr

Abstract—Business Process Modeling Notation (BPMN) is nowadays widely used by companies to represent their business processes. Such processes are usually designed and written by non-expert users for whom the main matter is to conceive a process corresponding to the needs of the companies. As the quality of the process is not the main design criterion, it can generally be optimised in several ways. For instance, reducing the financial cost of the process, its resource usage, or its execution time are classical optimisation axes. In this work, the considered BPMN processes are enriched with time and resources, and are executed multiple times. The proposed optimisation approach consists in reducing the execution time of these processes. To do so, the method presented in this paper consists in restructuring the processes by changing the position of their tasks. The goal of this restructuring is to limit the overuse of the resources and consequently reduce the execution time of the processes. To avoid generating unsuitable processes, the designer is involved in the restructuring phase, as (s)he validates each restructuring step. The proposed technique is fully automated by a tool that was implemented and applied on several examples for validation purposes.

1. Introduction

Nowadays, companies are making use of workflow modelling notations to represent their business processes. These processes are usually subject to changes due to the evolution of the companies needs throughout the years. Thus, optimising them is a key step in their lifecycle as it may lead to significant reductions of their operational cost, better usage of their resources, or important diminution of their execution time. In any case, optimising the business processes of a company is always beneficial for it.

Business process optimisation can be done in several ways. For instance, simulation techniques can be used to compute metrics of interest regarding the quality of the process. However, refining the process according to these metrics is almost always performed manually, which requires a user with an important knowledge of the needs of the company. Another possibility is to adjust the resources consumed by the process to make its execution smoother. This optimisation, known as *resource optimisation* [1][2], usually requires flexibility in the budget of the companies. Indeed, adjusting the resources may require to buy new equipment, or hire new employees. The approach proposed in this paper is automated, and does not require any change in the budget of the company.

In this approach, the main goal is to optimise the execution time of business processes described using the Business Process Modelling Notation (BPMN). A solution to optimise the execution time of a business process automatically and without impacting the company's budget consists in restructuring it by changing the position of its tasks. This structural modification is known as *process refactoring* [3][4]. Usually, this structural modification induces a better balance of the resource usage. Consequently, the competition to acquire the resources decreases, leading to a smoother execution of the process, and thus a shorter execution time.

The processes handled in this approach are enhanced with quantitative information that is necessary to precisely analyse and optimise them. Therefore, in this work, the processes also include time as a duration associated to tasks and an explicit description of the resources required to execute each task. It is also worth noting that a process is not executed once but multiple times resulting in multiple instances. For each execution/instance, each task needs to acquire the required (globally shared) resources to be able to execute. The restructuring, which consists in moving tasks from one place to another, is here performed step by step. At each step, a task of the process is elected and identified as task to move. The user designing the process can then decide whether this task should be moved or not. If (s)he decides to move the task, all the possible positions of this task in the process are computed. Among these generated processes, the one obtaining the greatest score is kept, and shown to the user. The user then decides whether the new version of the process makes sense to him. If it is the case, a new task is elected and proposed to the user, and so on. Otherwise, the new version of the process is discarded, returning to the version of the process generated in the previous step, and a new task is proposed to the user. As the user keeps control on the modifications that are performed, (s)he is more likely to have a better understanding of the final shape of the process. The whole approach is fully automated by a tool that was implemented and applied on hundreds of examples. The experimental results show an important gain and a short computation time on real-world examples.

Section 2 introduces the languages and models used in this paper. Section 3 presents the different steps of the refactoring approach. Section 4 describes the tool support and some experimental results assessing the performance of the approach. Section 5 compares the solution to related work and Section 6 concludes this paper.

2. MODELS

2.1 BPMN with Time and Resources

This work focuses on BPMN activity diagrams including the constructs related to control-flow modelling and behavioural aspects. Beyond those constructs, execution time and resources are also associated with tasks.

More precisely, the node types *event*, *task*, and *gateway*, and the edge type *sequence flow* are considered. Start and end events are used, respectively, to initialise 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 order. A task may have a duration or delay, expressed by default in *units of time* (UT). It can also be defined using probabilistic functions, in case of non-fixed durations. Resources are explicitly defined at the task level. A task can thus include, as part of its specification, the required resources. In such a case, it means that the task needs those resources to be able to execute. Once the resources are acquired, the task executes for the specified duration. In this approach, the acquisition of resources relies on a "first-come-first-served" strategy. This strategy is a parameter of the approach and can thus be replaced by another strategy according to the user's needs, without any impact on the results. If a task needs more replicas of a resource than available, it remains in a waiting state until the release of a sufficient number of replicas of the required resource.

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* (represented with the ✕ symbol) and *parallel* gateways (represented with the + symbol). A parallel gateway indicates the possibility of executing all of its paths at the same time, while an exclusive gateway represents a choice, thus only one of its paths is executed. As only one path of an exclusive gateway is executed, each of its paths has a probability of execution represented as a real value ranging from 0 to 1. The sum of the probability of each flow can not exceed 1. These probabilities are usually extracted from the execution traces of the process with the help of process mining techniques [5]. They can also be specified by the designer of the process. If they have not been extracted nor specified, they are considered as equal for each path of the gateway. In the rest of this paper, BPMN processes are represented as $B = (S_O, S_F)$ where S_O is a set of nodes (tasks, events, gateways, ...) and S_F a set of flows connecting these nodes (sequence flows).

Example. Figure 1 shows a BPMN process describing a bank account opening. As the reader can see, tasks have a duration and make use of resources. For instance, task *Create Profile* has a duration of 10UT and requires one replica of resources *database* and *bank advisor* to execute. Exclusive split gateways exhibit the probability of execution of their paths. For example, gateway Crepresenting one of $_1$ has one path executing with probability 0.1 and the other one executing with probability 0.9.

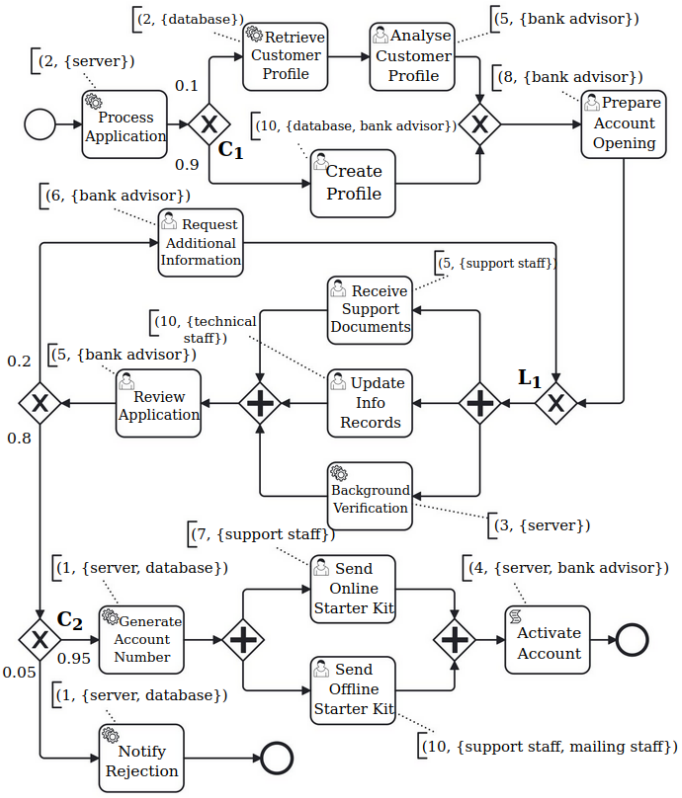


Figure 1: Example of Account Opening Process

2.2 Dependencies Between Tasks

In BPMN, tasks are naturally ordered by the sequence flows that are connecting them. Thus, two tasks connected by a sequence flow are dependent, as one must be executed before the other. As the solution presented in this paper performs a restructuring of the process, there is no guarantee regarding the final position of a task in the resulting process, compared to its position in the original one. Nonetheless, some tasks may have to remain in a specific order to preserve the meaning of the process (e.g., some product should be packaged before its delivery). Such *strong dependencies* can be given by the user or computed by analysing the data-flow graph corresponding to the BPMN process [6][7].

In the rest of this paper, a dependency or *partial order* between two tasks T_1 and T_2 is written as a pair (T_1, T_2). Two tasks are said to be dependent if they belong to a pair of dependencies, and non-dependent otherwise. T_1 is said to be a *predecessor* of T_2, and T_2 a *successor* of T_1.

Example. The account opening process presented in Figure 1 has several dependencies that should be preserved by the refactoring process: (i) (*Retrieve Customer Profile*, *Analyse Customer Profile*), (ii) (*Request Additional Information*, *Notify Rejection*), (iii) (*Process Application*, *Review Application*), (iv) (*Generate Account Number*, *Activate Account*), (v) (*Create Profile*, *Update Info Records*), (vi) (*Create Profile*, *Prepare Account Opening*) and (vii) (*Retrieve Customer Profile*, *Prepare Account Opening*). It is worth noting that other dependencies

exist in the process, such as (*Send Online Starter Kit*, *Activate Account*) but are not considered as mandatory to be preserved. Consequently, they may no longer exist in the resulting process.

2.3 Abstract Graphs

An abstract graph is an internal representation of a BPMN process that is used in this work as an intermediate format. It was originally introduced in [8] where the authors propose first to generate an abstract graph from a set of dependencies between tasks and then to generate the BPMN process corresponding to this abstract graph. This representation is defined recursively as a main graph, possibly containing conditional structures such as choices or loops as sub-graphs. It is worth noting that this representation has the same expressiveness than the subset of BPMN that is supported in this work.

Definition 1. *(Abstract Graph) An abstract graph is a (hierarchical) directed acyclic graph* (S_N, S_E) *where* S_N *is an ordered set of nodes* $(n_0,\ n_1,\ ...,\ n_n)$ *and* S_E *a set of directed edges* $(e_1 = n_0 \rightarrow n_1,\ e_2 = n_1 \rightarrow n_2,\ ...,\ e_n = n_{n-1} \rightarrow n_n)$ *connecting these nodes. A node* $n \in S_N$ *is defined as a pair* (S_T, S_G) *where* S_T *is a set of tasks and/or conditional structures and* S_G *is a set of abstract (sub-)graphs. A node has exactly one successor and one predecessor, except the first node which has no predecessor, and the last node that has no successor.*

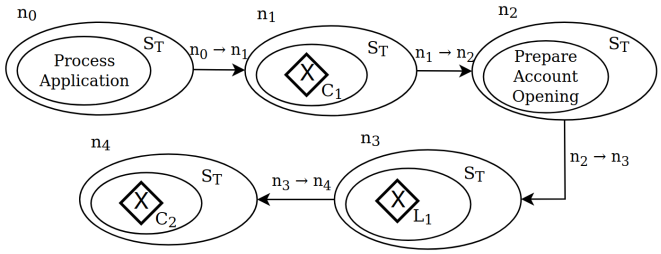
This definition is almost identical to the one given in [8], except that the one proposed here considers conditional structures. In this work, conditional structures belonging to abstract nodes contain themselves abstract graphs describing their behaviour. For instance, a choice is represented as a set of abstract graphs, each corresponding to one of its paths.

Definition 2. *(Choice Structure) Let* $G = (S_N, S_E)$ *be an abstract graph. A choice structure, denoted* $\langle\times\rangle_C$*, belonging to a node* $n \in S_N$ *is written as a set of abstract graphs* $\{G_1, G_2, ..., G_n\}$ *representing the different paths of the choice, enriched with their probability of execution* $p_i \in [0; 1]$*. Choice structures are written* $\langle\times\rangle_C = \{(G_1,\ p_1),\ (G_2,\ p_2), ...,\ (G_n,\ p_n)\}$.

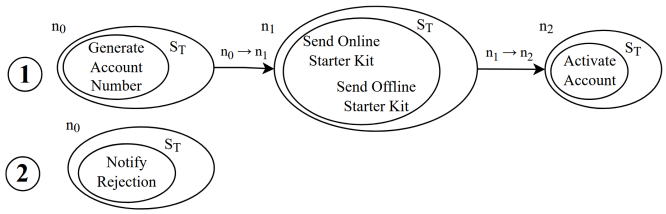
Similarly, a loop is defined as two abstract graphs. The first one represents the part of the loop that is always executed, while the second one represents the part of the loop that is conditionally executed, i.e., the part of the loop which goes back to its starting node.

Definition 3. *(Loop Structure) Let* $G = (S_N, S_E)$ *be an abstract graph. A loop structure, denoted* $\langle\times\rangle_L$*, belonging to a node* $n \in S_N$ *is represented by two abstract graphs* $\{G_{FL}, G_{LF}\}$*.* G_{FL} *represents the path going from the entry node of the loop to its exit node, while* G_{LF} *represents the path going from its exit node to its entry node. Both are enriched with their probability of execution* $p_{LF} \in [0; 1]$ *and* p_{FL} *= 1. It is written* $\langle\times\rangle_L = \{(G_{FL},\ p_{FL}),\ (G_{LF},\ p_{LF})\}$.

Example. Figure 2 shows the abstracts graphs corresponding to the account opening process presented in Figure 1. For the sake of space, in the rest of this paper, the empty sets of tasks/graphs belonging to the abstract nodes are omitted. As the reader can see, the main abstract graph shown in Figure 2a is rather simple as it does not contain the tasks belonging to the conditional structures of the process (i.e., $\langle\times\rangle_{C_1}$, $\langle\times\rangle_{C_2}$ and $\langle\times\rangle_{L_1}$). These tasks appear in the abstract sub-graphs of their respective conditional structures. This is for instance the case for task *Notify Rejection* that belongs to $\langle\times\rangle_{C_2}$, which is shown in Figure 2b. The abstract sub-graphs of $\langle\times\rangle_{L_1}$ and $\langle\times\rangle_{C_1}$, which are not shown in this figure, can be represented similarly.



(a) Main Abstract Graph



(b) Abstract Sub-Graphs Belonging to $\langle\times\rangle_{C_2}$

Figure 2: Abstract Graphs Corresponding to the Account Opening Process in Figure 1

2.4 Metrics

Several metrics can be computed on a BPMN process extended with quantitative aspects, such as execution times, synchronisation/waiting times, resource usage or total costs. In this work, the time taken by a process to execute is the most important one since it is the main optimisation goal of the approach. The *execution time* of a process corresponds to the difference between the timestamp at which an end event is reached and the timestamp at which the start event is triggered. This time varies depending on the structure of the process, the use of gateways or loops, but is always finite if the process is syntactically well-formed (i.e., if each execution scenario eventually ends up with an end event). This approach considers a notion of execution time called *average execution time*, which is only relevant in a multi-instance context. It represents

the time taken by one instance of the process to complete its execution, on average.

Definition 4. *(Average Execution Time) Let B be a BPMN process executed n times and ET_{B_1}, ET_{B_2}, ..., ET_{B_n} the execution times of each instance of B. The average execution time of B is defined as $AET_B = \frac{1}{n}\sum_{i=1}^{n} ET_{B_i}$.*

This execution time can be computed using simulation techniques [1]. These simulation techniques and the resulting execution times highly depend on the pool of shared resources that the process can use. Indeed, insufficient resources increase the competition between tasks, thus delaying their execution and increasing the execution time of the affected instances. The resulting execution times also depend on the *workload*, which is a couple (N, R) where N represents the number of instances of the process being executed and R the rate at which each process execution is started. In other words, each R units of time, a new instance of the process starts, until N instances have been triggered. This rate is also known as *inter-arrival time* (IAT).

It is worth noting that *synchronisation times* play an important role in this work. The synchronisation time induced by merge parallel gateways corresponds to the time elapsed between the end of execution of the path of the gateway having the shortest execution time and the end of execution of the path of the gateway having the longest execution time (thus resulting in the activation of that merge). These merge gateways are often seen as bottlenecks because they induce additional delays in a multi-instance context. On the other hand, adding more parallelism to a process may also speed up its execution. The solution proposed in this paper takes particularly care of this issue by adding parallelism when it does not induce such bottlenecks and by avoiding parallelism when it results in additional delays.

Example. Let us consider two processes both executing tasks A and B. Task A has a duration of 10UT, task B has a duration of 20UT, and both require one replica of resource r_1 to execute. The first process executes A and B in sequence, while the second one executes them in parallel. Both processes are executed 100 times, with an inter-arrival time of 5UT, and have access to two shared replicas of r_1. After simulating the two processes in these conditions, the first process obtains an AET of 580UT while the second one obtains an AET of 747UT. Thus, the addition of parallelism has increased the AET.

2.5 Optimality

The main goal of this work is to reduce the average execution time of the original process by applying refactoring techniques to it. In the best case, the average execution time of the refactored process is the shortest possible. In such a case, the refactored process is said to be *optimal*.

Definition 5. *(Optimal Process) Let B be a BPMN process and $\overset{R}{\Longrightarrow}$ be the operation generating all the possible refactored processes from a given BPMN process. If $\exists(B_{r_1}, B_{r_2}, ..., B_{r_n})$ such that $B \overset{R}{\Longrightarrow} (B_{r_1}, B_{r_2}, ..., B_{r_n})$, a process $B_{r_i} \in (B_{r_1}, B_{r_2}, ..., B_{r_n})$ is said to be optimal if and only if $AET_{B_{r_i}} = \min(AET_{B_{r_1}}, AET_{B_{r_2}}, ..., AET_{B_{r_n}})$.*

3. Refactoring

The approach proposed in this paper aims at automatically restructuring a BPMN process in order to optimise its average execution time. It takes as input a BPMN process with duration and used resources for each task, a pool of shared resources and an IAT. It is worth recalling that, in this work, a process is executed multiple times, each instance starting at a given rate called IAT. The refactoring technique does not change the tasks themselves, but the way they are organised within the process. Consequently, the semantics of the process changes over time. Nonetheless, the tasks belonging to the traces of the original process remain exactly the same throughout the refactoring steps. The only change resides in the position of these tasks in the traces. In this work, it was decided to involve the user in order to give her/him the opportunity to choose the modifications that seem relevant for her/him while discarding the ones that are not. By doing so, the user is aware of the changes performed on the process at each step, and is more likely to have a better understanding of the final process. In the end, the approach returns a new BPMN process whose average execution time is shorter (or equal to in the worst case) than the one of the original process. This section is divided as follows: Section 3.1 gives an overview of the approach, Section 3.2 details the task election process, Section 3.3 presents the refactoring patterns, Section 3.4 illustrates the completeness of the patterns, Section 3.5 discusses the semantics preserved by the patterns, and Section 3.6 concludes with the heuristic proposed to select the best process.

3.1 Overview

In this approach, it was decided to involve the user by making her/him validate or decline the changes performed on the process. This choice was motivated by several reasons, all guided by the idea that entirely restructuring a BPMN process and giving it back to the user may not be useful for her/him. First, the resulting BPMN process may not suit exactly the needs of the user. For instance, the user may have forgotten some dependencies between tasks, or may consider that some new positions of tasks are not relevant. Second, the resulting BPMN process may be syntactically far from the original one, which may prevent the user from fully understanding it. To avoid such issues, the user is involved at each important step of the approach in order to make her/him accept or decline each decision.

Figure 3 gives an overview of the approach. First, the original BPMN process is converted into its corresponding abstract graph. Then, one of the tasks of the process is elected and submitted to the user for validation. If the user wants this task

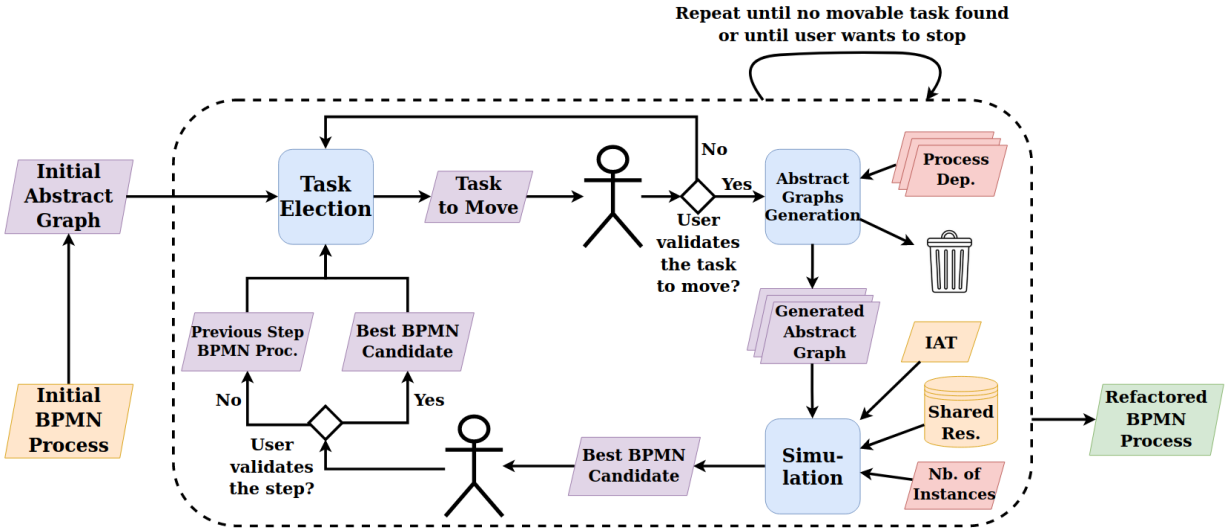


Figure 3: Overview of the Approach

to remain in its original position, (s)he can decline the offer, and another task will be elected and submitted to her/him, until (s)he validates. Once the user accepts to move the elected task, several patterns are applied to the abstract graph in order to obtain all the possible positions of that task in the process. Among the generated abstract graphs, those not respecting the dependencies of the original process are discarded. Then, each remaining abstract graph is simulated using the IAT, the pool of shared resources, and the number of instances, in order to obtain metrics of interest about it. According to these metrics, a score is attributed to each abstract graph to represent its quality. The abstract graph obtaining the highest score is then mapped to its equivalent BPMN process and shown to the user. If this new process, in which a task has been moved, seems relevant for the user, (s)he validates it and a new task is elected in this new version of the process. Otherwise, if the process does not seem relevant for the user, (s)he can refuse it and a new iteration will start with the process of the previous iteration as basis (or the initial one if no iteration was done). At any time, the user can decide to stop the iteration. If (s)he does so, the current version of the process is compared to the original one. If it has a shorter AET, it is returned to the user. Otherwise, the user keeps the original version of the process. If the user does not decide to stop, the iterations automatically end when all the tasks of the process have been proposed to be moved.

3.2 Task Election

Once the initial process has been transformed into its corresponding abstract graph, the approach proposes a task to move. This can be done in many ways: randomly, task with greatest duration first, task with greatest duration last, task with greatest resource usage first, task with greatest resource usage last, a mix of several criteria, etc. From an empirical study made on dozens of processes, the selection method that gave the best results consists in selecting the tasks in ascending order of duration. The task with the shortest duration is elected first, then the second shortest, and so on, until the task with the longest duration. Intuitively, the reason behind the good quality of this heuristic is that each task of the process is moved once and only once. Thus, finding its best position is crucial. However, the position of the first task moved may be subject to changes due to the repositioning of the other tasks. Consequently, the later a task is moved, the less its position is subject to changes. As tasks with long duration have a high impact on the execution time of the process, their position has to remain as stable as possible. Once the task to move has been selected, it is proposed to the user. If the user accepts to move this task, the refactoring patterns are applied to the abstract graph and the task.

3.3 Refactoring Patterns

In this approach, the process is restructured by changing the position of its tasks, one after the other. To do so, several patterns are applied on the abstract graph from which the elected task has been removed. Each pattern generates one or several abstract graphs, in which the position of the elected task is different. It is worth noting that, by applying all these patterns, all the possible positions of the elected task in the process are obtained. Let us now present the four patterns. For the sake of space, only the first pattern is formally defined. The reader interested in the formal definitions of the other patterns can find them at [9].

3.3.1 Pattern 1: Task in Sequence of Abstract Node or Task.

The first pattern consists in putting the task to move in sequence of any abstract node or task/conditional structure of the abstract graph. To do so, the task to move is put in a new abstract node containing only this task. Then, the first possibility consists in putting this abstract node in sequence between other abstract nodes of the graph. The second possibility is to put one of the tasks of an abstract node into a new abstract node, and to connect it to the abstract node containing the task to move. This generates an abstract graph which is then put in the set of sub-graphs of the considered abstract node. This operation is repeated recursively on all the sub-graphs of each node of the abstract graph. This pattern is formalised using the definition of abstract graphs.

Definition 6. *(Pattern 1) Let* $G = (S_N, S_E)$ *be an abstract graph,* $T \notin S_N$ *be the task to move, and*

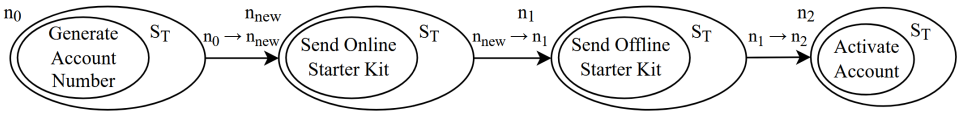


Figure 4: Example of Abstract Graph Generated by Pattern 1

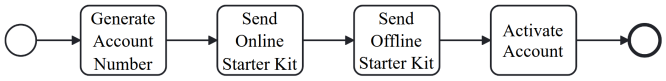


Figure 5: BPMN Process Corresponding to the Abstract Graph in Figure 4

$n_{new} = (\{T\}, \emptyset)$ *be a new abstract node containing only T. The set of abstract graphs generated by applying Pattern 1 to G and T, written* $gen_{P1}(G, T)$*, is defined as* $gen_{P1}(G, T) =$ **(i)** $\cup$ **(ii)** $\cup$ **(iii)** $\cup$ **(iv)** $\cup$ **(v)** $\cup$ **(vi)** *where*

(i) $= \bigcup_{i \in [1...|S_E|]} (S_N \cup \{n_{new}\}, S_E \setminus \{n_i \rightarrow n_{i+1}\} \cup \{n_i \rightarrow n_{new}, n_{new} \rightarrow n_{i+1}\})$ *represents the addition of the new node containing T in sequence with any other node of G,*

(ii) $= \{(S_N \cup \{n_{new}\},\ S_E \cup \{n_{new} \rightarrow n_0\})\}$ *represents the addition of the new node containing T before the first node of G,*

(iii) $= \{(S_N \cup \{n_{new}\},\ S_E \cup \{n_n \rightarrow n_{new}\})\}$ *represents the addition of the new node containing T after the last node of G,*

(iv) $= \bigcup_{n \in S_N} \bigcup_{t \in S_{T_n}} (S_N \setminus \{n\} \cup \{(S_{T_n} \setminus \{t\}, S_{G_n} \cup \{(\{n_t, n_{new}\}, \{n_t \rightarrow n_{new}\})\})\}, S_E)$ *– where* $n_t = (\{t\}, \emptyset)$ *– represents the addition of task T after any task of any node of G,*

(v) $= \bigcup_{n \in S_N} \bigcup_{t \in S_{T_n}} (S_N \setminus \{n\} \cup \{(S_{T_n} \setminus \{t\}, S_{G_n} \cup \{(\{n_t, n_{new}\}, \{n_{new} \rightarrow n_t\})\})\}, S_E)$ *– where* $n_t = (\{t\}, \emptyset)$ *– represents the addition of task T before any task of any node of G, and*

(vi) $= \bigcup_{n \in S_N} \bigcup_{g \in S_{G_n}} \bigcup_{g' \in gen_{P1}(g,T)} (S_N \setminus \{n\} \cup \{(S_{T_n}, S_{G_n} \setminus \{g\} \cup \{g'\})\}, S_E)$ *is the result of the recursive call of this function on each abstract sub-graph of each node of G.*

Example. Let us illustrate Pattern 1 on the abstract graph shown in Figure 2b and the task *Send Online Starter Kit*. This task is not dependent and can thus be put in sequence of any abstract node or task/conditional structure of the abstract graph. In particular, it can be put in sequence between nodes n_0 and n_1, as shown in Figure 4. To give a better understanding of the result of this pattern, Figure 5 shows the mapping between the generated abstract graph and its equivalent BPMN process. As the reader can see, task *Send Online Starter Kit* is now in sequence between tasks *Generate Account Number* and *Send Offline Starter Kit*.

3.3.2 Pattern 2: Task in Parallel of Ordered Combination of Nodes.

The second pattern consists in putting the task to move in parallel of any ordered combination of nodes of the abstract graph.

Definition 7. *(Non-Empty Ordered Power Set) Let* $G = (S_N, S_E)$ *be an abstract graph. The non-empty ordered power set of* S_N*, written* $\mathcal{P}^*_O(S_N)$*, is the set of combinations of nodes of* S_N *respecting their order of appearance in G, i.e.,* $\{s_0, s_1, ..., s_n\} \in 2^{S_N}$ *s.t.* $s_0 \rightarrow s_1 \in S_E$, $s_1 \rightarrow s_2 \in S_E$, ..., $s_{n-1} \rightarrow s_n \in S_E$.

For each ordered combination computed, a new abstract node is generated, containing the task to move and the ordered combination. The ordered combination is then removed from the original abstract graph and replaced by the new node to generate a new abstract graph. This operation is repeated recursively on all the sub-graphs of each node of the abstract graph. Similarly to pattern 1, the set of abstract graphs generated by this pattern is called $gen_{P2}(G, T)$.

Example. Let us illustrate Pattern 2 on the abstract graph shown in Figure 2b and the task *Send Online Starter Kit*. This task is not dependent and can thus be put in parallel of any ordered combination of nodes of the abstract graph. In particular, it can be put in parallel of nodes n_1 and n_2, as shown in Figure 6. To give a better understanding of the result of this pattern, Figure 7 shows the mapping of the abstract graph to its equivalent BPMN process. As the reader can see, task *Send Online Starter Kit* is now in parallel of tasks *Send Offline Starter Kit* and *Activate Account*, which are themselves in sequence.

3.3.3 Pattern 3: Task in Sequence of any Combination of Tasks and Subgraphs.

The third pattern consists in putting the task to move in sequence of any combination of tasks and sub-graphs of the abstract graph. First, the combinations of tasks and sub-graphs are computed for each abstract node of the graph. Then, the tasks belonging to the combination are put in the set of tasks of a new abstract node, and the sub-graphs belonging to the combination are put in the set of sub-graphs of the new node. Next, the task to move is put in the set of tasks of a new abstract node, that is connected to the abstract node containing the elements of the combination. This operation generates a new abstract graph, that is put in the set of sub-graphs of the currently processed node. This operation is repeated recursively on all sub-graphs of each abstract node of the graph. Similarly to patterns 1 & 2, the set of abstract graphs generated by this pattern is called $gen_{P3}(G, T)$.

3.3.4 Pattern 4: Insertion of Task Inside Choice.

The fourth and last pattern is slightly different from the previous ones. In order to compute all the possible positions of a task inside an abstract graph, the choice structures of the process must be carefully taken into account. A choice is, by definition, a structure composed of several paths among which only one will be executed. To preserve the semantics of the

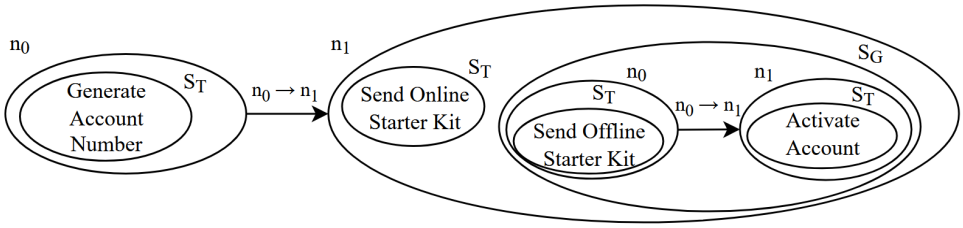


Figure 6: Example of Abstract Graph Generated by Pattern 2

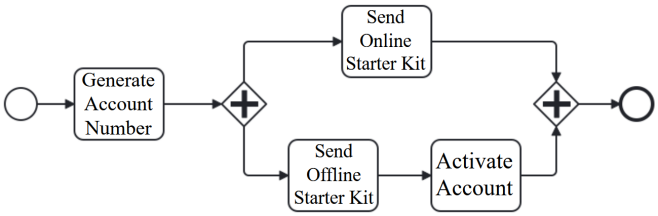


Figure 7: BPMN Process Corresponding to the Abstract Graph in Figure 6

process, the occurrences of any of its tasks must be preserved. However, inserting a task inside the path of a choice modifies the occurrences of the task, as the execution of the inserted task becomes conditional. The only way to properly insert a task inside a choice is to insert it in all the paths of the choice. By doing so, the execution of the task remains unconditional. Then, all the possible positions of the task in each path of the choice must be computed. To do so, patterns 1, 2 & 3 are applied to each path of the choice. This operation returns abstract graphs corresponding to all the possible positions of the task in each path of the choice. To obtain all the possible positions of the task at the level of the choice, the cartesian product between the sets of generated abstract graphs is computed.

Definition 8. *(Pattern 4) Let G be an abstract graph, $\langle\times\rangle_C = \{(G_1, p_1), ..., (G_n, p_n)\}$ be a choice structure belonging to G, and T be the task to insert. The set of choice structures generated by this pattern is $\prod_{i=1}^{n}(gen_{P1}(G_i, T) \cup gen_{P2}(G_i, T) \cup gen_{P3}(G_i, T))$.*

Finally, each generated choice structure replaces the original choice structure of the abstract graph. Similarly to patterns 1, 2 & 3, the set of abstract graphs generated by this pattern is called $gen_{P4}(G, T)$. It is worth noting that computing a cartesian product of sets often generates a large number of combinations. As the generated abstract graphs are simulated, this pattern is often discarded or partially discarded when the number of processes generated by the previous patterns exceeds a given threshold, which is a parameter of this approach.

3.4 Completeness

The four presented patterns are exhaustive, in the sense that they generate abstract graphs representing all the possible positions of the elected task without modifying the order of the remaining tasks. In other words, the generation of possible positions is *complete*.

Proposition 1. *(Completeness) Let G be an abstract graph, T be the task to move in G, and $\overset{T}{\Longrightarrow}$ be the operation applied to an abstract graph to generate a new abstract graph in which T has been moved to another position. The generation of positions is said to be complete, i.e., $\nexists G' \mid G \overset{T}{\Longrightarrow} G' \wedge G' \notin gen_{P1}(G,T) \cup gen_{P2}(G,T) \cup gen_{P3}(G,T) \cup gen_{P4}(G,T)$.*

The reader interested in the proof of completeness can find it at [9]. Finally, it is worth noting that these four patterns preserve the dependencies specified by the user. Indeed, once generated, all the abstract graphs are verified and those that do not respect the dependencies are discarded.

3.5 Preservation of Semantics

In BPMN, the semantics of a process corresponds to the set of all *execution traces* of this process.

Definition 9. *(Execution Trace) Let $B = (S_O, S_F)$ be a BPMN process and $S_A \subset S_O$ be the set of tasks of B. A trace τ is defined as a sequence of tasks $(t_0,\ t_1,\ ...,\ t_n) \in S_A$ representing one possible execution of B, i.e., a list of tasks executed to reach an end event of B from the initial event of B. The set of all traces of B is written T_B. A BPMN process may have an infinite set of traces, but each trace is finite (i.e., loops do not repeat infinitely).*

The refactoring patterns that were presented consist in moving a task from one place of the process to another. Thus, they modify the semantics of the original process. For instance, two tasks originally executed in sequence may be executed in parallel in the refactored process. Nonetheless, the patterns change the position of a single task in the process at once. This ensures a trace equivalence [10] (modulo the moved task) between each refactoring step. Globally, this means that each trace of the refactored process is a permutation of a trace of the original process, and vice versa.

Proposition 2. *(Traces Permutation Property) Let $B = (S_O, S_F)$ be a BPMN process, T_B be its set of traces, B' be a process generated by applying the refactoring patterns to B, and $\forall \tau \in T_B$, $\mathfrak{S}_\tau$ be the set of permutations of τ. The following properties are preserved by the patterns:*

1) *$\forall \tau \in T_B,\ \exists \tau' \in T_{B'} \mid \tau' \in \mathfrak{S}_\tau$, meaning that for each trace of B, there exists a permutation of this trace in B'.*
2) *$\forall \tau' \in T_{B'},\ \exists \tau \in T_B \mid \tau \in \mathfrak{S}_{\tau'}$, meaning that for each trace of B', there exists a permutation of this trace in B.*

Proof. (Sketch) By considering the syntax supported in this work, there are only four different ways to move a task of a process while unsatisfying the previous proposition. To do so, it must either enter a choice, exit a choice, enter a loop, or exit a loop. Let us now detail these four cases.

1) If a task enters a choice of B, its execution will become conditional, thus some original traces of B will have no permutation in B'. As the patterns are applied on abstract graphs and tasks, they never consider conditional structures. The only exception is pattern 4 that puts a task inside a choice. However, it introduces the task in each path of the choice. Thus, the execution of the task remains unconditional.
2) If a task exits a choice of B, its execution, which was originally conditional, will become unconditional. Thus, some traces of B' will have no permutation in the original process B. However, patterns are applied to the abstract graph being the most internal structure of the task to move. By construction, they can not be applied on outer structures, and thus can not put a task out of a choice in which it originally belonged to.
3) If a task enters a loop of B, it can be executed more than once, when it was not originally possible. Thus, some traces of B' will have no permutation in the original process B. Similarly to item 1), patterns are only applied on abstract graphs and tasks, and do not manage conditional structures. Thus, a task can never enter a loop.
4) If a task exits a loop of B, it can only be executed once, when it was originally possible to execute it several times. Thus, some original traces of B will have no permutation in B'. Similarly to item 2), as patterns are not applied on outer structures, they can not put a task out of a loop.

□

3.6 Generating the Best Refactoring Steps

Once the abstract graphs are generated by the patterns, the one leading to the best optimisation must be proposed to the user. To compute it, the first and naive method consists in keeping all the generated processes at each step, and repeating the generation for each task of the process. This method generates the whole tree of possibilities, each leaf corresponding to a candidate process. However, such an exhaustive exploration, despite returning the optimal process, is not appliable in a real-time context. Concretely, for a process containing 15 tasks and in which each task can be moved to 20 positions, the tree of possibilities contains $15^{20} = 3 \times 10^{23}$ leafs. As simulating a single process takes at least milliseconds, simulating such an important number of processes is not possible. Consequently, heuristics are required to reduce the number of processes explored at each step.

3.6.1 Heuristics.

To limit the size of the explosion, the processes that are far from the optimal one are discarded at each step. This can be done in several ways: a bounded depth-first search returning the process with shortest AET after some steps, a score attributed to each abstract graph to measure its quality, a structural analysis evaluating the balance of parallelism and sequentialisation, a mix between several ones, etc. Based on an empirical study made on dozens of examples, the option giving the best trade-off between quality of the result and execution time is the one for which a score is attributed to each abstract graph. This score is computed based on the metrics returned by the simulation of each generated abstract graph. In the end, the abstract graph obtaining the highest score is elected. By doing so, the number of abstract graphs generated and simulated at each step never exceeds the number of possible positions of the task being moved.

The computation of the score is based on variations of two different metrics: the AET and the resource usage of the process. It also keeps track of the history of the current path, that is, the processes that have been elected in anterior steps, to adjust the score. These two raw metrics serve as basis for the computation of the five main metrics composing the score. The first three make use of the AET, while the last two make use of the resource usage. The three metrics based on the AET are the *AET mean difference*, the *AET standard deviation difference*, and the *AET local difference*.

Definition 10. *(AET Mean Difference) Given a history H composed of n anterior processes $(p_1, ..., p_n)$ and a current process p_{n+1}, the AET mean difference is computed as $\delta_{\mu_{AET}} = \delta_{\mu_{AET_o}} - \delta_{\mu_{AET_n}}$, where $\delta_{\mu_{AET_n}} = \frac{1}{n+1} \sum_{i=1}^{n+1} AET_{p_i}$ and $\delta_{\mu_{AET_o}} = \frac{1}{n} \sum_{i=1}^{n} AET_{p_i}$.*

Definition 11. *(AET Standard Deviation Difference) Given a history H composed of n anterior processes $(p_1, ..., p_n)$ and a current process p_{n+1}, the AET standard deviation difference is computed as $\delta_{\sigma_{AET}} = \delta_{\sigma_{AET_o}} - \delta_{\sigma_{AET_n}}$, where $\delta_{\sigma_{AET_n}} = \sqrt{\frac{\sum_{i=1}^{n+1} (AET_{p_i} - \delta_{\mu_{AET_n}})^2}{n+1}}$ and $\delta_{\sigma_{AET_o}} = \sqrt{\frac{\sum_{i=1}^{n} (AET_{p_i} - \delta_{\mu_{AET_o}})^2}{n}}$.*

Definition 12. *(AET Local Difference) Given an anterior process p_n and a current process p_{n+1}, the AET local difference is computed as $\delta_{AET} = AET_{p_n} - AET_{p_{n+1}}$.*

The remaining metrics take into consideration the resource usage of the selected processes at each step. For the computation of these metrics, the pool of resources R_p used by the process p is considered, and a function $usg(r, p)$ returning the average usage of resource $r \in R_p$ over the execution of p is introduced. The two metrics based on the resource usage are the *resource usage mean difference* and the *resource usage local difference*.

Definition 13. *(Resource Usage Mean Difference) Given a history H composed of n anterior processes*

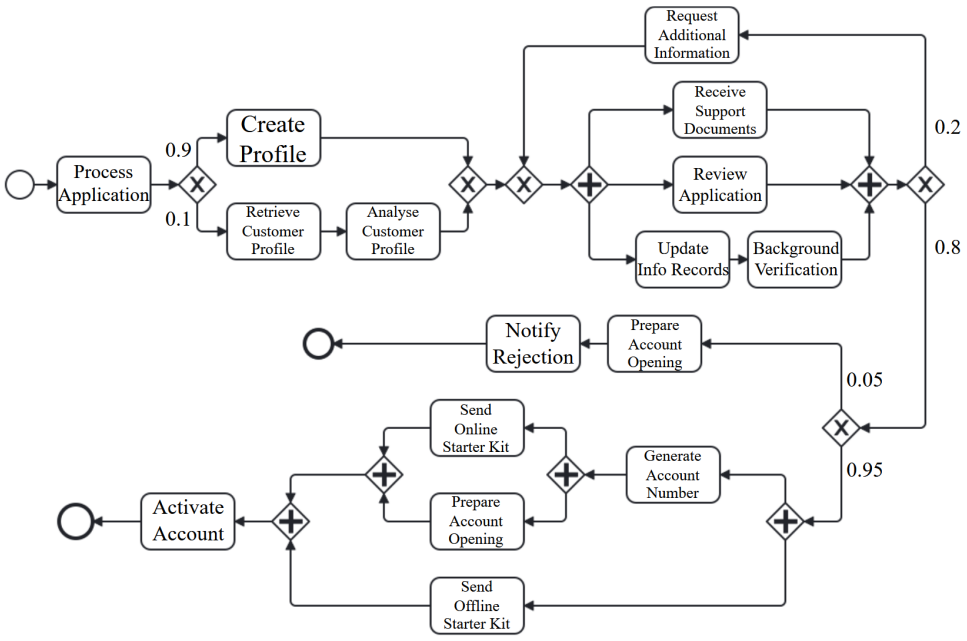


Figure 8: Refactored Account Opening Process

$(p_1, ..., p_n)$ *and a current process* p_{n+1}*, the resource usage mean difference is computed as* $\delta_{\mu_{res}} = \delta_{\mu_{res_n}} - \delta_{\mu_{res_o}}$*, where* $\delta_{\mu_{res_n}} = \frac{1}{n+1} \sum_{i=1}^{n+1} (\frac{1}{|R_p|} \sum_{r \in R_p} (usg(r, p_i)))$ *and* $\delta_{\mu_{res_o}} = \frac{1}{n} \sum_{i=1}^{n} (\frac{1}{|R_p|} \sum_{r \in R_p} (usg(r, p_i)))$.

Definition 14. *(Resource Usage Local Difference) Given an anterior process* p_n *and a current process* p_{n+1}*, the resource usage local difference is computed as* $\delta_{res} = \frac{1}{|R_p|} \sum_{r \in R_p} (usg(r, p_{n+1}) - usg(r, p_n))$.

These five metrics are then normalised to obtain a value between 0 and 1 for each of them. Finally, the *score* of the current process is computed as the weighted sum of these five metrics.

Definition 15. *(Score of a Process) Let* p *be a process,* $\delta_{\mu_{AET}}$*,* $\delta_{\sigma_{AET}}$*,* δ_{AET}*,* $\delta_{\mu_{res}}$ *and* δ_{res} *be the previously defined metrics computed on process* p*, and* ω_{AET}*,* ω_{res} *and* ω_{loc} *be the weights respectively attributed to the metrics based on AET, resource usage, and previous process information. The score of* p *is computed as* $s(p) = \omega_{AET} \times (\delta_{\mu_{AET}} + \delta_{\sigma_{AET}} + \omega_{loc} \times \delta_{AET}) + \omega_{res} \times (\delta_{\mu_{res}} + \omega_{loc} \times \delta_{res})$.

This score is a weighted sum between all the previously defined metrics, where the decrease of AET and the increase of resource usage gives a high score. Indeed, the objective is to use the resources as much as possible while avoiding an over-usage. The weights attributed to the metrics can be given by the user as an input of the approach. If none are given, the default ones, based on an empirical study made on dozens of examples, are $\omega_{AET} = 0.6$, $\omega_{loc} = 0.5$, $\omega_{res} = 1.0$. Finally, the process obtaining the highest score is shown to the user.

Example. Figure 8 shows the refactored account opening process obtained by applying all the refactoring steps proposed by the heuristic to the initial account opening process in Figure 1. For the sake of space, resource usage and duration of the tasks have been removed. As the reader can see, several positions have changed: task *Review Application* entered the parallel gateway next to it, or more interestingly, task *Prepare Account Opening* entered both paths of the final choice, to reach a parallel gateway in the second path. In this case, it may seem counter-intuitive for the user to put task *Prepare Account Opening* right before *Notify Rejection*, but this position of the task allows a significant optimisation in the other branch of the choice. Also, the (strong) dependencies of the process introduced in the example of Section 2.2 have been preserved. This refactoring allowed the user to obtain a process having an AET of 40.9UT instead of the original one having an AET of 51.9UT.

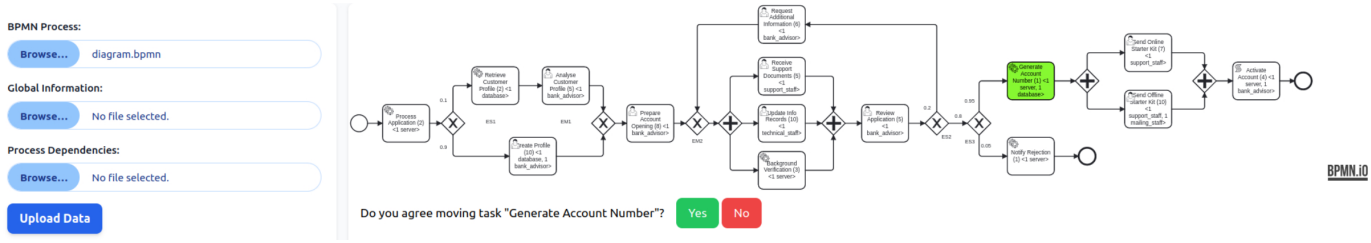
4. Tool and Experiments

This section gives information about the tool support of this approach, and describes the experiments conducted to evaluate and validate it.

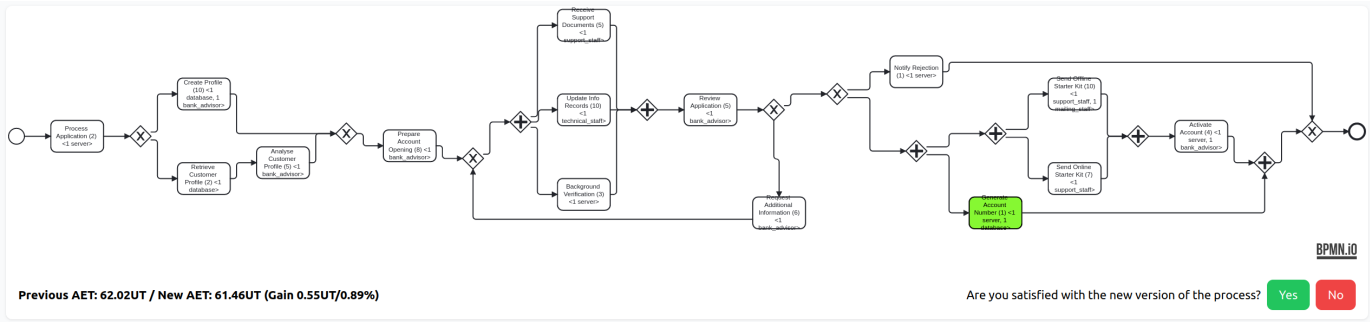
4.1 Tool Support

The approach has been fully implemented by a tool written in Java and consisting of approximately 15,000 lines of code. It has been fully tested and evaluated on several real-world examples and hundreds of handcrafted examples. It is freely available online at [9]. For a more user-friendly usage, it was embedded in the backend of a web server. Figure 9 shows two screenshots of the frontend of the web server that the user can use.

In Figure 9a, the user has uploaded her/his BPMN process, the global information (IAT, shared resources, number of instances) and the dependencies of the process. The BPMN process is then displayed using the "bpmn.io" API, and the



(a) Tool Proposing the Task to Move to the User



(b) Tool Showing the Generated Process to the User

Figure 9: Screenshots of the Tool Support of the Approach

tool proposes the first task to move, i.e., "Generate Account Number". It also asks the user whether (s)he agrees to move this task or not. If the user accepts, the tool then computes the best position of this task, and displays it on the screen, as shown in Figure 9b. Otherwise, it proposes another task until the user accepts to move a task. Here, as the user accepts, the resulting process is shown, and the gain of this new process compared to the previous one is displayed. The user is finally asked to accept or decline this new process, which triggers in both cases a new iteration.

4.2 Experiments

The experiments described in this section aim at assessing the performance of the tool, as well as the quality of the process returned by the score-based heuristic in terms of AET. To do so, the results obtained by the heuristic are compared to the ones obtained by the full exploration on several BPMN processes executed 100 times with multiple resources, both in terms of AET and computation time of the tool. This analysis has been conducted on an HP EliteBook x360 1030 G8 Notebook PC running with an Intel Core i5-1145G7 @ 2.60GHz VPRO and 16GB of RAM. The results of this analysis are given in Table 1. Column 1 presents the considered BPMN process with its name and its origin. Columns 2, 3, 4, 5 give information about the process, respectively, its number of tasks, choices, and dependencies, and its initial AET. Columns 6, 7, 8, 9 provide metrics about the process obtained by using the heuristic. They contain respectively the AET of the generated process, the gain in percentage compared to the original AET, the time taken by the tool to perform the whole computation, and the time taken by the tool to compute one step, on average. Column 10 shows the AET of the process obtained by using the full exploration, and column 11 its gain compared to the original process. As the full exploration time per step exceeds minutes or even dozens of minutes for certain processes, the execution time of the full exploration is not presented in the table. The star symbol (*) in column 6 (AET) highlights cases where the fourth pattern was (at least partially) discarded due to the important number of generated processes. It is worth noting that, as these results aim at assessing the performance of the tool and the quality of the heuristic, they were performed under the assumption that the user accepts all the proposed steps.

As the reader can see, the heuristic performs very well on real-world examples, as it returns processes with an AET close to the optimal one (and even the optimal one for example 6). The worst result obtained by the heuristic on real-world examples is for example 5, for which the gain of the heuristic is 33.5% worse than the gain of the full exploration. On average, the heuristic performs only 13.9% less well than the full exploration, while being much faster. The approach also performs in reasonable time on real-world examples, with a time per step reaching 5 seconds at most. On the first handcrafted example, containing 26 tasks, the approach still obtains a non-negligible gain of 37.5% in a reasonable time per step (15s). Finally, the approach shows some limitations on the last two handcrafted examples, which are two variants of a 51 tasks process with a different number of dependencies, and for which the execution time per step is no longer acceptable in a real-time context (more than 1m). It is worth noting that, as expected, the number of dependencies and the size of the process play an important role in the computational time of the approach.

5. Related Work

Beyond process refactoring, other optimisation techniques exist for BPMN processes. Task scheduling consists in priorising the execution of an instance's task among all the available instances, according to an orchestration policy. Similarly to our proposal, this method allows optimisation without any additional cost, as only the order in which the instances are executed is modified. However, it does not prevent nor allow to correct ill-formed processes, as the structure of the process remains the same throughout the optimisation phase. Other works, such as [1][2], consist in computing the optimal pool of resources for a given process. This usually requires flexibility in the budget of the companies, as one may have to hire a new employee or buy a new machine. Comparatively, our method does not require any budget adjustment or increase.

The rest of this section focuses on existing works on process refactoring. Reference [16] presents six common mistakes made by developers when modelling with BPMN. For each problem, the authors present best practices for avoiding these issues. As an example, the authors propose to use explicit gateways instead of using multiple incoming/outgoing sequence flows. Reference [17] presents a technique for detecting refactoring opportunities in process model repositories. The technique works by first computing activity similarity and then computing three similarity scores for fragment pairs of process models. Using these similarity scores, four different kinds of refactoring opportunities can be systematically identified. As a result, the approach proposes to rename activities or to introduce subprocesses. IBUPROFEN, a business process refactoring approach based on graphs, is presented in [18][19]. IBUPROFEN defines a set of 10 refactoring algorithms grouped into three categories: maximisation of relevant elements, fine-grained granularity reduction, and completeness. All these works mostly focus on syntactic issues and propose syntactic improvements of the process by, for instance, removing unreachable nodes or by merging consecutive gateways of the same type. They do not aim at providing any kind of optimisation regarding the process being designed as we do.

In [20], the authors present an approach for optimising the redesign of process models. It is based on capturing process improvement strategies as constraints in a structural-temporal model. Each improvement strategy is represented by a binary variable. An objective function that represents a net benefit function of cost and quality is then maximised to find the best combination of process improvements that can be made to maximise the objective. The BPMN subset used in [20] is very similar to the one we use in this paper. However, the approach is rather different since they compute optimal redesigns with respect to some constraints, whereas we propose process refactoring with respect to resource usage and execution times. Reference [8] proposes a semi-automated approach for helping non-experts in BPMN to model business processes using this notation. Alternatively, [21] presents an approach which combines notes taking in constrained natural language with process mining to automatically produce BPMN diagrams in real-time as interview participants describe them with stories. In this work, we tackle this issue from a different angle since we assume that an existing description of the process exists and that we want to automatically optimise it by updating its structure. In [3], the authors propose a refactoring procedure whose final goal is to reduce the total execution time of the process given as input. This solution relies on refactoring operations that reorganise the tasks in the process by taking into account the resources used by those tasks. However, this approach only considers single executions of the BPMN process and a single replica of each resource, while the approach proposed here applies refactoring techniques on processes executed multiple times, and making use of several replicas of each resource.

In [4], the authors present an approach aiming at optimising a BPMN process with refactoring techniques. The processes involved support a similar syntax and are also enriched with time and resources. However, the approach differs in many points. The first step of their approach consists in building the most parallel version of the process respecting its dependencies. Then, the authors make use of an internal calculation to compute the pool of resources needed by the process to execute without latencies. This pool is later compared to the available pool of resources. If the available pool is large enough, the most parallel process is returned to the user. Otherwise, some tasks are removed from parallel gateways

TABLE 1: Experimental results

BPMN Process	Characteristics				Heuristic				Full Exploration	
	Tasks	$\diamondsuit_C$	Dep.	AET	AET	Gain	Time	μ_{Time}	AET	Gain
1. Evisa App. [11]	8	1	3	36.1	20	44.6%	6.21s	0.88s	17.1	52.6%
2. Employee Recrui. [8]	10	1	9	30.9	21.4	30.7%	32s	0.32s	20.4	34.0%
3. Patient Diagnosis [12]	8	2	3	67.2	61.6	8.33%	5s	0.56s	60.4	10.1%
4. Employee Hiring [13]	11	2	7	24.7	19	23.1%	26s	2.36s	17.8	27.9%
5. Account Opening [14]	15	2	7	51.9	40.9	21.2%	1.25m	5s	34.2	34.1%
6. Perish. Goods Trans. [15]	16	2	10	15	13.2	12.0%	14s	1.75s	13.2	12.0%
7. Online Shipping [1]	24	3	27	85.9	70.3	18.2%	1.97m	4.9s	69.3	19.3%
8. Handcrafted 1	26	1	43	232	145*	37.5%	6.37m	15s	122	47.4%
9. Handcrafted 2a	51	1	43	323	244*	24.5%	58m	1.14m	183	43.3%
10. Handcrafted 2b	51	1	23	323	182*	43.7%	2h07	2.54m	99	69.3%

and put in sequence to limit synchronisation issues. The main difference between our approach and theirs is that we involve the user in the loop and thus perform the task moves one after the other. This difference requires to rethink the refactoring method by providing ways of finding the best position of a single task in the process (the patterns coupled to the heuristics here). Among other things, our approach also supports probabilistic IAT and tasks duration, and make use of simulation to obtain more accurate results.

6. Concluding Remarks

This paper has presented techniques aiming at optimising the execution time of a business process by applying iteratively refactoring operations on it. The approach considers BPMN processes enriched with time and resource usage which are executed multiple times. To optimise the execution time, the approach proposes to the user a task to move and a new position for it. Depending on the decision of the user, the task is moved or not. After modifying the position of each task, the iteration stops and the user obtains an optimised version of the original process. The whole approach is fully automated by using a tool that was implemented and applied to a set of real-world and handcrafted processes in order to evaluate the quality of its results and its performance. The experiments showed satisfactory results both in terms of optimisation and computation time.

This work offers several interesting perspectives. The first one is to allow the user to select the desired optimisation axis among several possible ones, such as execution time, resource usage or monetary cost. The second one is to enrich the model with data, in order to allow more precise definitions of the processes. The last one is to get rid of simulation in order to reduce the waiting time of the user and fasten the computation.

7. Acknowledgments

This work is supported by the French National Research Agency in the framework of the "France 2030" program (ANR-15-IDEX-0002) and by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01).

References

[1] F. Durán, C. Rocha, and G. Salaün, "A rewriting logic approach to resource allocation analysis in business process models," *Sci. Comput. Program.*, vol. 183, 2019.

[2] F. Durán, C. Rocha, and G. Salaün, "Resource provisioning strategies for BPMN processes: specification and analysis using maude," *J. Log. Algebraic Methods Program.*, vol. 123, p. 100711, 2021.

[3] F. Durán and G. Salaün, "Optimization of BPMN processes via automated refactoring," in *Proc. of ICSOC'22*, vol. 13740 of *LNCS*, pp. 3–18, Springer, 2022.

[4] Q. Nivon and G. Salaün, "Refactoring of multi-instance BPMN processes with time and resources," in *Proc. of SEFM'23*, LNCS, Springer, 2023.

[5] W. van der Aalst, *Process Mining*. Springer Berlin, Heidelberg, 2016.

[6] R. Eshuis and P. V. Gorp, "Synthesizing data-centric models from business process models," *Computing*, vol. 98, no. 4, pp. 345–373, 2016.

[7] F. Durán, C. Rocha, and G. Salaün, "Symbolic specification and verification of data-aware BPMN processes using rewriting modulo SMT," in *Proc. of WRLA'18* (V. Rusu, ed.), vol. 11152 of *LNCS*, pp. 76–97, Springer, 2018.

[8] Y. Falcone, G. Salaün, and A. Zuo, "Semi-automated modelling of optimized BPMN processes," in *Proc. of SCC'21*, pp. 425–430, IEEE, 2021.

[9] Q. Nivon, "Automated tool for step-by-step refactoring of multi-instance bpmn processes." https://github.com/QuentinNivon/Step-by-Step_Refactoring, 2023.

[10] R. Milner, *Communication and concurrency*. Prentice Hall, 1989.

[11] G. Salaün, "Quantifying the similarity of BPMN processes," in *Proc. of APSEC'22*, pp. 1–10, 2022.

[12] E. Bazhenova, F. Zerbato, B. Oliboni, and M. Weske, "From BPMN process models to DMN decision models," *Information Systems*, vol. 83, pp. 69–88, 2019.

[13] F. Durán, C. Rocha, and G. Salaün, "Computing the parallelism degree of timed BPMN processes," in *Proc. of FOCLASA'18*, pp. 1–16, 2018.

[14] Q. Nivon and G. Salaün, "Debugging of BPMN processes using coloring techniques," in *Proc. of FACS'22*, pp. 90–109, Springer, 2022.

[15] P. Valderas, V. Torres, and E. Serral, "Modelling and executing IoT-enhanced business processes through BPMN and microservices," *J. Syst. Softw.*, vol. 184, p. 111139, 2022.

[16] D. Silingas and E. Mileviciene, "Refactoring BPMN models: From 'bad smells' to best practices and patterns," in *BPMN 2.0 Handbook*, pp. 125–134, 2012.

[17] R. M. Dijkman, B. Gfeller, J. M. Küster, and H. Völzer, "Identifying refactoring opportunities in process model repositories," *Inf. Softw. Technol.*, vol. 53, no. 9, pp. 937–948, 2011.

[18] M. Fernández-Ropero, R. Pérez-Castillo, and M. Piattini, "Graph-based business process model refactoring," in *Proc. of the 3rd Int. Symposium on Data-driven Process Discovery and Analysis*, vol. 1027 of *CEUR Workshop Proceedings*, pp. 16–30, 2013.

[19] R. Pérez-Castillo, M. Fernández-Ropero, and M. Piattini, "Business process model refactoring applying IBUPROFEN. An industrial evaluation," *J. Syst. Softw.*, vol. 147, pp. 86–103, 2019.

[20] A. Kumar and R. Liu, "Business workflow optimization through process model redesign," in *Proc. of TEM'22*, LNCS, pp. 3068–3084, Springer, 2022.

[21] A. Ivanchikj, S. Serbout, and C. Pautasso, "From text to visual BPMN process models: design and evaluation," in *Proc. of MODELS'20*, pp. 229–239, ACM, 2020.