Refactoring of Multi-Instance BPMN Processes with Time and Resources

Quentin Nivon and Gwen Salaün

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

Abstract. Business process optimisation is a strategic activity in organisations because of its potential to increase profit margins and reduce operational costs. In this paper, we focus on a specific technique used for process optimisation known as process refactoring. In this work, a process is described using BPMN extended with quantitative aspects for modelling execution times and resources associated with tasks. A process is not executed once but multiple times, and multiple concurrent executions of a process compete for using the shared resources. In this context, we propose a refactoring approach whose goal is to reduce the total execution time of the process and optimise the usage of the shared resources. To do so, we first analyse the given process in terms of task dependency and resource usage, and then rely on these results to restructure the process and return an optimal version of it. This process refactoring technique is fully automated by a tool that we implemented and applied on several examples for validation purposes.

1 Introduction

Context. Process optimisation is a strategic activity in companies and organisations because it is a source of improvement in terms of throughput, resource usage and associated costs. However, this is a difficult task, which requires a precise description and understanding of these processes. Process optimisation can be achieved in different ways. A first option is to compute metrics of interest or precise recommendations to effectively change and improve manually the aforementioned processes. Another option is to automatically compute a new version of the process, which is an improved version of the original process. In both cases, optimisation focuses on one or several specific criteria (execution time, costs, resource usage, carbon footprint, etc.).

In this paper, we assume that processes are described using BPMN 2.0 (BPMN, as a shorthand, in the rest of this paper). BPMN was published as an ISO/IEC standard in 2013 and is nowadays extensively used for modelling and developing business processes. Additional quantitative information is required to be able to precisely analyse and then optimise the process given as input. Therefore, in this work, the process model also includes 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.

Motivation. When considering processes with time and resources, the two main optimisation criteria are execution time/throughput and better usage of resources (usually resulting in a reduction of associated costs). Operational research offers several techniques to solve such optimisation problems. However, they are difficult to apply in this context, due to the expressiveness of the model and the multiple instances of the process running in parallel, which would increase a lot the operational cost of such techniques. Another solution to this problem is *resource optimisation*, see e.g. [6, 8], but this solution usually implies some flexibility in terms of budget because the solution may propose to increase the number of certain resources so as to obtain better results. If the number of resources is fixed and cannot be updated, an alternative solution is to change the organisation of the tasks within the process. This solution is usually called *process refactoring*, and aims at restructuring the process to optimise the aforementioned criteria.

Refactoring a process manually is however a very difficult task. A naive solution could be to increase the level of parallelism, but this solution does not systematically work in the case of multiple executions of a process, which may increase the competition (thus time) to acquire the resources. Therefore, there is a need for automated refactoring techniques in order to optimise a process. Such techniques can be used in different contexts, for improving an existing process or at design time for optimising a new process before effectively deploying it.

Approach. Given a BPMN process model, we propose new optimisation techniques that aim at automatically restructuring the given process. These refactoring techniques generate a different process which has the shortest execution time in the best case, or is close to the shortest otherwise. This new process is said to be optimal or close to optimal, given a number of shared resources. The main idea is to adjust the structure of the process to avoid competition of resources and bottlenecks, to obtain smoother executions of multiple instances of the process. More precisely, in a first step, the approach transforms the process into an optimal version, structurally speaking, by putting as many tasks in parallel as possible. This intermediate process is optimal as it has the shortest execution time possible. We then compute the pool of resources needed by the process to execute without having concurrency issues while accessing resources. If this pool of resources is smaller than the actual number of shared resources, we return this process. If not, more resources would be required, but in this work we do not want to change the number of shared resources. Therefore, we change again the structure of the process to decrease the degree of parallelism of certain tasks and thus remove the identified competition on some specific resources. As a result, we return a refactored process whose execution time is close to optimal, or optimal in the best case. The whole approach is fully automated by a tool that we implemented. Even though the overall complexity of the approach is exponential, the experimental results obtained on several real-world and handcrafted examples were satisfactory, as none of the execution exceeded a few seconds.

Structure. Section 2 introduces the languages and models used in this paper. Section 3 presents the different steps of the refactoring approach proposed in this paper. Section 4 describes the tool support and some experimental results to assess

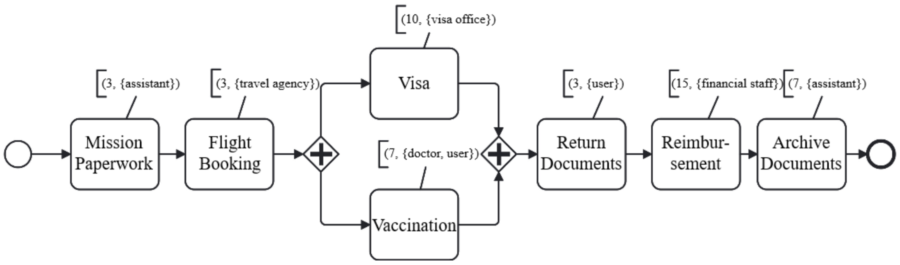


Fig. 1: Example of "Trip Organisation" process in BPMN [9]

the performance of the approach. Section 5 compares our solution to related work and Section 6 concludes this paper.

2 Models

BPMN with Time and Resources. In this paper, we focus on BPMN activity diagrams including the constructs related to control-flow modeling 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 describes two nodes executed one after the other in a specific execution order. A task may have a duration or delay, expressed by default in *units of time* (UT). 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 is going to execute for the specified duration. The acquisition of a resource is achieved in a "first-come-first-served" strategy. If a task needs more replicas of one or several resources than those available, it remains in a waiting state until the release of a sufficient number of replicas of the required resources.

Gateways are used to control the divergence and convergence of the execution flow. We consider in this work the two main kinds of gateways used in activity diagrams, namely, *exclusive* and *parallel* gateways. The difference between them is that only one outgoing flow of an exclusive gateway is executed (choice), while all the outgoing flows of a parallel gateway are executed. Data-based conditions for exclusive split gateways are modelled using probabilities associated to outgoing flows.

Example. Figure 1 shows an example of BPMN process enhanced with time and resources. Each task has a duration and makes use of resources. For example, task *Vaccination* takes 7 units of time (days here) to execute in average and requires one replica of resources *doctor* and *user*.

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. In this approach, we perform a restructuring of the process. Thereby, 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). Providing these dependencies is required in this work and is an information complementary to the BPMN process. They can be given by the user or computed by analysing the data-flow graph corresponding to the BPMN process [10, 5].

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. When there are several dependencies, they can be concisely represented using a *dependency graph*, which is a directed acyclic graph where each node corresponds to a task and each edge to a dependency between two tasks.

Example. Figure 1 shows an example of trip organisation process. In this example, the order of some tasks must not be changed by the refactoring techniques. For example, documents have to be returned by the user before being archived. Thus, task *Return Documents* must be executed before task *Archive Documents*.

Abstract Graphs. An abstract graph is an internal representation of a BPMN process that we use in this work as an intermediate format. It was originally introduced in [11] 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. It worth noting that this representation has the same expressiveness as the subset of BPMN that is supported in this work.

Definition 1. *(Abstract Graph) An abstract graph is a (hierarchical) directed graph* (S_N,S_E) *where* S_N *is a set of nodes and* S_E *a set of directed edges 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* S_G *a set of abstract (sub-)graphs.*

Given a set of dependencies, an abstract graph can be generated if and only if the two following conditions are satisfied:

- **Condition 1:** For any task *T”*, if *T”* is a common successor of two tasks *T* and *T’*, then the set of successors of *T* should be equal to the set of successors of *T’*.
- **Condition 2:** For any task *T*, if *T* is a common predecessor of two tasks *T’* and *T”*, then the set of predecessors of *T’* should be equal to the set of predecessors of *T”*.

If these conditions are not satisfied, several valid abstract graphs may be generated from the given dependencies.

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, we mostly consider the time taken by a process to execute since it is our main optimisation goal. The *execution time* of a process corresponds to the difference between the timestamp at which the last token has reached an end event and the timestamp at which the initial token has left the start event. This time varies depending on the structure of the process, the use of gateways or loops, but it 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 two notions of execution times. The first one, called *worst-case execution time*, corresponds to the longest time taken by a BPMN process to complete its execution. Indeed, conditional structures, such as loops or choices, may lead to several different execution times for a single process.

Definition 2. *(Worst-Case Execution Time) Let B be a BPMN process and* $S_{ET} = \{ET_1, ET_2, ..., ET_n\}$ *the set of all possible execution times of B. The worst-case execution time of B is defined as* $WCET_B = \max\,(S_{ET})$.

The second execution time considered, called *average execution time*, is only relevant in a multi-instance context. It represents the time taken by each instance of the process to complete its execution, on average.

Definition 3. *(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* $AET_B = \frac{1}{n}\sum_{i=1}^{n} ET_{B_i}$.

These execution times can be computed using simulation techniques [7]. These simulation techniques and the resulting execution times highly depends on the *workload* of a process, 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. 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 to merge parallel gateways corresponds to the time elapsed between the arrival of the first token through one of its incoming flows and the arrival of the last token (thus resulting in the activation of that merge). These merge gateways are often seen as bottlenecks because they induce additional delays. 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.

Optimality. The main goal of this work is to improve the average execution time of the original process by applying refactoring techniques to it. The final process, returned to the user, is in the best case *optimal*.

Definition 4. *(Optimal Process) Let B be a BPMN process,* S_D *the set of dependencies of B, and* $S_{B_D} = \{B_1, B_2, ..., B_n\}$ *the set of processes generated from B and satisfying* S_D. $\exists B_i \in S_{B_D}$, B_i *is optimal if and only if* $\min(AET_{B_1}, AET_{B_2}, ..., AET_{B_n}) = AET_{B_i}$.

3 Refactoring

Overview. The approach proposed in this paper aims at automatically restructuring a BPMN process in order to minimise 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. We recall that in this work, a process is executed multiple times, the beginning of execution of each instance being separated from the previous one by the IAT. Our refactoring technique does not change the tasks themselves, but the way they are organised within the process. The main idea behind our approach is that we increase the parallelism of the process as much as possible but not systematically. For instance, if adding parallelism results in the increase of resource competition or bottlenecks coming from synchronisation delays (at merge gateways), we prefer to keep a sequential organisation of tasks. At the end, the approach returns a new BPMN process whose average execution time is optimal or close to the optimal. It is worth noting that, except for the position of the tasks, the semantics of the process is preserved (i.e., a task that is not necessarily executed (choice), or that is possibly repeated (loop) has the same behaviour in the final process).

Figure 2 gives an overview of the mains steps of the approach. Beyond the previously mentioned inputs, our approach also needs as input a set of task dependencies. These dependencies correspond to some strong ordering of tasks that cannot be changed by our refactoring approach and thus must be preserved in the final process. They can be given by the user or computed by analysing the data-flow graph corresponding to the BPMN process [10, 5]. The first step (1) aims at computing an abstract graph corresponding to the given dependencies. This abstract graph is an intermediate format which is used throughout this work to simplify computations and restructuring that will finally lead to the resulting process. In some cases however, as explained in the previous section, the abstract graph cannot be generated because some conditions on the dependencies are not satisfied. Concretely, this means that several abstract graphs satisfy the given dependencies. Since the goal here is to generate an optimal process, we need to choose, among the possible graphs satisfying the conditions, the one that is optimal with respect to our goal. To do so, we explore all possible additional dependencies for which the conditions become true, and among all these solutions we keep the one making the resulting abstract graph exhibit the shortest execution time.

The next step (2) takes into account the shared resources and verifies whether they are sufficient to execute the current abstract graph smoothly, that is, without resource competition or synchronisation delays at merge gateways. If this is the case, the resulting BPMN is synthesised and is optimal. If the shared resources are not sufficient, two more steps are performed in the approach. Step (3) analyses the current abstract graph, identifies the different sources of competition/synchronisation issues, and rate them using a scoring system. Repeatedly, the task identified as the highest issue is removed from its parallel structure and put in sequence, until the shared resources allow a smooth execution of the process-to-be. Step (4) aims at refining the organisation of tasks within the abstract graph in order to find the best structure between the tasks that should be put in sequence and the others. When Steps (3) and (4) need to be applied, the resulting BPMN process may not be optimal, but is close to the optimal solution.

The rest of this section gives additional details on each step of the approach.

Computing Abstract Graph from Dependencies. The first step of this approach consists of computing the abstract graph corresponding to the process dependencies.

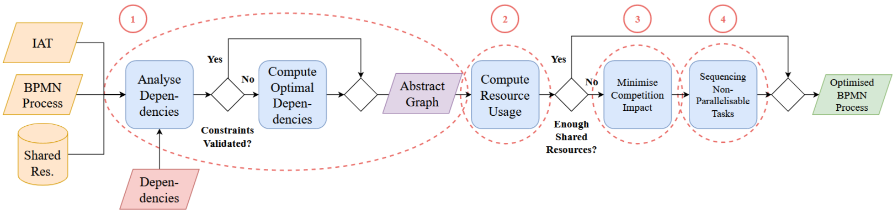


Fig. 2: Overview of the Approach

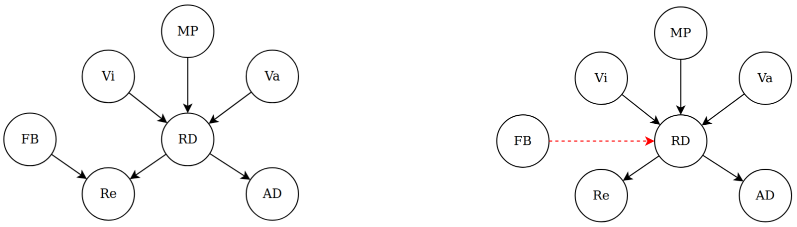
This can be done using the algorithm proposed in [11], when the conditions stated by the authors are respected. These conditions ensure the uniqueness of the generated abstract graph. Thus, several abstract graphs respecting the dependencies of the process may exist when the conditions are not satisfied. Since our main concern is to generate an optimal abstract graph in terms of execution time, we need to pick the one with the shortest execution time. To do so, our idea is to iteratively add new dependencies to the initial ones, until obtaining a set of dependencies that satisfies the conditions. By doing so, we obtain all the possible abstract graphs satisfying the dependencies. Then, we keep the one with the shortest execution time.

Algorithm 1 Optimal Dependencies Finding

```
    Inputs: $G = (G_V, G_E)$ (Dependency graph built from the initial dependencies)
    Output: $G_O$ (Optimal dependency graph validating the conditions)
$S_{CB} \leftarrow \{G\}$ ; $S_{NB} \leftarrow \emptyset$ ; $S_{PO} \leftarrow \emptyset$
while $S_{CB} \neq \emptyset$ do                                   ▷ Are there pending graphs?
    for $G_C \in S_{CB}$ do                                        ▷ Yes, iterate over each of them
        for $n \in G_{C_V}$ do                          ▷ Then over each node of the current graph
            $S_{ND} \leftarrow \mathit{findNonAlreadyDependentNodesOf}(n)$
            for $nd \in S_{ND}$ do                   ▷ Then over each non already dependent node
                $G'_C \leftarrow G_C.\mathit{copy}()$ ; $G'_C.\mathit{addDependencyBetween}(n, nd)$
                $G'_C.\mathit{removeShortestPathBetweenNodes}(n, nd)$
                if $\mathit{conditionsSatisfied}(G'_C)$ then $S_{PO} \leftarrow S_{PO} \cup \{G'_C\}$
                else $S_{NB} \leftarrow S_{NB} \cup \{G'_C\}$
                end if
            end for
        end for
    end for
    $S_{CB} \leftarrow S_{NB}$ ; $S_{NB} \leftarrow \emptyset$
end while
return $\mathit{findOptimalGraph}(S_{PO})$
```

This computation is performed by Algorithm 1. This algorithm takes as input the dependency graph built from the initial dependencies of the process, which is moved in the set of pending graphs S_{CB}. Then, for each pending graph, we iterate over its tasks (i.e., nodes) and add a new dependency between this task and another task that is not already dependent nor transitively dependent of the current task (line 7). As this new dependency may generate a path (i.e., a (transitive) dependency) between two nodes that were already connected, the shortest path between these nodes is removed (i.e., the edges composing it are removed) in order to avoid the duplication of dependencies, while preserving the original ones (line 8). If this new dependency makes the constraints be satisfied, the current graph is added to the set of possibly optimal graphs S_{PO}. Otherwise, it is added to the set of new pending graphs S_{NB}. At the end of an iteration, S_{NB} is put in S_{CB} and if it is not empty, a new iteration starts. When the while loop finishes (line 16), each set of dependencies validating the conditions has been built and put in the set of possibly optimal dependencies S_{PO}. Then, the algorithm generates the abstract graph corresponding to each possibly optimal set of dependencies and computes its worst-case execution time.The one with the shortest execution time is returned. The corresponding abstract graph is finally generated, and all the non-dependent tasks are put in parallel with it. Since we do not consider the resources in this step, this abstract graph is an optimal version of the original process in terms of execution time, as tasks have been put as much as possible in parallel. However, as this algorithm explores all possible combinations of dependencies, its complexity is exponential. Nonetheless, real-world BPMN processes are usually small, thus even smaller are their dependencies. Consequently, the execution time of this algorithm is rather short in practice.

Example. Let us consider the trip organisation process shown in Figure 1. According to the user, the dependencies that should be preserved by the approach are the ones shown in Figure 3(a). As the reader can see, these dependencies do not validate the conditions stated previously, because for example, task *Reimbursement* has two predecessors (*Flight Booking* and *Return Documents*) that do not have the same successors.



(a) Original Dependency Graph (b) Optimal Dependency Graph

Fig. 3: Evolution of the Dependency Graph of the "Trip Organisation Process"

The result of the execution of Algorithm 1 on this dependency graph is shown in Figure 3(b). As the reader can see, a new dependency has been added between tasks *Flight Booking* and *Return Documents*. The dependency between tasks *Flight Booking* and *Reimbursement* has been removed because these tasks are now transitively dependent through task *Return Documents*. These dependencies now validate the conditions, and the corresponding abstract graph is optimal, as it has the shortest worst-case execution time possible, that is 28UT.

Computation of Resource Usage. In the previous step, we have built an abstract graph that is an optimal representation of our initial process satisfying the dependencies that may exist between tasks. Nonetheless, this optimal graph has been built without considering the resources required by the tasks composing it to execute. Even if the execution time of the abstract graph may not vary a lot in a single instance context, it may drastically increase in the multi-instance context that we are dealing with, due to resource competition and synchronisation delays. The goal of this step is then to verify whether our optimal abstract graph can execute without such delays. To do so, we compute the resources needed by a single instance of the abstract graph to execute without resource competition. From this computation, we are able to compute the resources needed by multiple instances of the abstract graph to execute without resource competition. Then, we compare these resources to the shared ones. If the pool of computed resources is smaller than the pool of shared resources, the optimal abstract graph can execute without resource competition nor synchronisation delays. The corresponding BPMN process is then synthesised and returned to the user as optimal version of the original process. Otherwise, steps 3 & 4 of Figure 2 are performed to limit the delays induced by the resource competition. In the latter case, the BPMN process returned to the user is close to the optimal one.

The first part of this step consists of transforming the current abstract graph into a *tasks execution flow*. A tasks execution flow is a representation in which each task of the abstract graph is pictured as a box, for which the length represents the duration. Tasks are put one after the other if they are executed sequentially, and one above the other if they are executed in parallel, along the time axis. In case of choices or loops, their probability of execution is also indicated. From this representation, we know exactly which task is executed at each moment. The resource usage at any time is then computed directly, by considering the resource usage of each currently executing task. Nonetheless, we recall that we are dealing with several instances of the abstract graph executing at the same time. Thus, we are interested in computing the global resource usage of the abstract graph. To do so, the idea is to count the maximum number of instances running at the same time. This is feasible because the worst-case execution time of the abstract graph and the IAT are known. Once the maximum number of instances running at the same time has been computed, the instances are divided into three blocks. The first block contains the pivot instance, that is the middle one. In other words, this instance is the instance for which the number of instances that started before it is equal to the number of instances that will start after it. The second block contains the instances that were already running before the beginning of the pivot instance. The third block contains the instances that started running after the beginning of the pivot instance. Then, each instance in block 2 & 3 is shifted by a precise number of IAT in order to represent the progression of its execution compared to the one of the pivot instance. Thus, instances belonging to the second block are shifted by a negative number of IAT, while instances belonging to the third block are shifted by a positive number of IAT. Finally, the tasks execution flow of each running instance is traversed, and the tasks executing at their relative instant of time compared to the pivot instance are retrieved. Then, we know precisely which tasks are executing at any instant of time of the execution. The global resource usage of the abstract graph (i.e., for multiple instances) is deduced by considering the maximum usage of each resource over the execution.

Algorithm 2 Multi-Instances Resource Computation

```
    Input: $T_S$ (Tasks executing per instant of time (single instance))
    Output: $R_G$ (Global resource pool needed by the abstract graph)
 1: $T_M \leftarrow \emptyset$                          ▷ List of tasks executing per unit of time
 2: $NI \leftarrow \lceil \frac{WCET}{IAT} \rceil - 1$            ▷ Number of instances already running
 3: for t = 0; t < WCET; t++ do
 4:     $T_t \leftarrow \emptyset$                      ▷ List of tasks executing at time $t$
 5:     for i = -NI; i ≤ NI; i++ do
 6:         $t_r \leftarrow i \times IAT + t$                     ▷ Relative time of instance $i$
 7:         if 0 $\leq t_r <$ WCET then        ▷ Check that current instance is executing
 8:             $T_t \leftarrow T_t \cup T_S.get(t_r)$          ▷ Add the tasks executing at $t = t_r$
 9:         end if
10:     end for
11:     $T_M \leftarrow T_M \cup T_t$
12: end for
13: return $R_G \leftarrow extractGlobalPool(T_M)$
```

Algorithm 2 performs this computation. It takes as input the list of tasks executing per instant of time for a single instance. Then, it starts by computing the number of instances that were already running before the start of the pivot instance (line 2). By symmetry, it computes the number of instances that started executing after the beginning of the pivot instance. For each instant of time in the abstract graph execution, it iterates over the running instances. For each instance, it computes its relative instant of time t_r, that is its instant of time from the point of view of the pivot instance. Then, it verifies that this relative instant of time corresponds to an instant of time in which the current instance is running (i.e., has started and has not yet terminated). If this is the case, the tasks executing at this instant of time are added to the list of all tasks executing at this instant of time. This algorithm performs in linear time as it only traverses the list of tasks executing at each instant of time once.

Theorem 1. *(Optimality of BPMN Generation) Let A be the abstract graph generated from the original dependencies of the process, B the BPMN process synthesised from A, R_A the pool of available resources and R_C the pool of resources computed by Algorithm 2. Then, $R_C \leq R_A \implies B$ is optimal.*

Proof (sketch). In a first time, A is generated from the given set of dependencies, without considering the resources. Either the dependencies validate the conditions stated in Section 2, in which case A is optimal according to [11], or not, in which case the optimal dependencies are computed using Algorithm 1. Then, Algorithm 2 precisely computes the tasks executed at each instant of time of the execution of the multiple instances of B. It returns the maximum number of resources used

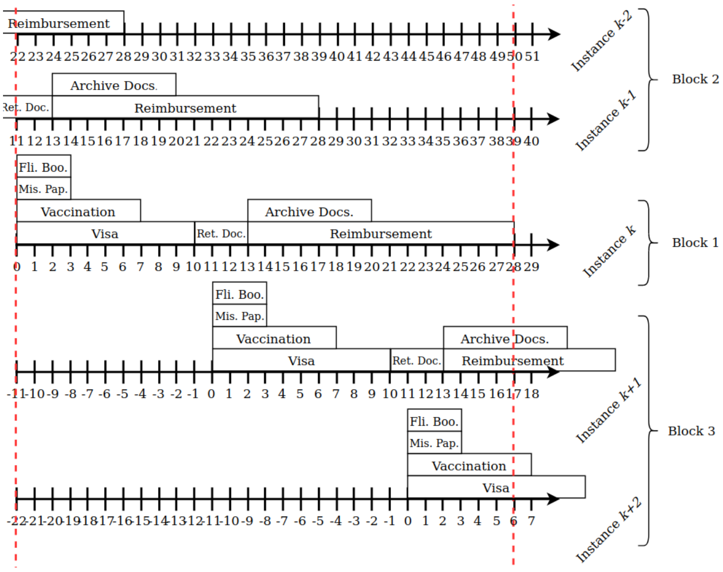


Fig. 4: Tasks Execution Flows of the Trip Organisation Process

at any instant of time, that we call R_C. If $R_A > R_C$, none of the instances will wait/compete to access a resource and execute a task, thus no delay will increase the execution time of the instances. Consequently, the execution time of B, which is the shortest execution time possible, is the same for each instance of B, i.e., $ET_{B_1} = ET_{B_2} = ... = ET_{B_n}$. By construction, the AET of B is then necessarily optimal: $AET_B = \frac{1}{n} \sum_{i=1}^{n} ET_{B_i} = ET_{B_1} = ET_{B_2} = ... = ET_{B_n}$ □

Example. Now, let us illustrate this computation on an example. Figure 4 shows the tasks execution flows corresponding to all the instances of the optimal trip organisation process running at the same time. The process has a worst-case execution time of 28UT, and is ran with an IAT of 11UT. The number of instances already executing when the pivot instance started is then $\lceil \frac{WCET}{IAT} \rceil - 1 = 2$. By symmetry, the same number of instances started executing after the pivot instance. Then, by progressing through the execution of the pivot instance (i.e., instance k), we are able to compute the tasks executed by all the running instances of the abstract graph. For example, at time $t = 10$, the pivot instance starts executing the task *Return Documents*. The corresponding relative time for instance $k-1$ is $t = 21$, as this instance started executing 11UT (1 × IAT) before the pivot instance. Thus, it is executing task *Reimbursement*. By repeating this procedure for each instance, we obtain the list of tasks executing at each instant of time for multiple instances. From this list, we compute the global pool of resources needed by the process to execute without waiting times, that is 2 *assistants*, 1 *doctor*, 2 *financial staffs*, 1 *travel agency*, 2 *users* and 1 *visa office*.

Minimising Resource Competition Impact. At this point of the approach, we have computed the resource usage of our optimal abstract graph, and found that the shared resources were not sufficient to avoid resource competition and synchronisation delays. This step consists of verifying whether the lack of certain resources will strongly impact the average execution time of the abstract graph or not, and, if it is the case, to identify all the tasks that should be removed from their parallel constructs and put in sequence. Such tasks are called *non-parallelisable tasks*. To do so, we compute for each insufficient resource a value called *absorbance*. This value is the ratio between the amount of time at which the usage of the resource is lower than the number of available replicas of the resource, and the amount of time at which the usage of the resource is greater than this number. If this value is below a certain threshold, we conclude that the lack of the resource will not impact the average execution time of the abstract graph. If this is the case for each insufficient resource, the abstract graph does not need any modification. Thus, it is mapped to its equivalent BPMN representation [11] and returned to the user. Otherwise, some tasks have to be removed from their parallel constructs and put in sequence to limit the resource competition and the synchronisation delays. To do so, a score is assigned to each task, according to its duration and its resource usage: the longer the duration the smaller the score, and the higher the cost the higher the score. Then, the task with the highest score is put in sequence, and the absorbance of the new abstract graph is computed. If it is below the threshold, the computation stops, and the current task is put in the set of non-parallelisable tasks. Otherwise, the second task with highest score is put in sequence, and so on. This computation finishes either when the absorbance of the abstract graph reaches a value lower than the threshold for each lacking resource, or when all the tasks using a lacking resource have been put in sequence.

Algorithm 3 Find Non-Parallelisable Tasks

```
Inputs: $P_O, P_S, G, T$ (Optimal resource pool, shared resource pool, abstract graph, set of tasks of the abstract graph)
Output: $S_{NP}$ (Set of non-parallelisable tasks)
$computeScores(T)$
$P_L \leftarrow computeLackingResources(P_O, P_S)$ ; $modified \leftarrow True$ ; $S_{NP} \leftarrow \emptyset$
while $modified = True$ do                                        ▷ Iterate until fix point
    $modified \leftarrow False$
    for $r \in P_L$ do                                  ▷ Iterate over each lacking resource
        $\delta_O \leftarrow computeWeightedOverUsageTimeOf(r, G)$
        $\delta_U \leftarrow computeWeightedUnderUsageTimeOf(r, G)$
        $A \leftarrow \frac{\delta_O}{\delta_U} \times 100$                      ▷ Compute absorbance of $r$
        if $A > Threshold$ then
            $T_{HS} \leftarrow getTaskWithHighestScore(r, T)$
            $T \leftarrow T \setminus \{T_{HS}\}$ ; $S_{NP} \leftarrow S_{NP} \cup \{T_{HS}\}$ ; $modified \leftarrow True$
            $putInSequence(T_{HS}, G)$
        end if
    end for
end while
return $S_{NP}$
```

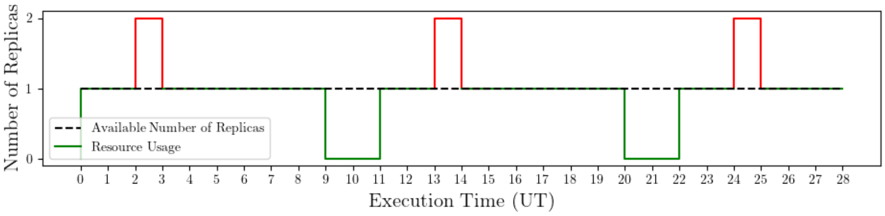


Fig. 5: Usage of Resource *assistant* Over Time

Algorithm 3 performs this computation. It takes as input the optimal pool of resources P_O, the pool of shared resources P_S, the abstract graph G, and the set of tasks of the abstract graph T. First, it assigns a score to each task (line 1). Then, it computes the set of lacking resources (line 2). For each lacking resource r, it computes its absorbance A (lines 6, 7, 8). If A exceeds the threshold, the task with the highest score using this resource T_{HS} is put in sequence and added to the list of non-parallelisable tasks S_{NP}, and a new iteration starts. When the algorithm reaches a fix point (i.e., no tasks were put in sequence in the previous iteration), the while loop breaks, and the list of non-parallelisable tasks S_{NP} is returned. In this work, according to a study made on many examples (real-world and handcrafted), we found that the threshold giving the best results was 100. Thus, this threshold is used as default threshold in the approach. Nonetheless, the user can still specify his own threshold as input. This algorithm runs in linear time as it performs at most $|T|$ iterations before finishing.

Example. Now, let us consider the optimal abstract graph of the trip organisation process to illustrate this step. In this example, we consider that the shared resources contain only one replica of the resource *assistant*, instead of the two required by the abstract graph to execute without resource competition. This means that tasks *Mission Paperwork* and *Archive Documents* will possibly be put in sequence, as they both require one replica of resource *assistant* to execute. Figure 5 shows the usage of the resource *assistant* over the execution time of the abstract graph. As the reader can see, the usage exceeds the total number of replicas by one between times 2 & 3, 13 & 14, and 24 & 25. Conversely, the usage is lower than the total number of replicas by one between times 9 & 11 and 20 & 22. The rest of the time, the replica is accessed without competition. The absorbance of resource *assistant* is then $\frac{1\times1+1\times1+1\times1}{2\times1+2\times1} = \frac{3}{4} = 75$. As this absorbance is lower than 100 (default threshold), the tasks *Mission Paperwork* and *Archive Documents* can remain in parallel. As no other resource is lacking, none of the tasks of the optimal abstract graph needs to be put in sequence. Thus, the BPMN process can directly be generated, as shown in Figure 6. As the reader can see, as no non-parallelisable tasks have been found, this process is optimal as it has the highest degree of parallelism while respecting the original dependencies.

Sequencing of Non-Parallelisable Tasks. This step is executed when the previous step has returned a list of non-parallelisable tasks. The goal of this step is to *isolate* these tasks in the abstract graph, while preserving the dependencies of the process. The principle of isolation is the following: each non-parallelisable task of the abstract graph is removed from its current abstract node, and put alone in a new abstract node. By doing so, the non-parallelisable tasks are not anymore in parallel with other tasks,

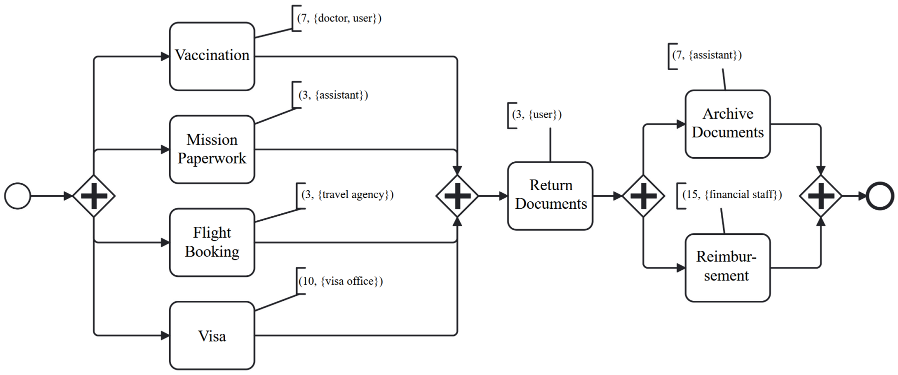
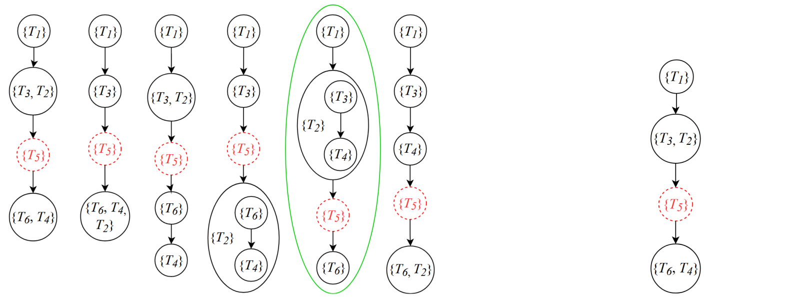


Fig. 6: Optimal Version of the Trip Organisation Process

as only elements inside the same abstract node are parallelised. This new abstract node is then added to the set of abstract nodes of the abstract graph. As the dependencies of the process must be preserved by our approach, we split this step in two cases. In the first case, none of the non-parallelisable tasks belong to the dependencies of the process. Thereby, each non-parallelisable task can simply be put in a new abstract node, which is connected either to the first or last node of the abstract graph (i.e., it becomes either the first or the last abstract node of the abstract graph). By doing so, the non-parallelisable tasks are kept all together in sequence while ensuring the highest level of parallelism for the rest of the abstract graph, and thus the shortest execution time.

In the second case, some non-parallelisable tasks belong to the dependencies of the process. Thus, they cannot simply be put at the beginning or at the end of the abstract graph, as such restructuring may no longer satisfy the dependencies. Instead of removing these tasks from their nodes as we do in the first case, we extract all the tasks that should have been put in parallel with them. By doing so, the non-parallelisable tasks now belong to isolated nodes that will no longer be parallelised. Nonetheless, we still have to manage the tasks that we have just extracted. These tasks may be put at several different places of the abstract graph while preserving the dependencies of the process. As the main concern of this approach is to minimise the execution time, we propose to compute all the places where these tasks can be put. To do so, we try to put these tasks in all possible abstract nodes of the graph while preserving the dependencies. In the end, we obtain several abstract graphs that satisfy the dependencies of the process. The one with the shortest execution time is then kept as best abstract graph. While having the advantage of returning the abstract graph with shortest execution time, this method also has the drawback of possibly taking a long time to be executed, if the number of valid combinations is large. As an alternative option, we propose an heuristic aiming at reducing this computational time. The idea of this heuristic is the following: instead of computing all the possible nodes that can host a task, we select the node having the closest greater execution time, if existing. Thus, the execution time of the selected node will not be impacted by its new task. If no such node exists, the task is put in the node having the closest execution time. More formally, considering a task T of duration d_t, and three nodes n_1, n_2 and n_3 of duration d_1, d_2 and d_3, the chosen node is the one minimising the quantity $q = d_i - d_t$ s.t. $q > 0$ for

$i \in \{1,2,3\}$. If $\nexists i$ s.t. $q > 0$, the chosen node is the one minimising the quantity $|d_i - d_t|$ for $i \in \{1,2,3\}$. This heuristic runs in linear time, but may generate an abstract graph that does not have the shortest execution time possible. Finally, the generated abstract graph is transformed into its equivalent BPMN process and returned to the user.



(a) Result of the Combination (b) Result of the Heuristic

Fig. 7: Generation of Abstract Graph with Non-Parallelisable Tasks

Example. Let us consider a BPMN process with six tasks $T_1, T_2, T_3, T_4, T_5, T_6$ in sequence, and the following dependencies: $(T_1, T_2), (T_1, T_3), (T_3, T_4), (T_3, T_5), (T_5, T_6)$. Each task respectively has a duration of 6UT, 12UT, 10UT, 2UT, 7UT, 1UT. Finally, T_5 has been marked as non-parallelisable during Step 3. Figure 7 shows the abstract graphs generated after this step, both for the combination method and the heuristic. Figure 7(a) shows the six different abstract graphs generated by the combination method. All of them have task T_5 in sequence in an isolated node, and respect the initial dependencies. According to the duration of each task, the fifth abstract graph (circled) has the shortest worst-case execution time (26UT), and is the one returned by the combination method. Figure 7(b) shows the abstract graph generated by the heuristic. As the reader can see, T_2 is now in parallel with T_3 and T_4 in parallel with T_6. The resulting abstract graph has an execution time of 27UT, which is close to the best one, but not the best.

4 Tool and Experiments

The approach has been fully implemented in Java. It consists of approximately 10,000 lines of code, and the tool is available online [16]. It has been tested on various handcrafted and real-world examples found in the literature. The experiments allowed us to evaluate our approach both in terms of usefulness and performance, by considering the gain of the optimised process in terms of AET and the time taken by the tool to execute. The number of instances for each process tested vary between 20 and 100.

Table 1 summarises these experiments. Column 1 gives the name of the process. Columns 2, 3 & 4 show several characteristics of the process (number of nodes, flows, types of resources, replicas of resources, IAT). Columns 5 & 6 provide respectively the AET of the initial process and of the optimised process. Column 7 shows the gain that was obtained by optimising the process. Column 8 states whether the available pool of resources is sufficient to execute the optimal version of the process or not. Column 9 gives the time taken by the tool to execute.

The results can be split in two parts: the processes having enough resources to execute the optimal version of the process, and the others. In the first case, the AET of the generated process is optimal, and is generally a significant improvement of the initial one (up to 46% for the first process). A lower gain only indicates that the initial process was already syntactically close to its optimal form, not that the approach does not perform well. In the second case, the gain is lower than in the first case (up to 15.5%), due to the decrease of parallelism induced by the sequencing of some tasks. Overall, the tool executes in less than 1s on real-world processes, which is satisfactory as this approach is executed at design time.

5 Related Work

In this section, we focus on existing works on process refactoring. [20] 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. [2] 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 [13, 18]. IBUPROFEN defines a set of 10 refactoring algorithms grouped into three categories: maximisation of relevant elements, fine-grained granularity reduction, and completeness. 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

Table 1: Experimental results

BPMN process	Nodes / Flows	Types / Number of Resources	IAT	Initial AET (UT)	Final AET (UT)	Gain	Sufficient Resources	Time (ms)
Perish. Goods Tran. [21]	24 / 26	9 / 17	6	26	14	46.2%	✓	563
Employee Hiring [4]	19 / 21	7 / 12	3	30	18	40.0%	✓	607
Trip Organisation [9]	11 / 11	6 / 11	7	41	28	31.7%	✓	588
Patient Diagnosis [1]	14 / 15	4 / 12	20	61	46	24.6%	✓	624
Shipment Process [12]	16 / 18	5 / 10	5	46	42	8.70%	✓	531
Evisa Application [19]	11 / 11	3 / 7	5	84	71	15.5%	✗	797
Employee Recruit. [11]	14 / 14	7 / 11	5	92	80	13.0%	✗	873
Account Opening [17]	22 / 25	6 / 17	8	67	63	5.97%	✗	732
Goods Delivery [3]	11 / 12	6 / 16	1	78	77	1.28%	✗	696

any kind of optimisation regarding the process being designed as we do. Moreover, they do not consider rich models as we do (including, e.g., time and resources).

In [15], 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 [15] is similar to the one we use in this paper in terms of expressiveness. However, the approach is rather different since they compute optimal redesigns with respect to some constraints, thus resulting in a change in the number or execution of tasks (e.g., by splitting a task into several ones, or by executing one task or another instead of these two tasks in sequence). In contrast, we do not change the semantics of the application but only the structure of the process (the order in which the tasks are executed).

[11] proposes a semi-automated approach for helping non-experts in BPMN to model business processes using this notation. Alternatively, [14] 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 [9], 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. This work assumes that processes are executed only once (single executions) and that only one replica of each resource is avalaible. In contrast, our approach applies refactoring techniques on processes that are executed multiple times, and for which several replicas of each resource can be available (no contraints on the number of resources).

6 Concluding Remarks

In this paper, we have proposed a solution to the optimisation of business processes using refactoring techniques. Processes are described using BPMN extended with time and resources, and are executed multiple times. The final goal of this approach is to restructure the tasks within the process in order to reduce the execution time and optimise the resource usage while avoiding bottlenecks. To do so, the approach applies several successive steps. The first steps aim at analysing the dependencies between tasks and the resources usage. From these first results, the approach determines whether some tasks have to be put in sequence to limit eventual bottlenecks, or if all the tasks can be put in parallel. Finally, it returns a process that is either optimal or close to the optimal. The whole approach is fully automated by using a tool we implemented, and was applied to a set of real-world processes in order to evaluate its usefulness and performance. The experiments show satisfactory results both in terms of optimisation and computation time. The main perspective of this work is to consider non-fixed IATs, such as IATs defined using probabilistic functions.

References

1. E. Bazhenova, F. Zerbato, B. Oliboni, and M. Weske. From BPMN process models to DMN decision models. *Information Systems*, 83:69–88, 2019.
2. R. M. Dijkman, B. Gfeller, J. M. Küster, and H. Völzer. Identifying Refactoring Opportunities in Process Model Repositories. *Inf. Softw. Technol.*, 53(9):937–948, 2011.
3. F. Durán, Y. Falcone, C. Rocha, G. Salaün, and A. Zuo. From Static to Dynamic Analysis and Allocation of Resources for BPMN Processes. In *Proc. of WRLA'22*, pages 1–18. Springer, 2022.
4. F. Durán, C. Rocha, and G. Salaün. Computing the Parallelism Degree of Timed BPMN Processes. In *Proc. of FOCLASA'18*, pages 1–16, 2018.
5. F. Durán, C. Rocha, and G. Salaün. Symbolic specification and verification of data-aware BPMN processes using rewriting modulo SMT. In V. Rusu, editor, *Proc. of WRLA'18*, volume 11152 of *LNCS*, pages 76–97. Springer, 2018.
6. F. Durán, C. Rocha, and G. Salaün. A Rewriting Logic Approach to Resource Allocation Analysis in Business Process Models. *Sci. Comput. Program.*, 183, 2019.
7. F. Durán, C. Rocha, and G. Salaün. A rewriting logic approach to resource allocation analysis in business process models. *Sci. Comput. Program.*, 183, 2019.
8. F. Durán, C. Rocha, and G. Salaün. Resource provisioning strategies for BPMN processes: Specification and analysis using maude. *J. Log. Algebraic Methods Program.*, 123:100711, 2021.
9. F. Durán and G. Salaün. Optimization of BPMN Processes via Automated Refactoring. In *Proc. of ICSOC'22*, volume 13740 of *LNCS*, pages 3–18. Springer, 2022.
10. R. Eshuis and P. V. Gorp. Synthesizing data-centric models from business process models. *Computing*, 98(4):345–373, 2016.
11. Y. Falcone, G. Salaün, and A. Zuo. Semi-automated Modelling of Optimized BPMN Processes. In *Proc. of SCC'21*, pages 425–430. IEEE, 2021.
12. Y. Falcone, G. Salaün, and A. Zuo. Probabilistic Model Checking of BPMN Processes at Runtime. In *Proc. of IFM'22*, pages 1–17. Springer International Publishing, 2022.
13. 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*, volume 1027 of *CEUR Workshop Proceedings*, pages 16–30, 2013.
14. A. Ivanchikj, S. Serbout, and C. Pautasso. From Text to Visual BPMN Process Models: Design and Evaluation. In *Proc. of MoDELS'20*, pages 229–239. ACM, 2020.
15. A. Kumar and R. Liu. Business Workflow Optimization through Process Model Redesign. In *Proc. of TEM'22*, LNCS, pages 3068–3084. Springer, 2022.
16. Q. Nivon. Automated Tool for Multi-Instance BPMN Processes Optimisation. `https://github.com/KyriuDev/MultiInstancesRefactoring`, 2023.
17. Q. Nivon and G. Salaün. Debugging of BPMN Processes Using Coloring Techniques. In *Proc. of FACS'22*, pages 90–109. Springer, 2022.
18. R. Pérez-Castillo, M. Fernández-Ropero, and M. Piattini. Business process model refactoring applying IBUPROFEN. An industrial evaluation. *J. Syst. Softw.*, 147:86–103, 2019.
19. G. Salaün. Quantifying the Similarity of BPMN Processes. In *Proc. of APSEC'22*, pages 1–10, 2022.
20. D. Silingas and E. Mileviciene. Refactoring BPMN Models: From 'Bad Smells' to Best Practices and Patterns. In *BPMN 2.0 Handbook*, pages 125–134. 2012.
21. P. Valderas, V. Torres, and E. Serral. Modelling and executing IoT-enhanced business processes through BPMN and microservices. *J. Syst. Softw.*, 184:111139, 2022.