Computing the Parallelism Degree of Timed BPMN Processes

Francisco Durán[1], Camilo Rocha[2], and Gwen Salaün[3]

[1] University of Málaga, Spain
[2] Pontificia Universidad Javeriana, Cali, Colombia
[3] Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, F-38000 Grenoble France

Abstract. A business process is a combination of structured and related activities that aim at fulfilling a specific organizational goal for a customer or market. An important measure when developing a business process is the *degree of parallelism*, namely, the maximum number of tasks that are executable in parallel at any given time in a process. This measure determines the peak demand on tasks and thus can provide valuable insight on the problem of resource allocation in business processes. This paper considers *timed* business processes modeled in BPMN, a workflow-based graphical notation for processes, where execution times can be associated to several BPMN constructs such as tasks and flows. An encoding of timed business processes into Maude's rewriting logic system is presented, enabling the automatic computation of timed degrees of parallelism for business processes. The approach is illustrated with a simple yet realistic case study in which the degree of parallelism is used to improve the business process design with the ultimate goal of optimizing resources and, therefore, with the potential for reducing operating costs.

1 Introduction

A business process is a collection of structured activities or tasks that produce a specific product and fulfill a specific organizational goal for a customer or market. A process aims at modelling activities, and their causal and temporal relationships by defining specific business rules that process executions have to comply with. The Business Process Model and Notation (BPMN) [10] is a graphical modeling language for specifying business processes. BPMN was published as an ISO standard in 2013 and has become the common notation for designing business processes.

Business process optimization is a strategic activity in organizations because of its potential to increase profit margins and reduce operating costs. Resource allocation is one of the main challenges in order to maximize resource usage, improve sharing, and detect bottlenecks with the final goal of optimizing processes. An important metric when modelling and developing a business process is its degree of parallelism, which is defined as the maximum number of tasks that are executable in parallel in the process. The degree of parallelism determines the

peak demand on tasks and provides a valuable guide for the problem of resource allocation in business processes [20]. Examples of such resources include physical objects, goods, robots, and employees.

This paper presents a solution for computing the degree of parallelism of business processes modeled in the BPMN notation. The focus here is on a subset of the BPMN notation that supports the main constructs of the language, including start/end events, sequence flows, tasks, and gateways. This subset also takes time features into account, making possible the association of timing attributes (e.g., duration) to sequence flows and tasks. A formal specification of this BPMN subset is provided in Maude's rewriting logic infrastructure [3], resulting in a formal timed semantics of the language under consideration. The automatic computation of the parallelism degree is achieved by using tools available from the Maude formal environment itself. A given BPMN process is encoded into Maude and all reachable states are automatically traversed to find the states with the maximum number of tokens: a token is the usual mechanisms employed for identifying a specific execution instance in the BPMN semantics. This approach has been applied to several real-world processes for validation purposes. In this paper, it is illustrated with a case study in which the degree of parallelism is used to optimize a process.

The organization of the rest of the paper is as follows. Section 2 introduces the BPMN notation with time features. Section 3 overviews the Maude rewriting logic framework. Section 4 presents the encoding of the BPMN subset considered in this work into Maude's rewriting logic. Section 5 focuses on the computation of the parallelism degree. Section 6 introduces a case study and shows how the approach can be used to optimize a BPMN process. Section 7 surveys related work and Section 8 concludes the paper.

2 BPMN with Time

This section explains the subset of BPMN considered in this paper, which focuses on behavioral aspects (start/end events, tasks, flows, gateways) enriched with time. The timed extension of BPMN was originally presented in [7].

A BPMN process is a directed graph with nodes as vertices and sequence flows as directed edges. A node is a start or end event, a task, or a gateway. Start and end events are used to initialize and terminate processes, respectively. A task represents an atomic activity, and has exactly one incoming and one outgoing flow. A gateway is used to control the split patterns (i.e., flow divergence) and merge patterns (i.e., flow convergence) of execution in a process. In this paper, a process is considered to have exactly one start event and at least one end event. The three main gateways available in BPMN are considered, namely, exclusive, parallel, and inclusive gateways. An exclusive gateway chooses one out of a set of mutually exclusive alternative incoming or outgoing branches. A parallel gateway creates concurrent flows for all its outgoing branches or synchronizes concurrent flows for all its incoming branches. In an inclusive gateway, any number of branches among all its incoming or outgoing branches may be

taken. Looping behaviors and unbalanced structure of the process (no strict correspondence between split and merge gateways) are supported in this work.

In addition to these classic BPMN constructs, time can be associated to tasks and flows. In this paper, time is interpreted as a duration of a task or a flow. When a flow has a duration d greater than zero, it means that the destination node is triggered after d units of time. If the duration is zero, that node is immediately triggered. Similarly, a task triggers its outgoing flow at once for a duration equal to zero and waits for d units of time when a duration d greater than zero is associated to that task.

Figure 1 summarizes the syntax of BPMN supported in this work, including examples of the timing constructs. In this paper, we assume that BPMN processes are syntactically correct. This can be enforced using existing works, e.g., [8], or using a BPMN engine, e.g., the Activiti BPM platform, Bonita BPM, or the Eclipse BPMN Designer.

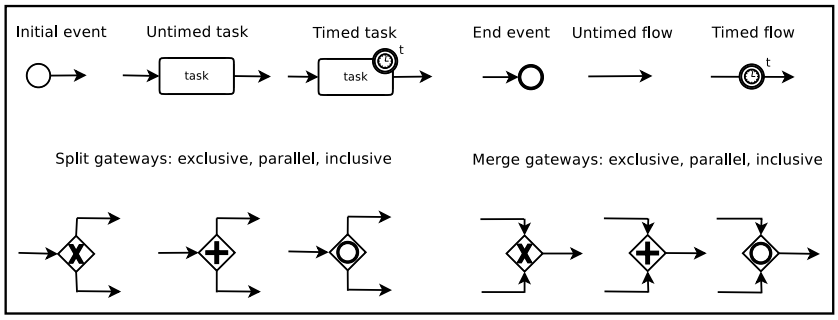


Fig. 1. BPMN syntax with time features

The informal semantics of BPMN is described in official documents [10, 18] and some attempts have been made to formalize it (e.g., [5, 16, 19, 21]). The execution semantics of BPMN is usually given by means of tokens representing how the execution of the process evolves over time. At the beginning of the process execution, there is exactly one token at the start event. A token can move along sequence flows. A token can also enter and leave a task by following the flow associated to that task. When a token arrives at a gateway, the execution behaves differently depending on the kind of gateway encountered. When a token arrives at a parallel split gateway, the token is consumed and one token is generated for every outgoing flow of the split gateway. When a token is consumed at an exclusive split gateway, only one token is created and assigned to one of its outgoing flows. In the case of an inclusive split gateway, when a token is consumed, some new tokens are generated and assigned to the outgoing flows. For the inclusive split gateway, the choice of outgoing branches to be activated depends on data-based conditions (e.g., "$x > 50$" is associated to one outgoing flow and "$x \leq 50$" is associated to the other flow) that can be evaluated to true or false. In this

work, we preferred to abstract away those data conditions and consider that all branches can be executed (we enumerate all possible combinations). Merge gateways usually act as synchronization points and can be triggered when all expected tokens have arrived. A process finishes its execution when all tokens have reached an end event.

3 Maude in a Nutshell

Rewriting logic [15] is a semantic framework that unifies a wide range of models of concurrency. Specifications in rewriting logic are called rewrite theories and they can be executed in Maude [3]. A rewrite logic theory is a tuple $(\Sigma, E \cup A, R)$, where $(\Sigma, E \cup A)$ is a membership equational logic [2] theory with Σ its signature, E a set of conditional equations, A a set of equational axioms (e.g., associativity, commutativity and identity) so that rewriting is performed modulo A, and R is a set of labeled conditional rules.

In rewriting logic, a distributed system is axiomatized by an equational theory describing the set of states as an algebraic data type and a collection of conditional rewrite rules specifying the concurrent transitions. Rewrite rules are written as `crl [`l`]:` t `=>` t' `if` C, with l a label, t and t' terms, and C a guard or condition. Rules describe the local, concurrent transitions that are possible in the system, i.e., when a part of the system state fits the pattern t, then it can be replaced by the corresponding instantiation of t'. The guard C acts as a blocking precondition in the sense that a conditional rule can only be fired if its condition is satisfied. Rules may be given without label or condition. Unlabelled and unconditional rules may be written as `rl` t `=>` t'.

Conditions are either a Boolean expression or a conjunction of equalities $u_i{=}v_i$, membership axioms $u_i{:}s_i$ or matching equations of the form $p_i{:=}u_i$, where u_i and v_i are terms, p_i are pattern terms (irreducible terms with variables), and s_i are sorts. In its simplest form, pattern terms are just variables, with a functionality equivalent to *where* statements in typical functional programs.

In the Maude language, object-oriented systems can be specified by object-oriented modules in which classes and subclasses are declared. A class is declared with syntax `class` $C \mid a_1 : S_1, \ldots, a_n : S_n$, where C is the name of the class, a_i are attribute identifiers, and S_i are the sorts of the corresponding attributes. The objects of a class C are then record-like structures of the form $< O : C \mid a_1 : v_1, \ldots, a_n : v_n >$, where O is the name of the object and v_i are the current values of its attributes. An object-oriented system, such as the one presented in this paper, evolves as the result of applying the rewrite rules on collections of objects in the system states.

4 Encoding into Rewriting Logic

In this section, the encoding of the subset of BPMN with time information is presented as a Real-Time Maude [17] specification. This Maude specification consists of two parts: the encoding of the process structure and the description

of the semantics of our BPMN subset using rewrite rules. In this section, the two parts of the encoding are surveyed. The interested reader is referred to [7] for a more in-depth presentation of this encoding and to [1] for the complete Maude specification, which includes all the rules and examples of BPMN processes.

As we will see in the rest of this section, the declarative style of Maude allowed us to encode BPMN execution semantics in a quite simple and elegant way. Moreover, Maude's formal environment is equipped with a large variety of analysis tools. The computation of the timed degree of parallelism relies on some of them as we will see in Section 5.

4.1 Process Encoding

Each BPMN process is translated into Maude for its analysis. This transformation is automated by applying a Python script we implemented as plugin of the VBPMN platform [12]. A BPMN process is represented in Maude as a set of flows and a set of nodes. A flow is represented as a term $\mathsf{flow}(\mathsf{sf}_i, t)$, with sf_i an identifier and t a duration (zero if there is no delay associated to that flow). There are different kinds of nodes: start, end, task, split, and merge. A start (end, resp.) node consists of an identifier and an output (input, resp.) flow identifier. A task node involves an identifier, a task description, two flow identifiers (input and output), and a duration (zero if no duration is associated to this task). A split node includes a node identifier, a gateway type (exclusive, parallel, or inclusive), an input flow identifier, and a set of output flow identifiers. A merge node includes a node identifier, a gateway type, a set of input flow identifiers, and an output flow identifier.

4.2 Execution Semantics

The execution semantics of BPMN constructs is usually described using tokens, which are associated to tasks and flows. The tokens circulate along those flows and tasks, and this evolution of tokens specifies the way a process executes. This token-based semantics is represented in rewriting logic using rewrite rules. We define one or several rewrite rules for each BPMN construct introduced in Section 2, modelling the different actions that may occur in the system, e.g., a token enters a task, a token moves along a flow, a token goes throw a gateway, etc. The rewrite rules are encoded once and for all and do not depend on the process specification.

Each rewrite rule applies on systems composed of a process object and a simulation object. The process object represents the BPMN process, and it does not change. The simulation object keeps information on the execution of the process: a set of tokens and a global time described using a natural number (discrete time). Each token is defined by the identifier of the flow or task it is associated to as well as a time corresponding to a duration. The simulation object may consist of several tokens at some point because parallel or inclusive split patterns generate several tokens as output given one token as input.

```
class Process | nodes : Set{Node}, flows : Set{Flow} .
class Simulation | tokens : Set{Token}, gtime : Time .
```

A tick rule is used to simulate the time evolution. This rule increases the global time and decreases all tokens' timers. The timing semantics forces the execution of actions by moving tokens in the process to a scheduler. The time cannot elapse when timers have reached zero time units, meaning that actions need to be triggered in the process.

We give now an informal introduction to the rewrite rules axiomatizing the process transitions for the BPMN subset considered in this work. As far as start/end events are concerned, it is assumed that the simulation object includes an initial token. The start rule (Figure 2) is triggered when this token is available (node identifier NId, line 6). When the startProc rule is applied, the initial token is consumed and another one is added to the set of current tokens (note lines 6 and 13), which indicates that the flow outgoing from the start event has been activated (FId). The time assigned to this new token is the duration of the flow FId (line 11).

```
1  rl [startProc] :
2    < PId : Process |
3        nodes : (start(NId, FId), Nodes),
4        flows : (flow(FId, T), Flows) >
5    < SId : Simulation |
6        tokens : (token(NId, 0), Tks),            --- init token available
7        Atts >
8    =>
9    < PId : Process |
10       nodes : (start(NId, FId), Nodes),
11       flows : (flow(FId, T), Flows) >
12   < SId : Simulation |
13       tokens : (token(FId, T), Tks),     --- token for FId with duration
14       Atts > .
```

Fig. 2. Start event rule

The end event rule is triggered when there is a token for the incoming flow with zero time duration. This token is consumed, terminating this flow's execution.

A task execution is encoded with two rules expressing the possibility that a task may take time if a duration is associated to it. An initiation rule activates the task when a token representing the incoming flow is available. In this case, a new token with the task identifier and the task duration is generated. A second rule is used for representing the task completion. This rule is triggered when there is a token for that task with time zero. In that case, this token is consumed and a new one is generated for the outgoing flow.

As far as gateways are concerned, the rewrite rules are different depending on the gateway. The exclusive and parallel gateways used in Section 6 for the case study are presented below (refer to [7] for details about inclusive gateways).

```
1  crl [mergeParallelGateway] :
2    < PId : Process |
3        nodes : (merge(NId, parallel, FIds, FId), Nodes),
4        flows : (flow(FId, T), Flows) >
5    < SId : Simulation | tokens : Tks, Atts >
6    =>
7    < PId : Process |
8        nodes : (merge(NId, parallel, FIds, FId), Nodes),
9        flows : (flow(FId, T), Flows) >
10   < SId : Simulation |
11       tokens : (token(FId, T), removeTokensParallel(FIds, Tks)), Atts >
12   if allTokensParallel(FIds, Tks) .   ---- all incoming flows activated
```

Fig. 3. Parallel merge gateway rule

The semantics of exclusive gateways is encoded with two rules. The rule for the exclusive split gateway executes when a token with time zero is available in the input flow and non-deterministically generates a token for one of the output branches. The exclusive merge gateway executes when there is one token for one of the incoming flows. In this case, the token is consumed and a token is generated for the merge outgoing flow.

The parallel split gateway rule is triggered when a token corresponding to the input flow is available. If so, the token is consumed and one token is added for each outgoing flow. The merge rule for the parallel gateway (Figure 3) is executed when there is a token for each incoming branch (function allTokensParallel in Figure 3, line 12). In that case, these tokens are removed (function removeTokensParallel, line 11) and a new token is generated for the outgoing flow.

5 Computing the Parallelism Degree with Maude

The encoding of the BPMN semantics in Maude can be used to simulate process executions. By using Maude's meta-programming capabilities, an interesting repertory of different measures related to the degree of parallelism of a process can be offered. The reader is referred to [3] for details on Maude and its reflective capabilities.

For the computation of the degree of parallelism, there is special interest in the search command: the process of searching for a term satisfying some conditions starting from an initial term is metarepresented by the built-in function metaSearch. This function takes as arguments the metarepresentation of a module, the metarepresentation of the starting term for search, the metarepresentation of the pattern to search for, the metarepresentation of a condition to be satisfied, the metarepresentation of the kind of search to carry on (the quoted identifier '* for a search involving zero or more rewrites), a bound value (maximum depth of the search), and a natural number indicating the solution of interest. In order to explore all possible reachable states, an algorithm has been implemented in Maude for iterating over all possible values of this solution number until the metasearch function fails to find any more states.

```
op parDegree : Module Term Bound -> Tuple{Nat, Nat} .
op parDegree : Module Term Term Bound Nat Nat -> Tuple{Nat, Nat} .

eq parDegree(M, T, B) = parDegree(M, T, 'St:Configuration, B, 0, 0) .
ceq parDegree(M, T, T', B, N, N1)
  = if RT == failure
    then (N, N1)
    else parDegree(M, T, T', s N,
           max(N1, downTerm(
                     getTerm(
                       metaReduce(M,
                         'getNumberOFTokens[getTerm(RT)])), INF)))
    fi
  if RT := metaSearch(M, T, T', nil, '*, B, N) .
```

Fig. 4. Degree of parallelism: the parDegree function

```
op metaSearch :
     Module Term Term Condition Qid Bound Nat ~> ResultTriple?
```

The parDegree function in Figure 4 computes the number of states and the (timed) degree of parallelism. Specifically, given a module M with the representation of the BPMN process to analyze, an initial state given by a term T, and a bound B, parDegree(M, T, B) will return a pair (N, PD) where N is the number of (different) reachable states up to the specified depth, and PD is the maximum degree of parallelism for that process.

Notice that given the representation of process states, in the presence of a loop there is the issue of nontermination. Therefore, the analysis is bounded up to some given depth, so that termination is always guaranteed. Theoretically, this bound may have an impact on the result because by missing executions an erroneous degree of parallelism could be computed. In practice, a large bound is chosen (100 for instance for the example presented in Section 6) in order to avoid such faulty results.

The parDegree function is implemented using an homonym function with three additional arguments: the target term (a variable of sort Configuration, so that any reachable state is considered), an index with the solution number to consider, and the provisional maximum degree of parallelism (zero at the beginning and updated every time a greater value is found). For each solution number N, the metaSearch operation is invoked. If the operation returns failure, the pair (N, PD) is returned with PD the maximum degree of parallelism. Notice the use of the getTerm operation to obtain the term component of the tuple returned by metaSearch and metaReduce, and the use of metaReduce to evaluate the auxiliary function getNumberOfTokens on the metaterm obtained as result of the search operation.

Complementarily to the maximum degree of parallelism, the minimum degree can be computed. Both values may help in scheduling the minimum and maximum amount of resources required for the execution of the process over time. To do so, the parDegreeTrace function in Figure 5 computes a map associating to each moment of time a pair (min, max) with the minimum and the

```
op parDegreeTrace : Module Term Term Bound -> Map{Time,Tuple{Nat,Nat}} .
op parDegreeTrace : Module Term Term Bound Nat Map{Time,Tuple{Nat,Nat}}
     -> Map{Time,Tuple{Nat,Nat}} .
op parDegreeTrace : Module Term Term Bound Nat Map{Time,Tuple{Nat,Nat}}
     Term -> Map{Time,Tuple{Nat,Nat}} .

eq parDegreeTrace(M, T, T', B) = parDegreeTrace(M, T, T', B, 0, empty) .
ceq parDegreeTrace(M, T, T', B, N, TMMM)
  = if RT:ResultTriple? == failure
    then TMMM
    else parDegreeTrace(M, T, T', B, N, TMMM, getTerm(RT:ResultTriple?))
    fi
  if RT:ResultTriple? := metaSearch(M, T, T', nil, '*, B, N) .
ceq parDegreeTrace(M, T, T', N, TMMM, T'')
  = parDegreeTrace(M, T, T', s N,
      if TMMM[G] == undefined
      then insert(G, (D, D), TMMM)
      else if D < p1 (TMMM[G])
           then insert(G, (D, p2 (TMMM[G])), TMMM)
           else if D > p2 (TMMM[G])
                then insert(G, (p1 (TMMM[G]), D), TMMM)
                else TMMM
                fi
           fi
      fi)
  if D := downTerm(getTerm(metaReduce(M, 'getNumberOFTokens[T'']')), INF)
  /\ G := downTerm(getTerm(metaReduce(M, 'getTime[T'']')), INF) .
```

Fig. 5. Degree of parallelism along execution: the parDegreeTrace function

maximum number of tokens seen at that time. The function is similar to the above parDegree function. The main difference is that in this case the function produces a mapping that assigns a pair (min, max) to each instance of time, collecting the minimum and maximum numbers of tokens in each of the visited states with that time as current time. Pairs (min, max) are represented as elements of sort Tuple{Nat, Nat}, defined with constructor (_,_) and projection operations p1_ and p2_. Maude's built-in maps are defined as a set of pairs, with empty as empty mapping, and operations _[_] and insert to, respectively, consult and update values. Given a variable TMMM of sort Map{Time, Tuple{Nat, Nat}}, we can consult the value associated to some time G with TMMM[G]. If the map TMMM does not associate a value to a given key G, TMMM[G] will return the value undefined.

6 Case Study

Figure 6 presents the case study used in this paper to illustrate the proposed approach. It is a simplified version of an employee hiring process in a company. This process focuses on the different tasks to be carried out once the employee has successfully passed the interview. The process thus starts by some paperwork that has to be accomplished by the employee. (S)He has to see the doctor for medical check-up. If the employee needs visa, (s)he should also apply for working visa. If all the submitted documents are not satisfactory, the company

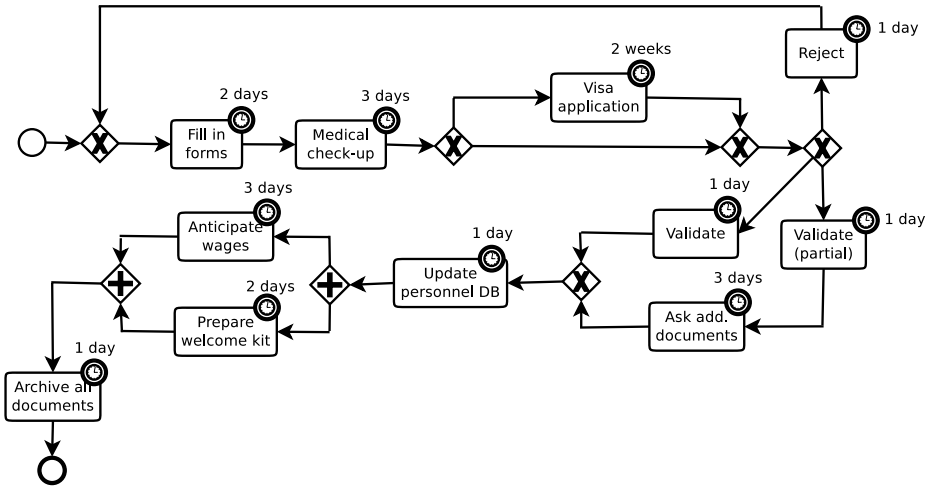


Fig. 6. BPMN process for employee hiring

may ask for them again. If everything is fine, all documents are accepted as is. In some cases, the company can validate the files but asks the employee to provide additional documents or information. The employee is then added to the personnel database and, in parallel, Human Resources (HR) anticipate wage payment while an assistant prepares the welcome kit (office, badge, keys, etc.). Finally, all provided documents are archived properly by HR.

It is worth noting that this process exhibits different kinds of gateways (exclusive and parallel), looping behaviors, and time associated to tasks. The rest of this section focuses on the timed degree of parallelism by analyzing how this measure can be used to optimize the workflow in terms of execution time.

The degree of parallelism for this example, obtained by using the approach presented in the previous sections, is 2.

```
reduce parDegree(upModule('VERIF, false),
        'initSystem.Configuration, 'St:Configuration, 100) .
result Tuple{Nat,Nat}: (1710,2)
```

This comes from the final part of the process where a parallel split/merge is used. If a closer look is taken at this part of the process, it can be seen that it takes 5 days to compute the final four activities. However, these tasks involve different people: the assistant is in charge of preparing the welcome kit, the technical staff updates the DB, and HR are in charge of the two other activities (anticipate wages, archive all documents). So this final part of the process could be organized differently. The employee information can be stored in the DB (prerequisite to other tasks), and then "anticipate wages" and "prepare welcome kit" tasks are performed in parallel. Archiving all documents is independent and could be achieved in another parallel branch. A second version of the process is given in Figure 7.

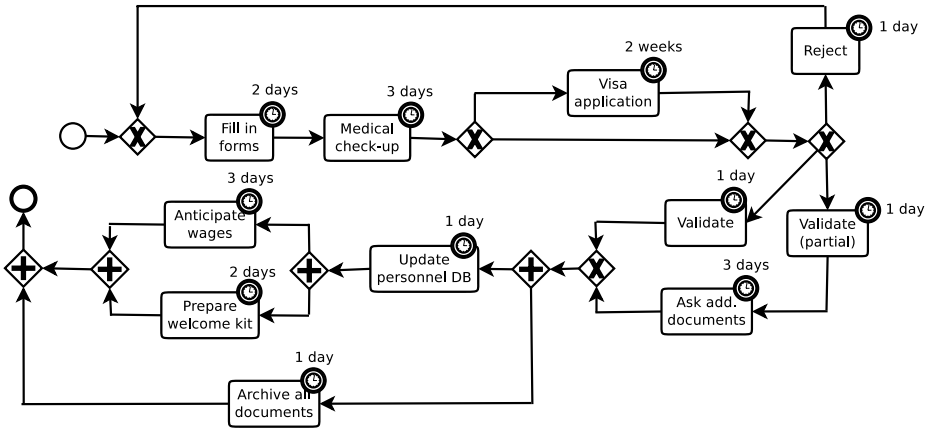


Fig. 7. BPMN process for employee hiring (V2)

When applying the computation of the timed degree of parallelism to the second version of this process, the degree is 3. This is because, although archiving all documents are completed before the internal parallel split is triggered, that token waits at the parallel merge level for the other branch to complete.

Related to that, the two tasks carried out by HR (anticipate wages and archive all documents) do not overlap and the execution time of this part of the workflow is reduced by one day (going from 5 days in the original version of the process to 4 days in this new version). It is worth observing that execution times can be automatically computed with the approach proposed in this paper too.

In the initial part of the workflow, the first three activities involve the employee. These tasks are quite time-consuming because of the appointment with a doctor (3 days in average in our model) and the visa application (2 weeks in average). However, most of the time the employee is available, (s)he is just waiting due to external constraints. Therefore, those 3 activities could be executed in parallel as shown in Figure 8. In this case, the degree of parallelism for that part of the process jumps to 3 and the execution time goes from 19 days to 14 days. More generally, the degree of parallelism of this third version (Figure 8) is 3 for the top part and 3 for the bottom part.

One can wonder whether the bottom part could be improved a little bit more by increasing the degree of that part if the used resources allows it. This is actually the case, because the "ask additional documents" task is achieved by the employee and is independent of the rest of this part of the process. In its current form, this task even delays the execution of the final part of the workflow. A possible optimization is to execute this task in parallel with the rest of the final activities. This makes the degree of parallelism, in this part of the workflow, increase to 4, resulting in saving 3 days with respect to the former version of the process. Figure 9 gives the resulting process after the three optimizations.

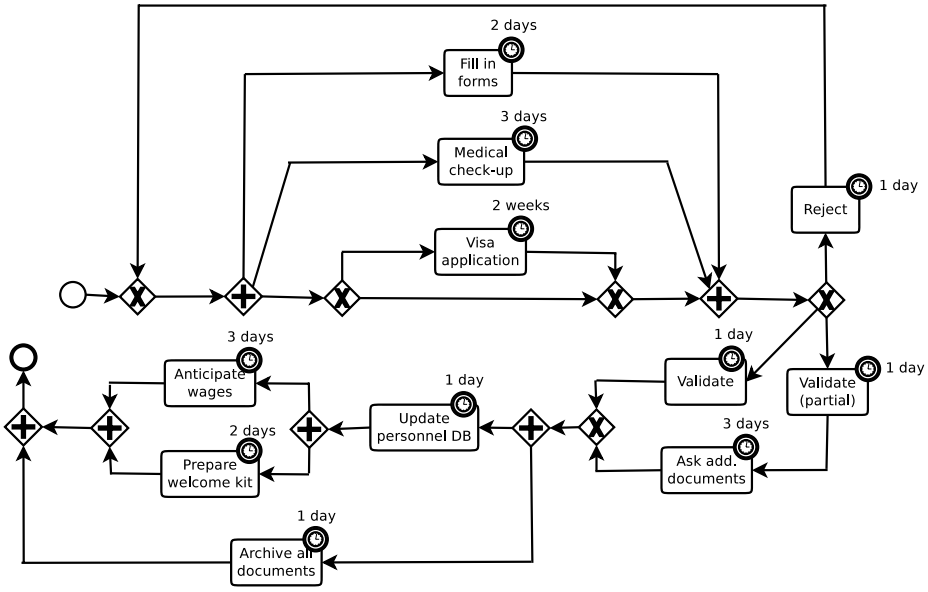


Fig. 8. BPMN process for employee hiring (V3)

Assuming that the documents are not rejected, note that the overall maximum execution time was of 28 days in the original version of the process and drops to 19 days in the final version.

The variation of the minimum and maximum degree of parallelism is also worth looking at in order to better allocate required resources over time. By using the parDegreeTrace function, a sequence of these values can be computed for each execution time (discrete time). Figure 10 shows the graphical representation for the last version of the running example (the process depicted in Figure 9). In this process, it can be observed how the maximum degree of parallelism fluctuates between 3 and 4, whilst the minimum degree varies from 0 to 3.

Last but not least, we made experiments to see how our approach scales. The main factor of explosion regarding the computation time is not the number of tasks in the process but the number of gateways, which increases the parallelism of the process and thus the number of possible executions that need to be explored. We applied our approach to large examples consisting of more than 20 gateways, including multiple nested parallel and inclusive gateways. For those examples, it took several minutes to compute the degree of parallelism. It is worth saying that we built these examples for evaluation purposes, but we have never seen a real-world process with so many nested gateways.

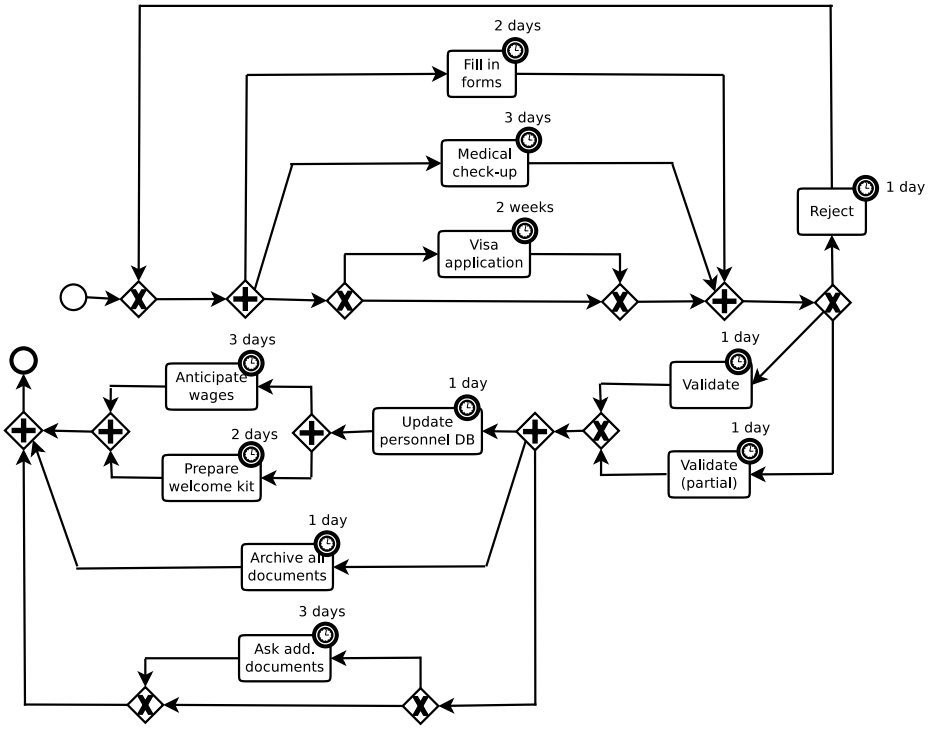


Fig. 9. BPMN process for employee hiring (V4)

7 Related Work

Two categories of related work are surveyed: (i) those proposing solutions for computing the degree of parallelism of BPMN processes, and (ii) those using rewriting logic and Maude for specification and verification of BPMN processes.

The degree of parallelism for BPMN can be computed by reasoning on Petri net models and determining the bound of a Petri net, which is the maximum number of tokens in a marking of the net. However, to do so, the reachability graph for the net should be constructed entirely. The reachability problem for some specific Petri nets, such as conflict-free Petri nets and 1 safe live free-choice nets [9], is NP-complete. Note that for arbitrary Petri nets, this problem is much harder [14]. This is probably the reason why, to the best of authors' knowledge, there is no work on degree computation with Petri nets in the literature.

In [20] several algorithms are proposed for directly calculating the degree of parallelism of a BPMN process without transforming it to another model. In this work, a duration constraint is associated to each task, i.e., a task is required to be completed within a certain time frame. Furthermore, a task must begin immediately after the completion of its precedent task. Without considering in-

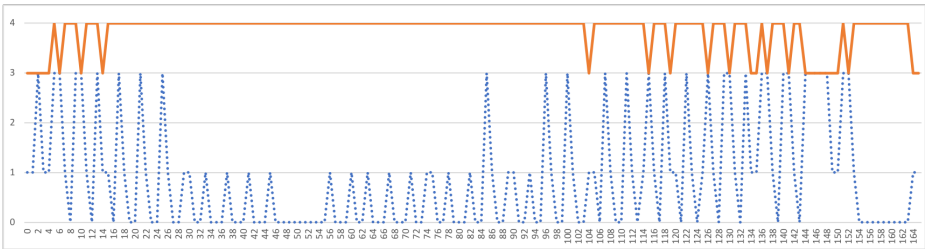


Fig. 10. Max-Min degree for the employee hiring process (V4)

clusive gateways, they deal with three special cases of BPMN processes: with only one type of gateways; without split exclusive gateway nor cycles; with only two types of gateways. Each case is treated with a different algorithm. The solution proposed in this paper focuses on BPMN processes with time too, and allows for the automatic computation of the degree of parallelism for complex BPMN structures, i.e., combining different gateways and cycles, without imposing restrictions on the structure of processes.

In [13], the authors propose an approach to automatically measure the degree of parallelism for BPMN processes. They rely on a formal model for BPMN processes defined in terms of Labelled Transition Systems, obtained through process algebra encodings. The degree of parallelism is then computed by using model checking techniques and dichotomic search. The main difference with respect to the approach presented in this work is that the subset of BPMN considered in [13] does not support timing features.

Several research contributions have used rewriting logic and Maude to formalyze and analyze BPMN processes. El-Saber and Boronat [8] propose a translation of BPMN into rewriting logic with a special focus on data-based decision gateways. They provide mechanisms to avoid structural issues in workflows such as flow divergence by introducing the notion of "well-formed" BPMN process. Kheldoun *et al.* [11] propose high-level Petri nets and to use Maude's LTL model checker for, respectively, specifying BPMN processes and analyzing behavioral properties. Both works do not support time features. Corradini *et al.* [4] present BProVe, a tool for the verification of business processes modeled in BPMN. The tool accepts BPMN processes in standard notation and can perform checks of soundness and safeness on them, as defined in [22] and [5], respectively, using Maude's LTL model checker.

In a previous work [7], the idea of specifying BPMN with time using Maude's rewriting logic was introduced by some of the authors of the present paper. However, little attention was paid to the degree of parallelism. This is the focus of the current paper, which presents how the parallelism degree of timed BPMN processes can be automatically computed, and how it can be used as a measure to improve and optimize a process in practice. In the present paper, it is also shown how to compute the variation between the minimum and maximum degree of

parallelism of timed BPMN processes. More recently, the authors have developed a rewriting logic executable specification of BPMN with time and probabilities supporting the automatic analysis of stochastic properties via statistical model checking [6].

8 Concluding Remarks

This paper contributed a mechanical approach to the key question of business process optimization. Business processes are described using a subset of BPMN supporting the main behavioral constructs (including, start/end events, flows, tasks, gateways) and time features. This BPMN subset was formalized using rewriting logic, resulting in a formal and executable semantics of the language. In a second step, the timed degree of parallelism has been computed. This measure can be useful for better understanding a business process and for improving the execution time of a process. The parallelism degree is computed automatically using Maude's metaprogramming capabilities. A realistic case study has been used to illustrate the approach, which can in general guide process refactoring tasks with optimization purposes in mind.

As far as future work is concerned, a first perspective is to integrate an explicit description of the resources (e.g., HR, assistant, and employee in the case study in Section 6) at the BPMN model level. To enable the automatic computation of new metrics, such as resource occupancy and average execution time, the verification framework would need to be extended for considering multiple concurrent executions of a process. A second perspective could focus on the thorough automation of the approach presented here. It is true that in its current form, the refactoring task is guided by the parallelism degree results, but it is ultimately manually applied. Measuring the degree of parallelism could be part of a more general methodology where other measures and additional information (e.g., regarding the resources) would drive the entire automated refactoring of the process for optimization purposes.

Acknowledgments. The authors would like to thank the anonymous reviewers for their valuable comments on an earlier draft of this paper. F. Durán has been partially supported by Spanish MINECO/FEDER project TIN2014-52034-R. The work of C. Rocha was partially supported by CAPES, Colciencias, and Inria via the STIC AmSud project "EPIC: EPistemic Interactive Concurrency" (Proc. No 88881.117603/2016-01), and by Capital Semilla 2017, project "SCORES: Stochastic Concurrency in Rewrite-based Probabilistic Models" (Proj. No. 020100610).

References

1. `http://maude.lcc.uma.es/MaudeBPMN/`.
2. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and Proof in Membership Equational Logic. *Theoretical Computer Science*, 236(1):35–132, 2000.

3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.
4. F. Corradini, F. Fornari, A. Polini, B. Re, F. Tiezzi, and A. Vandin. BProVe: A Formal Verification Framework for Business Process Models. In *Proc. of ASE*, pages 217–228. IEEE Computer Society, 2017.
5. R. Dijkman, M. Dumas, and C. Ouyang. Semantics and Analysis of Business Process Models in BPMN. *Information and Software Technology*, 50(12):1281–1294, 2008.
6. F. Durán, C. Rocha, and G. Salaün. Stochastic analysis of BPMN with time in rewriting logic. *Science of Computer Programming*, 168:1–17, Dec. 2018.
7. F. Durán and G. Salaün. Verifying Timed BPMN Processes using Maude. In *Proc. of COORDINATION*, volume 10319 of *LNCS*, pages 219–236. Springer, 2017.
8. N. El-Saber and A. Boronat. BPMN Formalization and Verification using Maude. In *Proc. of BM-FA'14*, pages 1–8. ACM, 2014.
9. J. Esparza. Reachability in Live and Safe Free-choice Petri Nets is NP-complete. *Theoretical Computer Science*, 198:211–224, 1998.
10. ISO/IEC. International Standard 19510, Information technology – Business Process Model and Notation. 2013.
11. A. Kheldoun, K. Barkaoui, and M. Ioualalen. Specification and Verification of Complex Business Processes - A High-Level Petri Net-Based Approach. In *Proc. of BPMN*, volume 9253 of *LNCS*, pages 55–71. Springer, 2015.
12. A. Krishna, P. Poizat, and G. Salaün. VBPMN: Automated Verification of BPMN Processes. In *Proc. of IFM'17*, volume 10510 of *LNCS*, pages 323–331. Springer, 2017.
13. R. Mateescu, G. Salaün, and L. Ye. Quantifying the Parallelism in BPMN Processes using Model Checking. In *Proc. of CBSE'14*, pages 159–168. ACM, 2014.
14. E. Mayr. An Algorithm for the General Petri Net Reachability Problem. *SIAM Journal on Computing*, 13(3):441–460, 1984.
15. J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
16. L. E. M. Morales, M. I. Capel, and M. A. Pérez. Conceptual Framework for Business Processes Compositional Verification. *Information and Software Technology*, 54(2):149–161, 2012.
17. P. C. Ölveczky and J. Meseguer. Semantics and Pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.
18. OMG. *Business Process Model and Notation (BPMN) – Version 2.0*. January 2011.
19. P. Poizat and G. Salaün. Checking the Realizability of BPMN 2.0 Choreographies. In *Proc. of SAC'12*, pages 1927–1934. ACM Press, 2012.
20. Y. Sun and J. Su. Computing Degree of Parallelism for BPMN Processes. In *Proc. of ICSOC'11*, volume 7084 of *LNCS*, pages 1–15. Springer, 2011.
21. P. Wong and J. Gibbons. A Process Semantics for BPMN. In *Proc. of ICFEM'08*, volume 5256 of *LNCS*, pages 355–374. Springer, 2008.
22. M. T. Wynn, H. M. W. Verbeek, W. M. P. van der Aalst, A. H. M. ter Hofstede, and D. Edmond. Business Process Verification - Finally a Reality! *Business Process Management Journal*, 15(1):74–92, 2009.