Probabilistic Runtime Enforcement of Executable BPMN Processes

Yliès Falcone, Gwen Salaün, Ahang Zuo

Probabilistic Runtime Enforcement of Executable BPMN Processes

Yliès Falcone, Gwen Salaün, and Ahang Zuo

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

Abstract. A business process is a collection of structured tasks corresponding to a service or a product. Business processes do not execute once and for all, but are executed multiple times resulting in multiple instances. In this context, it is particularly difficult to ensure correctness and efficiency of the multiple executions of a process. In this paper, we propose to rely on Probabilistic Model Checking (PMC) to automatically verify that multiple executions of a process respect some specific probabilistic property. This approach applies at runtime, thus the evaluation of the property is periodically verified and the corresponding results updated. However, we go beyond runtime PMC for BPMN, since we propose runtime enforcement techniques to keep executing the process while avoiding the violation of the property. To do so, our approach combines monitoring techniques, computation of probabilistic models, PMC, and runtime enforcement techniques. The approach has been implemented as a toolchain and has been validated on several realistic BPMN processes.

1 Introduction

Business processes are structured tasks that model a specific service or product. Such processes are present in any company or institution worldwide, and there is a need for better controlling these processes to reduce costs and improve throughput. Many companies model their services and processes, thereby increasing their level of automation. One of the challenges in this context is to ensure the quality, correctness, and efficiency of these processes. In this paper, we assume that processes are described using Business Process Model and Notation (BPMN) [20], the standard business process modelling language. BPMN processes are not executed once but multiple times, resulting in multiple instances.

In this study, we focus on quantitative analysis of processes, which is particularly useful for computing probabilistic properties or other metrics related to time, costs or resource usage. More precisely, we use probabilistic model checking (PMC) to automatically verify that multiple executions of a process respect probabilistic properties [15]. In the context of BPMN processes, probabilistic properties help verifying that some task usage does not go above a certain threshold or for computing how many resources have to be associated with specific tasks to execute the process smoothly. Evaluating a probabilistic property is strongly related to the number of process instances being executed. Therefore, PMC should

be applied at runtime to analyse the current execution of running instances. The property is periodically verified, and the corresponding results are updated.

In this paper, we not only verify probabilistic properties on BPMN processes using PMC at runtime, but also enforce the process executions to not violate the property. To do so, we rely on runtime verification and enforcement techniques. Runtime verification [3, 10] is a technique to verify whether system's executions satisfy a given correctness property at runtime. Runtime Enforcement (RE) [12, 13] is complementary to runtime verification and provides techniques that can intervene in the system at runtime to ensure that the behaviour of the system respects the expected properties. In this paper, the system consists in the multiple executions of a process and we want these executions to always satisfy a given property. This is possible by catching the flow of executions of these process instances and by changing it (when the property is violated) using correcting actions (such as buffering or reordering specific tasks).

More precisely, we introduce probabilistic runtime enforcement, allowing BPMN processes to satisfy a given probabilistic property at runtime. To achieve this, we first convert the BPMN process into a formal model represented by a Labelled Transition System (LTS). We then monitor the multiple executions of the process and extract the corresponding traces (one trace per process instance). Based on these execution traces, we can annotate the LTS model of the process by adding execution probabilities to transitions of the LTS, thus obtaining a Probabilistic Transition System (PTS) model. It is worth noting that recent actions are taken into account to compute this PTS but are not effectively released and considered executed. Probabilistic model checking is then used to verify whether the PTS model satisfies the given property. If the property is satisfied, all recent actions are released. If the property is violated, the enforcement mechanism is triggered and the aforementioned recent actions are retained, removed or re-ordered to avoid the property violation. This approach was fully implemented and its effectiveness was validated on several examples of processes and properties.

The contributions of this work can be summarised as follows:

- A novel algorithm, which analyses (possibly incomplete) execution traces and builds a Probabilistic Transition System.
- A probabilistic enforcement mechanism, which avoids probabilistic property violation when executing multiple process instances.
- An entire toolchain supporting the whole approach and its validation on realistic processes.

The organisation of this paper is as follows. Section 2 introduces the background notions required to this work. Section 3 presents the probabilistic enforcement approach for BPMN. Section 4 describes the toolchain automating all the approach steps, illustrates the approach with a case study, and presents experimental results. Section 5 surveys related work, and Section 6 concludes.

2 Background

This section outlines the fundamental concepts, such as BPMN, Labelled Transition System (LTS), Probabilistic Transition System (PTS), execution traces, and probabilistic properties.

2.1 Business Process Model and Notation

Business Process Model and Notation (BPMN) is a widely used workflow-based notation for describing and modelling business processes [20]. The syntax of a BPMN process is defined as a graph-based structure, where vertices or nodes represent various elements such as events, tasks, and gateways, and edges or flows connect these nodes. Figure 1 introduces the key elements of the BPMN notation.

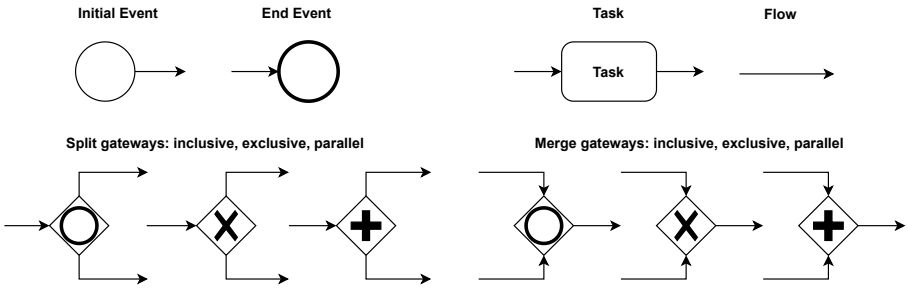

Fig. 1: Excerpt from the BPMN notation.

The diagram includes the *initial event* and the *end event*, which serve to initialise and terminate processes, respectively. It is assumed that there is only one initial event, which corresponds to the initiation of a process and at least one end event, which corresponds to the completion of a process. *Task* represents an atomic activity and typically has only one incoming flow and one outgoing flow, denoting the sequence of activities within the process. *Gateways* are used to describe the control flow of the process. There are two patterns for each gateway type: the split pattern and the merge pattern. The split pattern consists of a single incoming flow and multiple outgoing flows. The merge pattern consists of multiple incoming flows and a single outgoing flow. Several types of gateways are available, such as exclusive, parallel, and inclusive gateways. An exclusive gateway corresponds to a choice among several flows. A parallel gateway executes all possible flows at the same time. An inclusive gateway executes one or several flows. The choice of flows to execute in exclusive and inclusive gateways depends on the evaluation of data-based conditions.

This paper focuses on the multiple executions of a single process, known as process instances. Each instance is characterised by an identifier and by the list

of tasks executed by this instance. It is assumed that each instance eventually completes, thus resulting in a finite list of tasks.

2.2 LTS & PTS

Labelled and Probabilistic Transition Systems are used in this paper as semantic models for BPMN. Moreover, they allow the automated analysis of the corresponding BPMN processes.

Definition 1 (LTS). *A Labelled Transition System (LTS) is a tuple* $\langle Q, \Sigma, q_{init}, \Delta \rangle$*, where:* Q *is a finite set of states,* Σ *is a finite set of labels/actions,* q_{init} *is the initial state,* $\Delta \subseteq Q \times \Sigma \times Q$ *is a transition relation, where* $(q, a, q') \in \Delta$ *represents a possible transition from state* q *to state* q' *with label* a*, also written* $q \xrightarrow{a} q'$*.*

Probabilities are useful for making explicit the likelihood of executing specific tasks in a process. Therefore, we also use Probabilistic Transition Systems [23], an extension of the LTS model that incorporates probabilities for transitions.

Definition 2 (PTS). *A Probabilistic Transition System (PTS) is a tuple* $\langle S, A, s_{\text{init}}, \delta, P \rangle$ *such that* $\langle S, A, s_{\text{init}}, \delta \rangle$ *is a labelled transition system as per Definition 1 and* $P : \delta \to [0, 1]$ *is the probability labelling function.*

$P(s \xrightarrow{a} s') \in [0, 1]$ is the probability for the system to move from state s to state s', performing action a. For each state s, the sum of the probabilities associated with its outgoing transitions is equal to 1, that is $\forall s \in S : \sum_{s' \in S} P(s, a, s') = 1$. When using LTS or PTS as a semantic model of a BPMN process, the set of labels or alphabet refers to the set of tasks appearing in the BPMN process.

2.3 Execution Traces

A process can be executed multiple times, resulting in multiple instances. Each process instance being executed can be in one of the following three states: waiting state, running/ongoing state, and completed state. Any (ongoing or completed) instance consists of a sequence of tasks within the process. Every time an instance executes, it results in an execution trace of tasks.

Definition 3 (Execution Trace). *An execution trace* ($\sigma_{\mathcal{T}}$) *refers to a sequence of tasks that are executed in a specific order by a specific process instance.*

It is worth noting that in the rest of this work, an execution trace can be completed or not. In the latter case, this is due to the fact that the process instance is still running and has not completed yet.

Several operations can be performed on execution traces. Assuming an execution trace σ of length n and an execution trace σ' of length m, we define the following primitive operations:

- **Size:** $Size(\sigma) = |\sigma|$.
- **Index:** $\sigma[i]$ is the ith element in σ, $i < n$
- **Slice:** $\sigma[0...i] = \sigma[0].\sigma[1].\cdots.\sigma[i-1]$, $i \leq n$.
- **Concatenation:** $Concat(\sigma, \sigma') = \sigma[0...n].\sigma'[0...m]$.
- **Reorder:** $Reorder(\sigma, \sigma') = \sigma'[0...m].\sigma[0...n]$.

2.4 Probabilistic Properties

The Model Checking Language (MCL) [26] is a branching-time temporal logic that is suitable for expressing properties of concurrent systems using actions. It extends the alternation-free μ-calculus [9] with regular expressions, data-based constructs, and fairness operators. A probabilistic property is a specification or requirement that expresses a probabilistic behaviour of a system or model being analysed. In this paper, probabilistic properties are used to describe the requirements for the probability of execution of a task or a set of combined tasks in a BPMN process. We use MCL to describe probabilistic properties using the prob R is op [?] E end prob construct [24], where R is a regular formula that describes transition sequences, op is a comparison operator such as "<", "≤", ">", "≥", "=", "<>", and E is a real number that represents a probability. Given an MCL probabilistic property and a PTS model, we use the CADP Probabilistic Model Checker [24] in order to evaluate the property on the PTS model.

3 Probabilistic Runtime Enforcement

Our approach takes two inputs, a BPMN model and a probabilistic property, and produces as output a list of safe-to-execute tasks, in the sense that they do not violate the given property. This approach consists of three parts: the *monitoring* part, the *transformation* part, and the *probabilistic runtime enforcement* mechanism (Figure 2). First, monitoring is used to observe the multiple executions of the given process, in particular to retrieve the tasks executed by each process instance (resulting in execution traces). Second, the input BPMN model is transformed into its corresponding semantic model, namely an LTS. This step is performed only once. Finally, the probabilistic runtime enforcement mechanism consists of two modules. The first module corresponds to Probabilistic Model Checking (PMC), which determines whether a new version of the PTS violates the given probabilistic property. The second module corresponds to the enforcer, which is activated only when the probabilistic model checking returns false. In such a case, the enforcer applies appropriate techniques to modify the input trace (e.g., by retaining some tasks and not executing them immediately), and thus avoid property violation.

3.1 Monitoring

Monitoring techniques are useful to observe and monitor the current status of the BPMN process executions. More precisely, we monitor process executions

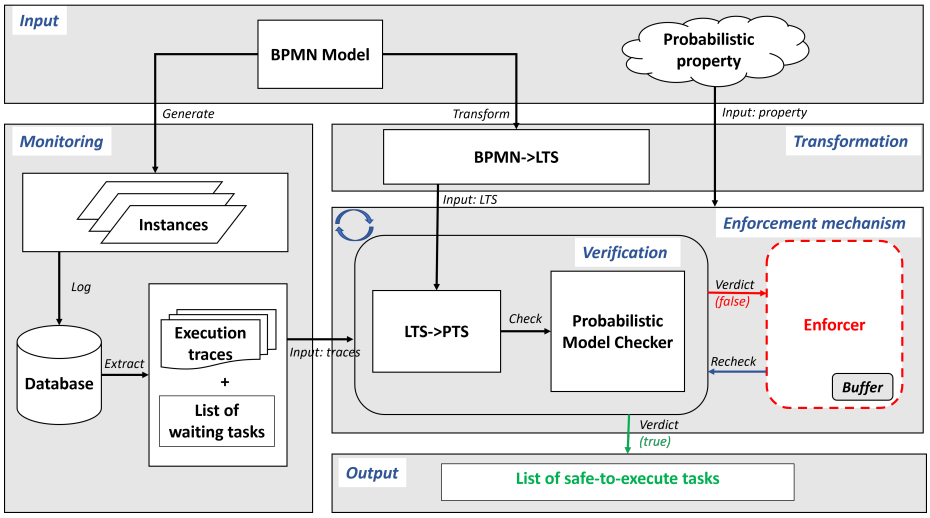

Fig. 2: Approach Overview.

from an instance perspective since the main goal is to extract all traces executed by ongoing process instances on a given period.

Figure 3 illustrates the monitoring process of a BPMN process at runtime, which involves observing every generated instance for that process. Multiple instances can execute concurrently, and all information related to the execution of one process instance is stored in a database. To retrieve execution traces for all process instances, we rely on extraction techniques at varying levels of granularity. As shown in the figure, each instance execution trace is composed of a process ID, an instance ID, a set of tasks, a start time, and an end time.

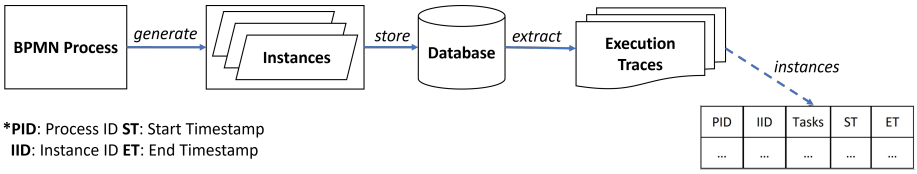

Fig. 3: Runtime monitoring of multiple executions of a BPMN process.

Since we focus here on long-running process executions, it does not make sense to retrieve all execution traces from the beginning. Therefore, the extraction is triggered for a specific time window. This operation is repeated periodically, thus resulting in a *sliding window* algorithm. Algorithm 1 aims at extracting the execution traces for all instances that are either in progress or have already finished during a specified time window. The algorithm takes as input

the process ID, the checkpoint timestamp, and the window duration. It first initialises an empty list for the output traces. Then, it retrieves all execution traces associated with the process ID using the *getTraces()* method, which extracts all execution traces as illustrated in Figure 3. For each instance, it checks whether its *endTime* property is None (instance still running), or less than or equal to the start of the window. If so, it appends the execution trace to the output trace list. Finally, the algorithm returns as output a set of traces executed on that window. The time complexity of this algorithm is $\mathcal{O}(n)$, where n is the number of instances in the process.

Algorithm 1 Get traces in the sliding window

Inputs: Process ID PID, Checkpoint Timestamp ts, window duration td
Output: Execution traces $\mathcal{T}$
1: $\mathcal{T} := []$
2: $\mathcal{T}_{all} := PID.getTraces()$
3: **for** each $Tr \in \mathcal{T}_{all}$ **do**
4: **if** $Tr.endTime$ is None or $Tr.endTime \leq ts - td$ **then** $\mathcal{T}.append(Tr)$
return $\mathcal{T}$

3.2 Transforming BPMN into LTS

LTS is a semantic model that shows all possible execution paths for a process. To transform BPMN into LTS, we rely on an existing approach that first translates BPMN into the LNT process algebraic specification language, and then transforms it into an LTS by using CADP compilers [17]. For more information on the transformation process from BPMN to LTS, please refer to [22, 27].

3.3 Transforming LTS into PTS

The transformation process from an LTS to a PTS consists of two steps. The initial step aims at traversing all provided instances and identifying all the possible execution paths for each instance (Algorithm 2). In a second step, a counter is added to each transition of the LTS, thus allowing us to track the number of times each transition is executed. This facilitates the calculation of the probability value associated with executing each transition. Finally, the output model is represented as a PTS (Algorithm 3).

An execution path is a sequence of transitions in the LTS that matches with the execution trace of an instance. When an instance has been successfully completed, there exists only one corresponding execution path. The LTS may exhibit non-deterministic behaviour due to the presence of inclusive gateways in the BPMN model. Therefore, when considering unfinished instances, we calculate the execution probabilities of all relevant paths and normalize these probabilities.

Algorithm 2 takes as input an LTS and an execution trace of an instance $\mathcal{T}_{tasks}$ (i.e. a list of tasks), and finds all feasible execution paths in the LTS that

satisfy the given execution trace. The algorithm uses a depth-first search (DFS) approach to traverse the LTS, starting from the initial state. It compares the tasks in the transitions of the LTS with the tasks in the ordered sequence of tasks to determine feasible paths. The algorithm maintains a stack to keep track of the current state and partial paths, and recursively explores all possible transitions from the current state until it reaches a state that fully matches the ordered sequence of tasks. Given that it is a non-deterministic model, it then backtracks to explore other possible transitions and continues the exploration process until all paths have been exhaustively explored. The time complexity of the algorithm is $\mathcal{O}(|Q| \times |\Delta|)$, where $|Q|$ represents the number of states in the LTS and $|\Delta|$ represents the number of transitions in the LTS.

Algorithm 2 Get all execution paths of an instance in LTS (FindPaths)

Inputs: LTS $= \langle Q, \Sigma, q_{init}, \Delta \rangle$, an execution trace $\mathcal{T}_{tasks} = [t_1, t_2, \ldots, t_n]$
Output: A list of paths (resultPaths)

```
1: resultPaths := []
   return DFS(LTS, T_tasks, q_init, [], resultPaths)

2: function DFS(LTS, tasks, q_current, currentPath, resultPaths)
3:     if Size(tasks) == 0 then
4:         return resultPaths.append(currentPath)
5:     else
6:         task := tasks[0]; restTasks := tasks[1:]
7:         Q_next := {q' ∈ Q | (q_current, task, q') ∈ Δ}
8:         for all q_next ∈ Q_next do
9:             nextPath := currentPath
10:            nextPath.append((q_current, task, q_next))
11:            DFS(LTS, restTasks, q_next, nextPath, resultPaths)
```

Algorithm 3 takes as input an LTS and a list of execution traces $\mathcal{I}$, and computes a PTS representing the probability distribution of transitions between states of the LTS based on the occurrence of tasks in the set of execution traces. The algorithm first initialises a counter for each transition in the LTS, which records the number of times the transition is taken in the execution trace (line 1). Then, for each execution trace in the list, the algorithm computes the set of possible execution paths in the LTS that correspond to the execution trace (line 5). If there is only one path, the algorithm increments the counter for each transition in the path by 1 (lines 6 to 7). If there are multiple paths, the algorithm increments the counter for each transition in each path by 1, but also keeps track of the number of execution traces that have multiple paths to avoid double-counting (lines 10 to 11). Finally, the algorithm computes the probability of each transition by dividing its counter by the sum of counters for all transitions with the same source state and event (line 12). The resulting probabilities are normalised so that they sum to 1 (line 13). The algorithm returns the PTS,

which consists of the set of states, tasks, and transitions of the LTS, along with the computed probabilities for each transition. The time complexity of this algorithm is $\mathcal{O}(|\mathcal{I}| \times |Q| \times |\Delta|)$, where $|\mathcal{I}|$ is the number of execution traces, $|Q|$ represents the number of states in the LTS, and $|\Delta|$ represents the number of transitions in the LTS.

Algorithm 3 Computation of PTS (ComputePTS)

Inputs: LTS $= \langle Q, \Sigma, q_{init}, \Delta \rangle$, a list of execution traces $\mathcal{I} = [I_1, I_2, \ldots, I_n]$
Output: PTS $= \langle S, A, s_{init}, \delta, P \rangle$

for each $(q, a, q') \in \Delta$ **do** $cnt((q, a, q')) := 0$
$Paths := [], counter := 0$ ▷ *counter* records the number of unfinished traces
for all $I_i \in \mathcal{I}$ **do**
  $\mathcal{T}_{tasks} := I_i.getTasks()$
  $Paths :=$ FindPaths(LTS, $\mathcal{T}_{tasks}$) ▷ FindPaths (Algorithm 2)
  if *Size*(Paths) == 1 **then**
    for each $(s, a, s') \in Paths[0]$ **do** $cnt((s, a, s')) := cnt((q, a, q')) + 1$
  else
    $counter := counter + 1$
    for each $Path \in Paths$ **do**
      for each $(s, a, s') \in Path$ **do** $cnt((s, a, s')) := cnt((q, a, q')) + 1$

$P := \{(s, a, s') \mapsto cnt((s, a, s'))/$ ▷ calculate probabilities
$(\sum_{q \in S, a' \in A, (s, a', q) \in \delta} cnt((s, a', q)) - counter) \mid (s, a, s') \in \delta\}$
$P := Normalisation(P)$
return $\langle S, A, s_{init}, \delta, P \rangle$

3.4 Critical Tasks

In this subsection, we describe how to define and compute *critical actions/tasks* given an LTS model of a BPMN process and a probabilistic property. Critical tasks refer to specific tasks that play a crucial role in determining whether a system's behaviour violates or satisfies a given property. This notion is at the heart of the enforcement techniques presented in the next subsection.

The notion of critical task used here is inspired by the notion of *last action of the property* introduced in [16]. This paper states that the violation of a property by a given model is somehow triggered when the last action of the property is executed by the model. In other words, if the last action is not executed, the model does not violate the property. Depending on the actions used in the probabilistic property (including the last action), we can identify one or more execution paths in the LTS, including the actions of the property, where each path consists of an ordered list of transitions. We then traverse this set of paths and for each path we search for the last state (the closest to the end of the path) corresponding to a choice between several transitions. This state

s is particularly important because it is the last opportunity to avoid reaching the last action (of the property) and thus violating the property. The actions or tasks for all transitions outgoing from state s are *candidates* to critical tasks. At this point, the operator of the property needs to be considered. If the operator is less than ("<" or "≤"), there is one critical task, corresponding to the transition outgoing from s and leading to the last action. If the operator is greater than (">" or "≥"), the critical tasks correspond to all transitions outgoing from s and leading to actions other than the last one. If the operator is "=" or "<>", the critical tasks correspond to all tasks appearing on transitions outgoing from s.

Algorithm 4 Computation of critical tasks in LTS (ComputeCriticalTasks)

Inputs: LTS = $\langle Q, \Sigma, q_{init}, \Delta \rangle$, Probabilistic property (pp)
Output: A set of Critical Tasks ($CTasks$)

```
 1: CTasks := {}, T_tasks := pp.getTasks()
 2: Paths := FindPaths(LTS, T_tasks)                    ▷ FindPaths (Algorithm 2)
 3: for each path ∈ paths do
 4:     reversedPath := Reverse(path)
 5:     for each transition (s, task, s') in reversedPath do
 6:         Δ_s ⊆ {(s, a, q) ∈ Δ | q ∈ Q}
 7:         if Size(Δ_s) > 1 then
 8:             if pp.operator() is " > " or " ≥ " then
 9:                 CTasks := CTasks ∪ {a ∈ Σ \ task | ∃q ∈ Q, (s, a, q) ∈ Δ_s}
10:             else if pp.operator() is " < " or " ≤ " then
11:                 CTasks := CTasks ∪ {task}
12:             else
13:                 CTasks := CTasks ∪ {a ∈ Σ | ∃q ∈ Q, (s, a, q) ∈ Δ_s}
14:             break
    return CTasks
```

Algorithm 4 presents a method for computing the critical tasks ($CTasks$) given an LTS and a probabilistic property (pp). The algorithm starts by initialising $CTasks$ as an empty set and extracts the set of all tasks $\mathcal{T}_{tasks}$ included in the probabilistic property. Next, it calls FindPaths (Algorithm 2) to find all paths in the LTS that include the tasks in $\mathcal{T}_{tasks}$ (line 2). For each path found, the algorithm reverses it and iterates over the transitions in reverse order. For each transition t represented as $(s, task, s')$, the algorithm selects the set of outgoing transitions from state s in the LTS, denoted by Δ_s (line 6). If the size of Δ_s is greater than 1, the algorithm checks the operator specified in pp (lines 7 to 13). If the operator is either $>$ or $\geq$, the algorithm adds to $CTasks$ the set of all actions a in Σ that have outgoing transitions from state s and do not correspond to the task in $task$ (lines 8 to 9). If the operator is $<$ or $\leq$, the algorithm adds the task $task$ to $CTasks$ (lines 10 to 11). Otherwise, the algorithm adds to $CTasks$ the set of all actions a in Σ that have outgoing transitions from state s (line 13). Finally, the algorithm breaks out from the loop for the current

path. The algorithm returns the set of critical tasks *CTasks* as output. The time complexity of this algorithm is $\mathcal{O}(f(n) \times |\Delta|)$, where $f(n)$ is the time complexity of the FindPaths algorithm and $|\Delta|$ is the number of transitions in the LTS.

3.5 Probabilistic Runtime Enforcement (PRE)

The enforcement mechanism (EM) requires as input a probabilistic property φ and an LTS (Fig. 4). It is triggered right after the monitoring component. At runtime, it periodically receives a list of execution traces and a list of waiting tasks (waiting to be executed) from the monitoring component, and produces as output a list of tasks (to be executed) whose execution does not cause the violation of the probabilistic property, as verified using PMC techniques.

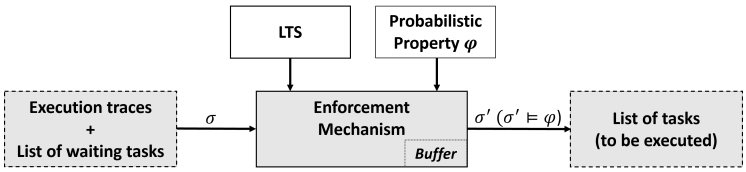

Fig. 4: Overview of PRE.

The enforcement techniques used in this paper rely on two operations: *reordering* and *buffering*. Reordering techniques correspond to a change in the order of application of some of the tasks received as input. Buffering techniques rely on a FIFO buffer $\mathcal{B}$, which stores critical tasks when necessary. Buffering techniques aim at delaying the execution of specific tasks by adding them temporarily to the buffer $\mathcal{B}$ and taking them out of the buffer when their execution does not induce the violation of the property.

Algorithm 5 presents the enforcement mechanism in detail. The algorithm takes as input a list of (waiting) tasks, a probabilistic property φ, and an LTS. It returns a list of tasks to be executed (in the best case, the same sequence of tasks given as input) that satisfies φ. The idea is to update the PTS by merging the execution traces and the tasks to be executed (waiting tasks and tasks in the buffer), and to use PMC techniques to determine whether these new tasks would still preserve the satisfaction of the property. If the executions of these tasks would violate the property, buffering or reordering techniques are triggered.

The algorithm is initialised when the EM is called for the first time. Initialisation consists of (i) computing the critical tasks using the ComputeCriticalTasks algorithm (Algorithm 4) and storing them in the global variable *ct*, and (ii) initialising the buffer $\mathcal{B}$ to empty. The ComputeCriticalTasks algorithm computes the tasks of the process that can avoid the property violation and thus will be stored in the buffer $\mathcal{B}$ by the enforcer when necessary. When the enforcement mechanism is used for the first time, the list of tasks to be processed only consists of the waiting tasks. Later on, each time enforcement is used, the list

Algorithm 5 Enforcement Mechanism

Inputs: a list of execution traces $\mathcal{T}$, a list of waiting tasks $\sigma_{\mathcal{T}}$, a probabilistic property φ, an LTS.
Output: a list of tasks to be executed $\sigma'_{\mathcal{T}}$

if EM is not initialised **then** ▷ ct and $\mathcal{B}$ are **Global** variables.
  $ct :=$ ComputeCriticalTasks(LTS, φ) ▷ Algorithm 4
  $\mathcal{B} := [], \sigma := \sigma_{\mathcal{T}}$ ▷ Initialise Buffer $\mathcal{B}$
else
  $\sigma_{buffer} := \langle task \mid task \in \mathcal{B}.getTasks() \rangle$ ▷ All tasks in Buffer
  $\sigma := Concat(\sigma_{buffer}, \sigma_{\mathcal{T}})$ ▷ Concatenation
return $\sigma'_{\mathcal{T}} :=$ EM(LTS, $\mathcal{T}$, σ, φ, ct)

function EM(LTS, $\mathcal{T}$, σ, φ, ct)
  if Check(LTS, $\mathcal{T}$, σ, φ) **then**
    $\sigma_s := \langle task \mid task \in \sigma \wedge task \in \mathcal{B}.getTasks() \rangle$
    RemovefromBuffer(σ_s) ▷ Buffering: (Remove)
    return σ
  else
    $\sigma_1 := \langle task \mid task \in \sigma \wedge task \in ct \rangle$, $\sigma_2 := \langle task \mid task \in \sigma \wedge task \notin \sigma_1 \rangle$
    $\sigma_r := Reorder(\sigma_1, \sigma_2)$ ▷ Reordering
    if Check(LTS, $\mathcal{T}$, σ_r, φ) **then**
      $\sigma_s := \langle task \mid task \in \sigma_r \wedge task \in \mathcal{B}.getTasks() \rangle$
      RemovefromBuffer(σ_s) ▷ Buffering: (Remove)
      return σ_r
    else
      $\sigma', \sigma'' :=$ Bisection(σ_1) ▷ Binary-Search
      $\sigma_a := \langle task \mid task \in \sigma'' \wedge task \notin \mathcal{B}.getTasks() \rangle$
      AddtoBuffer(σ_a) ▷ Buffering: (Add)
      $\sigma_b := Concat(\sigma_2, \sigma')$ ▷ Concatenation
      EM(LTS, $\mathcal{T}$, σ_b, φ, ct)

function Check(LTS, $\mathcal{T}$, σ, φ) ▷ Probabilistic model checking
  return UpdatePTS(LTS, $\mathcal{T}, \sigma$) $\models \varphi$? **true** : **false**

function UpdatePTS(LTS, $\mathcal{T}$, σ) ▷ Transforming LTS into PTS
  $\mathcal{I} := []$
  for each $task \in \sigma$, *in order* **do** $I := task.getInstance()$ ▷ I: Execution trace
    $I.append(task)$, $\mathcal{I}.append(I)$
  for each $\tau \in \mathcal{T}$ **do** $I := \tau.getInstance()$
    if $I \notin \mathcal{I}$ **then** $\mathcal{I}.append(I)$
  return ComputePTS(LTS, $\mathcal{I}$) ▷ ComputePTS (Algorithm 3)

function Bisection(σ) ▷ Binary-Search
  $n := Size(\sigma)$; $m := \lfloor n/2 \rfloor$
  return $\sigma[0...m]$, $\sigma[m...n]$

of tasks to be processed is obtained by concatenating all the tasks in the buffer with the tasks in the waiting list (line 6). Function EM then starts processing this list of tasks. The CHECK function first verifies whether the given execution traces and the given list of tasks satisfy the property by using PMC. If this function returns true, all the tasks are removed from the buffer and the algorithm returns the tasks in the buffer and the waiting tasks (lines 8 to 11). Otherwise, the enforcement techniques are triggered. First, reordering techniques are applied as follows. The list of tasks is reordered by favouring (and thus executing first) the non-critical tasks, which are placed at the beginning of the list. Then, the PTS is built again, and PMC called to check whether ordering differently the tasks to be executed avoid the property violation (line 15). If the result is true, the buffer is emptied, and the list of tasks is returned. If the result is false, reordering techniques are not enough, and in such a case, the mechanism then executes some of the tasks only partially. To identify the subset of tasks that can be executed without violating the property, we use the BISECTION function (lines 34 to 36). This function helps to avoid an exhaustive exploration of all possible combinations of tasks (and calling PMC for each solution), which would be too costly and time-consuming. This function divides the list of critical tasks into two parts. The algorithm then puts the second part into the buffer and recursively calls the EM function for this new list of tasks, which is the list of non-critical tasks (computed on line 13) concatenated with the first part returned by the BISECTION function (lines 20 to 24). The algorithm ends when the verdict of PMC is true and returns a list of safe-to-execute tasks.

The time complexity of this algorithm is $\mathcal{O}(\log|\sigma_{\mathcal{T}}| \times f(|\sigma_{\mathcal{T}}|))$, where $|\sigma_{\mathcal{T}}|$ is the size of the given list of tasks, and $f(|\sigma_{\mathcal{T}}|)$ represents the time complexity of using PMC.

3.6 Characteristics

This paper proposes enforcement mechanism that is online, untimed, and operational, meaning it utilises real-time system traces, disregards physical time intervals, and offers a practical implementation guide. This mechanism has three main characteristics: *soundness*, *monotonicity*, and *transparency*. PRE refers to the probabilistic enforcement mechanism, PRE.buff is the buffer $\mathcal{B}$, $\neg E$(PRE.buff) means that the buffer was not triggered, PRE.out refers to the output of the mechanism, and CHECK refers to the probabilistic model checking function.

Proposition 1 states that the tasks in each trace generated by the mechanism do not violate the properties of the system by their execution.

Proposition 1 (Soundness)
$\forall\sigma : \text{PRE}(\text{LTS}, \mathcal{T}, \sigma, \varphi).\text{out} = \sigma'_{\mathcal{T}} \implies \textsc{Check}(\text{LTS}, \mathcal{T}, \sigma'_{\mathcal{T}}, \varphi) == \textbf{\textit{true}}$

Proof (Sketch). If the PMC's verdict is false, the execution monitor does not produce any tasks as output to maintain soundness.

Proposition 2 states that the enforcer's output sequence consistently grows with respect to the number of non-critical tasks in the input sequence.

Proposition 2 (Monotonicity)
$\forall t \in \sigma, t' \in \sigma', t, t' \notin ct : size(\sigma) \leq size(\sigma') \implies size(\text{PRE}(\text{LTS}, \mathcal{T}, \sigma, \varphi).\text{out}) \leq size(\text{PRE}(\text{LTS}, \mathcal{T}, \sigma', \varphi).\text{out})$

Proof (Sketch). The buffer exclusively stores critical tasks. Therefore, as the number of non-critical tasks in the input increases, the length of the output of the mechanism also increases.

The execution monitor is transparent, which means that it only intervenes if the input tasks to be executed violate the property.

Proposition 3 (Transparency)
$\text{PRE}(\text{LTS}, \mathcal{T}, \sigma, \varphi).\text{out} = \sigma'_{\mathcal{T}}, \neg E(\text{PRE.buff}) \implies \text{PRE}(\text{LTS}, \mathcal{T}, \sigma'_{\mathcal{T}}, \varphi).\text{out} = \sigma$

Proof (Sketch). Since there is no suppression operation in the enforcement mechanism, all tasks in the input σ are the same as in the output $\sigma'_{\mathcal{T}}$ when the buffer is not triggered.

4 Tool Support & Evaluation

This section first presents the toolchain that automates the different steps of our approach. We then provide a practical illustration of the approach and tools using a case study. Finally, additional experiments are presented to evaluate the tools' performance on a series of realistic examples.

4.1 Tool

Figure 5 gives an overview of the toolchain. As far as the inputs are concerned, we rely on the open-source tool Activiti [2] to specify and execute BPMN processes. Probabilistic properties are described using MCL. The monitoring techniques are implemented in Java and aim at extracting the required information about execution traces from a MySQL database. The transformation from BPMN processes to LTS models is performed using an open-source tool called VBPMN [21]. The annotation of the LTS model with probabilities, thus resulting in a PTS model, is implemented in Java. PMC is computed using the CADP probabilistic model checker, which takes as input an MCL probabilistic property and a PTS, and returns a Boolean value. Finally, the enforcer is also implemented in Java and applies the correction when necessary on the input flow of tasks using the techniques (reordering and buffering) presented in Section 3.

4.2 Case Study

The approach is illustrated using the shipment process of a hardware retailer [25]. Figure 6 shows the BPMN process of this example, whose final goal is to deliver goods. More precisely, this process starts when there are goods ready for shipment. Two tasks are then executed concurrently: one involves packaging the

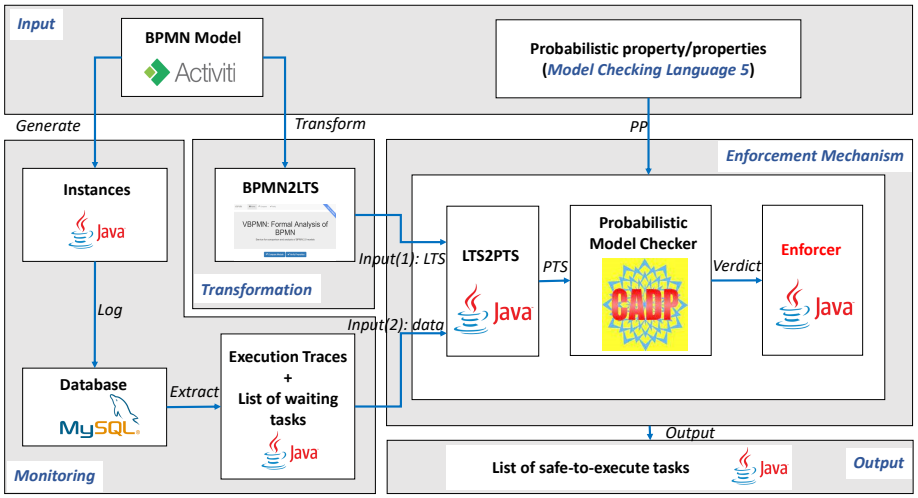

Fig. 5: Toolchain overview.

goods (T7) while the other determines whether a normal or special shipment is required (T1). Based on that decision, the first option verifies the need for additional insurance (T2), followed by the opportunity to purchase additional insurance (T4) and/or complete a post-label (T5). Another option is to request quotes from carriers (T3), followed by assigning a carrier and preparing the paperwork (T6). Finally, the package is transferred to a designated pick-up area (T8).

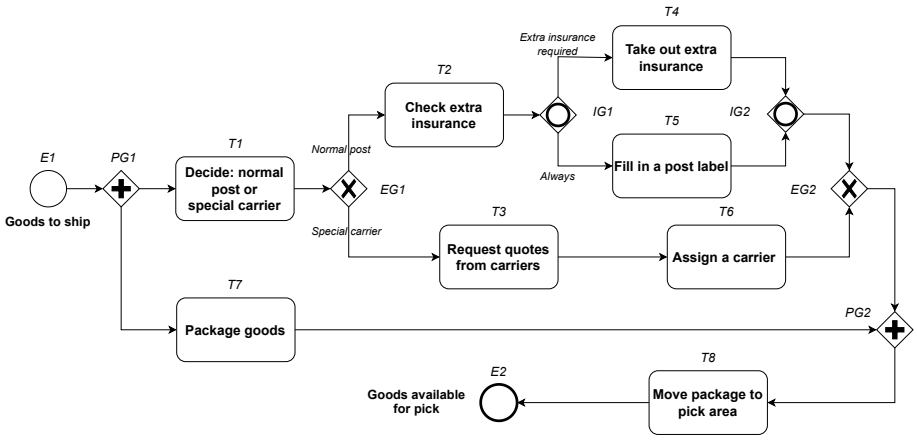

Fig. 6: BPMN shipment process of a hardware retailer.

For illustration purposes, we choose a property checking that the probability of executing task T4 after task T2 is less than 0.5. This is important because the choice of taking extra insurance (T4) comes with a cost, and if this decision is taken too often (more than half of the time here), this could result in high expenses on a short period of time. This property is expressed in MCL as follows: prob true*. T2. true*. T4 is $<$ 0.5 end prob. As the question mark symbol is used, the model checker returns a Boolean value indicating the property's truthfulness and a numerical value representing the probability of executing T4 after T2.

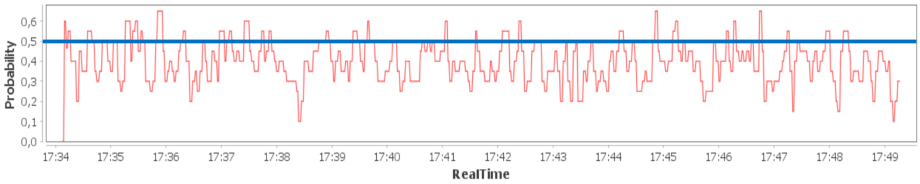

Fig. 7: Experiments on the case study *without* enforcement.

We have conducted two series of experiments with this running example, one without the enforcement mechanism (results are shown in Figure 7) and the other with enforcement (Figure 8). The same randomized workload of 2000 instances was used for each experiment. These experiments show that, without enforcement techniques, there is a 7% risk of violating the property, resulting in a satisfaction rate of 93%. In other words, the property is violated 7% of the time, which corresponds to the situations where the curve goes above the probability threshold represented as an horizontal line in Figure 7. On the other hand, Figure 8 shows that with enforcement, the instance executions keep satisfying the given probabilistic property, resulting in a 100% satisfaction rate and no violation of the property. In practice, this allows one to delay payment of extra insurance over time and thus avoids peaks of extra expenses.

Fig. 8: Experiments on the case study *with* enforcement.

4.3 Experiments

The goal of this section is to evaluate the correctness and performance of the enforcement approach. The correctness is calculated as the percentage of probabilistic properties violated during the running process, while the performance is measured by the average execution time (AET) of an instance. AET is computed by summing the execution time of each instance and by dividing this value by the number of instances. To conduct these experiments, we relied on a set of BPMN processes taken from the literature. Each process was executed 1000 times, resulting in 1000 instances. The time taken between the startup of two new process instances was computed using an exponential distribution with a lambda value of 5 ($\lambda = 5$). These experiments were performed on an Ubuntu OS laptop with a 1.7 GHz Intel Core i5 processor and 8 GB of RAM.

The results of these experiments are presented in Table 1. Each row gives the results for a given process by providing a description, its size in terms of number of tasks and gateways, the size of the corresponding LTS in terms of number of states and transitions, the correctness results without (a) and with (b) enforcement, and the AET without/with enforcement. The correctness value corresponds to the satisfaction rate as a percentage (%). The second is described as the unit of time for AET.

Table 1: Experimental results for some case studies.

No.	BPMN Process	Characteristics		PTS		Correctness	AET (s)
		Tasks	Gateways	States	Transitions		
1	Shipment [25]	8	2 ⊗ + 2 ○ + 2 ⊕	18	38	(a) 93%	0.65
						(b) 100%	1.38
2	Shipment [25]	8	2 ⊗ + 2 ○ + 2 ⊕	18	38	(a) 47%	0.68
						(b) 100%	2.23
3	Shopping [22]	22	8 ⊗ + 2 ○ + 2 ⊕	59	127	(a) 93%	0.94
						(b) 100%	1.98
4	Shopping [22]	22	8 ⊗ + 2 ○ + 2 ⊕	59	127	(a) 54%	0.97
						(b) 100%	3.76
5	AccoutOpening [22]	15	3 ⊗ + 2 ○ + 2 ⊕	20	33	(a) 89%	0.56
						(b) 100%	1.58
6	Online-Shop [22]	19	7 ⊗ + 2 ○	36	74	(a) 96%	1.98
						(b) 100%	4.52
7	Multi-Inclusives [22]	8	6 ○	141	1201	(a) 85%	3.42
						(b) 100%	11.44
8	Booking [22]	11	2 ⊗ + 4 ○	53	252	(a) 88%	2.42
						(b) 100%	6.17

Table 1 first shows that without enforcement techniques, the resulting correctness results present a satisfaction rate below 100%, whereas this rate is systematically of 100% when enforcement is used. As for AET, the execution time is longer when using enforcement techniques. The time increases when the percentage of satisfaction of the property decreases. For instance, examples 1 and 2 use the same process but different properties. The percentage of property violations of example 1 is lower than example 2; therefore, the latter takes more time when using enforcement because it takes more time for the process instances to complete. Similar results can be observed for examples 3 and 4. Although the enforcement mechanism increases the execution time of the process, it systematically ensures that the process executes while preserving the given property.

5 Related Work

In this section, we first compare with existing works on probabilistic verification of business processes, and then we focus on enforcement techniques.

The approaches proposed in [5,6] deal with Bayesian networks to infer the relationship between different events. As an example, the authors in [6] introduce a BPMN normal form based on Activity Theory that can be used for representing the dynamics of a collective human activity from the perspective of a subject. This workflow is then transformed into a Causal Bayesian Network that can be used for modelling human behaviours and assessing human decisions. In [18, 19], the authors present a framework for modelling and analysing business workflows. These workflows are described with a subset of BPMN extended with probabilistic nondeterministic branching and general-purpose reward annotations. An algorithm translates such models into Markov Decision Processes (MDP) written in the syntax of the PRISM model checker. This enables quantitative analysis of business processes for properties such as transient/steady-state probabilities, reward-based properties, and best- and worst-case scenarios. These properties are verified using the PRISM model checker. This work supports design time analysis but does not focus on the dynamic execution and runtime verification of processes. The approach in [8] extends BPMN with time and probabilities. Specifically, the authors expect that a probability value is provided for each flow involved in an inclusive or exclusive split gateway. These BPMN processes are then transformed to rewriting logic and analysed using the Maude statistical model checker PVeStA. The authors in [15] propose to compute probabilities from execution traces of executable BPMN and apply probabilistic model checking techniques at runtime to analyse a given property. In this work, we also rely on PMC, but we go beyond the analysis of BPMN processes, because when the property is not satisfied, we apply techniques for enforcing the satisfaction of the property.

As far as runtime enforcement is concerned, existing techniques usually rely on common techniques including buffering, reordering, healing and discarding actions or events [1, 4, 12, 14]. Buffering rely on storing events that violate certain property in a buffer, which helps delaying their execution. Reordering was

used in several works for favouring or delaying the execution of some actions. Healing is a technique that enforces properties by repairing or inserting new events to ensure compliance. Suppression of events ensures property enforcement by discarding specific events. In the context of BPMN processes, removing specific tasks or artificially adding other tasks is meaningless due to the overall goal of the running processes, explaining why we made use of reordering and buffering techniques only. The authors of [11, 13] focus on developing runtime enforcement techniques for timed properties, without targetting any specific application area. In [7], the authors study runtime monitoring and enforcement of first-order LTL properties over data evolution using an automata-based technique. Their approach is based on the construction of a first-order automaton that is able to perform the monitoring incrementally and by using exponential space in the size of the property. This theoretical work does not focus on BPMN probabilistic processes, nor on probabilistic properties.

6 Conclusion

In this paper, we have proposed a probabilistic execution enforcement mechanism for BPMN processes at runtime. The BPMN process is first transformed into an LTS model. This model is periodically annotated with the execution probability of each transition in the LTS, resulting in a PTS model. This step is achieved by supervising the multiple executions of the BPMN process and extracting the corresponding execution traces. When new instances are triggered, new tasks are waiting to be executed. We check whether the execution of these tasks will not violate the given probabilistic property. If it is the case, the enforcement techniques are activated by either buffering or reordering tasks in order to avoid the violation of the property. All the steps of the approach are automated by a toolchain consisting of tools we implemented or reused. Experiments show the correctness of the approach, which preserves the truthfulness of the property, and a slight overhead in terms of performance, which comes from the time needed to apply enforcement techniques.

The two main perspectives of this work are as follows. The first one is to extend the PRE mechanism in order to minimise the frequency of verifications by considering the PMC results. The second future work focuses on applying PMC results to dynamically adjust the resource allocation necessary for efficient process execution.

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

References

1. Aceto, L., Cassar, I., Francalanza, A., Ingólfsdóttir, A.: On Runtime Enforcement via Suppressions. In: 29th International Conference on Concurrency Theory (CONCUR 2018). pp. 34:1–34:17. https://doi.org/10.4230/LIPIcs.CONCUR.2018.34

2. Activiti: Open source business automation (accessed December 2021), https://www.activiti.org/
3. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. Lectures on Runtime Verification: Introductory and Advanced Topics pp. 1–33 (2018). https://doi.org/10.1007/978-3-319-75632-5_1
4. Basin, D., Klaedtke, F., Zălinescu, E.: Runtime Verification over Out-of-Order Streams. ACM Trans. Comput. Logic **21**(1) (oct 2019). https://doi.org/10.1145/3355609
5. Ceballos, H.G., Cantu, F.J.: Discovering causal relations in semantically-annotated probabilistic business process diagrams. In: Global Conference on Artificial Intelligence, GCAI. pp. 29–40 (2018). https://doi.org/10.29007/nd7r
6. Ceballos, H.G., Flores-Solorio, V., Garcia, J.P.: A probabilistic BPMN normal form to model and advise human activities. In: International Workshop on Engineering Multi-Agent Systems. pp. 51–69. Springer (2015). https://doi.org/10.1007/978-3-319-26184-3_4
7. De Masellis, R., Su, J.: Runtime enforcement of first-order LTL properties on data-aware business processes. In: Service-Oriented Computing: 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings 11. pp. 54–68. Springer (2013). https://doi.org/10.1007/978-3-642-45005-1_5
8. Durán, F., Rocha, C., Salaün, G.: Stochastic analysis of BPMN with time in rewriting logic. Science of Computer Programming **168**, 1–17 (2018). https://doi.org/10.1016/j.scico.2018.08.007
9. Emerson, E., Jutla, C.S., Sistla, A.: On model checking for the mu-calculus and its fragments. Theoretical Computer Science **258**(1), 491–522 (2001). https://doi.org/10.1016/S0304-3975(00)00034-7
10. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. Engineering dependable software systems pp. 141–175 (2013). https://doi.org/10.3233/978-1-61499-207-3-141
11. Falcone, Y., Jéron, T., Marchand, H., Pinisetty, S.: Runtime enforcement of regular timed properties by suppressing and delaying events. Sci. Comput. Program. **123**, 2–41 (2016). https://doi.org/10.1016/j.scico.2016.02.008
12. Falcone, Y., Mounier, L., Fernandez, J.C., Richier, J.L.: Runtime enforcement monitors: composition, synthesis, and enforcement abilities. Formal Methods in System Design **38**, 223–262 (2011). https://doi.org/10.1007/s10703-011-0114-4
13. Falcone, Y., Pinisetty, S.: On the Runtime Enforcement of Timed Properties. In: Proceedings of the Runtime Verification 2019 conference, pp. 48–69. Springer (Oct 2019). https://doi.org/10.1007/978-3-030-32079-9_4
14. Falcone, Y., Salaün, G.: Runtime Enforcement with Reordering, Healing, and Suppression. In: SEFM 2021 - 19th IEEE International Conference on Software Engineering and Formal Methods. pp. 1–20. IEEE, Virtual, United Kingdom (Dec 2021). https://doi.org/10.1007/978-3-030-92124-8_3
15. Falcone, Y., Salaün, G., Zuo, A.: Probabilistic Model Checking of BPMN Processes at Runtime. In: iFM 2022 - International Conference on integrated Formal Methods. pp. 1–17. Lugano, Switzerland (Jun 2022). https://doi.org/10.1007/978-3-031-07727-2_11
16. Faqrizal, I., Salaün, G.: Counting Bugs in Behavioural Models using Counterexample Analysis. In: FormaliSE 2022 - International Conference on Formal Methods in Software Engineering. pp. 1–11. Pittsburgh, United States (May 2022). https://doi.org/10.1145/3524482.3527647

17. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. Int. J. Softw. Tools Technol. Transf. **15**(2), 89–107 (2013). https://doi.org/10.1007/s10009-012-0244-z
18. Herbert, L., Sharp, R.: Precise quantitative analysis of probabilistic business process model and notation workflows. Journal of Computing and Information Science in Engineering **13**(1), 011007 (2013). https://doi.org/10.1115/1.4023362
19. Herbert, L.T., Sharp, R.: Quantitative analysis of probabilistic BPMN workflows. In: International Design Engineering Technical Conferences and Computers and Information in Engineering Conference. vol. 45011, pp. 509–518. American Society of Mechanical Engineers (2012). https://doi.org/10.1115/DETC2012-70653
20. ISO/IEC: International standard 19510, information technology – business process model and notation. (2013)
21. Krishna, A., Poizat, P., Salaün, G.: VBPMN: Automated Verification of BPMN Processes. In: 13th International Conference on integrated Formal Methods (iFM 2017). Turin, Italy (Sep 2017). https://doi.org/10.1007/978-3-319-66845-1_21
22. Krishna, A., Poizat, P., Salaün, G.: Checking Business Process Evolution. Science of Computer Programming **170**, 1–26 (Jan 2019). https://doi.org/10.1016/j.scico.2018.09.007
23. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Information and Computation **94**(1), 1–28 (1991). https://doi.org/10.1016/0890-5401(91)90030-6
24. Mateescu, R., Requeno, J.I.: On-the-Fly Model Checking for Extended Action-Based Probabilistic Operators. International Journal on Software Tools for Technology Transfer **20**(5), 563–587 (Oct 2018). https://doi.org/10.1007/s10009-018-0499-0
25. Mateescu, R., Salaün, G., Ye, L.: Quantifying the Parallelism in BPMN Processes using Model Checking. In: The 17th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE 2014). Lille, France (Jun 2014). https://doi.org/10.1145/2602458.2602473
26. Mateescu, R., Thivolle, D.: A Model Checking Language for Concurrent Value-Passing Systems. In: Cuellar, J., Maibaum, T. (eds.) FM 2008. Lecture Notes in Computer Science, vol. 5014, pp. 148–164. Springer Verlag, Turku, Finland (May 2008). https://doi.org/10.1007/978-3-540-68237-0_12
27. Poizat, P., Salaün, G., Krishna, A.: Checking Business Process Evolution. In: 13th International Conference on Formal Aspects of Component Software (FACS). Besançon, France (Oct 2016). https://doi.org/10.1007/978-3-319-57666-4_4