Probabilistic Model Checking of BPMN Processes at Runtime

Yliès Falcone, Gwen Salaün, and Ahang Zuo

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

Abstract. Business Process Model and Notation (BPMN) is a standard business process modelling language that allows users to describe a set of structured tasks, which results in a service or product. Before running a BPMN process, the user often has no clear idea of the probability of executing some task or specific combination of tasks. This is, however, of prime importance for adjusting resources associated with tasks and thus optimising costs. In this paper, we define an approach to perform probabilistic model checking of BPMN models at runtime. To do so, we first transform the BPMN model into a Labelled Transition System (LTS). Then, by analysing the execution traces obtained when running multiple instances of the process, we can compute the probability of executing each transition in the LTS model, and thus generate a Probabilistic Transition System (PTS). Finally, we perform probabilistic model checking for verifying that the PTS model satisfies a given probabilistic property. This verification loop is applied periodically to update the results according to the execution of the process instances. All these steps are implemented in a tool chain, which was applied successfully to several realistic BPMN processes.

1 Introduction

A business process describes a set of structured tasks that follow a specific order and thus results in a product or service. The business process model and notation (BPMN), proposed by OMG, is the de facto standard for developing business processes [15]. BPMN relies on a graphical workflow-based notation that describes the structured tasks in a business process and the relationships between these tasks.

The BPMN standard was quickly adopted by industry and academia, even though several flaws were identified. One of them regards the lack of formal semantics. Several approaches proposed to use Petri nets or automata-based languages for filling this gap. Related to formal semantics, the lack of formal analysis techniques appeared as another weakness. The final goal is to provide (ideally automated) verification techniques and tools for ensuring that processes respect some functional and non-functional properties of interest (e.g. the absence of deadlocks, the execution of the process within a reasonable amount of time, the occupancy of resources, etc.). All these checks are particularly useful for optimising processes and thus reducing the costs associated with their execution.

In this paper, we tackle the problem of computing the probability of executing certain tasks or combination of tasks when running the processes. The possibility of executing one task or another comes from the use of different kinds of gateways in the BPMN process (e.g. exclusive gateways). These probabilities are difficult to determine, especially when multiple instances of the process are executed at the same time. In that case, since resources are necessary for executing some specific tasks, knowing these probabilities is of prime importance for better adjusting the corresponding resources and thus converging to an optimal allocation of resources. It is worth noting that before executing the process multiple times, the developer has often no clear idea regarding the probability of executing some task or a specific sequence of tasks. Therefore, there is a need for automated techniques that can compute (and update) at runtime these probabilities, thus allowing the verification of probabilistic properties (e.g. what is the probability to execute task T? Is the probability to execute task T1 followed by T2 higher than 40%?).

In this work, we define an approach to perform probabilistic model checking of BPMN processes at runtime. To do so, we assume that a process is described using an executable version of BPMN. The process can be executed multiple times, each execution of the process is called an instance. Different instances may perform different tasks in the process. Our approach first monitors these executions to extract from the corresponding logs the probability of executing each individual task. These probabilities are used to build a semantic model of the BPMN process where these probabilities appear explicitly. This model is called a Probabilistic Transition System (PTS). Then, given a probabilistic property expressed in a dedicated temporal logic and this PTS, a probabilistic model checker is called for verifying whether the property is true/false or for computing the expected probability of that property. Note that this approach is not applied once and for all, because more instances of the process can keep executing including variations in terms of frequency of the executed tasks. Based on these variations, the probability of each transition of the LTS evolves over time. Therefore, the PTS is updated periodically, and the model checker is called again. The result of our approach is thus not a single value, but a dynamic curve indicating the evolution of the property evaluation over time.

To summarise, the main contributions of this work are as follows:

- Monitoring techniques for extracting at runtime relevant information about the execution of multiple instances of a process.
- Periodic computation of a Probabilistic Transition System by analysing execution logs resulting of the monitoring of the process.
- Integrated toolbox for probabilistic model checking of BPMN processes at runtime.
- Validation of our approach on a large set of realistic BPMN processes.

The remainder of this paper is organised as follows. In Section 2, we describe the concepts and definitions used in the subsequent sections. In Section 3, we present the approach in detail. Section 4 focuses on the tool support and the experiments performed for validation purposes. Section 5 describes related work. Finally, in Section 6, we present our conclusions and future work.

2 Models

In this section, we introduce the preliminary concepts.

BPMN. Business process model and notation (BPMN) is a workflow-based notation for describing business processes [15]. Originally, it was a modelling notation, but recent frameworks also allow the execution of such processes using a process automation engine or by translation to an executable language. The syntax of a BPMN process is given by a graph-based structure where vertices (or nodes) correspond to events, tasks and gateways, and edges (or flows) connect these nodes.

Figure 1 describes a fragment of the BPMN notation showing the main elements. Events include the initial/start event and the end event, which are used to initialise and terminate processes. We assume there is only one start event, and at least one end event. A task is an atomic activity containing only one incoming flow and one outgoing flow. Gateways are used to describe the control flow of the process. There are two patterns for each type of gateway: 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.

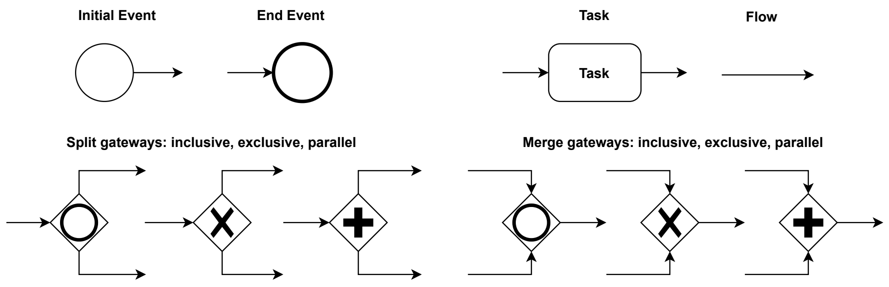

Initial Event
End Event
Task
Flow
Task
Split gateways: inclusive, exclusive, parallel
Merge gateways: inclusive, exclusive, parallel

Fig. 1. Excerpt of the BPMN notation

In this paper, we consider multiple executions of a single process. Each execution is called an *instance* and is characterised by an identifier and the list of consecutive tasks executed by this process. We assume that a BPMN process cannot run infinitely and that each instance terminates at some point. Therefore, the list of tasks associated to an instance is always finite.

LTS. We use Labelled Transition Systems as a semantic model of BPMN processes, as described in [20, 23, 17].

Definition 1 (LTS). *A labelled transition system (LTS) is a tuple* $\langle Q, \Sigma, q_{\text{init}}, \Delta\rangle$ *where:* Q *is a set of states;* Σ *is a finite set of labels/actions;* $q_{\text{init}} \in Q$ *is the initial state;* $\Delta \subseteq Q \times \Sigma \times Q$ *is the transition relation.*

A transition $(q, a, q') \in \Delta$, written $q \xrightarrow{a} q'$, means that the system can move from state q to state q' by performing action a.

PTS. We also need a more expressive model than LTS because we want to associate transitions with probabilities. We therefore rely on Probabilistic Transition Systems [18], which is a probabilistic extension of the LTS model.

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.*

$P(s \overset{a}{\to} s') \in [0, 1]$ 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 to its outgoing transitions is equal to 1, that is $\forall s \in S : \sum_{s' \in S} P(s, a, s') = 1$.

MCL. Model Checking Language (MCL) [21] is an action-based branching-time temporal logic suitable for expressing properties of concurrent systems. MCL is an extension of alternation-free μ-calculus [6] with regular expressions, data-based constructs, and fairness operators. We rely on MCL for describing probabilistic properties, using the following construct [19]: prob R is op [?] E end prob, where R is a regular formula on transition sequences, op is a comparison operator among "<","≤",">","≥", "=", "<>", and E is a real number corresponding to a probability. MCL is interpreted over a PTS model.

3 Probabilistic Model Checking of BPMN

This section first gives an overview of the different steps of our approach. Then, we present with more details the solution for monitoring BPMN processes and the computation of a probabilistic model from the execution traces observed during the monitoring step.

3.1 Overview

Recall that before executing a process, it is unclear how often a certain task or combination of tasks are executed. This is of prime importance for adjusting the resources necessary for executing the tasks involved in a process. The goal of our approach is to analyse the multiple instances of a process at runtime to precisely measure the probabilities of executing the tasks involved in a process, and thus to evaluate automatically probabilistic properties on that process.

Our approach takes as input a BPMN process and a probabilistic property, and returns as output the verdict returned by the model checker. The verdict indicates whether the property holds on the system. Such a verdict is obtained by passing the process and the property to a model checker. This verdict is periodically updated, since the process keeps on executing, and our approach runs as long as there are new instances of the process completing. Figure 2 overviews the approach. First, we monitor and analyse the multiple instances resulting of the execution of the BPMN process. These instances are used to compute the probability of execution for each task. Then, these probabilities are added to the LTS semantic model obtained from the BPMN process, resulting in a PTS. Finally, we call a model checker to verify that the PTS satisfies the given probabilistic property. Since the process keeps running, the probability of each task and thus the PTS are periodically updated. The period is a parameter of the approach. Every time the PTS is updated, the model checker is called again. Let us now give a little more details on the three main parts of the approach.

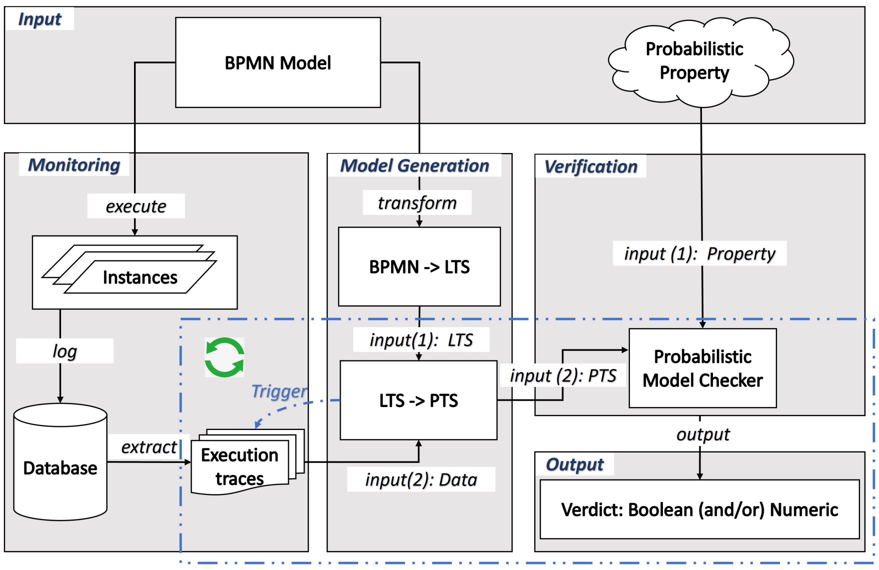


Fig. 2. Overview of the approach

Monitoring. The monitoring part focuses on the data streams generated by the execution of the BPMN process. A BPMN process may be executed multiple times, each of its executions produces an instance. Each time a new instance completes (meaning that the process has terminated), the information about that instance execution is stored into a database. We have implemented a technique for extracting from this database the events related to a set of instances that have completed, and we convert these events into execution traces (one trace per process instance). This extraction is applied periodically, where the period can be a duration (e.g. every hour) or a fixed number of completed instances (e.g. when 100 instances have completed).

Model Generation. The first step of this part is to generate an LTS from the BPMN process. This LTS stands as a semantic model, and exhibits all possible execution paths for the given process. There are several ways to transform BPMN to LTS. Here, we rely on an existing work, which proposes to first transform BPMN into the LNT (LOTOS New Technology) [10] process algebra. Since LNT operational semantics maps to LTS, the generation of that LTS is thus straightforward. Due to lack of space, the reader interested in more details regarding the transformation from BPMN to LTS can refer to [20, 23, 17]. Note that this transformation from BPMN to LTS is only computed once. In a second step, by analysing the execution traces built during the monitoring stage, we compute the probabilities of executing each task involved in the process, and add these probabilities to the LTS, which thus becomes a PTS. This PTS is updated periodically, every time a new set of execution traces is provided by the monitoring techniques.

Verification. This step of the approach takes as input a probabilistic model (PTS) and a probabilistic property, and computes as output a Boolean or numerical verdict depending on the property. This check is performed by using an existing model checker (the latest version of the CADP model checker [9] in this work). Since the PTS is updated periodically, the model checker is thus called whenever this update takes place. Therefore, the final result does not consist of a single value, but all successive values are gathered on a curve, which is dynamically updated every time the model checker is called with a new PTS.

3.2 BPMN Process Monitoring

In this section, we introduce monitoring techniques for BPMN processes at runtime. These techniques are useful because a process is usually not executed only once. Instead, a process can be executed multiple times. Each execution of the process is called an instance. An instance of the process can be in one of the following states: *initial* means that the instance is ready to start (one token in the start event), *running* means that the instance is currently executing and is not yet completed, *completed* means that all tokens have reached end events. Tokens are used to define the behaviour of a process. When a process instance is triggered, a token is added to the start node. The tokens move through nodes and flows of the process. When a token meets a split gateway (e.g. parallel gateway), it may be divided into multiple ones, depending on the type of split gateway. On the contrary, when multiple tokens meet a merge gateway (e.g. inclusive gateway), they are merged into a single token depending on the type of merge gateway. An identifier is used to characterise a specific instance, and this identifier is associated to all nodes (e.g. tasks) executed by this instance.

Monitoring techniques (see Fig. 3 for an overview) aim at analysing the information stored in a database, and extracting for each instance the corresponding execution trace. An execution trace corresponds to a list of tasks executed by this specific instance. The order of execution of these tasks is established by using timestamps at which each task is executed. These timestamps are computed by the process execution engine (Activiti [1] in this work), which relies on a global clock. The execution trace corresponding to a specific instance can be computed only when the instance is in its *completed* state.

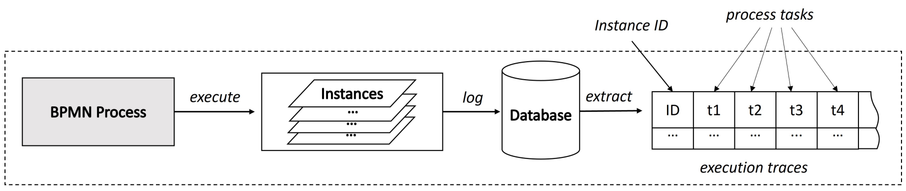


Fig. 3. Runtime monitoring of multiple executions of a BPMN process

Since new instances can execute at any time, we should extract execution traces periodically. There are several possible strategies that can be followed by taking into account different criteria. In this work, we propose to use one of the two following strategies:

– the time-based strategy means that the trace extraction is performed every fixed period of time;
– the instance-based strategy is based on the number of instances, and the trace extraction is triggered whenever the total number of new completed instances reaches a certain value.

It is worth noting that a hybrid strategy combining these two strategies is also an option, e.g. we extract traces whenever 100 instances have completed or every hour if after one hour less than 100 instances have completed. In addition, the choice of these different strategies may have a different impact on the actual results.

There are two similar algorithms for extracting execution traces depending on the strategy. We illustrate below with the algorithm relying on the time-based strategy. The first goal of this algorithm is to extract the relevant completed instances of this process from the database. These instances are then traversed in order to generate the corresponding execution traces.

Let us now go through the algorithm to give more details. Algorithm 1 describes the execution of the time-based extraction of execution traces. The inputs of the algorithm are the process identifier (pid), a timestamp (ts), and a time duration (td). This timestamp is the start time of the period to identify the new instances that have completed. The output is a set of execution traces ($\mathcal{T}$).

Algorithm 1 Algorithm for extracting execution traces

```
Inputs: A process ID ($pid$), a timestamp ($ts$), and a time duration ($td$)
Output: A set of execution traces ($\mathcal{T}$)
1: $\mathcal{I} := \emptyset$, $\mathcal{T} := \emptyset$
2: $\mathcal{I} := getInstances(pid)$
3: for each $I \in \mathcal{I}$ do
4:     if $I.hasEndEvent()$ and $ts < I.endts() \leq ts + td$ then
5:         $\mathcal{T} := \mathcal{T} \cup I.computeSortedTasks()$
   return $\mathcal{T}$
```

Algorithm 1 first connects to the database and retrieves all the instances corresponding to the process identifier by using function *getInstances*(). These instances are stored in variable $\mathcal{I}$. Each instance consists of the identifier of the instance and a set of tasks (lines 1 to 2). These instances are traversed to keep only those that have completed during the last period of time (presence of an end event and completion time lower than timestamp + duration, line 4). The resulting instances are all eligible instances. For each completed instance, function *computeSortedTasks*() sorts the tasks using their completion times, and returns an execution trace consisting of the instance identifier and an ordered list of tasks (line 5). The algorithm finally returns the set of execution traces $\mathcal{T}$.

The time complexity of the algorithm is $\mathcal{O}(n \times m \times \log m)$, where n is the number of completed instances over a period, and m is the maximum number of tasks executed by an instance ($\mathcal{O}(m \times \log m)$ is the complexity of the timsort algorithm used for sorting tasks).

3.3 Transforming LTS into PTS

Given a BPMN process, we can generate its LTS semantic model using existing techniques such as [20, 23, 17]. The LTS model exhibits all possible execution paths of the input BPMN process. This generated model is non-deterministic, and it has only one final state[1]. In this section, we show how by analysing execution traces (one trace per instance) extracted during the monitoring of the process, we can extend this LTS with probabilities of execution for each transition included in this LTS model. These probabilities are added as annotations to the transitions of the LTS, which thus becomes a PTS.

Before explaining how we generate a PTS given an LTS and a set of execution traces, it is worth noting that, similarly to trace extraction, the PTS should be

[1] A final state is a state without outgoing transitions. If an LTS exhibits several final states, these states can be merged into a single one, resulting into an LTS strongly bisimilar [22] to the original one.

updated periodically as well due to the execution of multiple instances. Therefore, this part of the approach also relies on one of the two aforementioned strategies (time or instance-based strategy) for defining the period.

Algorithm 2 takes as input the LTS model corresponding to the BPMN process and a set of execution traces, and returns as output a PTS model. The main idea of the algorithm is to count the number of times each transition is executed using the information from the execution traces. This is achieved by associating a counter to each transition and by traversing the execution traces one after the other. Essentially, each time a task appears in an execution trace, we increment the counter of the corresponding transition. After traversing all execution traces, we compute the probability of executing each transition outgoing from a state by using the associated counter value. We augment the LTS model with these probabilities to obtain the PTS model.

Algorithm 2 Algorithm for transforming LTS into PTS

```
Inputs: LTS = $\langle Q, \Sigma, q_{\text{init}}, \Delta \rangle$, a finite set of execution traces $\mathcal{T} = \langle T_1, T_2, \ldots, T_n \rangle$
Output: PTS = $\langle S, A, s_{\text{init}}, \delta, P \rangle$
 1: $S := Q, A := \Sigma, s_{\text{init}} := q_{\text{init}}, \delta := \Delta$
 2: $Path := [\,], Fpaths := [\,], Bpaths := [\,], T_{\text{tasks}} := [\,]$   /* [ ] indicates an empty list */
 3: for each $(s, a, s') \in \Delta$ do $cnt((s, a, s')) := 0$
 4: for each $T \in \mathcal{T}$ do
 5:     $\mathcal{Q}_{\text{current}} := \{q_{\text{init}}\}$, $\mathcal{Q}_{\text{next}} := \emptyset$, $\mathcal{Q}_{\text{pre}} := \emptyset$, $T_{\text{tasks}} := T.getTasks()$
 6:     for each $task \in T_{\text{tasks}}$ do
 7:         $q_{\text{succ}} := \{q' \in Q \mid \exists q \in \mathcal{Q}_{\text{current}}, (q, task, q') \in \Delta\}$
 8:         if $task \neq T_{\text{tasks}}[T_{\text{tasks}}.length() - 1]$ then
 9:             $\mathcal{Q}_{\text{next}} := q_{\text{succ}}$, $\mathcal{Q}_{\text{current}} := \mathcal{Q}_{\text{next}}$
10:         else
11:             $q_{\text{next}} := \{q \in Q \mid \exists q \in q_{\text{succ}} \text{ and } q \neq q', (q, task, q') \in \Delta\}$
12:             $\mathcal{Q}_{\text{next}} := q_{\text{succ}} \setminus q_{\text{next}}$
13:         for each $(q, task, q') \in \Delta, q \in \mathcal{Q}_{\text{current}}, q' \in \mathcal{Q}_{\text{next}}$ do
14:             $Fpaths.append((q, task, q'))$
15:     for each $task \in T_{\text{tasks}}.reverseOrder()$ do
16:         $\mathcal{Q}_{\text{pre}} := \{q \in Q \mid \exists q' \in \mathcal{Q}_{\text{next}}, (q, task, q') \in \Delta\}$
17:         for each $(q, task, q') \in \Delta, q \in \mathcal{Q}_{\text{pre}}, q' \in \mathcal{Q}_{\text{next}}$ do
18:             $Bpaths.append((q, task, q'))$
19:         $\mathcal{Q}_{\text{next}} := \mathcal{Q}_{\text{pre}}$
20:     $Path := Fpaths \cap Bpaths$
21:     for each $(s, a, s') \in Path$ do $cnt((s, a, s')) := cnt((s, a, s')) + 1$
22: $P := \{(s, a, s') \mapsto cnt((s, a, s'))/\sum_{q \in S, a' \in A, (s, a', q) \in \delta} cnt((s, a', q)) \mid (s, a, s') \in \delta\}$
    return $\langle S, A, s_{\text{init}}, \delta, P \rangle$
```

Let us now present with more details how this algorithm for generating the PTS model works. The PTS model is first initialised, and a counter (initialised to 0) is added to each transition of the LTS model (lines 1 to 3). The algorithm starts by traversing the set of execution traces $\mathcal{T}$. For each execution trace, the algorithm proceeds in three steps: (a) traversing the tasks of the execution trace, (b) finding the corresponding valid path into the LTS model, (c) increasing the value of the counters. As a final step, all execution traces are traversed for computing the probability of each transition. $\mathcal{Q}_{\text{current}}$ is the set of current states in the LTS during the traversal, $\mathcal{Q}_{\text{next}}$ is the set of successor states of a current state, and $\mathcal{Q}_{\text{pre}}$ is the set of predecessor states of a current state. We now present these steps with more details:

(a) *Traversing the tasks of the execution trace* (lines 5 to 14). Since the LTS may exhibit non-deterministic behaviours, this step (and the following one) computes the valid path in the LTS corresponding to an execution trace. This step relies on a forward traversal of the LTS (from initial state to final state). Each execution trace T consists of an identifier and a sequence of tasks T_{tasks}. For each trace, these tasks are handled one after the other, and by using transitions Δ, the successor states for each current state are obtained until all tasks of the current execution trace have been traversed. We use *Fpaths* (Forward-paths) to record the sequence of transitions in the LTS corresponding to the execution paths of the current execution trace.

(b) *Finding the corresponding valid path into the LTS model* (lines 15 to 20). This step relies on a backward traversal of the LTS (from final state to initial state). Therefore, we start by reversing the sequence of tasks for the current execution trace. By using this reversed list and the final state which is stored in the last $\mathcal{Q}_{\text{next}}$ of the previous step, we then traverse backwards to the initial state. We use *Bpaths* (Backward-paths) to record all the transitions from the final state to the initial state (lines 15 to 19). Next, we take the intersection of each element in *Fpaths* and *Bpaths*, and store the result in *Path*. This intersection operation eliminates the invalid paths, or more precisely the invalid transitions, in *Fpaths* and *Bpaths*. Thus, the *Path* variable finally stores all the transitions of the LTS model corresponding to the current execution trace (line 20).

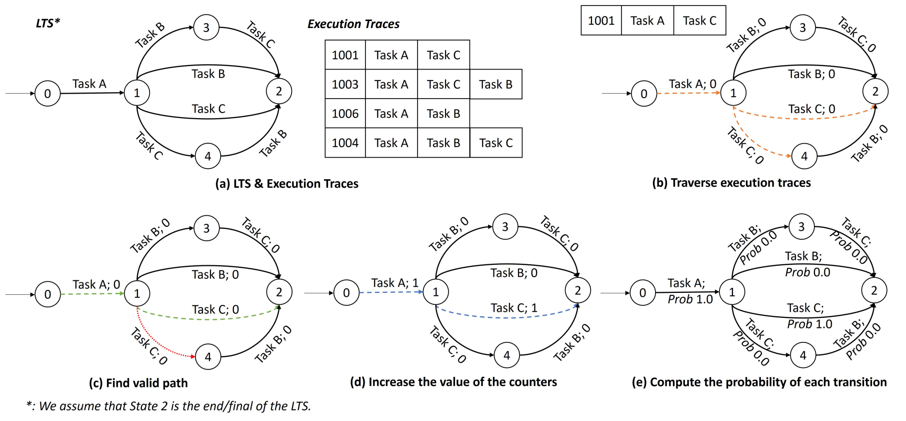
(c) *Increasing the value of the counters* (line 21). The values of the counters for the transitions in *Path* are increased by 1.

(d) *Computing the probability of each transition* (line 22). The probability of each transition is computed. To do so, the value of each transition counter is divided by the total number of transitions with the same starting state.

When we have traversed all the execution traces, the algorithm finally returns the resulting PTS.

The time complexity of this algorithm is $\mathcal{O}(|\mathcal{T}| \times n \times |\Delta|)$, where $|\mathcal{T}|$ is the number of execution traces, n is the number of tasks in the longest trace, and $|\Delta|$ is the number of transitions in the LTS/PTS.

Figure 4 illustrates the execution of the algorithm. Figure 4(a) depicts the input of the algorithm: an LTS and a set of execution traces, where the number in the first column (e.g. 1003) is the identifier of the execution trace. In this example, we assume that State 2 is the end/final state of the LTS. Figure 4(b) depicts an example of traversing an execution trace, where the dashed lines


LTS*
Execution Traces
(a) LTS & Execution Traces
(b) Traverse execution traces
(c) Find valid path
(d) Increase the value of the counters
(e) Compute the probability of each transition
*: We assume that State 2 is the end/final of the LTS.
Task A; 0
Task B; 0
Task C; 0
Task A; 1
Task C; 1
Task A; Prob 1.0
Task B; Prob 0.0
Task C; Prob 1.0
Task C; Prob 0.0
Task B; Prob 0.0

1001	Task A	Task C	
1003	Task A	Task C	Task B
1006	Task A	Task B	
1004	Task A	Task B	Task C

1001	Task A	Task C

Fig. 4. Example describing the execution of the algorithm

indicate all possible transitions. Figure 4(c) depicts an example of filtering the invalid paths in it based on the paths obtained in the previous step, which is indicated by dotted lines. Dashed lines are used to represent the valid path. In this example, after the previous step, we get a total of two paths. One path contains two transitions of ($0 \xrightarrow{\text{Task\_A}} 1$) and ($1 \xrightarrow{\text{Task\_C}} 4$). The final state reached by this path is 4, which is not the final state of the LTS. Therefore, this path is invalid. For the other one, its final state is the final state of the LTS, and hence, this is a valid path. Figure 4(d) shows each relevant transition coming from the valid path (dashed lines) whose counter is incremented by 1. Figure 4(e) then describes the computation of the probability for each transition of the LTS. Finally, the PTS corresponding to the LTS extended with probabilities is returned.

4 Tool Support

In this section, we present the tool chain automating the different steps of our approach. We then illustrate the application of these tools to a case study, and end with additional experiments to evaluate performance of the tools on a set of realistic examples.

4.1 Tool

Figure 5 overviews the tool chain. First, we use the Activiti framework [1] for developing and executing BPMN processes. Activiti is a lightweight Java-centric open source tool. When running a BPMN process once or several times, all data related to these executions are stored into a MySQL database.

Beyond a BPMN process, the second input required by our approach is a probabilistic property. In this work, the property is specified using the MCL [21] temporal logic, which is one of the input languages of the CADP toolbox [9]. CADP is a toolbox for the design and verification of asynchronous concurrent systems. Note that the approach can take several properties as input, not just a single one. We also use the Script Verification Language [8] (SVL), which is convenient for automating all verification tasks, particularly when there are several properties given as input.

The VBPMN tool [17] is used for transforming BPMN into LTS. The generation of the PTS from the analysis of the execution traces is automated by a Python program we implemented. The property is then evaluated by calling the CADP probabilistic model checker [19]. As a result, it returns either a Boolean or Numerical value. Since the BPMN process keeps executing (multiple instances), the PTS is updated periodically according to an update strategy defined in Section 3. Whenever the PTS is updated, the model checker is called again. The final result is thus not a single value, but a series of values, which we represent as a curve (x: time or number of instances, y: verification result). This curve is drawn using Matplotlib, which is a plotting library for the Python programming language.

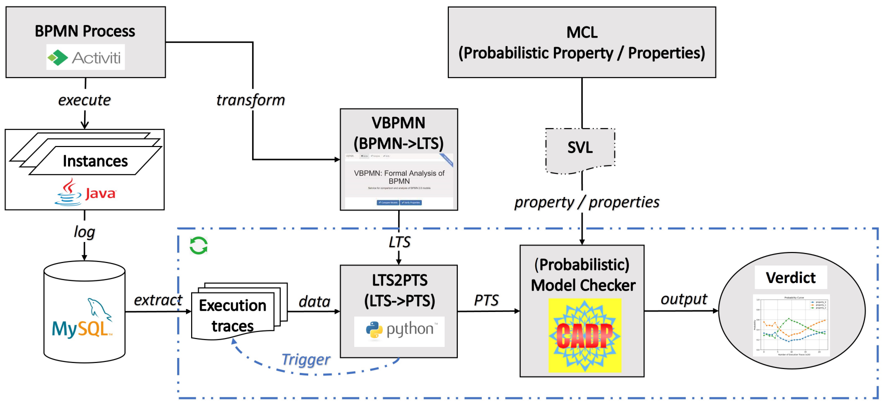


Fig. 5. Overview of the tool chain

4.2 Case Study

Let us illustrate our approach with the shipment process of a hardware retailer, which comes from [20]. This example, shown in Figure 6, describes a realistic delivery process of goods. More precisely, this process starts when there are goods to be shipped (E1). Two tasks are then processed in parallel (PG1), one is the packaging of the goods (T7) and the other decides whether the goods require normal or special shipment (T1). Depending on that decision, a first option checks whether there is a need for additional insurance (T2), followed by the possibility to buy additional insurance (T4) and/or fill in a post label (T5). A second option is to request quotes from carriers (T3), followed by the assignment of a carrier and preparation of the paperwork (T6). Before completing the whole process, the package is moved to a pick-up area.

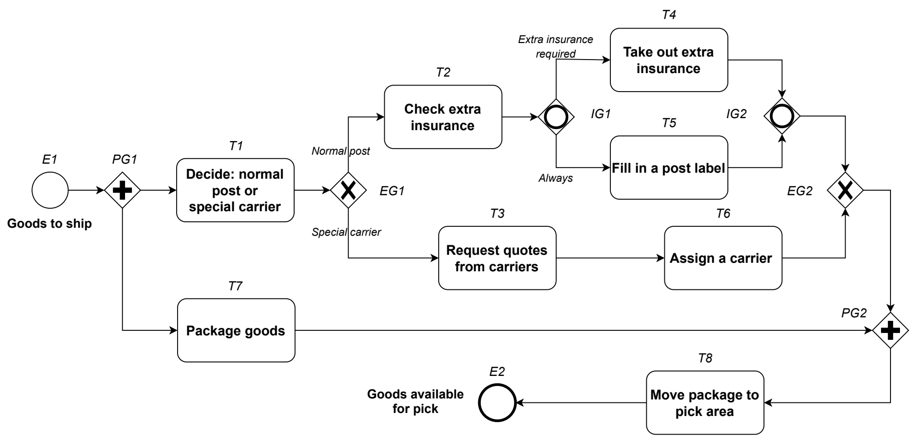


Fig. 6. BPMN shipment process of a hardware retailer

Probabilistic Property. For illustration purposes, we choose a property checking that the probability of executing task T4 after task T2 is less than 0.4. This is important because the choice of taking extra insurance (T4) comes with a cost, and if this decision is taken too often (e.g. more than four times out of ten), this may become a problem in terms of budget. This property is written in MCL as follows: `prob true*. T2. true*. T4 is < ? 0.4 end prob`. Since we use the '?' symbol, the model checker returns both a Boolean value (indicating whether the property is true or false) and a numerical value (indicating the probability to execute T4 after T2).

Simulation. We implemented a simulator in Java in charge of executing many instances of the BPMN process, varying the order and frequency of task executions in order to simulate a realistic operating environment. Figure 7 shows the Boolean and numerical results for a simulation consisting of 1400 instances, executed over a period of about 4 minutes, where the property is the one mentioned earlier. The update strategy used here relies on the number of completed instances. Every time there are ten completed instances, we compute again the execution probability of each transition of the LTS, generate a new PTS, and we call the model checker to obtain a new result. Figure 7 shows a variation of the truth and numerical values of the evaluated property over time. This is due

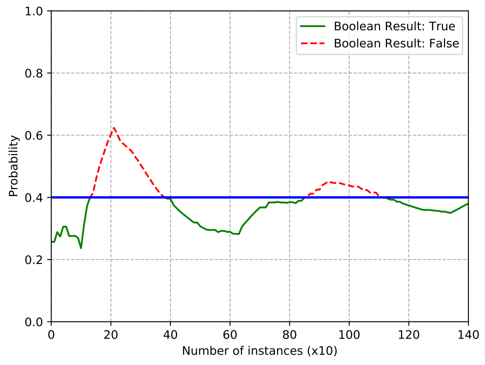


Fig. 7. Simulation results for the shipment process

to our simulator, which favours the execution of some specific tasks during some periods, resulting in the curve given in the figure.

4.3 Additional Experiments for Performance Evaluation

The goal of this section is to measure the execution times of the different steps of our approach in practice. To do so, we rely on a set of realistic BPMN processes found in existing papers and frameworks shown in Table 1. We used an Ubuntu OS laptop running on a 1.7 GHz Intel Core i5 processor with 8 GB of memory to carry out these experiments. Table 1 shows the results of these experiments. Each process is characterised by its size (number of tasks and gateways), the size of the generated PTS (number of states and transitions), and the execution time of each step is decomposed as follows:

(1) Time for transforming the BPMN process into an LTS (executed only once);
(2) Time for extracting a certain number of execution traces (100 in these experiments) from the database;
(3) Time for analysing these execution traces and for computing the PTS;
(4) Time for verifying the property on that PTS using the CADP model checker.

Let us now focus on the results presented in Table 1 for each step. The first step focuses on the transformation of the BPMN process to an LTS model. Table 1 shows this can be a time-consuming step compared to the other ones. This computation time depends on the structure of the BPMN process and increases with the number of parallel and inclusive gateways (in particular when they are nested). Rows 8 and 11 in the table illustrate this point. However, it is worth noting that this step of the approach is only executed once at the beginning, so this extra-time is not really a problem. The second step focuses on the computation time for connecting to the database and extracting a certain number of execution traces (100 in these experiments) from it. We can see that

Table 1. Experimental results for some case studies

No.	BPMN Process	Characteristics				PTS		Time (s)			
		Tasks	Gateways			States	Transitions	(1)	(2)	(3)	(4)
			Exclusive	Inclusive	Parallel						
1	Shipment [20]	8	2	2	2	18	38	15	0.32	0.61	1.45
2	Recruitment [7]	10	1	0	6	19	31	25	0.21	0.64	1.32
3	Shopping [17]	22	8	2	2	59	127	50	0.26	1.12	1.68
4	AccoutOpeningV2 [17]	15	3	2	2	20	33	31	0.25	0.71	0.84
5	Publish [17]	4	0	2	0	16	61	22	0.42	0.58	0.77
6	Car [17]	10	2	0	2	18	31	18	0.44	0.67	0.84
7	Online-Shop [17]	19	7	2	0	36	74	41	0.33	0.79	1.21
8	Multi-Inclusive [17]	9	0	6	0	47	194	42	0.23	2.32	1.39
9	Multi-Exclusive [17]	8	6	0	0	6	9	22	0.01	0.51	0.82
10	Multi-Parallel [17]	8	0	0	6	15	22	12	0.12	0.63	0.67
11	Multi-InclusiveV2 [17]	12	0	6	0	141	1201	78	0.25	4.29	1.58
12	Booking [17]	11	2	4	0	53	242	22	0.19	2.37	1.24

the computation time of this step is less than 0.5 seconds for all the examples in the table. The third step aims to analyse these execution traces, calculating the probabilities of each transition for building a PTS by annotating the previously computed LTS. The algorithm (and its complexity) for computing that PTS was presented in Section 3. According to our experiments, we can see that this time increases with the size of the LTS (number of transitions). It takes less than a second to compute this step for most examples and it is slightly longer for a few examples (about 4 seconds for example Multi-InclusiveV2 for instance). The final computation time corresponds to the verification of the PTS model by calling the probabilistic model checker. We can see in the table that it takes about 1 or 2 seconds for each example to make this computation.

To conclude, these experiments show that, except for the LTS computation that might be costly, the other steps of the approach are computed in a reasonable time for realistic processes. This shows that conducting probabilistic model checking of BPMN processes at runtime is feasible. Last but not least, the sum of the times observed for steps (2), (3) and (4) could be used to obtain a lower bound value to the period of time used by the time-based strategy. Indeed, this would not make sense to use as period a value that would be smaller than this lower bound.

5 Related Work

In this section, we overview some existing research efforts proposing probabilistic models and analysis for BPMN.

The approaches in [3, 2] focus on the use of Bayesian networks to infer the relationship between different events. As an example, [3] introduces 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.

The approach in [4] 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. This work is extended in [5] to explicitly model and analyse resource allocation. This series of works allows one to compute at design time generic properties, such as average execution times, synchronisation times or resource usage, whereas the goal of this paper is to compute probabilistic properties at runtime by dynamically analysing the executions of multiple process instances.

The approach in [14] presents a framework for the automated restructuring of workflows that allows minimising the impact of errors on a production workflow. To do so, they rely on a subset of BPMN extended to include the tracking of real-valued quantities associated with the process (such as time, cost, temperature), and the modelling of probabilistic- or non-deterministic branching behaviour, and the introduction of error states. The main goal of this approach is to reduce the risk of production faults and restructure the production workflows for minimising such faults.

In [23, 16], the authors first propose to give a formal semantics to BPMN via a transformation to Labelled Transition Systems (LTSs). This is achieved via a transformation to process algebra and use of existing compilers for automatically generating the LTS from the process algebraic specification. Once the LTS model is generated, model checking of functional properties is possible as well as comparison of processes using equivalence checking. This work does not provide any probabilistic model for BPMN nor any kind of quantitative analysis.

In [13, 12], the authors present a framework for modelling and analysis of 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.

Statistical model checking [11], which uses simulation and statistical methods, facilitates the generation of approximate results to quantitative model checking. Although it has a low memory requirement, the cost is expensive if high accuracy is required. In comparison, probabilistic model checking produces highly accurate results, despite the potential problem of state explosion.

6 Conclusion

We have presented a new approach that allows BPMN analysts to automatically carry out probabilistic model checking of BPMN processes at runtime. This approach takes as input an executable BPMN process and one (or several) probabilistic property. To evaluate this property, we build a probabilistic model (PTS) by analysing the execution traces extracted from the multiple execution of this process. This analysis allows us to annotate the LTS semantic model corresponding to the BPMN process with probabilities, thus obtaining a PTS model. Finally, we can call the probabilistic model checker with the probabilistic model and the property. Since the process keeps executing, the probabilistic model is updated periodically and the model checker is called periodically as well. Therefore, we do not return a single value as a result but a curve displaying the successive truth or numerical values returned by the model checker. Our approach is fully automated by a tool chain consisting of existing and new tools. The tool chain was applied to several realistic examples for validation purposes.

As far as future work is concerned, we first plan to take advantage of the results computed by our approach to effectively adjust resource allocation depending on the runtime analysis results. This requires having an explicit description of resources associated with tasks and dynamically modifying the resource allocation with respect to the analysis results. A second perspective is to not only analyse properties at runtime, but predict the result of the evaluation of these properties in the near future. This would allow the anticipation of changes in the resource allocation for instance. This prediction can be achieved by relying on the computed probabilistic model or by using machine learning techniques.

Acknowledgements. This work was supported by the Région Auvergne-Rhône-Alpes within the "*Pack Ambition Recherche*" programme, the H2020-ECSEL-2018-IA call – Grant Agreement number 826276 (CPS4EU), the French ANR project ANR-20-CE39-0009 (SEVERITAS), and LabEx PERSYVAL- Lab (ANR-11-LABX-0025-01).

References

1. Activiti: Open source business automation (accessed December 2021), https://www.activiti.org/
2. 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)
3. 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)
4. Durán, F., Rocha, C., Salaün, G.: Stochastic analysis of BPMN with time in rewriting logic. Science of Computer Programming 168, 1–17 (2018)
5. Durán, F., Rocha, C., Salaün, G.: Analysis of resource allocation of BPMN processes. In: International Conference on Service-Oriented Computing. pp. 452–457. Springer (2019)

6. Emerson, E.A., Lei, C.L.: Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In: LICS (1986)
7. Falcone, Y., Salaün, G., Zuo, A.: Semi-automated Modelling of Optimized BPMN Processes. In: Proc. of SCC'21. pp. 425–430. IEEE (2021)
8. Garavel, H., Lang, F.: SVL: A scripting language for compositional verification. In: Kim, M., Chin, B., Kang, S., Lee, D. (eds.) Proc. of FORTE'01. IFIP Conference Proceedings, vol. 197, pp. 377–394. Kluwer (2001)
9. 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)
10. Garavel, H., Lang, F., Serwe, W.: From LOTOS to LNT. In: Katoen, J.P., Langerak, R., Rensink, A. (eds.) ModelEd, TestEd, TrustEd, Lecture Notes in Computer Science, vol. 10500, pp. 3–26. Springer (Oct 2017)
11. Herault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate probabilistic model checking. In: Proc. 5th International Conference on Verification, Model Checking and Abstract Interpretation. vol. 2937, pp. 73–84 (01 2004)
12. 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)
13. 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)
14. Herbert, L.T., Hansen, Z.N.L., Jacobsen, P.: Automated evolutionary restructuring of workflows to minimise errors via stochastic model checking. In: Probabilistic Safety Assessment and Management conference 2014 (2014)
15. ISO/IEC: International standard 19510, information technology – business process model andnotation. (2013)
16. Krishna, A., Poizat, P., Salaün, G.: VBPMN: automated verification of BPMN processes (tool paper). In: International Conference on Integrated Formal Methods. pp. 323–331. Springer (2017)
17. Krishna, A., Poizat, P., Salaün, G.: Checking business process evolution. Sci. Comput. Program. 170, 1–26 (2019)
18. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Information and computation 94(1), 1–28 (1991)
19. 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)
20. 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)
21. Mateescu, R., Thivolle, D.: A model checking language for concurrent value-passing systems. In: International Symposium on Formal Methods, FM (2008)
22. Milner, R.: Communication and Concurrency. Prentice Hall (1989)
23. Poizat, P., Salaün, G., Krishna, A.: Checking business process evolution. In: International Conference on Formal Aspects of Component Software, FACS'16. LNCS, vol. 10231, pp. 36–53 (2016)