

Verifying Timed BPMN Processes Using Maude

Francisco Durán, Gwen Salaün

▶ **To cite this version:**

Francisco Durán, Gwen Salaün. Verifying Timed BPMN Processes Using Maude. 19th International Conference on Coordination Languages and Models (COORDINATION), Jun 2017, Neuchâtel, Switzerland. pp.219-236, 10.1007/978-3-319-59746-1_12 . hal-01538104

HAL Id: hal-01538104
https://inria.hal.science/hal-01538104

Submitted on 13 Jun 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Verifying Timed BPMN Processes using Maude

Francisco Durán[1] and Gwen Salaün[2]

[1] University of Málaga, Spain
[2] University of Grenoble Alpes, LIG, CNRS, France

Abstract. A business process is a collection of structured activities producing a particular product or software. BPMN is a workflow-based graphical notation for specifying business processes. Formally analyzing such processes is a crucial challenge in order to avoid erroneous executions of the corresponding software. In this paper, we focus on timed business processes where execution time can be associated to several BPMN constructs. We propose an encoding of timed business processes into the Maude language, which allows one to automatically verify several properties of interest on processes such as the maximum/minimum/average execution time or the timed degree of parallelism that provides a valuable guide for the problem of resource allocation. The analysis is achieved using the rewriting-based tools available in Maude, which also provides other techniques (*e.g.*, reachability analysis and model checking) for verifying BPMN specifications. We applied our approach on a large set of BPMN processes for evaluation purposes.

1 Introduction

Business Process Model and Notation (BPMN) [14] is a graphical modelling language for specifying business processes. A business process is a collection of structured activities or tasks that produce a specific product and fulfill a specific organizational goal for a customer or market. More precisely, a process aims at modelling activities, their causal and temporal relationships, and specific business rules that process executions have to comply with. Business process modelling is an important area in software engineering since it supports the development of workflow-based software, such as information and distributed systems. BPMN is the *de facto* notation for designing business processes and was published as an ISO standard in 2013.

When modelling processes using BPMN, many questions arise: is my workflow precisely modelling what I expect from it? Is my workflow free of errors and bugs? Are certain properties of interest preserved? What is the degree of parallelism of my process? What is the minimum execution time of my workflow? All these questions are meaningful, but they are not that simple to answer, particularly when modelling complex processes involving many tasks and intricate combinations of gateways. Some of these questions (and corresponding computations) may even turn out to be undecidable if the whole expressiveness of BPMN is considered (*e.g.*, cyclic behaviours, data aspects, or time).

In this paper, we focus on software development based on a subset of BPMN where we can model process behaviours (tasks, sequence flows, gateways) and time aspects (duration associated to tasks and flows). We propose automated analysis techniques for verifying that certain properties of interest are satisfied for timed business processes modelled with BPMN. In this work, we focus on properties that are application independent, which allows us to provide press-button verification techniques without requiring any input from the developer. Properties of interest are for instance the minimum/maximum/average execution time of a process and the timed degree of parallelism, which is a valuable information for resource allocation. Our approach also enables one to carry on other kinds of analysis such as reachability analysis to search, *e.g.*, for deadlock states, or state-based LTL model checking to verify the satisfaction of temporal properties (safety and liveness). In these cases, since the properties depend on the input process, they have to be provided by the developer, who can reuse well-known patterns for timed properties as those presented in [15, 12].

Our approach relies on an encoding of the BPMN execution semantics into the rewriting-logic-based language Maude [7]. The three challenges of this encoding were to properly translate all gateways (including the inclusive merge gateway), to describe time durations and passing, and to support loops and unbalanced workflows. Unbalanced workflows are those processes that exhibit an unbalanced structure with no exact correspondence between split and merge gateways. The expressive power of the Maude language allowed us to model these features in a uniform way. Moreover, Maude is equipped with a large variety of analysis tools, which can be used for automatically verifying properties of interest such as the aforementioned execution time measures and the timed degree of parallelism. We applied our approach on many business processes for validation purposes and verification times turn out to be reasonable for real-size examples.

To sum up, the main contributions of this work with respect to existing results on this topic are the following: (i) an encoding into Maude of a subset of BPMN including time aspects, inclusive gateways, looping behaviours, and unbalanced workflows; (ii) automated analysis techniques for verifying properties of interest on timed BPMN models using reachability analysis and model checking tools; and (iii) tool support for automating the transformation to Maude and validation of the approach by application to many BPMN processes.

The organization of the rest of the paper is as follows. Section 2 introduces the BPMN notation and Maude. Section 3 explains our Maude encoding of the considered subset of BPMN, with emphasis on the handling of time. In Section 4, we present our techniques for automatically analyzing properties on BPMN processes. This section also presents experimental results. Section 5 surveys related work and Section 6 concludes the paper.

2 Preliminaries

2.1 BPMN

BPMN is a graphical notation for modelling business processes as collections of related tasks that produce specific services or products for particular clients. BPMN is an ISO/IEC standard [14], and can be executed by using different process interpretation engines (*e.g.*, Activiti, Bonita BPM, or jBPM). The semantics of BPMN is described informally in official documents [23, 14], and some attempts have been made for giving a formal semantics to BPMN (see, *e.g.*, [9, 29, 24, 20, 17]).

In this paper, our goal is not to consider the whole expressiveness of the BPMN language, but to concentrate on the BPMN elements related to control-flow modelling and on time aspects that can be represented in BPMN constructs. This enables us to focus on those aspects and show how automated analysis is possible for them. Specifically, we consider the node types *event*, *task*, and *gateway*, and the edge type *sequence flow*. Start and end events are used, respectively, to initialize and terminate processes. A task represents an atomic activity that has exactly one incoming and one outgoing flow. A gateway is used to control the divergence and convergence of the execution flow. A sequence flow describes two nodes executed one after the other, *i.e.*, imposing the execution order.

Gateways are crucial since they are used to model control flow branching in BPMN and therefore influence the overall process execution. There are five types of gateways in BPMN: *exclusive*, *inclusive*, *parallel*, *event-based* and *complex* gateways. We consider all of them except complex gateways, because they are used to model complex synchronization behaviours especially based on data control, and we do not take data objects, nor conditions on flows outgoing of split gateways, into account.

Gateways with one incoming branch and multiple outgoing branches are called *splits*, *e.g.,* split inclusive gateway. Gateways with one outgoing branch and multiple incoming branches are called *merges*, *e.g.,* merge parallel gateway. An exclusive gateway chooses one out of a set of mutually exclusive alternative incoming or outgoing branches. For an inclusive gateway, any number of branches among all its incoming or outgoing branches may be taken. A parallel gateway creates concurrent flows for all its outgoing branches or synchronizes concurrent flows for all its incoming branches. For an event-based gateway, it takes one of its outgoing branches or accepts one of its incoming branches based on events. In the following, we call the branches that are taken by a gateway during an execution as *active* branches. In this work, we support unbalanced workflows, meaning that each merge gateway does not necessarily have a corresponding split gateway with an exact correspondence of the branches among outgoing and incoming flows. We also support workflows with looping behaviours.

The execution semantics of BPMN constructs can be described using tokens as depicted in Figure 1. The start event node can be triggered once at any moment, which creates a token in its outgoing sequence flow. Whenever a token is present in the incoming sequence flow of an end event, this execution flow

can terminate by consuming this token. If a token is in a sequence flow, then the destination node for this sequence flow can be triggered. The semantics of gateways is also given, emphasizing that specific care should be taken when considering inclusive split/merge gateways since all possible combinations should be generated and all triggered branches should be awaited for at the merge synchronization point. The inclusive merge is particularly problematic from a semantic point of view as discussed in [6]. We will show in the next section how our Maude executable semantics allows the encoding of these BPMN gateways.

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

In this paper, we assume that BPMN processes are syntactically correct. This can be enforced using existing works, *e.g.*, [11], or using a BPMN engine, *e.g.*, the Activiti BPM platform, Bonita BPM, or the Eclipse BPMN Designer.

Running example. The process we use as running example (Figure 2) aims at monitoring the organization of a business trip. The process starts by reserving flight tickets and by completing the mission paperwork. Flight booking may take some time, because in many companies this task is subcontracted to a third party company. Once the flight tickets are issued, accommodation reservation and other additional services (insurance, vaccines, etc.) are tackled in parallel. Visa process is initiated only when all reservations (flights and hotel) are completed and when the paperwork is finished. Once all the aforementioned prerequisites of the trip are completed, the mission details are stored in a specific database.

2.2 Maude and Real-Time Maude

Real-Time Maude [22] is a rewriting-logic-based specification language and formal analysis tool that extends the Maude system [7] to support the formal specification and analysis of *real-time systems*. Real-Time Maude provides support for symbolic simulation through timed rewriting, and time-bounded temporal logic model checking and search for reachability analysis.

Rewriting logic [18] is a logic of change that can naturally deal with states and non-deterministic concurrent computations. A rewrite logic theory is a tuple $(\Sigma, E \cup A, R)$, where $(\Sigma, E \cup A)$ is a *membership equational logic* [3] theory with Σ its signature, E a set of conditional equations, A a set of equational axioms such as associativity, commutativity and identity, so that rewriting is performed *modulo* A, and R is a set of labeled conditional rules. In rewriting logic, a distributed system is axiomatized by an equational theory, describing its set of states as an algebraic data type, and a collection of conditional rewrite rules, specifying its dynamics. Rewrite rules are written $\mathsf{crl}\ [l] : t \Rightarrow t'\ \mathsf{if}\ C$, with l the rule label, t and t' terms, and C a condition. We may have rules without label

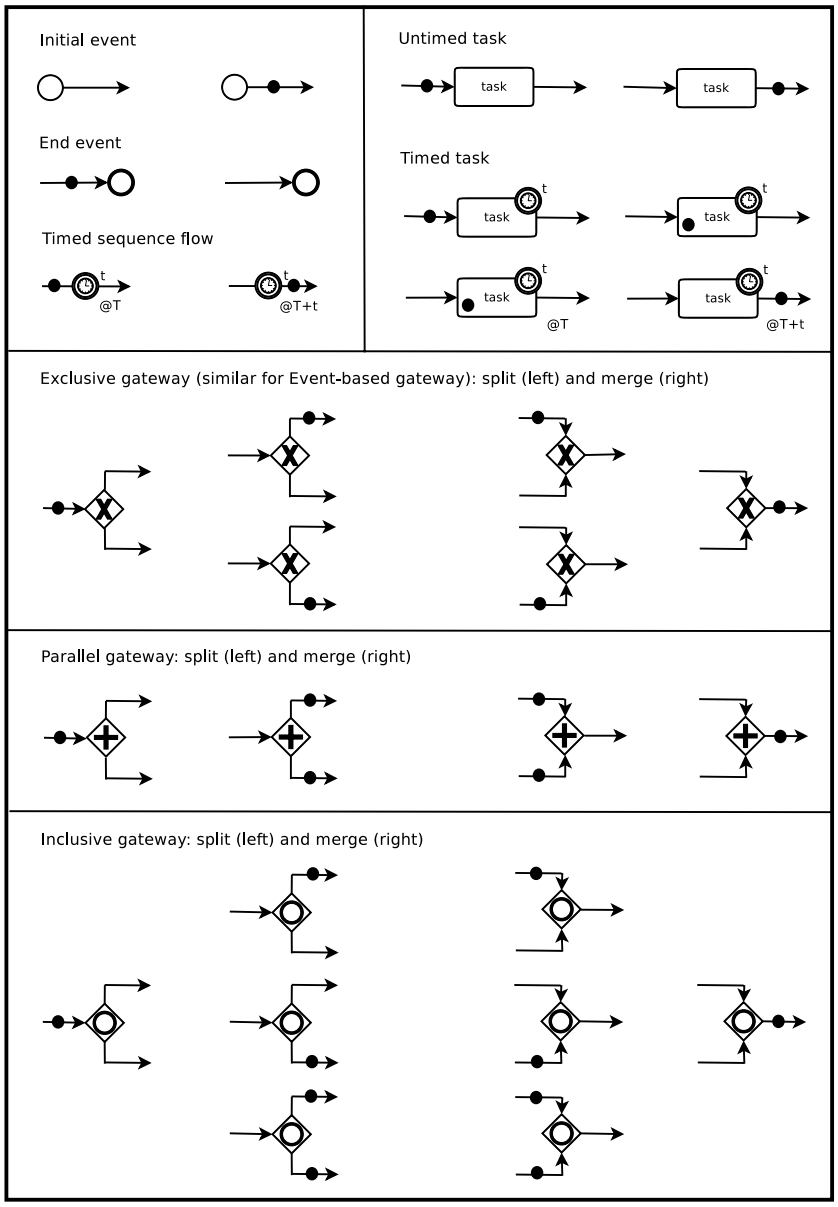


Fig. 1. BPMN Execution Semantics

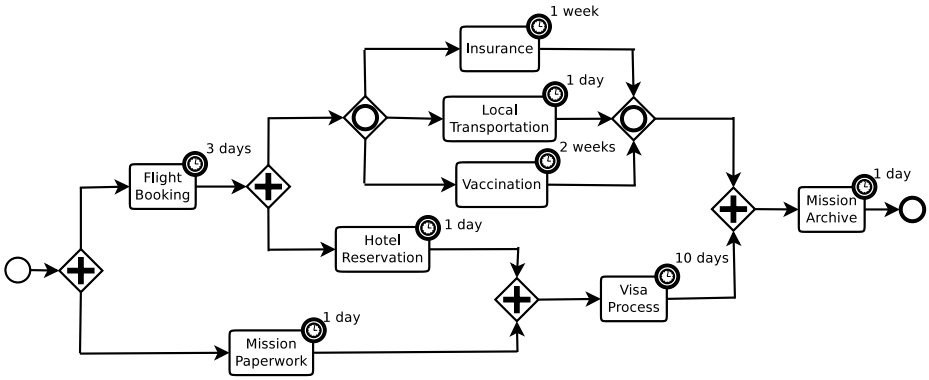


Fig. 2. BPMN Running Example

or condition. An unlabelled unconditional rule would be written `rl` t `=>` t'. Rules describe the local, concurrent transitions that are possible in the system, *i.e.*, when a part of the system state fits the pattern t, then it can be replaced by the corresponding instantiation of t'. The guard C acts as a blocking precondition, in the sense that a conditional rule can only be fired if its condition is satisfied.

In Maude, object-oriented systems are specified by object-oriented modules in which classes and subclasses are declared. A class is declared with the syntax `class` C `|` a_1`:`S_1`,` ... `,` a_n`:`S_n, where C is the name of the class, a_i are attribute identifiers, and S_i are the sorts of the corresponding attributes. Objects of a class C are then record-like structures of the form `<` O `:` C `|` a_1`:`v_1`,` ... `,` a_n`:`v_n `>`, where O is the name of the object, and v_i are the current values of its attributes.

In a concurrent object-oriented system, the concurrent state has the structure of a multiset made up of objects and messages. Such state evolves by concurrent rewriting using rules that describe the effects of the communication events. The general form of such rewrite rules is:

```
crl [r] :
  < O1 : C1 | atts1 > ... < On : Cn | attsn >
  M1 ... Mm
  =>
  < Oi1 : C'i1 | atts'i1 > ... < Oik : C'ik | atts'ik >
  < Q1 : C''1 | atts''1 > ... < Qp : C''p | atts''p >
  M'1 ... M'q
  if Cond .
```

where r is the rule label, $M_1...M_m$ and $M'_1...M'_q$ are messages, $O_1...O_n$ and $Q_1...Q_p$ are object identifiers, $C_1...C_n$, $C'_{i_1}...C'_{i_k}$ and $C''_1...C''_p$ are classes, $i_1...i_k$ is a subset of $1...n$, and $Cond$ is a condition (the rule's *guard*). The result of applying such a rule is that: (a) messages $M_1...M_m$ disappear, *i.e.*, they are consumed; (b) the state, and possibly the classes of objects $O_{i_1}...O_{i_k}$ may change; (c) all the other objects O_j vanish; (d) new objects $Q_1...Q_p$ are created; and (e) new messages $M'_1...M'_q$ are created, *i.e.*, they are posted.

Real-Time Maude provides a sort Time to model the time domain, which can be either discrete or dense time (we use discrete time in this paper). Given a constructor {_,_} of sort GlobalSystem, time passing is modelled by rewrite rules known as *tick rules*:

```
crl [l] : {t, T} => {t', T + τ} if C .
```

where t is the state of the system, T is its global time, and τ is the *duration* of the rewrite. Since tick rules affect the global time, in Real-Time Maude time elapse is usually modeled by one single tick rule, and the system dynamic behaviour by instantaneous transitions [22]. Although there are other strategies, the most flexible one models time elapse by using two functions, namely, delta, which defines the effect of time elapse over every model element, and mte (maximal time elapse), which defines the maximum amount of time that can elapse before any action is performed.

```
crl [tick] : {t, T} => {delta(t), T + τ} if τ := mte(t) /\ τ > 0 /\ C .
```

In Maude, rule conditions may be given as a short-circuited conjunction of conditional terms using operator /\. In the previous rule, Boolean expressions and assignments are used as conjuncts (see [7] for further details).

3 Encoding into Maude

In this section, we present our encoding of our subset of BPMN into Maude. This encoding consists of two parts: the syntactic encoding of a BPMN process into Maude and the set of rewrite rules encoding the BPMN execution semantics. The encoding of the BPMN process into Maude (Section 3.1) depends on the example, so the corresponding Maude code has to be generated for each new process. This transformation is fully automated by applying a Python script we implemented. The rewrite rules have been encoded once and for all, and we will present in Section 3.2 the rules corresponding to the handling of some of the constructs in our BPMN processes. The complete Maude specification with all the rules and examples of BPMN processes is available online [1].

3.1 Process Encoding

We represent a BPMN process as a set of flows and a set of nodes. A flow is represented as a term flow(sf$_i$, t), with sf$_i$ an identifier and t a duration. If there is no duration associated to a flow, the duration value is zero. We distinguish different kinds of nodes: start, end, task, split, and merge. A start (end, resp.) node consists of an identifier and an output (input, resp.) flow identifier. A task node involves an identifier, a task description, two flow identifiers (input and output), and a duration (zero if no duration is associated to this task). A split node includes a node identifier, a gateway type (exclusive, parallel, inclusive, or event-based), an input flow identifier, and a set of output flow identifiers. A merge node includes a node identifier, a gateway type, a set of input flow identifiers, and an output flow identifier.

```
eq fls = ( flow(sf1, 0), flow(sf2, 0), ... ) .                    --- flows

eq nds = ( start(initial, sf1),                                   --- nodes
           end(final, sf13),
           split(g1, parallel, sf1, (sf2, sf14)),
           ...
           task(t7, "transportation", sf6, sf9, 1),
           task(t8, "vaccination", sf7, sf10, 14) ) .

eq initSystem
  = < p : Process | nodes : nds, flows : fls >
    < s : Simulation | tokens : token(initial, 0), gtime : 0 > .
```

Fig. 3. Running Example in Maude

These constructs are illustrated in Figure 3, which shows an excerpt of the representation of the running example. Constants fls and nds represent, respectively, its set of flows and its set of nodes.

3.2 Execution Semantics

The execution of BPMN activities is modeled using tokens, which are associated to tasks and flows, and circulate along activities as the execution evolves. For instance, split gateways produce tokens for their outgoing flows, and merge gateways collect tokens from their incoming flows and produce one single token. This simple approach allows us to support unbalanced workflows where there is no strict correspondence between splits and merges, as well as looping behaviours.

The execution semantics of BPMN is defined using Maude rewrite rules, which operate on systems composed of a process object and a *simulation* object. The process object represents the BPMN process, and it does not change. The simulation object keeps information on the execution of the process.

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

A simulation involves a set of tokens and a global time (gtime) described using a natural number (discrete time). Tokens are used to represent the evolution of the workflow during its execution. These tokens correspond to flow or task identifiers, plus a time that express a delay, used to model duration of flows and tasks. Thus, a token token(t8, t) indicates that the task t8 has a token, and that such task will be completed in t time units. The operator initSystem in Figure 3 represents the initial state for the process introduced in Figure 2.

Tick rule. A *tick* rule is necessary to simulate the time evolution, which is modelled by the increase of the global time and the decrease of the tokens' timers. Given appropriate definitions of functions mte and delta, the tick rule is written as in Figure 4. The delta function is straightforward, since it just increments the global time present in the simulation object of the indicated amount of time and decrements the timers of the tokens by the same amount. The mte function is more subtle. Although one could think that it is enough taking the smallest of the

```
crl [tick] :
  < PId : Process | nodes : Nodes, Atts >
  < SId : Simulation | tokens : Tks, gtime : T >
  =>
  < PId : Process | nodes : Nodes, Atts >
  < SId : Simulation |
      tokens : delta(Tks, T1),           --- updates all tokens
      gtime : (T + T1) >                 --- increments the global time
  if T1 := mte(Nodes, Tks)
  /\ 0 < T1 .
```

Fig. 4. Tick Rule

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

Fig. 5. Start Event Rule

tokens' delays, notice that parallel and inclusive merges may require additional delays in the incoming branches, requiring a more intricate calculation. A parallel merge is not activated until all its incoming flows are active, meaning that there can be tokens with time zero that have to wait until all these flows get their tokens. The case of inclusive merges is similar, although in this case we do not know beforehand how many tokens are to be expected. Thus, each incoming flow must be traversed backwards to check whether that flow must be awaited for or discarded in the calculation of the mte function.

The semantics we choose for describing time obliges to execute actions and move tokens in the process as soon as possible. The time cannot elapse when there are timers to zero and thus possible actions to be triggered in the process.

Start / end events. We assume that the initial set of tokens includes a token token(initial, 0). Thus, the start rule (Figure 5) is triggered when this token is available (line 6). When the startProc rule is applied, the initial token is consumed and another one is added to the set of current tokens (line 13), which indicates that the flow outgoing from the start event has been activated (FId). The time assigned to this new token is the delay of the flow FId (line 11).

The end event rule is triggered when there is a token for the incoming flow with zero time duration. In that case, the simulation consumes this token without generating new ones, which terminates this flow execution. Note that there is no specific rules for flows. It is enough to have tokens representing flow activations

```
1  rl [execTask] :
2    < PId : Process |
3        nodes : (task(NId, TaskName, FId1, FId2, T), Nodes),
4        flows : Flows >
5    < SId : Simulation |
6        tokens : (token(NId, 0), Tks),    --- token available with 0 time
7        Atts >
8    =>
9    < PId : Process |
10       nodes : (task(NId, TaskName, FId1, FId2, T), Nodes),
11       flows : Flows >
12   < SId : Simulation |              --- new token for outgoing flow FId2
13       tokens : (token(FId2, retrieveTimeFlow(FId2, Flows)), Tks),
14       Atts > .
```

Fig. 6. Task Completion Rule

and the tick rule we have presented before in this section makes the time evolves for these tokens, thus for these flows once they are activated.

Tasks. A task execution is encoded with two rules to express the possibility that a task may take time if a duration is associated to it. An initiation rule activates the task when a token representing the incoming flow is available. In that case, we generate a new token with the task identifier and the task duration. A second rule is used for representing the task completion. This rule is triggered when there is a token for that task with time zero. In that case, this token is consumed and a new one is generated for the outgoing flow (Figure 6).

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

The parallel split gateway rule is triggered when a token corresponding to the input flow is available. If so, the token is consumed and one token is added for each outgoing flow. The merge rule for the parallel gateway (Figure 7) is executed when there is a token for each incoming branch (function allTokensParallel in Figure 7, line 12). In that case, these tokens are removed (function removeTokensParallel, line 11) and a new token is generated for the outgoing flow.

The semantics of inclusive gateways is more intricate [6]. An inclusive split gateway can trigger any number of outgoing flows (at least one). To do so, we generate tokens for a non-deterministic number of outgoing flows to simulate the concurrent execution of those flows. The inclusive merge gateway is one of the most subtle parts of this encoding. This gateway is triggered when *all expected tokens* are available. However, we cannot know beforehand the number of active branches, and therefore, the only way is to traverse the process backwards and look for active branches (available tokens), similar to the procedure described for the mte function. Function allTokensInclusive (line 14, Figure 8) ex-

```
crl [mergeParallelGateway] :
  < PId : Process |
      nodes : (merge(NId, parallel, FIds, FId), Nodes),
      flows : (flow(FId, T), Flows) >
  < SId : Simulation | tokens : Tks, Atts >
  =>
  < PId : Process |
      nodes : (merge(NId, parallel, FIds, FId), Nodes),
      flows : (flow(FId, T), Flows) >
  < SId : Simulation |
      tokens : (token(FId, T), removeTokensParallel(FIds, Tks)), Atts >
  if allTokensParallel(FIds, Tks) .   ---- all incoming flows activated
```

Fig. 7. Parallel Merge Gateway Rule

```
crl [mergeInclusiveGateway] :
  < PId : Process |
      nodes : (merge(NId, inclusive, FIds, FId), Nodes),
      flows : (flow(FId, T), Flows) >
  < SId : Simulation | tokens : Tks, Atts >
  =>
  < PId : Process |
      nodes : (merge(NId, inclusive, FIds, FId), Nodes),
      flows : (flow(FId, T), Flows) >
  < SId : Simulation |
      tokens : (token(FId, T), removeTokensInclusive(FIds, Tks)),
      Atts >
  if atLeastOneToken(FIds, Tks)
  /\ allTokensInclusive(FIds,Tks,(merge(NId,inclusive,FIds,FId),Nodes)).
```

Fig. 8. Inclusive Merge Gateway Rule

plores the process upstream looking for active flows and deduces whether all the expected tokens are present in order to fire the merge gateway or if other tokens must be expected before executing this gateway. To avoid unnecessary computations, this checking is only performed when a token has reached the gateway (atLeastOneToken, line 13, Figure 8). Once this rule is executed, all expected tokens are consumed and a fresh token is added for the outgoing flow.

4 Rewriting-based Verification of Timed Processes

In this section, we successively present the verification of properties on timed processes, other kinds of analysis (simulation, reachability, model checking), and experimental results. It is worth stressing that by using an encoding into an existing framework (Maude here), we can reuse and take advantage of all the existing tools without having to develop new algorithms (*from scratch*) for computing execution times and checking timed properties.

Verification of timed properties. There are several properties of interest to be checked on timed processes. We focused on the minimum/maximum/average execution time and on the degree of parallelism in this work. These metrics are independent of any concrete BPMN process instance, which makes these checks generic and easily reusable.

Given a module M including a BPMN process to analyze and an initial system I (Process and Simulation objects), the function execTime(M, I) generates all solutions (states where an end node has been reached) and computes their minimum, maximum and average execution times. The calculation of these values relies on the search and meta-programming capabilities of Maude. The search takes place following a breadth-first strategy. In order to avoid infinite runs of our system, which may happen when processes include infinite loops, one can either bound the search depth or the global time. By using Maude's facilities, solutions are considered one by one, making the computation more efficient and saving storage space.

As far as the parallelism degree is concerned, for a specific process, we traverse all reachable states (and not only the final solutions) to search the state with the maximum number of tokens, which corresponds to the degree of parallelism.

Simulation and reachability analysis. Simulation is very useful for exploring system executions. In Maude, simulation relies on rewriting, which consists in successively applying equations and rewrite rules on an initial term (a BPMN process here), with the possibility of using some strategy language to guide the execution. Since systems may be rewritten in many different ways, Maude also provides a *search* command, which allows us to explore the reachable state space up to a certain depth. Thus, we can perform analysis on the reachability of states satisfying certain conditions, *e.g.*, when searching for deadlock states or other undesired situations. For example, given our running example in Figure 2, and its corresponding Maude representation InitSystem in Figure 3, the following search command checks that there is no reachable final state with tokens in it, which shows that there is no deadlock.

```
> search InitSystem =>! Conf such that getNumberTokens(Conf) =/= 0 .
No solution.
```

Notice the use of '=>!' to limit the check to final states. Variants of this command allows us to carry on other types of search.

Model checking. We can also take advantage of our encoding for using other analysis tools available in the Maude system. For instance, Maude's Linear Temporal Logic (LTL) explicit-state model checker [10] can be used for analyzing all possible executions of a business process. Maude's model checker allows one to check whether every possible behaviour starting from a given initial state (the start node in BPMN) satisfies a given LTL property. It can be used to check safety and liveness properties of systems when the set of states reachable from an initial state is finite. Full verification of invariants in infinite-state systems can be accomplished by verifying them on finite-state abstractions [19] of the original infinite-state system, that is, on an appropriate quotient of the original system whose set of reachable states is finite. In our context, beyond classic properties such as deadlock-freeness, the properties that can be verified depend on the example and should be specified by the developer, *e.g.*, a certain task is always achieved after another specific task. In order to make the property writing easier, the developer can rely on well-known patterns as those presented in [15, 12] for timed properties.

Table 1. Experimental Results

BPMN	Size					Exploration		Proc. exec. time			Parall.	Analysis
Proc.	Tasks	Flows	✕	+	○	Sol.	States	Min	Max	Avg	Degree	Time
1	8	19	-	4	2	2	138	15	17	16	5	0.07s
2	7	14	2	2	-	2	59	4,837	5,322	5,079	2	0.02s
3	8	17	-	5	-	1	44	3,863	3,863	3,863	4	0.02s
4	8	16	-	-	4	3	127	3,288	5,095	3,913	3	0.05s
5	12	24	-	-	6	6	1,051	2,902	3,900	3,547	6	0.6s
6	20	39	-	-	8	35	8,760	4,529	8,222	6,423	7	10.3s
7	20	43	-	6	6	7	2,653	5,649	7,341	6,453	7	3.7s
8	40	87	14	9	2	24	28,327	7,619	9,235	8,332	7	49.7s
9	40	87	12	9	4	24	55,693	7,619	9,235	8,332	8	1m48s
10	40	87	10	9	6	24	288,025	7,619	9,235	8,332	12	24m20s
11	16	31	-	-	2	13	225,378	1,370	3,024	2,274	13	6m23s
12	213	215	4	6	4	22	5,844	4,189	21,199	17,367	6	16.8s

For instance, given propositions FlightBooking and VisaProcess, which are true in states in which the process is executing these respective tasks, *i.e.*, there is a token in the corresponding task, we can check that the visa request is always processed after a flight booking as follows:

```
> reduce modelCheck(InitSystem, [](FlightBooking -> <> VisaProcess)) .
result Bool: true
```

Experimental evaluation. We made experiments on about 100 examples, some of them taken from the literature on this topic, *e.g.*, [31, 27, 24], or handcrafted for testing some special structures such as multiple nested gateways. Our main goal was to see how our verification approach scales in terms of time and explored state space depending on the size of the input process. We used a Mac OS laptop running on a 2.9 GHz Intel Core i5 processor with 16 GB of memory. We present in Table 1 some of these results. The table gives for each process the number of tasks, the number of sequence flows, and the number of gateways. The exploration is characterized giving the number of solutions and the total number of states. As for verification, for each example, we give the results for process execution times and for the degree of parallelism. The last column shows the analysis time for the parallelism degree calculation, which is the operation that takes longer.

First of all, it is worth noting that we made experiments varying time durations (durations between 0 and 10 units of time, between 0 and 100, between 0 and 1000). This does not impact analysis times because the function mte avoids an execution where the time would elapse unit by unit. Therefore, this function speeds up the time passing whatever maximum time is considered for task and flow duration. Regarding the experimental results presented in Table 1, we set flow and task durations between 0 and 1000 units of time.

Example 1 is the running example. The minimum time (15 days) is obtained when the vaccination task is not executed in the inclusive gateway. When this task is required, we obtain the maximum time (17 days). The degree of parallelism (5) corresponds to the case where both hotel reservation and paperwork tasks are not yet completed, and the three tasks in the inclusive gateway are all triggered in parallel.

The analysis time is short for small and medium size examples, even when there are several nested inclusive or parallel gateways (see examples 5-7 in the table). Example 3 exhibits the same minimum, maximum, and average times because it involves only parallel gateways, and in that case, all behaviours are systematically executed. We made experiments with variants of the same example (rows 8-10) to observe how our approach scales. These examples are quite large, involving 40 tasks and more than 20 gateways (most of them nested). We can see how, by increasing the number of inclusive gateways and reducing the number of exclusive gateways, the analysis time goes from 52 seconds (example 8) to over 24 minutes (example 10). These times are long because the number of states to explore is rather large. Let us emphasize that we may encounter realistic processes larger than those ones in terms of number of tasks, but we have not seen yet a real example with as many nested gateways as in those examples (8-10). Example 12 shows an example with more than 200 tasks. This process mainly exhibits sequential behaviours. In that case, we can see that the number of states is lower and thus the analysis time is quite short (16 seconds).

5 Related Work

Several works focus on providing formal semantics and verification techniques for business processes using Petri nets. [16] proposes to formalize business processes and more specifically composition of Web services using Petri nets. Decker and Weske present in [8] an extension of BPMN 1.0 (iBPMN) in the direction of interaction modelling. They also propose a formal semantics for iBPMN in terms of interaction Petri nets. [9] presents a mapping from BPMN to Petri nets that enables the static analysis of BPMN models. [26] presents a double transformation from BPMN to Petri nets and from Petri nets to mCRL2. This allows one to use both Petri nets based tools and the mCRL2 toolset for searching deadlocks, livelocks, or checking temporal properties. [2] describes how BPMN processes can be represented using the Reo coordination language, which admits formal analysis using model checking and bisimulation techniques. Compared to these results, our encoding also gives a semantics to BPMN by translation to Maude, yet it was not our primary goal. The main difference with respect to these related works is our focus on timed aspects.

Another line of works aimed at using process algebras for formalizing and verifying BPMN processes. The authors of [29] present a formal semantics for BPMN by encoding it into the CSP process algebra. They show in [30] how this semantic model can be used to verify compatibility between business participants in a collaboration. This work was extended in [31] to propose a timed semantics

of BPMN with delays. [21, 5, 20] focus on the semantics formalized in [29, 31] and propose an automated transformation from BPMN to timed CSP, as well as composition verification techniques for checking properties using the FDR2 model checker. In [25], the authors present an encoding of an untimed subset of BPMN into the LNT process algebra for supporting the analysis of process evolution. [24, 13] address the issue of checking whether a BPMN choreography is realizable by computing participant implementations using projection. We go one step farther compared to these related works because we provide verification techniques for a timed version of BPMN including all main gateways (in particular inclusive gateways), loops, and unbalanced structures for workflows.

Another paper [11] attempted to translate BPMN to Maude for verification purposes. In this work, the authors focus on data objects semantics and data-based decision gateways, and provide new mechanisms to avoid structural issues in workflows such as flow divergence. To do so, they introduce the notion of well-formed BPMN processes, which allows one to guarantee structural properties of the workflows. This paper mainly handles syntactic issues and aims at avoiding incorrect syntactic patterns. The main difference compared to our contributions here is that the authors have a specific interest on data-centric workflows whereas we look at behavioural and timed features of processes. One can also take advantage of our Maude encoding and verification framework for checking other timed properties, such as those presented in [4, 28] (sojourn time, synchronization time, waiting time).

[17] proposes a general approach for computing the degree of parallelism of BPMN processes using model checking techniques. To do so, the authors propose a transformation to process algebra and a low-level model for BPMN processes based on Labelled Transition Systems. However, the subset of BPMN considered in [17] makes abstraction of times and durations possibly associated to tasks and flows. [27] focuses on timed aspects and proposes several algorithms for directly calculating the degree of parallelism of a BPMN process. In this work, a duration constraint is associated to each task. They do not consider inclusive gateways and propose different algorithms for special cases of processes, *e.g.*, processes with only one type of gateways or acyclic processes with only parallel gateways. Our work focuses on BPMN processes with time constraints too, but we associate durations not only to tasks but also to flows. In addition, we consider any combination of gateways as well as cyclic processes.

6 Concluding Remarks

BPMN is now widely used by companies for supporting the workflow-based development of their information and management systems. However, we are still far from having press-button analysis techniques integrated in the existing modelling and development BPMN frameworks. This work is a contribution in that direction, that is, to provide automated techniques for analyzing BPMN processes. In this paper, we have focused on a subset of BPMN containing the main behavioural constructs (tasks, sequence flows, gateways) and time aspects

associated to flows and tasks. We have proposed an encoding of this BPMN subset into the input language of the rewriting-based system Maude. Maude was expressive enough for representing unbalanced BPMN processes with moderate effort using a token-based semantics. The whole approach consisting of the translation to Maude and of the verification of several properties of interest on concrete BPMN processes is fully automated. In particular, we have showed how several measures of execution time (minimum, maximum, average) or the degree of parallelism can be computed with Maude. Several other tools can be used, such as simulation, reachability analysis, or LTL model checking. Our encoding and verification approach was validated through experiments we achieved on a significant number of BPMN processes, showing that these checks are completed in a reasonable time for real-size examples.

A first perspective of this work is to extend our approach with other BPMN constructs. We have focused in this paper on the behavioural part of BPMN, which allows us to formally analyze important properties, and we have discarded data aspects. Dataless models are over-approximations of the corresponding processes. This may generate false negative results, that is, our approach may return that a process has a deadlock for example whereas it is not the case because the blocking case actually never occurs. We plan to take data into account and in particular conditions that may be associated to outgoing flows for split gateways. As far as activities are concerned, we would like to support not only tasks but also interactions and message sending/reception. This extension would require to accept the description of distributed systems using BPMN collaboration diagrams. Finally, we intend to extend our time analysis capabilities by considering inter-activity and inter-process temporal constraints (*e.g.*, process deadline, timers, or time conflicts).

References

1. `http://maude.lcc.uma.es/MaudeBPMN/`.
2. F. Arbab, N. Kokash, and S. Meng. Towards Using Reo for Compliance-Aware Business Process Modeling. In *Proc. of ISoLA'08*, volume 17 of *Communications in Computer and Information Science*, pages 108–123. Springer, 2008.
3. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and Proof in Membership Equational Logic. *Theoretical Comput. Sci.*, 236(1):35–132, 2000.
4. R. Bruni, A. Corradini, G. L. Ferrari, T. Flagella, R. Guanciale, and G. Spagnolo. Applying Process Analysis to the Italian eGovernment Enterprise Architecture. In *Proc. of WS-FM'11*, volume 7176 of *LNCS*, pages 111–127. Springer, 2011.
5. M. I. Capel and L. E. M. Morales. Automating the Transformation from BPMN Models to CSP+T Specifications. In *Proc. of SEW'12*, pages 100–109. IEEE Computer Society, 2012.
6. D. R. Christiansen, M. Carbone, and T. T. Hildebrandt. Formal Semantics and Implementation of BPMN 2.0 Inclusive Gateways. In *Proc. of WS-FM'10*, volume 6551 of *LNCS*, pages 146–160. Springer, 2011.
7. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify,*

Program and Verify Systems in Rewriting Logic, volume 4350 of *LNCS*. Springer, 2007.
8. G. Decker and M. Weske. Interaction-centric Modeling of Process Choreographies. *Information Systems*, 36(2):292–312, 2011.
9. R. Dijkman, M. Dumas, and C. Ouyang. Semantics and Analysis of Business Process Models in BPMN. *Inf. Softw. Technol.*, 50(12):1281–1294, 2008.
10. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL Model Checker. In *Proc. of WRLA'02*, volume 71 of *ENTCS*, pages 115–142. Elsevier, 2002.
11. N. El-Saber and A. Boronat. BPMN Formalization and Verification using Maude. In *Proc. of BM-FA'14*, pages 1–8. ACM, 2014.
12. V. Gruhn and R. Laue. Patterns for Timed Property Specifications. *Electr. Notes Theor. Comput. Sci.*, 153(2):117–133, 2006.
13. M. Güdemann, P. Poizat, G. Salaün, and L. Ye. VerChor: A Framework for the Design and Verification of Choreographies. *IEEE Trans. Services Computing*, 9(4):647–660, 2016.
14. ISO/IEC. International Standard 19510, Information technology – Business Process Model and Notation. 2013.
15. S. Konrad and B. H. C. Cheng. Real-time Specification Patterns. In *Proc. of ICSE'05*, pages 372–381. ACM, 2005.
16. A. Martens. Analyzing Web Service Based Business Processes. In *Proc. of FASE'05*, pages 19–33, 2005.
17. R. Mateescu, G. Salaün, and L. Ye. Quantifying the Parallelism in BPMN Processes using Model Checking. In *Proc. of CBSE'14*, pages 159–168. ACM, 2014.
18. J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Comput. Sci.*, 96(1):73–155, 1992.
19. J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational Abstractions. In *Proc. of CADE'03*, volume 2741 of *LNCS*, pages 2–16. Springer, 2003.
20. L. E. M. Morales, M. I. Capel, and M. A. Pérez. Conceptual Framework for Business Processes Compositional Verification. *Information & Software Technology*, 54(2):149–161, 2012.
21. L. E. M. Morales, M. I. C. Tuñón, and M. A. Pérez. A Formalization Proposal of Timed BPMN for Compositional Verification of Business Processes. In *Proc. of ICEIS'10*, volume 73 of *Lecture Notes in Business Information Processing*, pages 388–403. Springer, 2010.
22. P. C. Ölveczky and J. Meseguer. Semantics and Pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.
23. OMG. *Business Process Model and Notation (BPMN) – Version 2.0*. January 2011.
24. P. Poizat and G. Salaün. Checking the Realizability of BPMN 2.0 Choreographies. In *Proc. of SAC'12*, pages 1927–1934. ACM Press, 2012.
25. P. Poizat, G. Salaün, and A. Krishna. Checking Business Process Evolution. In *Proc. of FACS'16*, LNCS. Springer, 2016.
26. I. Raedts, M. Petkovic, Y. S. Usenko, J. M. van der Werf, J. F. Groote, and L. Somers. Transformation of BPMN Models for Behaviour Analysis. In *Proc. of MSVVEIS'07*, pages 126–137, 2007.
27. Y. Sun and J. Su. Computing Degree of Parallelism for BPMN Processes. In *Proc. of ICSOC'11*, volume 7084 of *LNCS*, pages 1–15. Springer, 2011.
28. W. M. P. van der Aalst, A. Adriansyah, and B. F. van Dongen. Replaying History on Process Models for Conformance Checking and Performance Analysis. *Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery*, 2(2):182–192, 2012.

29. P. Wong and J. Gibbons. A Process Semantics for BPMN. In *Proc. of ICFEM'08*, volume 5256 of *LNCS*, pages 355–374. Springer, 2008.
30. P. Wong and J. Gibbons. Verifying Business Process Compatibility. In *Proc. of QSIC'08*, pages 126–131. IEEE, 2008.
31. P. Y. H. Wong and J. Gibbons. A Relative Timed Semantics for BPMN. *Electr. Notes Theor. Comput. Sci.*, 229(2):59–75, 2009.