Symbolic Specification and Verification of Data-aware BPMN Processes using Rewriting Modulo SMT

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

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

Abstract. The *Business Process Model and Notation* (BPMN) is the standard notation for modeling business processes. It relies on a workflow-based language that allows for the modeling of the control-flow graph of an entire process. In this paper, the main focus is on an extension of BPMN with data, which is convenient for describing real-world processes involving complex behavior and data descriptions. By considering this level of expressiveness due to the new features, challenging questions arise regarding the choice of the semantic framework for specifying such an extension of BPMN, as well as how to carry out the symbolic simulation, validation, and assess the correctness of the process models. These issues are addressed first by providing a symbolic executable rewriting logic semantics of BPMN using the rewriting modulo SMT framework, where the execution is driven by rewriting modulo axioms and by querying SMT decision procedures for data conditions. Second, reachability properties, such as deadlock freedom and detection of unreachable states with data exhibiting certain values, can be specified and automatically checked with the help of Maude, thanks to its support for rewriting modulo SMT. The approach presented in this paper has been validated on realistic processes and it is illustrated with a running example.

1 Introduction

Business processes are omnipresent in companies all around the world. A business process is a collection of structured activities or tasks that produce a specific product and fulfil a specific organizational goal for a customer or market. A process aims at modeling activities, their causal and temporal relationships, and specific business rules that process executions have to comply with. In this context, business process modeling is of prime importance to analyze and control business processes.

Business processes are usually described using workflow-based notations. BPMN is one of these notations. It was normalized by ISO/IEC in 2013 [16] and has become the *de facto* standard for modeling business processes. BPMN is a quite expressive notation that describes the order in which a set of activities is executed. Beyond basic operators (e.g., beginning, end, sequence), the notion of gateways allows designers to specify different evolutions of the process control flow (e.g., choice, parallel, message-based). In this work, there is the particular interest on data descriptions, which take two different forms in BPMN processes, namely: (i) as variables that are initialized and modified during the process execution, and (ii) as conditions that may be used

in gateways to decide the branches to be triggered at runtime. Representing data and conditions in BPMN is crucial for enabling the modeling of real-world processes, where data is pervasive and comes from different sources. BPMN tools such as the Activiti or Bonita platforms provide support for the definition of data-aware workflows. However, as it can be seen in the related work section of this paper, just a few formal specification and verification approaches consider this level of expressiveness.

The research presented in this paper takes a step farther by supporting the symbolic specification, execution, and analysis of data-aware processes with an external and non-deterministic environment (e.g., interaction with user input or sensor probing). This basically means that process variables can be initialized using assignments or by means of lookup operations, so that their values are non-deterministically provided by the environment. The consideration of such open systems, however, poses new challenges that include the potential infinitely-branching nondeterminism due to the environment. Queries over such systems are, in general, beyond the reach of ground rewriting and would require, for instance, inductive techniques over the rewrite relation.

Given such a general and symbolic modeling language for BPMN processes, there are several questions that arise from a correctness point of view. For example, is it possible to verify that a given process is deadlock free for any possible input and interaction with the environment? Are there parts of the workflow that are never reached or cannot be reached during execution under some initial conditions? Are there possible executions of the process leading to a state where the variables have specific values? It can be difficult to answer these questions when considering data ranging over possibly infinite domains. Indeed, most approaches on the analysis and verification of workflows are based on process over-approximation in which data is abstracted, either just by removing it or by replacing it with stochastic information. Although these approaches are useful in the formal analysis of workflows, they can miss important information hidden in the data. E.g., they can miss deadlocks due to data-based conditions that are removed or identify false livelocks since all loops in a process without data are always nonterminating.

The main contributions of this paper are a formal rewriting-based symbolic semantics for BPMN with data support and automated verification techniques for checking properties of interest, such as the ones abovementioned. The encoding of BPMN is made using the rewriting logic framework and is fully executable in the Maude system [5]. It comprises BPMN operators such as end/start events, sequence flows, and gateways annotated with data constraints, and support for looping behavior and unbalanced split/join composition (i.e., no systematic correspondence between split and join gateway patterns). Although some important BPMN features, such as events or exceptions, are not considered, the current supported subset is quite expressive.

The symbolic analysis and verification techniques focus on the behavioral aspect of the BPMN processes, which is described as a control-flow graph with data annotations. Data variables and data-based conditions are supported with the help of the recently developed rewriting modulo SMT approach [30], which is well suited to model and analyze reachability properties of infinite-state open systems that exhibit both internal and external nondeterminism due to the environment. In particular, data variables are logical variables under the control of an SMT-solver and data conditions are constraints

over these variables that can be checked for satisfiability with SMT-based decision procedures. This approach ultimately enables the automatic verification of existential reachability properties of BPMN processes with a potential infinite number of initial states. They include deadlock freedom, detection of unreachable states, and reachability of certain states based on data analysis relative to an initial constraint on the data variables. The Maude specification presented here and several examples are available at `http://maude.lcc.uma.es/BPMN-SMT`.

The organization of the rest of this paper is as follows. Section 2 introduces BPMN with data and the running example. The example has design issues on purpose, with the intention of identifying them with the help of the proposed formal analysis approach. Section 3 gives some background about rewriting logic and rewriting modulo SMT. Section 4 introduces the Maude encoding of the considered subset of BPMN, with emphasis on the handling of data. In Section 5, analysis techniques for automatically verifying properties on data-aware BPMN processes are presented and illustrated with the help of the running example; it ends with a proposal to correct such an example. Section 6 surveys related work and Section 7 concludes the paper.

2   BPMN with Data

BPMN is a workflow-based notation for modeling business processes as collections of tasks that produce specific services or products for particular clients. BPMN is an ISO/IEC standard [16], and can be executed by using different process interpretation engines (e.g., Activiti, Bonita BPM, or jBPM).

In this paper, our goal is not to consider the whole expressiveness of the BPMN language, but to concentrate on its main elements related to control-flow modeling and on data aspects that can be represented in BPMN constructs (variables and conditional flows). By focusing on these aspects, we show how automated analysis is possible for them. Specifically, we consider the node types *event*, *task*, and *gateway*, and the edge type *sequence flow*. Figure 1 illustrates the syntax of BPMN supported in this work, including examples of data assignments and conditions at exclusive/inclusive split gateways.

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.

In BPMN, variables are global to the process. Their initialization and modification is possible at the task level using assignment (:=). Values from the environment can be read using a `lookup` operator. In this paper, we consider basic datatypes (integer, real and Boolean) with usual functions on them. As an example, integers can be manipulated using functions +, -, *, etc. These variables are also used to define conditions in *gateways*.

Gateways are crucial since they are used to model control flow branching and therefore influence the overall process execution. In this paper, we support the three main types of gateways, namely, *exclusive*, *inclusive* and *parallel* gateways. Gateways with
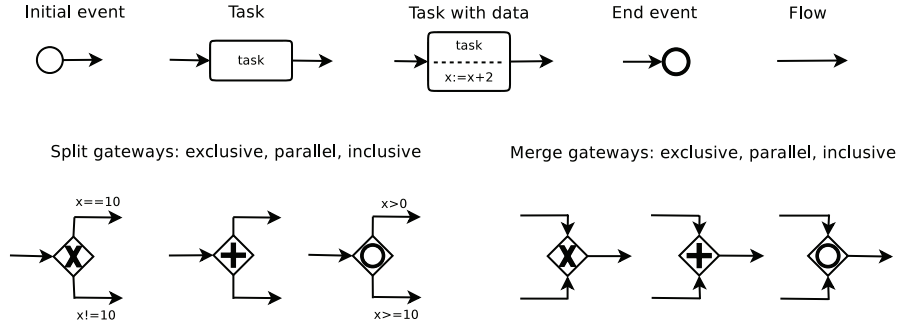
Fig. 1. BPMN syntax augmented with data variables and conditions

one incoming branch and multiple outgoing branches are called *splits*, e.g., inclusive split gateway. Gateways with one outgoing branch and multiple incoming branches are called *merges*, e.g., parallel merge gateway. An exclusive gateway chooses one out of a set of mutually exclusive alternative incoming or outgoing branches depending on the evaluation of conditions on their outgoing flows. For an inclusive gateway, any number of branches among its incoming or outgoing branches may be taken depending on the evaluation of flow conditions. A parallel gateway creates concurrent flows for its outgoing branches or synchronizes concurrent flows for its incoming branches. In this work, we support unbalanced workflows, meaning that each merge gateway does not necessarily have a corresponding split gateway with a correspondence of the branches among outgoing and incoming flows. We also support workflows with looping behaviors.

We assume that BPMN processes are syntactically correct. This can be enforced using existing works, e.g., [12], or using a BPMN engine, as the Activiti or Bonita platforms. Note that any well-typed expression may be used in assignments, using variables, literal and operators, any valid Boolean expression may be used in conditions, and split (resp. merge) gateways may have any positive number of outgoing (resp. incoming) branches.

The semantics of BPMN is described informally in official documents [16, 24], and several attempts have been made for giving a formal semantics to subsets of BPMN (see Section 6). The semantics of BPMN is usually described using *tokens* representing the evolution of a process execution. A token can enter and leave a task by following the incoming and outgoing flows associated to that task. Tokens are created and consumed at gateways. When a token arrives at a parallel split gateway, it is consumed and one token is generated for every outgoing flow of the split gateway. When a token is consumed at an exclusive split gateway, only one token is created and assigned to the outgoing flow whose condition is evaluated to true. In the case of an inclusive split gateway, when a token is consumed, some new tokens are generated and assigned to the outgoing flows (one for each outgoing flow whose corresponding condition is evaluated to true). We assume that a process starts its execution with exactly one token located at its start event, and finishes its execution when all tokens are at end events. In the following, we will
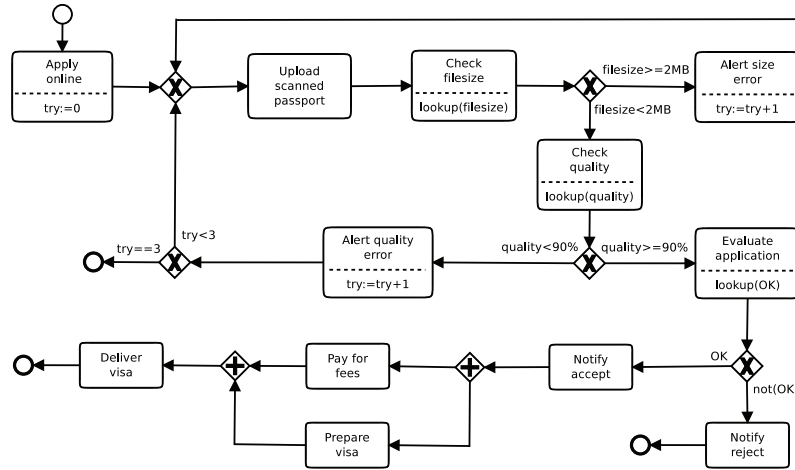
Fig. 2. Running example: e-visa application process

say that a task or flow (or branch) is *active* if it has a token associated with it. E.g., the branches a gateway follows during an execution will be called the active branches.

Running example. We use as running example an online e-visa service described as a BPMN process shown in Figure 2. The workflow models the on-line application for a visa, in which the user has to fulfil several forms, provide an electronic copy of her passport and pay the corresponding fees. The process starts with the client initiating the application by filling some basic information. A scanned version of the passport must then be provided. The calculation of the size and quality of the uploaded file is modeled as reading from the environment its size and quality using the `lookup` operator. If the size of the uploaded file does not respect the size limit (2MB), the user may try again to submit another file. Once the file size is within the size limit, the application checks for image quality. Here, again, if the quality is not good enough, the user can upload another scanned version of her passport. The user has up to three attempts to upload the scanned copy of her passport with valid file size and quality. When both the size limit and scan quality respect the imposed thresholds, the request is evaluated and a result is notified to the user (accept or reject). The evaluation is performed by a human agent, which provides her response as an input to the system. In case of acceptance, the user has to pay for fees and the bureau in charge of visa delivery prepares the requested document. Both activities are achieved in parallel because they are independent. Finally, an electronic version of the visa is delivered by email.

In addition to classic BPMN elements, data are used at different places of the process, e.g., for keeping track of the number of attempts or for storing the size and quality of the uploaded files. In those usages, the case study includes several interesting data-related features: different variable types (integer, real and Boolean), variable manipulation (assignments, arithmetic expressions and predicates), data-based decisions

(depending on input data file size and quality), and external decisions (application evaluation by agent).

As the careful reader has already possibly noticed, and as we will see in Section 5, the process in Figure 2 has several design problems, e.g., the number of attempts for uploading the passport is not correctly handled and checked. The remaining sections of this paper will show how the proposed analysis techniques can detect these issues.

3   Rewriting Logic and Rewriting Modulo SMT in a Nutshell

This section briefly explains order-sorted rewriting logic and rewriting modulo SMT, summarizing Sections 2–5 in [30]. Rewriting logic [21] is a semantic framework that unifies a wide range of models of concurrency. Maude [5] is a language and tool to support the formal specification and analysis of concurrent systems in rewriting logic. Notation on terms, term algebras, and equational theories is used as in, e.g., [1, 13].

An *order-sorted signature* Σ is a tuple $\Sigma=(S, \leq, F)$ with a finite poset of sorts $(S, \leq)$ and a set of function symbols F typed with sorts in S. The binary relation $\equiv_{\leq}$ denotes the equivalence relation $(\leq \cup \geq)^+$ generated by $\leq$ on S and its point-wise extension to strings in S^*. For any sort $s \in S$, the expression $[s]$ denotes the connected component of s, that is, $[s] = [s]_{\equiv_{\leq}}$. A *top sort* in Σ is a sort $s \in S$ such that for all $s' \in [s]$, $s' \leq s$. Let $X = \{X_s\}_{s \in S}$ denote an S-indexed family of disjoint variable sets with each X_s countably infinite. The *set of terms of sort s* and the *set of ground terms of sort s* are denoted, respectively, by $T_{\Sigma}(X)_s$ and $T_{\Sigma,s}$; similarly, $T_{\Sigma}(X)$ and T_{Σ} denote, respectively, the set of terms and the set of ground terms. $\mathcal{T}_{\Sigma}(X)$ and $\mathcal{T}_{\Sigma}$ denote the corresponding order-sorted Σ-term algebras. A *substitution* is an S-indexed mapping $\theta : X \longrightarrow T_{\Sigma}(X)$ that is different from the identity only for a finite subset of X and such that $\theta(x) \in T_{\Sigma}(X)_s$ if $x \in X_s$, for any $x \in X$ and $s \in S$. Substitutions extend homomorphically to terms in the natural way. A substitution θ is called *ground* if and only if $ran(\theta) = \emptyset$. The application of a substitution θ to a term t is denoted by $t\theta$.

An *equational theory* is a tuple (Σ, E), with Σ an order-sorted signature and E a finite collection of (possibly conditional) Σ-equations. An equational theory $\mathcal{E} = (\Sigma, E)$ induces the congruence relation $=_{\mathcal{E}}$ on $T_{\Sigma}(X)$ defined for $t, u \in T_{\Sigma}(X)$ by $t =_{\mathcal{E}} u$ if and only if $\mathcal{E} \vdash t = u$, where $\mathcal{E} \vdash t = u$ denotes $\mathcal{E}$-provability by the deduction rules for order-sorted equational logic in [22]. The expressions $\mathcal{T}_{\mathcal{E}}(X)$ and $\mathcal{T}_{\mathcal{E}}$ (also written $\mathcal{T}_{\Sigma/E}(X)$ and $\mathcal{T}_{\Sigma/E}$) denote the quotient algebras induced by $=_{\mathcal{E}}$ on the term algebras $\mathcal{T}_{\Sigma}(X)$ and $\mathcal{T}_{\Sigma}$, respectively. $\mathcal{T}_{\Sigma/E}$ is called the *initial algebra* of (Σ, E).

A *rewrite theory* is a tuple $\mathcal{R} = (\Sigma, E, R)$ with (Σ, E) an order-sorted equational theory and R a finite set of Σ-rules. $\mathcal{R} = (\Sigma, E, R)$ is called a *topmost rewrite theory* if it has a top sort *Conf* such that no operator in Σ has *Conf* as argument sort and each rule $l \rightarrow r$ **if** $\phi \in R$ satisfies $l, r \in T_{\Sigma}(X)_{Conf}$ and $l \notin X$. A rewrite theory $\mathcal{R}$ induces a rewrite relation $\rightarrow_{\mathcal{R}}$ on $T_{\Sigma}(X)$ defined for every $t, u \in T_{\Sigma}(X)$ by $t \rightarrow_{\mathcal{R}} u$ if and only if there is a rule $(l \rightarrow r$ **if** $\phi) \in R$ and a substitution $\theta : X \longrightarrow T_{\Sigma}(X)$ satisfying $t =_E l\theta$, $u =_E r\theta$, and $E \vdash \phi\theta$. The tuple $\mathcal{T}_{\mathcal{R}} = (\mathcal{T}_{\Sigma/E}, \rightarrow_{\mathcal{R}}^*)$ is called the *initial reachability model of $\mathcal{R}$* [3].

Appropriate requirements are needed to make an equational theory $\mathcal{E}$ *admissible*, i.e., *executable* in rewriting languages such as Maude [5]. In this paper, it is assumed that the equations of $\mathcal{E}$ can be decomposed into a disjoint union $E \uplus B$, with B a collection

of regular and linear structural axioms (such as associativity, and/or commutativity, and/or identity) for which there exists a *matching algorithm modulo B* producing a finite number of B-matching solutions, or failing otherwise. Furthermore, it is assumed that the equations E can be oriented into a set of (possibly conditional) sort-decreasing, operationally terminating, confluent rewrite rules $\overrightarrow{E}$ modulo B. The rewrite system $\overrightarrow{E}$ is *sort decreasing* modulo B if and only if for each $(t \rightarrow u \text{ if } \gamma) \in \overrightarrow{E}$ and substitution θ, $ls(t\theta) \geq ls(u\theta)$ if $(\Sigma, B, \overrightarrow{E}) \vdash \gamma\theta$. The system $\overrightarrow{E}$ is *operationally terminating* modulo B [10] if and only if there is no infinite well-formed proof tree in $(\Sigma, B, \overrightarrow{E})$. Furthermore, $\overrightarrow{E}$ is *confluent* modulo B if and only if for all $t, t_1, t_2 \in T_\Sigma(X)$, if $t \rightarrow^*_{E/B} t_1$ and $t \rightarrow^*_{E/B} t_2$, then there is $u \in T_\Sigma(X)$ such that $t_1 \rightarrow^*_{E/B} u$ and $t_2 \rightarrow^*_{E/B} u$. The term $t \downarrow_{E/B} \in T_\Sigma(X)$ denotes the *E-canonical form* of t modulo B so that $t \rightarrow^*_{E/B} t \downarrow_{E/B}$ and $t \downarrow_{E/B}$ cannot be further reduced by $\rightarrow_{E/B}$. Under sort-decreasingness, operational termination, and confluence, the term $t \downarrow_{E/B}$ is unique up to B-equality. For a rewrite theory $\mathcal{R}$, the rewrite relation $\rightarrow_\mathcal{R}$ is undecidable in general, even if its underlying equational theory is admissible, unless conditions such as coherence [31] are given (i.e, whenever rewriting with $\rightarrow_{R/E\cup B}$ can be decomposed into rewriting with $\rightarrow_{E/B}$ and $\rightarrow_{R/B}$).

A rewrite theory $\mathcal{R} = (\Sigma, E \uplus B, R)$ modulo a built-in subtheory $\mathcal{E}_0 = (\Sigma_0, E_0 \uplus B_0) \subseteq (\Sigma, E \uplus B)$ is a topmost rewrite theory with a signature of built-ins $\Sigma_0 = (S_0, \leq_0, F_0)$. The *ground rewrite relation* $\rightarrow_\mathcal{R}$ induced by a rewrite theory with built-ins $\mathcal{R}$ is the topmost rewrite relation defined for any $t, u \in T_{\Sigma, Conf}$ as follows: $t \rightarrow_\mathcal{R} u$ if and only if there is a rule $l \rightarrow r \text{ if } \phi$ in R and a ground substitution $\sigma : X \longrightarrow T_\Sigma$ such that $t =_{E \uplus B} l\sigma$, $u =_{E \uplus B} r\sigma$, and $\mathcal{T}_{\mathcal{E}_0} \models \phi\sigma$.

The symbolic rewrite relation induced by a rewrite theory with built-ins $\mathcal{R}$ operates over pairs $(t \, ; \varphi)$, called *constrained terms*, where $t \in T_\Sigma(X_0)_{Conf}$ is a term, $\varphi \in QF_{\Sigma_0}(X_0)$ is a constraint of built-ins, and $X_0 \subseteq X$ are the variables with sorts in S_0. Each formula in $QF_{\Sigma_0}(X_0)$ is a Boolean combination of atoms, where an atom is a Σ_0-equation with variables in X_0. For any term $t \in T_\Sigma(X)_{Conf}$ and constraint φ, the *denotation* $[\![t]\!]_\varphi$ is defined as $[\![t]\!]_\varphi = \{t' \in T_{\Sigma, Conf} \mid (\exists \sigma : X \longrightarrow T_\Sigma) \; t' =_E t\sigma \wedge \mathcal{T}_{\mathcal{E}_0} \models \varphi\sigma\}$. Given a rewrite rule $((l \, ; \phi_l) \rightarrow (r \, ; \phi_r) \text{ if } \phi) \in R$, with $l, r \in T_\Sigma(X)_{Conf}$ and $\phi \in QF_{\Sigma_0}(X_0)$, a constrained term $(t \, ; \phi_t) \in T_\Sigma(X_0)_{Conf} \times QF_{\Sigma_0}(X_0)$ symbolically rewrites to a constrained term $(u \, ; \phi_u) \in T_\Sigma(X_0)_{Conf} \times QF_{\Sigma_0}(X_0)$ (denoted by $(t \, ; \phi_t) \rightsquigarrow_\mathcal{R} (u \, ; \phi_u)$) if and only if there is a substitution θ such that: (a) $l\theta =_{E \uplus B} t$ and $r\theta =_{E \uplus B} u$, (b) $\mathcal{T}_{\Sigma/E \uplus B} \models (\phi_l \wedge \phi_t\theta) \Leftrightarrow \phi_u$, and (c) ϕ_u is $\mathcal{T}_{\Sigma/E \uplus B}$-satisfiable. Condition (a) can be solved by matching as in the definition of $\rightarrow_\mathcal{R}$ above. Condition (b) can be met by setting ϕ_u to be $\phi_l \wedge \phi_t\theta$. Condition (c) is checked with the help of decision procedures available from an SMT solver via the function *check-sat*. Observe that, up to the choice of the semantically equivalent φ_u, the symbolic relation $\rightsquigarrow_\mathcal{R}$ is deterministic in the sense of being determined by the rule and the substitution θ (here it is assumed that variables in the rules are disjoint from the ones in the target terms). The reader is referred to [30] for details about rewriting modulo SMT and the correspondence between $\rightarrow_\mathcal{R}$ and $\rightsquigarrow_\mathcal{R}$.

```
< pid : Process |
  nodes :
    (start(initial, sf1),
      end(final1, sf25),
      end(final2, sf24),
      end(final3, sf14),
      task(t1, "Apply online", sf1, sf2, try := 0),
      task(t2, "Upload scanned passport", sf3, sf4),
      task(t3, "Check filesize", sf4, sf5, lookup(filesize)),
      task(t4, "Alert size error", sf6, sf8, try := try + 1),
      task(t5, "Check quality", sf7, sf9, lookup(quality)),
      task(t6, "Evaluate application", sf11, sf15, lookup(OK)),
      task(t7, "Alert quality error", sf10, sf12, try := try + 1),
      task(t8, "Notify accept", sf16, sf18),
      task(t9, "Pay for fees", sf19, sf21),
      task(t10, "Deliver visa", sf23, sf24),
      task(t11, "Notify reject", sf17, sf25),
      task(t12, "Prepare visa", sf20, sf22),
      merge(m1, exclusive, (sf2, sf8, sf13), sf3),
      merge(m2, parallel, (sf21, sf22), sf23),
      split(s1, exclusive, sf5, (sf6, filesize >= 2) (sf7, filesize < 2)),
      split(s2, exclusive, sf9, (sf10, quality < 9/10) (sf11, quality >= 9/10)),
      split(s3, exclusive, sf15, (sf17, OK === false) (sf16, OK === true)),
      split(s4, parallel, sf18, (sf19, sf20)),
      split(s5, exclusive, sf12, (sf13, try < 3) (sf14, try === 3)))
  flows :
    (flow(sf1), flow(sf2), ..., flow(sf24), flow(sf25)) >
```

Fig. 3. Running example: representation in Maude of the e-visa application process

4   Symbolic Specification

The symbolic semantics of BPMN is defined as a rewrite theory $\mathcal{R}$, with built-ins $\mathcal{E}_0$ and topsort Conf. A symbolic state is a constrained term $(t \,; \varphi)$, where t is a configuration of objects and φ a constraint. The objects in t represent the set of nodes and flows of a process, and keep information of the execution of the process. The constraint φ ranges over the built-in sorts Boolean, Integer, and Real, of Booleans, integers, and reals, respectively, and it maintains the bookkeeping of constraints accumulated during execution. For example, a constraint φ can encode the possible values for an integer variable that are possible after the process is executed.

Figure 3 includes a representation in Maude of the e-visa application process as a Process object. This object represents the static part of the process, and identifies its nodes (attribute nodes) and flows (attribute flows). The process pid is parametric on 4 built-in variables, namely, OK (of sort Boolean), try and filesize (of sort Integer), and quality (of sort Real). Thus the constraints accumulated during the execution of this process range over these 4 variables.

The initial node of the process is represented as the term start(initial, sf1) specifying that the initial node has outgoing flow sf1. Task nodes are specified by an identifier, a label, an input flow, an output flow, and a list of assignments. An assignment has the form x := E, where x is a variable, and E an expression over the variables with the same sort of x. In this particular example, x is a variable in the set {OK, try, filesize, quality} and E an expression over the same set of variables. For in-

```
< sim : Simulation |
    tokens : token(sf1),
    constr : true,
    varidx : (v(OK) |-> 0, v(try) |-> 0, v(filesize) |-> 0, v(quality) |-> 0) >
```

Fig. 4. Running example: simulation in Maude of the e-visa application process

stance, the assignment `try := try + 1` in task `t4` indicates that the value represented by the variable `try` is to be incremented in one unit. For the reading of external variables we use the `lookup` operator. E.g., variables `filesize` and `quality` are read in tasks `t3` and `t5`, respectively. Split nodes are specified by an identifier, a type (`exclusive`, `inclusive`, or `parallel`), an incoming flow, and a nonempty list of outgoing flows. Exclusive and inclusive gateways are equipped with constraints associated to the outgoing flows: these represent conditions over the data of the process. For instance, in the exclusive split gateway `s2`, the list

$$(\text{sf10}, \text{ quality } < 9/10) \; (\text{sf11}, \text{ quality } >= 9/10)$$

specifies that the flow `sf10` is triggered when the quality of the picture is below 90% and that the flow `sf11` is taken otherwise. Merge nodes are specified by an identifier, a type (the same types of split gateways), a set of incoming flows, and an outgoing flow. End nodes are specified by an identifier and an incoming flow. Flows are specified just by an identifier.

Figure 4 shows a representation in Maude of an execution state of the process as the object `sim`. This object gathers the dynamic information used along execution, and has attributes identifying the set of tokens (attribute `tokens`), the constraint accumulated during execution (attribute `constr`), and a map (of sort `Map{SymbVar, Nat}`) for indexing the built-in variables present in the state (attribute `varidx`). In this example, the token to be processed next is at flow `sf1`, the constraint is `true` (i.e., the empty one), and all variables in the process are indexed at 0.

The index of a variable can increase during execution, which is key because an expression over the built-ins is evaluated with respect to a given variable indexing. Since the variables in a process are "mathematical" variables, they need to be treated with a predicate-transformer approach (opposed to the imperative programming-like approach). For example, given the variable indexing `v(try) |-> 4`, the expression `try + 1` is interpreted as `try#4 + 1`, meaning that the interpretation of the increment of `try` by one unit is to be done with respect to the 'latest' version of the variable. When an assignment such as `try := try + 1` is evaluated, it creates a constraint that is added to the global constraint and depends on the given variable indexing. For example, with the variable indexing `v(try) |-> 4`, the evaluation of `try := try + 1` results in the constraint `try#5 === try#4 + 1`, meaning that the "new" value of `try` corresponds to its previous value incremented by one. The `lookup` operator works similarly, although in this case there is no restriction on the new value.

The concurrent transitions of $\mathcal{R}$ are specified by rewrite rules that update the simulation object `sim`. Specifically, there are 12 rewrite rules that model the different actions

```
crl [execTask] :
    < PId : Process |
          nodes : (task(NId, TaskName, FId1, FId2, AsgL), Nodes),
          flows : (flow(FId2), Flows),
          Atts >
    < SId : Simulation |
          tokens : (token(NId), Tks),
          constr : B,
          varidx : VMp,
          Atts1 >
 => < PId : Process |
          nodes : (task(NId, TaskName, FId1, FId2, AsgL), Nodes),
          flows : (flow(FId2), Flows),
          Atts >
    < SId : Simulation |
          tokens : (token(FId2), Tks),
          constr : (B and B1),
          varidx : VMp1,
          Atts1 >
 if (VMp1, B1) := prepare-update(AsgL, VMp) .
```

Fig. 5. Execution of a task with a list of assignments

that may happen in the system, e.g., start or end the execution of a process, handle the
arrival of a token to split and merge gateways, tasks, and flows. For illustration purposes,
we explain in the rest of this section two of these rules.

Figure 5 shows the rewrite rule that handles execution of a task. Intuitively, ex-
ecuting a task results in a new symbolic state in which the constraint is updated by
accumulating the constraints that result from the list of assignments associated to the
task, if possible. Since handling an assignment changes the variable indexing kept in
the object sim, a new version of the variable indexing term needs to be computed. Both
the constraint resulting from a list of assignments and the new variable indexing are
computed with the help of the auxiliary function prepare-update. For example, the
evaluation of the term

```
prepare-update((X := X + 1) (Y := Y + X), (v(X) |-> 4, v(Y) |-> 0))
```

returns the pair

```
((v(X) |-> 5, v(Y) |-> 1), X#5 === X#4 + 1 and Y#1 === Y#0 + X#5)
```

where the index of both variables X and Y is incremented, and the constraint chains
the sequential assignment performed on these variables. Note that this rule does not
check the satisfiability of the constraint B1 resulting from the assignment in conjunction
with the constraint B from the state; this is because if the latter is satisfiable, so is their
conjunction since the newly constrained variables in B1 are fresh with respect to B.

One interesting case for symbolic specification can be observed in the semantics
of inclusive gateways, axiomatized by the rule in Figure 6. In a version without data,
when a token arrives at a split gateway, any number of outgoing flows are selected and
assigned newly created tokens that start executing right away. However, in a symbolic
version with data, the situation is more complex because *all* possible flow selections

```
crl [splitGatewayInclusive] :
    < PId : Process |
        nodes : (split(NId, inclusive, FId, LTIB), Nodes),
        flows : Flows,
        Atts >
    < SId : Simulation |
        tokens : (token(FId), Tks),
        constr : B,
        varidx : VMp,
        Atts1 >
 => < PId : Process |
        nodes : (split(NId, inclusive, FId, LTIB), Nodes),
        flows : Flows,
        Atts >
    < SId : Simulation |
        tokens : (tokens(IL), Tks),
        constr : (B and B2),
        varidx : VMp,
        Atts1 >
 if ILL1 . IL . ILL2 := gen(project1(LTIB))
 /\ B1 := get-constr(IL, LTIB)
 /\ B2 := process-expression(VMp, B1)
 /\ check-sat(B and B2) .
```

Fig. 6. Arrival of a token to an inclusive split gateway

need to be considered and checked for satisfiability relative to the global constraint. E.g, if there are n outgoing flows, then there are at most $2^n - 1$ possible selections satisfiable relative to the global constraint. Each one of these scenarios, computed by the auxiliary function gen, can be identified by the set of the m constraints ($0 < m \leq n$) associated to the selected outgoing flows, say Γ, and the set of the $n - m$ constraints associated to the outgoing flows that are not selected, say Δ. Then, the constraint B1 associated to such selection is logically equivalent to the following conjunction of constraints:

$$\bigwedge_{\gamma \in \Gamma} \gamma \ \wedge \ \bigwedge_{\delta \in \Delta} \neg \delta.$$

Such a constraint needs to be interpreted with respect to the current variable indexing, resulting in a new constraint, say B2; this is done with the help of the auxiliary function process-expression. Finally, a split is possible if the constraint representing the symbolic split is compatible with the constraint B accumulated in the state, i.e., if check-sat(B and B2) evaluates to true.

5   Symbolic Execution and Reachability Analysis

Symbolic execution and reachability analysis can be useful for exploring infinitely many system executions at once. In the rewriting modulo SMT implementation available from Maude, symbolic execution of the rewrite theory $\mathcal{R}$ presented in Section 4 uses a combination of term rewriting, matching modulo axioms, and SMT solving. This consists in applying equations and rewrite rules from an initial term (e.g., from an initial state such as the one in Figure 3) and querying the SMT solver for checking satisfiability of constraints at each rewrite step, when necessary. The full potential of rewriting

modulo SMT can be exploited for solving existential reachability queries in the initial model $\mathcal{T}_{\mathcal{R}}$ of a rewrite theory $\mathcal{R}$ modulo built-ins $\mathcal{E}_0$. The type of *existential reachability* question that rewriting modulo SMT can answer can be formulated as follows:

are there states in $[\![t]\!]_\phi$ that can reach a state in $[\![u]\!]_\psi$?

This question is especially useful for symbolically proving or disproving safety properties, such as inductive invariants and deadlock freedom of $\mathcal{T}_{\mathcal{R}}$: when $[\![u]\!]_\psi$ is a set of *bad* states, the goal is to check whether reaching a state in $[\![u]\!]_\psi$ is possible.

In the rest of this section, we show how to use symbolic rewriting and symbolic reachability analysis for verifying properties of interest on data-aware processes encoded into Maude's rewriting logic. This is illustrated using the e-visa application process introduced in Section 2. Note that, beyond toy examples and the e-visa application process we use as running example in this paper, we have also applied our approach to two other examples we found in the literature: a drug store process [15] and a DMN process on application file handling [25, Section 11.2].

5.1 Symbolic Rewriting

As for regular rewrite theories, rewriting may be useful for gathering a first understanding of the whereabouts of our systems. Consider the running example in Figure 2. For the initial state corresponding to the terms in Figures 3 and 4, the following rewrite command in Maude symbolically executes the process for ten consecutive rewrite steps:

```
rew [10] initSystem(v(OK:Boolean) v(TRY:Integer) v(FSIZE:Integer) v(QUAL:Real)) .
```

The `initSystem` operator generates the initial state depending on the specified list of variables. Maude's output to this command corresponds to one of the possible states reached after ten rewrite steps; in this case, the output is the following term:

```
< p : Process | nodes : ..., flows : ... >
< s : Simulation |
    tokens : token(t2),
    constr : (TRY#1:Integer === 0 and
              FSIZE#1:Integer >= 2 and
              TRY#2:Integer === TRY#1:Integer + 1,
    varidx : (v(OK) |-> 0, v(TRY) |-> 2, v(FSIZE) |-> 1, v(QUAL) |-> 0) >
```

In this state, there is exactly one token, at task t2, the "Upload scanned passport" task. The constraint indicates that the TRY variable has value 1 and that FSIZE#1 is at least 2. After a first upload of an oversized file, task "Alert size error" was executed, where the TRY variable got increased, and then moved to the task t2. Note that the index of variable TRY is set to 2 because of the initial assignment on this variable in the "Apply online" task, and its update in "Alert size error". The index for FSIZE is 1.

A constraint like the one in this final state can also be interpreted as the conditions on the initial state and interactions to lead us to such state. Although a symbolic state may represent an infinite number of concrete states, as a symbolic path may represent an infinite number of concrete executions, in this case, we can say that any execution in which we upload an oversized file will lead to this state in ten rewriting steps.

5.2 Symbolic Reachability Analysis

Beyond symbolic rewriting, symbolic reachability is useful in order to answer a number of interesting questions such as the following ones:

- What are the reachable states after execution of *n* rewrite steps? Notice that each rewrite represent an infinite number of possible executions.
- Is it possible for an input variable to be assigned a certain value? As an example, can we check whether our running example can reach a state in which the TRY variable takes a value greater than 3?
- Are there deadlocks in the process?
- Does a process have unreachable flows or tasks?

In the rest of this section, we will see how reachability analysis can be used to answer these questions. Consider the following existential query, where $\mathcal{R}$ is the rewrite theory presented in Section 4 corresponding to the running example in Figure 2:

$$\mathcal{T}_{\mathcal{R}} \models (\exists \texttt{OK:Boolean, TRY:Integer, FSIZE:Integer, QUAL:Real, St:Conf})$$

$$\texttt{initSystem(v(OK) v(TRY) v(FSIZE) v(QUAL))} \to_{\mathcal{R}}^{*} \texttt{St}$$

$$\wedge \text{ "St is } \to_{\mathcal{R}}\text{-irreducible"} \wedge \text{ "TRY = 3 in St"}.$$

This query asks whether we can reach from our initial state an irreducible state (w.r.t. the rewrite relation) in which the value of the TRY variable can be 3. Observe that infinitely many states need to be considered in the query because some of the variables range over infinite domains (e.g., integer and real numbers). This means that such a query, in general, is beyond the reach of ground rewriting and would require, for instance, inductive techniques over the rewrite relation. However, the following search command can be issued in Maude to find a proof (or a counterexample) of the reachability query for $\to_{\mathcal{R}}$ using the symbolic rewrite relation $\rightsquigarrow_{\mathcal{R}}$:

```
search [1] initSystem(v(OK) v(TRY) v(FSIZE) v(QUAL)) =>! St
  such that
    check-sat(get-constr(St) and process-expression(get-varidx(St), try === 3)) .
```

Note the use of [1] and =>! to indicate, respectively, our interest in finding exactly one witness to the existential reachability query and that such a witness needs to be irreducible (i.e., no rule can be applied to it). The function calls get-constr(St) and get-varidx(St) return, respectively, the constraint and the variable indexing from any execution state St. The satisfiability of the constraint TRY === 3 needs to be checked against the variable indexing at each corresponding execution state; this is achieved by invoking the auxiliary function process-expression. The above search command generates the following output:

```
Solution 1 (state 155)
states: 171  rewrites: 2111 in 68ms cpu (70ms real) (30874 rewrites/second)
St --> < p : Process | nodes : ..., flows : ... >
       < s : Simulation |
           tokens : empty,
           constr : (TRY#1:Integer === 0 and
                     FSIZE#1:Integer >= 2 and
                     TRY#2:Integer === TRY#1:Integer + 1 and
                     FSIZE#2:Integer >= 2 and
```

```
                        TRY#3:Integer === TRY#2:Integer + 1 and
                        FSIZE#3:Integer < 2 and
                        QUAL#1:Real < 9/10 and
                        TRY#4:Integer === TRY#3:Integer + 1 and
                        TRY#4:Integer === 3),
              varidx : (v(OK) |-> 0, v(TRY) |-> 4, v(FSIZE) |-> 3, v(QUAL) |-> 1) >
```

This means that there is at least one ground execution from an initial state that reaches an irreducible state where the value of the TRY variable is 3. Indeed, the constraint tells us how to reach such a state: three files must be uploaded, the first two with size over 2 MB and the third with a quality under the threshold. Observe that this solution is actually a final state because all tokens have been processed. Note also the ellipses for brevity. The process does not change along the execution. The indication of requesting only one solution is important. Without it, Maude would have kept giving more and more solutions, in which any number of oversized passport files are uploaded due to the unguarded loop. This clearly points out a design error. A corrected version of the process is given below.

As mentioned earlier, the symbolic specification $\mathcal{R}$ presented in Section 4 can be used to automatically check for other safety properties such as deadlock freedom. In general, a reachability query associated to having a deadlock in $\mathcal{T}_\mathcal{R}$ can be cast as the following satisfaction relation:

$$\mathcal{T}_\mathcal{R} \models (\exists \vec{x}, \mathtt{St:Conf})\, \mathtt{initSystem}(\vec{x}) \xrightarrow{*}_\mathcal{R} \mathtt{St}\, \wedge$$

$$\text{``St is} \xrightarrow{}_\mathcal{R}\text{-irreducible''}\, \wedge$$

$$\text{``St is not final''}.$$

In this formula, $\vec{x}$ denotes the list of variables input to the process specified by $\mathcal{R}$. The condition on the irreducibility of the state St is the same one appearing in the previous reachability goal. In the symbolic semantics $\mathcal{R}$, a state is considered final whenever all tokens in the state have been consumed. This is of practical importance because checking for final states can be decided by checking the contents of the token set, namely, by checking if the set is empty. The following search command in Maude can be used to find deadlocks in the running example process:

```
Maude> search [1] in RUN :
          initSystem(v(OK) v(TRY) v(FSIZE) v(QUAL)) =>! St
            such that get-tokens(St) =/= empty .
Solution 1 (state 249)
states: 265   rewrites: 2852 in 97ms cpu (99ms real) (29183 rewrites/second)
St --> < p : Process | nodes : ..., flows : ... >
       < s : Simulation |
         tokens : token(sf12, 0),
         constr : (TRY#1:Integer === 0 and
                   FSIZE#1:Integer >= 2 and
                   TRY#2:Integer === TRY#1:Integer + 1 and
                   FSIZE#2:Integer >= 2 and
                   TRY#3:Integer === TRY#2:Integer + 1 and
                   FSIZE#3:Integer >= 2 and
                   TRY#4:Integer === TRY#3:Integer + 1 and
                   FSIZE#4:Integer < 2 and
                   QUAL#1:Real < 9/10 and
                   TRY#5:Integer === TRY#4:Integer + 1),
         varidx : (v(OK)|-> 0, v(TRY)|-> 5, v(FSIZE)|-> 4, v(QUAL)|-> 1) >
```

The witness provided shows that a deadlock can actually be reached. The constraint in the deadlock state tells us that by uploading a file with poor quality after three oversized files, we reach a state in which there is a token at the flow sf12, the outgoing flow of task "Alert quality error". Note that in that state the variable TRY has value 4, and none of the alternative branches of the following exclusive split can be triggered.

Let us now illustrate the check of whether certain flows or tasks are reachable. The following command proves that the task "Pay for fees" cannot be reached with an oversized file.

```
search [1,500]
  initSystem(v(OK) v(TRY) v(FSIZE) v(QUAL))
  =>! St
  such that token-at("Pay for fees", St)
            and
            check-sat(
              get-constr(St)
              and
              process-expression(get-varidx(St), gen-intvar("FSIZE") > 2)) .
No solution.
states: 8846  rewrites: 117728 in 7818ms cpu (8151ms real) (15057 rewrites/second)
```

In this section we have included the results provided by Maude on the number of executions and time obtained when executing all the rewrite and search commands. If we look at the last search command above, we can see that the exploration of 500 states took around 8 seconds. This number is rather low for Maude, if compared to standard rewriting/search. Notice however that symbolic rewriting/search involves additional satisfiability checks during the rewriting process, which are handled by invocation to back end SMT solvers.

Last but not least, let us give a corrected version of the running example (Figure 7) where the problem coming from an erroneous handling of the number of attempts has been resolved. This was achieved using an exclusive split gateway before the scanned passport upload, which checks for the number of attempts. If this number is greater than three, the process terminates. Note that all the properties mentioned beforehand in this section are satisfied by this new version of the process as we will show in the final part of this section.

There are now only eight possible ways to reach a final state with the variable TRY with value 3, showing the different combinations for uploading an invalid file at most 3 times:

```
search initSystem(v(OK) v(TRY) v(FSIZE) v(QUAL))
       =>! St
       such that check-sat(get-constr(St) and
                 process-expression(get-varidx(St), gen-intvar("TRY") === 3)) .

Solution 1 (state 146)
states: 158  rewrites: 2042 in 62ms cpu (64ms real) (32520 rewrites/second)
St --> < p : Process | nodes : ..., flows : ... >
       < s : Simulation |
         tokens : empty,
         gtime : 0,
         constr : (TRY#1:Integer === 0 and
                   TRY#1:Integer < 3 and
                   FSIZE#1:Integer >= 2 and
                   TRY#2:Integer === TRY#1:Integer + 1 and
                   TRY#2:Integer < 3 and
                   FSIZE#2:Integer >= 2 and
                   TRY#3:Integer === TRY#2:Integer + 1 and
```
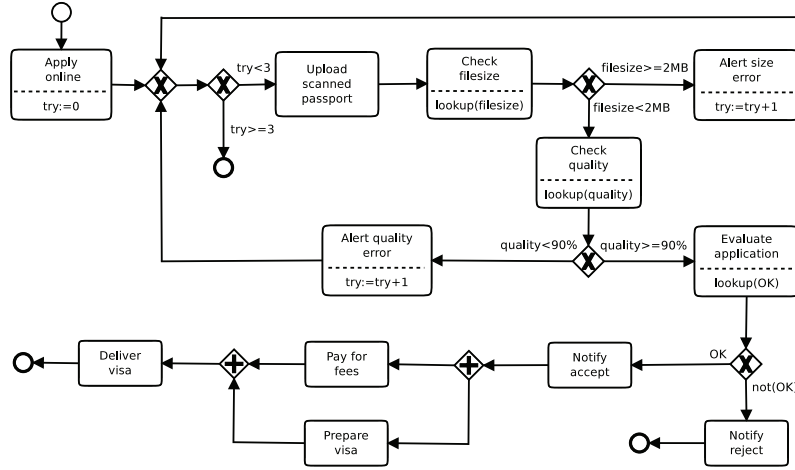
Fig. 7. Running example: e-visa application process (V2)

```
                     TRY#3:Integer < 3 and
                     FSIZE#3:Integer >= 2 and
                     TRY#4:Integer === TRY#3:Integer + 1 and
                     TRY#4:Integer >= 3),
          varidx : (v(OK) |-> 0, v(TRY) |-> 4, v(FSIZE) |-> 3, v(QUAL) |-> 0) >

...

Solution 8 (state 259)
states: 267   rewrites: 3457 in 134ms cpu (136ms real) (25739 rewrites/second)
St --> < p : Process | nodes : ..., flows : ... >
        < s : Simulation |
          tokens : empty,
          gtime : 0,
          constr : (TRY#1:Integer === 0 and
                     TRY#1:Integer < 3 and
                     FSIZE#1:Integer < 2 and
                     QUAL#1:Real < (9/10).Real and
                     TRY#2:Integer === TRY#1:Integer + 1 and
                     TRY#2:Integer < 3 and
                     FSIZE#2:Integer < 2 and
                     QUAL#2:Real < (9/10).Real and
                     TRY#3:Integer === TRY#2:Integer + 1 and
                     TRY#3:Integer < 3 and
                     FSIZE#3:Integer < 2 and
                     QUAL#3:Real < (9/10).Real and
                     TRY#4:Integer === TRY#3:Integer + 1 and
                     TRY#4:Integer >= 3),
          varidx : (v(OK) |-> 0, v(TRY) |-> 4, v(FSIZE) |-> 3, v(QUAL) |-> 3) >
```

The new version of the process is deadlock free:

```
search [1,500]
  initSystem(v(OK) v(TRY) v(FSIZE) v(QUAL))
  =>! St
  such that get-tokens(St) =/= empty .

No solution.
states: 293   rewrites: 3333 in 123ms cpu (126ms real) (26882 rewrites/second)
```

Finally, as for the previous version, there is no way to reach the "Pay for fees" task (t9) with an oversized file:

```
search [1,500] initSystem(v(OK) v(TRY) v(FSIZE) v(QUAL))
       =>! St
       such that token-at("Pay for fees", St)
                  and
                  check-sat(
                    get-constr(St)
                    and
                    process-expression(get-varidx(St), gen-intvar("FSIZE") > 2)) .

No solution.
states: 293  rewrites: 3530 in 133ms cpu (135ms real) (26518 rewrites/second)


search [1,500] initSystem(v(OK) v(TRY) v(FSIZE) v(QUAL))
       =>! St
       such that token-at(t9, St)
                  and
                  check-sat(
                    get-constr(St)
                    and
                    process-expression(get-varidx(St), gen-intvar("FSIZE") > 2)) .

No solution.
states: 293  rewrites: 3490 in 132ms cpu (134ms real) (26390 rewrites/second)
```

6   Related Work

Several works have focused on providing rigorous definitions and formal semantics for business processes using Petri nets, process algebras, or abstract state machines, see, *e.g.*, [7, 8, 14, 18–20, 26, 29, 33, 34]. The main differences with respect to these related works are our focus on data-aware workflow models and the fact that our encoding gives a symbolic semantics to BPMN by translation to Maude.

As far as data-based analysis is concerned, Decision Model and Notation (DMN) is a recent OMG standard for modeling decisions in an interchangeable format. DMN can be used into workflow-based notations for representing conditions. [4] proposes a formal semantics of DMN decision tables, a notion of DMN table correctness, and algorithms that check the detection of overlapping rules and missing rules. These algorithms have been implemented in the DMN toolkit and validated through empirical evaluation. Our modeling language provides the same expressiveness as decision tables existing in DMN, but our analysis techniques go further since they allow to verify properties of interest on the whole flow of control taking data and conditions into account.

Herbert and Sharp [15] choose to simplify the modeling of exclusive/inclusive split gateways by considering probabilities instead of conditions. They propose an algorithm for translating a BPMN subset extended with probabilistic information into the guarded command language used by PRISM. This enables model checking of quantitative properties of business processes such as transient probabilities, occurrence of events, and best-/worst-case scenarios. [15] uses the notion of *rewards*, from Markov Models [32], as annotated values that can be used to keep track of quantities of interest (e.g., execution times, number of iterations, etc.) in processes.

In [27], Prandi et al. propose a formalization for BPMN models which supports rewards and probabilistic elements. They propose a conversion of BPMN models into

a model expressed in the Calculus for Orchestration of Web Services (COWS) [28], which can then be analyzed by using model checking.

[23] focuses on the analysis of choreography models. The main property of interest in that context is called *conformance* and aims at checking whether the distributed implementation and the choreography behave identically. The authors mainly focus on data description. Their approach supports choreographies extended with conditions and relies on SMT solving for conformance checking.

Several authors have used rewriting logic and Maude to model and analyze BPMN processes. El-Saber and Boronat [12] propose a translation of BPMN into rewriting logic with a special focus on data-based decision gateways. They provide mechanisms to avoid structural issues in workflows such as flow divergence by introducing the notion of "well-formed" BPMN process. Their approach aims at avoiding incorrect syntactic patterns whereas we propose automated analysis at the semantic level. Kheldoun et al. [17] propose high-level Petri nets and to use Maude's LTL model checker for, respectively, specifying BPMN processes and analyzing behavioral properties. They also focus on handling exceptions and activity cancellation. Durán and Salaün used Maude to represent BPMN processes enriched with time features in [11]. In this paper, they show how real-time analysis of such BPMN processes could be performed. Specifically, they used simulations, reachability analysis and model checking, and calculate certain properties such as minimum and maximum expected response times, maximum degree of parallelism, and synchronization times. Corradini et al. present in [6] their tool BProVe, a friendly tool for the verification of business processes modeled in BPMN. The tool accepts BPMN processes in standard notation and can perform checks of soundness and safeness on them, as defined in [35] and [9], respectively, using Maude's LTL model checker.

7   Concluding Remarks

This paper focused on the BPMN modeling language enhanced with constructs for supporting data (variables and conditions). A a symbolic semantics for BPMN with data was proposed using Maude's rewriting logic framework. The transformation to Maude enables the verification of process models, including properties such as deadlock freedom and reachability of certain states based on data constraints. The verification task can be automated by using rewriting modulo SMT techniques. The approach was used on several use cases, including an online e-visa application process.

As far as future work is concerned, a first perspective is to increase the number of properties to verify. A first step in this direction is the verification of livelock absence. As initially proposed in [30], the approach presented in this work can also benefit of LTL model checking based on rewriting modulo SMT. The challenge here is in implementing effective mechanisms for dealing with the state explosion problem, such as the ones presented in [2]. The authors are also considering equational abstractions for rewriting modulo SMT. A second perspective aims at combining some previous work on timed BPMN [11] with the approach for data-based process models presented here. This will certainly result in a richer language for process modeling and the likely need of new verification techniques, useful both for timing and data-based analysis.

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge Univ. Press, 1999.
2. K. Bae and C. Rocha. Guarded terms for rewriting modulo SMT. In J. Proença and M. Lumpe, editors, *Formal Aspects of Component Software*, pages 78–97. Springer International Publishing, 2017.
3. R. Bruni and J. Meseguer. Semantic Foundations for Generalized Rewrite Theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006.
4. D. Calvanese, M. Dumas, U. Laurson, F. M. Maggi, M. Montali, and I. Teinemaa. Semantics and Analysis of DMN Decision Tables. In *Proc. of BPM*, volume 9850 of *LNCS*, pages 217–233. Springer, 2016.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.
6. F. Corradini, F. Fornari, A. Polini, B. Re, F. Tiezzi, and A. Vandin. BProVe: A Formal Verification Framework for Business Process Models. In *Proc. of ASE*, pages 217–228. IEEE Computer Society, 2017.
7. G. Decker and M. Weske. Interaction-centric Modeling of Process Choreographies. *Information Systems*, 36(2):292–312, 2011.
8. R. Dijkman, M. Dumas, and C. Ouyang. Semantics and Analysis of Business Process Models in BPMN. *Information and Software Technology*, 50(12):1281–1294, 2008.
9. R. M. Dijkman, M. Dumas, and C. Ouyang. Semantics and analysis of business process models in BPMN. *Information & Software Technology*, 50(12):1281–1294, 2008.
10. F. Durán, S. Lucas, C. Marché, J. Meseguer, and X. Urbain. Proving Operational Termination of Membership Equational Programs. *Higher-Order and Symbolic Computation*, 21(1-2):59–88, 2008.
11. F. Durán and G. Salaün. Verifying Timed BPMN Processes using Maude. In *Proc. of CO-ORDINATION*, volume 10319 of *LNCS*, pages 219–236. Springer, 2017.
12. N. El-Saber and A. Boronat. BPMN Formalization and Verification using Maude. In *Proc. of BM-FA*, pages 1–8. ACM, 2014.
13. J. A. Goguen and J. Meseguer. Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations. *Theoretical Computer Science*, 105(2):217–273, 1992.
14. 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.
15. L. Herbert and R. Sharp. Using Stochastic Model Checking to Provision Complex Business Services. In *Proc. of HASE*, pages 98–105. IEEE, 2012.
16. ISO/IEC. International Standard 19510, Information technology – Business Process Model and Notation. 2013.

17. A. Kheldoun, K. Barkaoui, and M. Ioualalen. Specification and Verification of Complex Business Processes - A High-Level Petri Net-Based Approach. In *Proc. of BPMN*, volume 9253 of *LNCS*, pages 55–71. Springer, 2015.
18. F. Kossak, C. Illibauer, V. Geist, J. Kubovy, C. Natschläger, T. Ziebermayr, T. Kopetzky, B. Freudenthaler, and K. Schewe. *A Rigorous Semantics for BPMN 2.0 Process Diagrams*. Springer, 2014.
19. A. Martens. Analyzing Web Service Based Business Processes. In *Proc. of FASE*, pages 19–33, 2005.
20. R. Mateescu, G. Salaün, and L. Ye. Quantifying the Parallelism in BPMN Processes using Model Checking. In *Proc. of CBSE*, pages 159–168. ACM, 2014.
21. J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
22. J. Meseguer. Membership Algebra as a Logical Framework for Equational Specification. In *Proc. of WADT'97*, volume 1376 of *LNCS*, pages 18–61. Springer, 1998.
23. H. N. Nguyen, P. Poizat, and F. Zaïdi. A Symbolic Framework for the Conformance Checking of Value-Passing Choreographies. In *Proc. of ICSOC*, volume 7636 of *LNCS*, pages 525–532. Springer, 2012.
24. Object Management Group. *Business Process Model and Notation (BPMN) – V. 2.0*. January 2011.
25. Object Management Group. *Decision Model and Notation Specification (DMN) – V. 1.1*. May 2016.
26. P. Poizat and G. Salaün. Checking the Realizability of BPMN 2.0 Choreographies. In *Proc. of SAC*, pages 1927–1934. ACM, 2012.
27. D. Prandi, P. Quaglia, and N. Zannone. Formal Analysis of BPMN via a Translation into COWS. In *Proc. of COORDINATION*, volume 5052 of *LNCS*, pages 249–263. Springer, 2008.
28. R. Pugliese and F. Tiezzi. A Calculus for Orchestration of Web Services. *J. Applied Logic*, 10(1):2–31, 2012.
29. 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*, pages 126–137, 2007.
30. C. Rocha, J. Meseguer, and C. Muñoz. Rewriting Modulo SMT and Open System Analysis. *Journal of Logical and Algebraic Methods in Programming*, 86(1):269 – 297, 2017.
31. P. Viry. Equational Rules for Rewriting Logic. *Theoretical Computer Science*, 285(2):487–517, 2002.
32. D. J. White. *Markov Decision Processes*. John Wiley & Sons, 1993.
33. P. Wong and J. Gibbons. A Process Semantics for BPMN. In *Proc. of ICFEM*, volume 5256 of *LNCS*, pages 355–374. Springer, 2008.
34. P. Wong and J. Gibbons. Verifying Business Process Compatibility. In *Proc. of QSIC*, pages 126–131. IEEE, 2008.
35. M. T. Wynn, H. M. W. Verbeek, W. M. P. van der Aalst, A. H. M. ter Hofstede, and D. Edmond. Business Process Verification - Finally a Reality! *Business Process Management Journal*, 15(1):74–92, 2009.