Information Sciences 385–386 (2017) 39–54

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Formal verification of complex business processes based on high-level Petri nets

Ahmed Kheldoun[a], Kamel Barkaoui[b,*], Malika Ioualalen[a]

[a] MOVEP, Computer Science Department, USTHB, Algiers, Algeria
[b] CEDRIC-CNAM, 292 Rue Saint-Martin 75141, Cedex 03 Paris, France

ARTICLE INFO

Article history:
Received 23 October 2015
Revised 28 November 2016
Accepted 31 December 2016
Available online 31 December 2016

Keywords:
Business process modelling
BPMN
RECATNets
Conditional rewriting logic
Maude language and tool

ABSTRACT

The Business Process Modeling Notation (BPMN) has been widely used as a tool for business process modeling. However, BPMN suffers from a lack of standard formal semantics. This weakness can lead to inconsistencies, ambiguities, and incompletenesses within the developed models. In this paper, we propose a formal semantics of BPMN using recursive ECATNets. Owing to this formalism, a large set of BPMN features such as cancellation, multiple instantiation of subprocesses and exception handling can be covered while taking into account the data flow aspect. The benefits and usefulness of this modelling are illustrated through three examples. Moreover, since recursive ECATNets semantics is defined in terms of conditional rewriting logic, one can use the Maude LTL model checker to verify several behavioral properties related to BPMN models. The work presented in this document has been automated through the development of a first prototype.

1. Introduction

1.1. Overview and motivation

The standard Business Process Modeling Notation (BPMN) [19] has been established as the de-facto standard for modeling business processes. It provides a standard notation easily understandable that supports the business process management while being able to represent complex processes semantics. Nevertheless, despite the various advantages of BPMN, it suffers from a lack of formal semantics which can lead to inconsistencies, ambiguities, and incompletenesses within the developed models. Furthermore, BPMN brings additional features drawn from a range of sources including Workflow Patterns [22] which are able to define: (1) subprocesses that may be executed multiple times concurrently; and (2) subprocesses that may be interrupted as a result of exceptions. These features increase the types of semantic errors that can be found in BPMN models. As a result, many researchers proposed formal methods to build formal description and verification models of business processes. However, one of the weaknesses of these proposals is their lack of support for modeling complex BPMN business processes involving exception, cancellation and multiple instantiation of subprocesses. For that, we need an expressive modeling formalism that allows, on one hand, to specify their dynamic structure, and on the other hand, to check the control-flow correctness of these business processes while taking into account their data flow aspect.

* Corresponding author.
E-mail address: kamel.barkaoui@cnam.fr (K. Barkaoui).

http://dx.doi.org/10.1016/j.ins.2016.12.044
0020-0255/

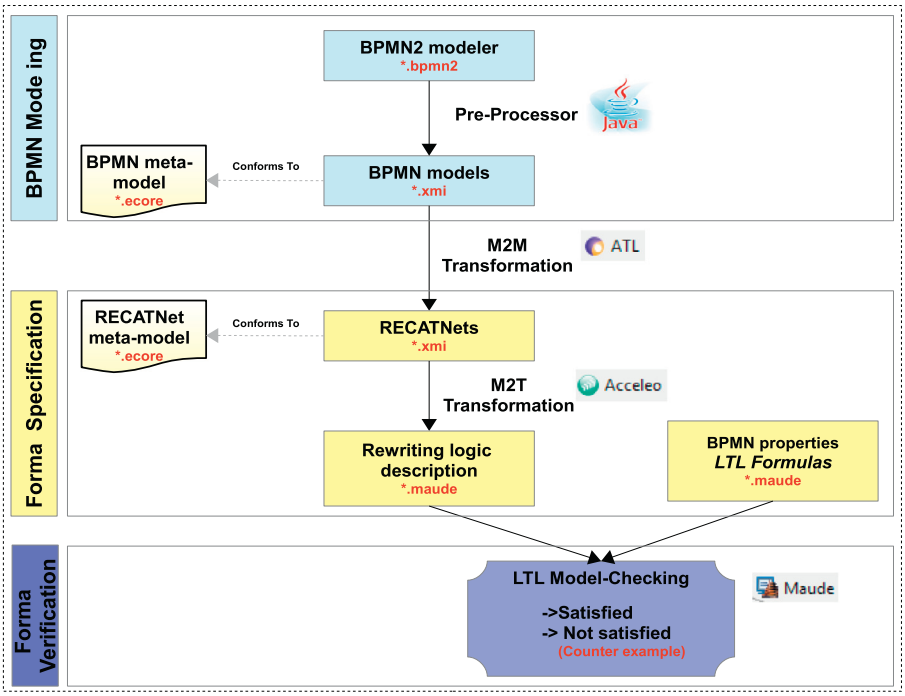


Fig. 1. Overview of the proposed approach.

In this paper, we propose the use of Recursive ECATNets [3] to cope with the formal verification of complex BPMN models. The Recursive ECATNets model offers practical mechanisms for handling the most advanced BPMN constructs (involving exception, cancellation and multiple instances). Since Recursive ECATNets semantics can be expressed in terms of conditional rewriting logic [17], we can use the Maude LTL model checker [8] to investigate several behavioral properties of business processes.

1.2. Proposed approach

The proposed approach for the verification of BPMN models is based mainly on Meta-modeling and Model Transformations. It is achieved automatically into four steps : (1) transformation of BPMN models created by a graphical editor like eclipse BPMN2 Modeler (URL: http://eclipse.org/bpmn2-modeler/) into XML Metadata Interchange (XMI) files conforming to our proposed BPMN meta-model, (2) transformation of obtained BPMN XMI files into RECATNets using the ATLAS Transformation Language (ATL) [13] where two meta-models are defined for BPMN and RECATNet (URL: http://recatnets.cnam.fr/), (3) transformation of obtained RECATNets into rewriting logic description using the transformation tool Acceleo (URL: http://www.eclipse.org/acceleo/), (4) checking the properties of business processes expressed as rewrite theories by using the Maude LTL model checker. This is summarized in Fig. 1.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 provides an overview about BPMN and Recursive ECATNets. Section 4 presents the mapping rules from BPMN to Recursive ECATNet. Section 5 shows two examples for the proposed mapping. Section 6 presents the formal semantics of the mapping rules. Section 7 presents the RECATNet semantics in terms of rewriting logic. Section 8 presents the developed prototype. Section 9 presents an overview of the verification process using the Maude LTL model checker. Finally, Section 10 concludes and gives some further research directions.

2. Related work

Many researchers have tried to deal with formal modeling and verification of business processes using BPMN models. Morimoto [18] presents an extended survey of the existing verification techniques of BPMN diagrams and compares them with each other with respect to motivations and methods. Nevertheless, none of the cited works take into account the following key features of BPMN : (1) cancellation of subprocesses; (2) parallel multi-instance subprocesses; and (3) exception handling in the context of subprocesses that are executed multiple times concurrently.

Petri nets often are a topic in verification of business processes using BPMN models. Dijkman et al. [6] propose a mapping from a core set of BPMN to labelled Petri nets. This output is represented in the PNML language [4] and can subsequently be used to verify BPMN processes by using the open source tool WofBPEL [20]. The proposed mapping for exception handling is

Table 1
Features supported by BPMN semantics.

BPMN 2.0 object	Dijkman et al. [6]	Ou-Yang & Lin [5]	Ye et al. [27]	Our Approach
Events				
Start events	X	X	X	X
End events	X	X	X	X
Timer events			X	X
Error events (catch)	X		X	X
Error events (throw)	X		X	X
Cancel events			X	X
Message events	X		X	X
Gateways				
Parallel fork (AND-Split)	X	X	X	X
Parallel join (AND-Join)	X	X	X	X
Data-based gateway (XOR-Split)	X	X	X	X
Event-based gateway (XOR-Split)	X		X	X
Merge gateway (XOR-Join)	X	X	X	X
Activities				
Task (atomic activity)	X	X	X	X
Subprocess	X		X	X
Loop activity	X		X	X
Static multiple instance activity			X	X
Dynamic multiple instance activity			X	X
Data objects				
Data object				X
Transformation into	PN	CPN	YAWL	RECATNet
***Soundness*property**	X	X		X

very complicated and does not work properly in the case where an activity within the subprocess is enabled multiple times concurrently. Ou-Yang [5], propose a Petri-net-based approach to evaluate the feasibility of a BPMN model. This approach enables to reveal deadlocks and infinite loops. It consists in manually translating the BPMN model to a modified BPEL4WS representation, and then to XML-based representation of Colored Petri nets (CPNXML) that can be verified using CPN Tools. However, only, simple BPMN contructs are taken into account.

An important piece of related work is YAWL (Yet Another Workflow Language) [1]. YAWL is a workflow language based on Petri nets but extended with additional features to facilitate the modelling of complex workflows involving cancellation and multiple instantiation of subprocesses. In [27], an approach is proposed to convert automatically business processes to YAWL nets and to verify subsequently them by YAWL verification tools like WofYAWL [23]. YAWL is expressive enough; however, what is clearly missing is the handling of data concepts, which is a key concept in business processes [16]. In contrast, in our model, the data types can be distinguished because the model used the expressive power of abstract data types. Also, modelling a synchronization of a dynamic number of process instances depending on their termination state can't be done using the multiple instantiation constructors of YAWL where the terminated process instances are synchronized following their number and not following their states. Another important weakness of YAWL is its semantics constructors, which is expressed in terms of Colored Petri nets and in terms of reset nets [7] to describe the cancellation construction. Therefore, regarding BPMN analysis, the *soundness* property is not decidable for YAWL specifications. In contrast, our model allows the modelling of cancellation constructs via cut steps execution while the *soundness* property remains decidable, if the state space generated from the specified model is finite [11]. Table 1 summarizes the BPMN features that are supported by the different semantics discussed above.

Very recently, Roa et al. [12] propose an approach based on anti-patterns for the verification of block-structured collaborative business processes specified in UP-ColBPIP language [24]. Also, the authors claim that their approach can be adapted for the case of BPMN language. The proposed approach seems very useful since it covers several complex control flow, but it presents several limits in specifying well-known constructs of BPMN language such as subprocess, dynamic multi-instances, cancellation of several instances and data object. In addition, the authors do not show how the approach can be applied for the verification of complex properties specified in temporal logic.

There are other research on transforming BPMN into π-calculus [21], CSP [26] and Maude [9]. In [21], a mapping from subset of BPMN to π-calculus is proposed. However, this mapping does not take into account error handling, which is a key feature of BPMN. In [26], CSP is used to define the BPMN semantics, followed by a refinement procedure for property checking. However, the authors do not show how can the CSP semantics be used to detect various types of errors. Also, it is unclear how data is modeled. El-Saber and Boronat [9] formalizes a subset of BPMN in Maude, but does not consider features such as : static/dynamic multiple instance and cancellation of a subprocess. Also, there is no tool provided for automatic generation of Maude description from a BPMN models.

The advantages of our proposed model are : (1) adequate for handling the most advanced BPMN constructs such as multiple instantiation, cancellation of subprocesses and exceptional behaviors (2) providing a hierarchical and modular specifi-

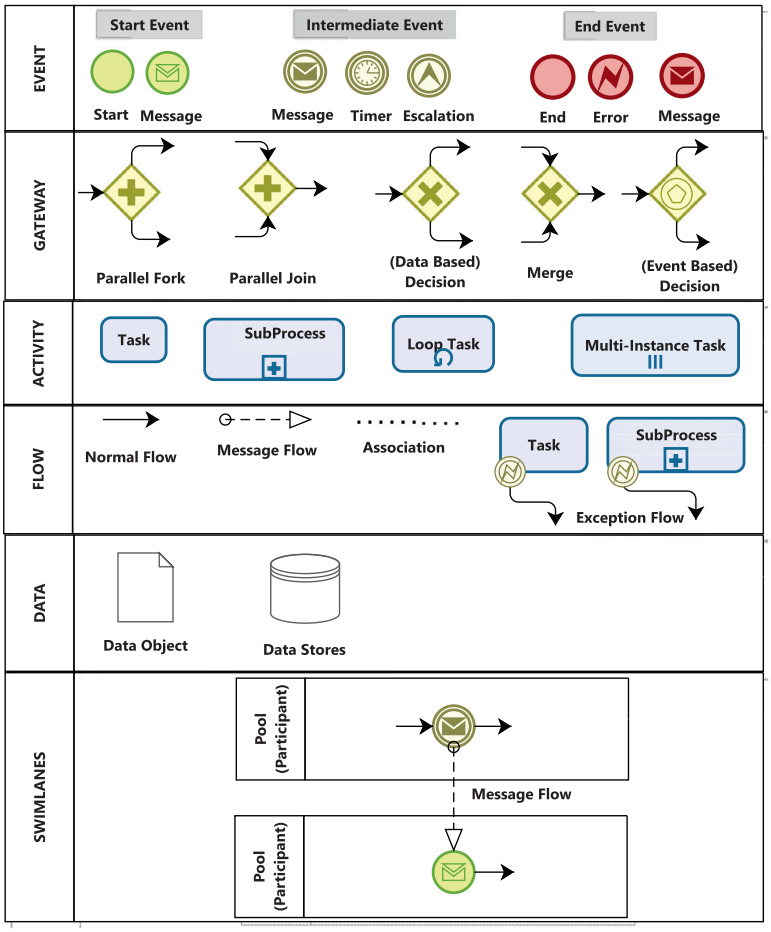


Fig. 2. A core subset of BPMN elements.

cation (3) allowing distributed execution of modeling business processes and (4) its semantic may be defined in terms of conditional rewriting logic [17] and therefore the model checker Maude [8] can be used to investigate several behavioral properties.

3. Background

This section gives background on the three related concepts. First, we summarize the BPMN 2.0 notation principles. Next, we explain the concept of Recursive ECATNets as the formal mathematical modeling language. Finally, we give an overview of system maude and its language rewriting logic.

3.1. BPMN overview

A BPMN process, which is using the core subset of BPMN elements shown in Fig. 2 is referred to as a core BPMN process. In this paper, we only consider the well-formed core BPMN models [6].

Definition 1. A core BPMN process is a tuple $B = \langle O, F, Me, Cond, \mathcal{X}, Ass \rangle$ where :

- O is a set of objects, which can be partitioned into disjoint sets of flow objects FO and data objects DO,
  - FO can be partitioned into disjoint sets of activities A, events E and gateways G,
  - A can be partitioned into disjoint sets of tasks T and subprocess invocation activities S,
  - E can be partitioned into disjoint sets of start events E^s, intermediate events E^i and end events E^e,
  - E^i can be partitioned into disjoint sets of intermediate message events E^i_M, intermediate timer events E^i_T and intermediate error events E^i_R,
  - E^e can be partitioned into disjoint sets of end non-error events E^e_N and end error events E^e_R,

– G can be partitioned into disjoint sets of parallel fork gateways G^F, parallel join gateways G^J, data-based *XOR* decision gateways G^D, event-based *XOR* decision gateways G^E and *XOR* merge gateways G^M.

- $F \subseteq FO \times FO$ is the control flow relation,
- $Me \subset eq(T \cup E^i_M \cup E^e) \times (T \cup E^i_M \cup E^s) \setminus (FO \times FO)$ is the message flow relation,
- $Cond$: $F \cap (G^D \times O) \rightarrow C$ is a function which maps sequence flows emanating from data-based *XOR* gateways to conditions[1],
- $\mathcal{X} : E^i \rightarrow A$ is a function assigning an intermediate event to an activity, such that the occurrence of the event interrupts the performance of the activity,
- $Ass \subseteq DO \times A$ is the data object association relation.

3.2. RECATNet overview

Recursive ECATNets (abbreviated RECATNets) [2,3] are a kind of high level algebraic Petri nets combining the expressive power of abstract data types and Recursive Petri nets [10]. RECATNets are proposed to specify flexible concurrent systems where functionalities of discrete event systems such as abstraction, dynamicity, preemption and recursion are preponderant.

In what follows, we denote by $Spec = \langle \Sigma, E \rangle$ an algebraic specification of an abstract data type associated to a RECATNet, where $\Sigma = \langle S, OP \rangle$ is its multi-sort signature (S is a finite set of sort symbols and OP is a finite set of operations, such that $OP \cap S = \phi$). E is the set of equations associated to $Spec$. A set of variables associated to $Spec$ is a family $X = (X_s)_{s \in S}$ with X_s, the set of variables of sort s, where $OP \cap X = \phi$. We denote by $T_\Sigma(X)$, the set of Σ-terms with variables in the set X and by $T_{\Sigma,\, s}(X)$ the set of Σ-terms with variables in the set X_s. The multisets of Σ-terms are denoted by $[T_\Sigma(X)]_\oplus$, where the multiset union operator ($\oplus$) is associative, commutative and admits the empty multiset ϕ as the identity element.

Definition 2. A recursive ECATNet [3] is a tuple $RECATNet = \langle Spec, P, T, sort, Cap, IC, CT, TC, \Omega, I, \Upsilon, ICT, K \rangle$ where:

- $Spec = (\Sigma, E)$ is a many sorted algebra, where the sorts domains are finite (with $\Sigma = (S, OP)$), and X is a set of variables associated to $Spec$,
- P is a finite set of places,
- $T = T_{elt} \cup T_{abs}$ is a finite set of transitions ($T \cap P = \phi$) partitioned into elementary and abstract ones,
- sort: $P \rightarrow S$, is a mapping called a sort i.e type assignment,
- Cap: is a P-vector on capacity places: $p \in P$, $Cap(p)$: $T_\Sigma(\phi) \rightarrow \mathbb{N} \cup \{\infty\}$,
- IC: $P \times T \rightarrow [T_{\Sigma,\, sort(p)}(X)]_\oplus$ maps a multiset of terms for every input arc,
- CT: $T \times P \rightarrow [T_{\Sigma,\, sort(p)}(X)]_\oplus$ maps a multiset of terms for every output arc (t, p) where $t \in T_{elt}$,
- TC: $T \rightarrow [T_{\Sigma,\, bool}(X)]$ maps a boolean expression for each transition,
- Ω: $P \times T_{abs} \rightarrow [T_{\Sigma,\, sort(p)}(X)]_\oplus$ maps a multiset of terms for every starting marking associated to $t \in T_{abs}$ according to place p,
- $I = I_{cut} \cup I_{pre}$ is a finite set of indices, called termination indices, dedicated to cut steps and preemptions (interruptions) respectively,
- Υ is a family, indexed by I_{cut}, of effective semi-linear sets of final markings,
- ICT: $T_{abs} \times P \times I \rightarrow [T_{\Sigma,\, sort(p)}(X)]_\oplus$ maps a multiset of terms for every output arc (t, p, i) where $t \in T_{abs}$ and $i \in I$,
- K: $T_{elt} \rightarrow T_{abs} \times I_{pre}$, maps a set of interrupted abstract transitions, and their associated termination indexes, for every elementary transition.

Let's illustrate, through the net presented in the upper part of Fig. 3, the symbols and concepts used in RECATNets definition :

1. The algebraic specification *SpecRECATNet* associated to this RECATNet defines a set of data structures represented by the following sorts : *Data, CoupleData* and *Token*. Also, the algebraic specification defines a set of variables X and constantes. For instance, *Rq* is a variable of sort *Data* (i.e. $Rq \in X_{Data}$) and *Ok* is a constante of sort *Data*,
2. Each place in the RECATNet is associated a sort. For instance, places p_0, p_1 and p_3 are associated respectively the sorts *Token, Data* and *CoupleData*,
3. An elementary transition is represented by a filled rectangle; its name is possibly followed by a set of terms $(t', i) \in T_{abs} \times I$. Each term specifies an abstract transition t', which is under the control of t, associated with a termination index to be used when interrupting t' consequently to a firing of t. For instance, t_{cancel} is an elementary transition, where its firing preempts instances of threads created by the firing of t_1 and the associated termination index is $< 2 >$.
4. An abstract transition t is represented by a double rectangles; its name is followed by the starting marking $\Omega(t)$. For instance, t_1 is an abstract transition and $\Omega(t_1) = <p_4, Rq>$ means that any thread, named refinement net, created by firing of t_1 starts with one token, i.e. *Rq* in place p_4.

[1] a condition C is a boolean function, which operates over a set of propositional variables.

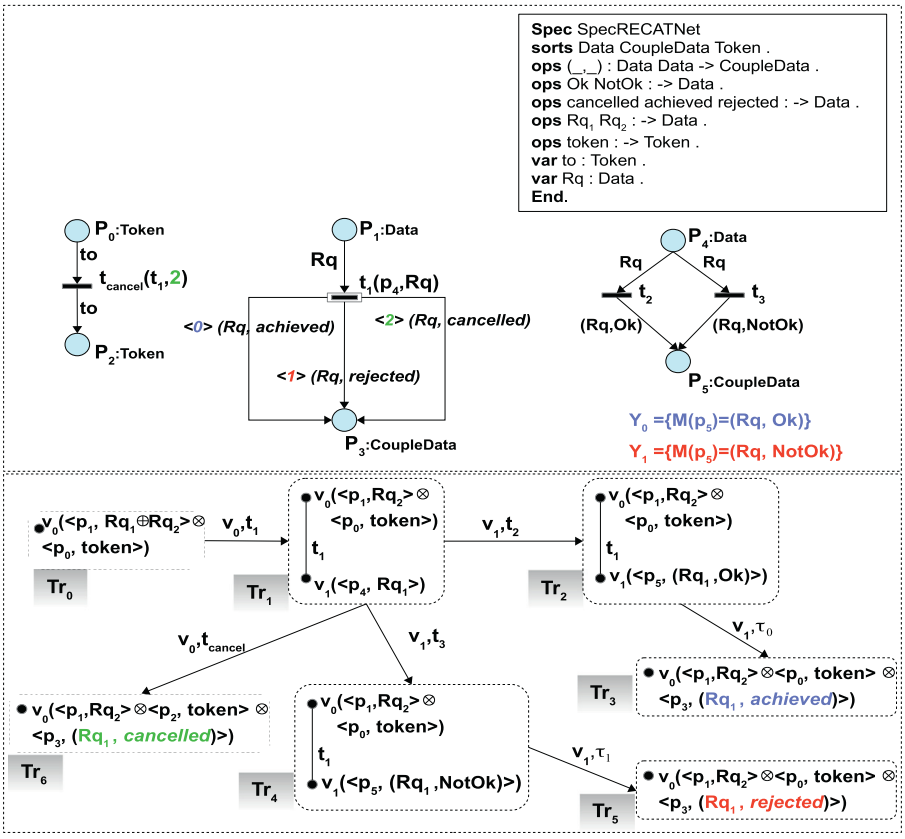


Fig. 3. Example of a RECATNet and three firing sequences.

5. Any termination set can be defined based on place marking. For instance, Y_0 specifies the final marking of threads, such that the place p_5 is marked by a couple of (Rq, Ok).
6. The set I of termination indices is deduced from the indices used to subscript the termination sets and from the indices bound to elementary transitions i.e. interruption. For the example, $I = \{0, 1, 2\}$, more precisely, $I_{cut} = \{0, 1\}$ and $I_{pre} = \{2\}$.

Informally, a RECATNet generates during its execution a dynamical tree of marked threads called *extended marking* [3], which reflects the global state of such net. This latter denotes relation between generated threads, where each one of them having its own execution context.

Let's now consider again the net of Fig. 3 with initial marking $< p_1, Rq_1 \oplus Rq_2 > \otimes < p_0, token >$ to illustrate the firing sequence notion. The graphical representation of any extended marking Tr is a tree, where an arc $v_1(m_1) \rightarrow v_2(m_2)$ labelled by t_{abs} means that v_2 is a child of v_1 created by firing the abstract transition t_{abs}, and m_1 (resp. m_2) is the marking of v_1 (resp. v_2). In the lower part of Fig. 3, note that the initial extended marking Tr_0 is reduced to a single node v_0, whose marking is $< p_1, Rq_1 \oplus Rq_2 > \otimes < p_0, token >$. From the initial extended marking Tr_0, the abstract transition t_1 is enabled; its firing is achieved as follows: the consumption of tokens specified by the precondition of t_1 (i.e. $IC(t_1)$ from the place p_1) and the creation of a thread modelled by a refinement net, which starts its evolution with an initial marking $\Omega(t_1) = (p_5, Rq)$. The obtained extended marking after firing the abstract transition t_1 is denoted by Tr_1. Note that the extended marking Tr_1 contains a new node v_1 marked by the starting marking $\Omega(t_1)$. Then, the firing of the elementary transition t_2 from the node v_1 of Tr_1 leads to an extended marking Tr_2, having the same structure as Tr_1, but only the marking of node v_1 is changed. From the node v_1 in Tr_2, the cut step τ_0 is enabled; its firing leads to an extended marking Tr_3 by removing the node v_1 and change the marking on its node father, i.e. v_0 by adding $ICT(t_1, 0) = (p_3, (Rq_1, achieved))$. Similarly, from the node v_1 in Tr_1, the elementary transition t_3 is enabled; its firing leads to an extended marking Tr_4 having the same structure as Tr_1, but only the marking of node v_1 is changed. Now, from the node v_1 in Tr_4, the cut step τ_1 is enabled; its firing leads to an extended marking Tr_5 by removing the node v_1 and change the marking on its node father, i.e. v_0, by adding $ICT(t_1, 1) = (p_3, (Rq_1, rejected))$. Also, from the node v_0 in Tr_1, the elementary transition t_{cancel} with associated interruption $(t_1, 1)$ is enabled; its firing leads to an extended marking Tr_6 by removing the node v_1 and changing the marking on its node father, i.e. v_0, by adding $ICT(t_1, 2) = (p_3, (Rq_1, cancelled))$.

The analysis of a RECATNet is based on constructing its state space, named extended reachability graph, which is used for checking properties such as reachability, deadlock and liveness. Furthermore, RECATNets semantics can be expressed in rewriting logic, and the Maude LTL model checker can be used to check LTL properties.

3.3. Maude and rewriting logic

Maude is a high-performance reflective language and system supporting executable specification and declarative programming in rewriting logic [17]. It has been developed at SRI International (URL: http://maude.cs.uiuc.edu/). A system, under Maude, is represented using membership equational logic describing its set of states and a set of rewrite rules representing its state transitions. Maude's basic programming statements are equations and rules, and have in both cases a simple rewriting semantics in which, instances of the left-hand side pattern are replaced by corresponding instances of the right-hand side. One aim using Maude is its LTL model checker, which can be used to verify properties specified in LTL when the set of reachable states from an initial state in a system module is finite.

4. Mapping BPMN into RECATNets

In this section, we show the mapping rules from BPMN constructs, which contain events, activities, gateways, exception, cancellation, message flows and data objects to RECATNet modules. Owing to the formalism of RECATNet, the large set of advanced BPMN features can be covered straightforwardly.

4.1. Mapping events and gateways

Events represented something that happened in the process and required a reaction. A *start event* indicates where a particular process starts. So, in Fig. 4(a), *start event* is mapped into an elementary transition with one input place. An *end event* ends the flow of the process and it has no successors. Therefore, in Fig. 4(b), an *end event* is mapped into one transiton with only one output place. Intermediate events indicate where an event occur somewhere between the start and end of a process. As shown in 4(c), an intermediate event, such as *timer event* is mapped into an elementary transition.

Gateways are used to control the divergence and convergence of sequence flows in a Process. A *parallel Fork* gateway, known as **AND-Split**, allows to split one sequence flow into two or more paths that can run in parallel within the process. So, in Fig. 4(d), a *parallel Fork* gateway is mapped into one elementary transition connected with only one input place and a set of output places as paths that can run in parallel. A diverging exclusive *event-based* gateway, known as **XOR-Split**, is used to create alternative paths within a process flow, where only one of the alternatives will be chosen. Therefore, in Fig. 4(h), this *event-based* gateway is mapped into a set of elementary transitions as the number of alternative paths, sharing in common one input place. In our previous paper [14], we gave details explanations about the mapping rules of the other types of gateways. Note that, the places with dashed borders are used to link the different generated RECATNet modules. These dashed places are created by mapping of sequence flows, except for the output flows of *event-based* gateways, as shown in Fig. 4(i).

4.2. Mapping activities

An activity can be a *task* or a *subprocess*. A *task* is an atomic activity, standing for work to be performed. Therefore, a *task* is mapped as shown in Fig. 4(j), into an elementary transition. The firing of the elementary transition models the execution of the task. A *subprocess* may be viewed as an independent BPMN process. In Fig. 4(k), we depict the mapping of calling a *subprocess* via a subprocess invocation activity. The invocation of a subprocess is mapped into an abstract transition $t_{CallSubProc}$, where its starting marking is the initial marking of the RECATNet module associated to the invoked subprocess. Also, a semi-linear set of final markings is defined, which indicates the end flows of the *subprocess*. For instance, the set $\Upsilon_0 = \{M(p_{eSubProc}) > 0\}$ indicates that a *subprocess* ends when the place $p_{eSubProc}$ is marked. Once a final marking is reached by the invoked subprocess, according to the semi-linear set of final markings Υ_0, a cut step closes the corresponding RECATNet module, and produces tokens in the appropriate output place of the abstract transition $t_{CallSubProc}$.

In BPMN, an activity may have attributes specifying its additional behavior, such as looping and parallel multiple instances. In [14], we have defined the mapping rules of these advanced activities into RECATNets.

4.3. Exception handling

In BPMN, exception handling is captured by exception flow. An exception flow originates from an exception event attached to the boundary of an activity, which is either a task or a subprocess. Fig. 4(n) depicts the mapping of an exception associated with a task and a subprocess. The RECATNet associated to a subprocess shows the mapping of an exception associated with an atomic activity i.e. *Task*. The occurrence of the exception may only interrupt the normal flow at the point when it is ready to execute the task. Hence, the mapping of an exception associated with a task is an elementary transition t_{exTask}, where its firing will avoid to run the atomic activity *Task* and generates a token representing the type of exception in place p_{exTask}.

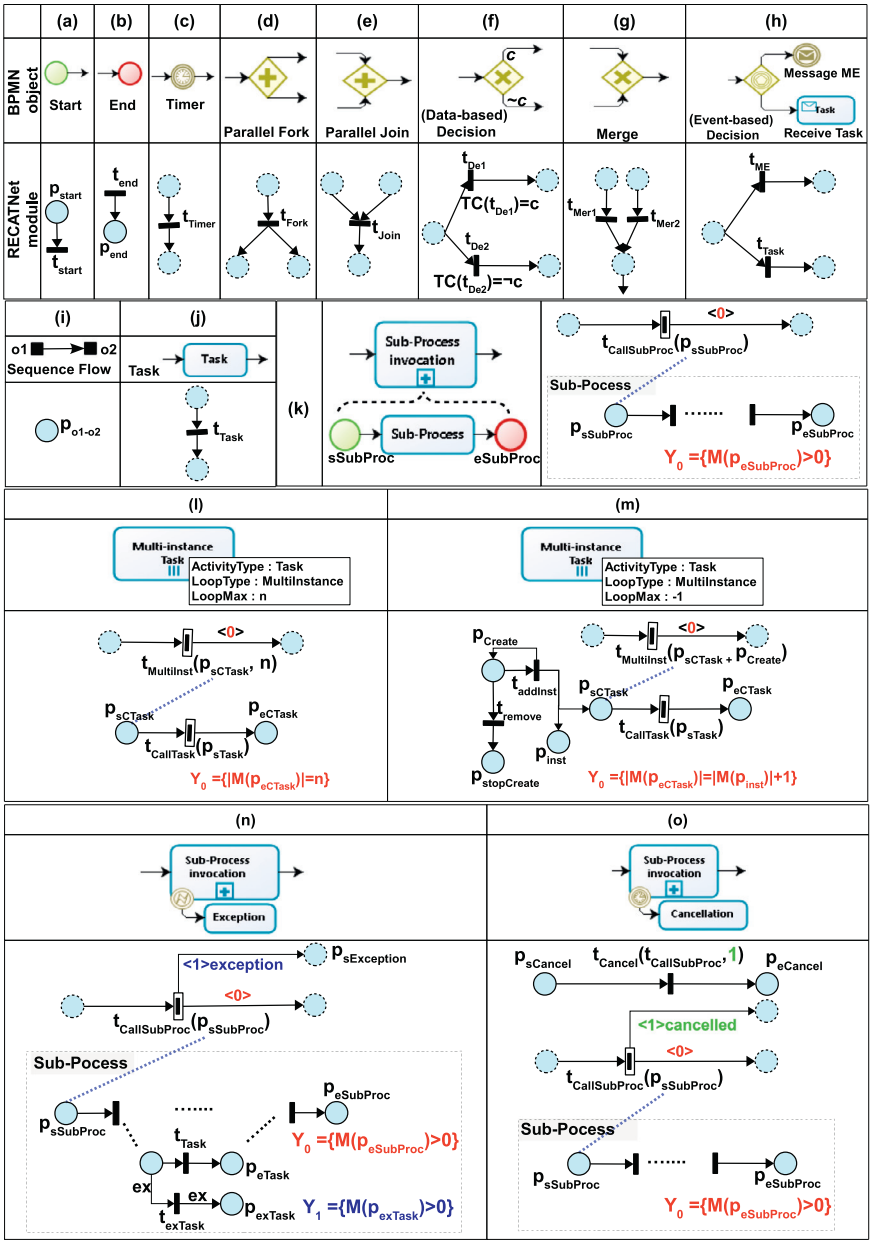


Fig. 4. Mapping of BPMN features (events, activities, gateways, exception and cancellation) to RECATNets. The symbol ■ is used to express any valid flow object of BPMN model (i.e., an *activity*, or an *event*, or a *gateway*).

For the exception handling associated to a subprocess, the occurrence of the exception event will interrupt the execution of the normal flow within the subprocess, when it is active (running). This mapping is straightforward into RECATNet. Firstly, a semi-linear set of final markings is defined, it indicates the occurring of such exception in a subprocess. For instance, as shown in Fig. 4(n), the semi-linear set of final markings $\Upsilon_1 = \{M(p_{exTask}) > 0\}$ indicates the occurring of an exception associated to the atomic activity *Task*. So, the subprocess may end with two types of terminations. The first way of termination is when the subprocess ends properly by marking the place $p_{eSubProc}$, which needs to define the set of terminations Υ_0. The second way of termination is when the subprocess ends upon the occurrence of an exception, i.e. marking the place p_{exTask} which needs to define the set of terminations Υ_1. Secondly, we need to add another output place for the abstract transition, which models the invocation of the subprocess to capture the exception. For instance, as shown in Fig. 4(n), the asbtract transition $t_{CallSubproc}$, which can invoke the subprocess, has a second output place $p_{sException}$, with index of termination $< 1 >$. It means that, when the subprocess reachs a final marking in Υ_1, a cut step interrupts the subprocess and produces tokens in the output place $p_{sException}$ associated to the index of termination $<1>$.

In addition, the mapping of cancellation activity into RECATNet, as shown in Fig. 4(o), is straightforward, thanks to the preemption and cut step which are key features of RECATNet. The cancellation of a subprocess is very difficult for mapping using other types of Petri net since, this requires to freeze each transition of the model (i.e. removing all tokens from pre-condition places of transitions). In RECATNet, each node in the extended marking is associated to a running subprocess. In fact, for cancellation construct, we remove the node models the behavior of the subprocess instead of removing tokens from places of that subprocess. In addition, the termination of the removed subprocess is captured easily by its parent process. More details of this mapping rule can be found in [14].

4.4. Mapping message flows

The BPMN 2.0 specification defines a message flow as follows [19] : "*a Message Flow is used to show the flow of messages between two participants that are prepared to send and receive them*". For example, a collaborative emergency response process [15] is a typical business process where message flows are necessary. In fact, in this type of business model, several emergency organizations can involve and need to collaborate between them.

In an abstract way, a message flow can be mapped to a place with an incoming arc from the transition modelling a send action and an outgoing arc to the transition modelling a receive action, as shown in Fig. 5(a) and (b). The interchenge message is represented in RECATNet by an algebraic term M of sort *MessageType*. We distinguish a special case in mapping a message flow, where a start event receives a message. In this case, for each received message, a business process is instantiated. Indeed, as shown in Fig. 5(c) and (d), the message flow is mapped to an arc linking the transition that models the send message and the place that models the start event. Another special case in mapping a message flow is, when a task in a subprocess *SP* receives a message. In this case, as shown in Fig. 5(e), the message flow is mapped to a place with an incoming arc from the transition that models the send message and an outgoing arc to the abstract transition t_{CallSP} that models the call of a subprocess *SP*. Furthermore, another place p_{SP-Tr} of sorte *MessageType* is added as an input of transition t_{Tr} that models the receive task *Tr*. In fact, the place p_{SP-Tr} will be marked by the received message M when calling the subprocess *SP* by firing the above abstract transition (i.e. $\Omega(t_{CallSP}) = (p_{sSP} \otimes [M]p_{SP-Tr})$).

4.5. Mapping data objects

When executing a business process, there may be data produced, either during or after the end of the process. For example, a successful execution of the *Place Order task* will produce data like *purchase order, invoice, receipt*, etc. In BPMN, data can be modeled by several types of data objects such as *DataObjects, DataInputs, DataOutputs* and *DataStores*. They share common general attributes with other objects like *name* and *in/out* associations which connect it to the other objects in the process. Also, each data object is in a certain state at any time during the execution of the business process. This state changes through the execution of activities. In Fig. 6, we propose a mapping of BPMN data objects to RECATNets. All data objects have a common *name* are mapped to one place associates a sort *DOState*. In addition, a directed association from a data object to an activity is mapped as a precondition of the activity and an association from an activity towards a data object is mapped as postcondition of the activity. In Fig. 6(a), the activity *Task* reads and changes the state of the same data object. In Fig. 6(b), the activity *Task* reads a state from a data object and changes the state of another data object. In Fig. 6(c), the activity *Task* reads a non specified state for the input data object. In this case, a variable S is used as an input condition IC of the input arc of the activity.

5. Case studies

In this section, we illustrate our approach for mapping BPMN processes into RECATNets through three examples, where a translation to Petri nets is not feasible.

5.1. Travel request process

Fig. 7(a) depicts the business process for Travel Request process. First, the requester enters the information related to the travel, then the administrative department has to manage the bookings for the employee and send the information related to them, once they have been confirmed. The administrative department can manage the car, hotel and flight bookings simultaneously as requested by the employee. When completed, the subprocess finishes. However, many situations can arise during the booking process. Suppose that the administrative department has successfully confirmed the car and hotel booking, but when the flight is going to be booked, an error arises (connection error, etc.). Thus, the sub-process will have to be finished and an exception flow has to be enabled for the main process. In addition, to make the travel request process more flexible, the employee must be able to cancel the booking process at any moment, if necessary. The RECATNet derived from the travel request process is depicted in Fig. 7(b). The obtained RECATNet contains one abstract transition $t_{CallBook}$, where its firing will call the sub-net modeling the booking process. As shown in Fig. 7(b), the sub-net associated to the booking process models the car, hotel and flight bookings by three elementary transitions $t_{CarBook}$, $t_{HotelBook}$ and $t_{FlightBook}$. Then, to each type of booking is associated an elementary transition to model exceptions. For instance, the elementary transition $t_{exFlightBook}$ models the exceptions occurred during flight booking. The abstract transition $t_{CallBook}$ has three outgoing labelled

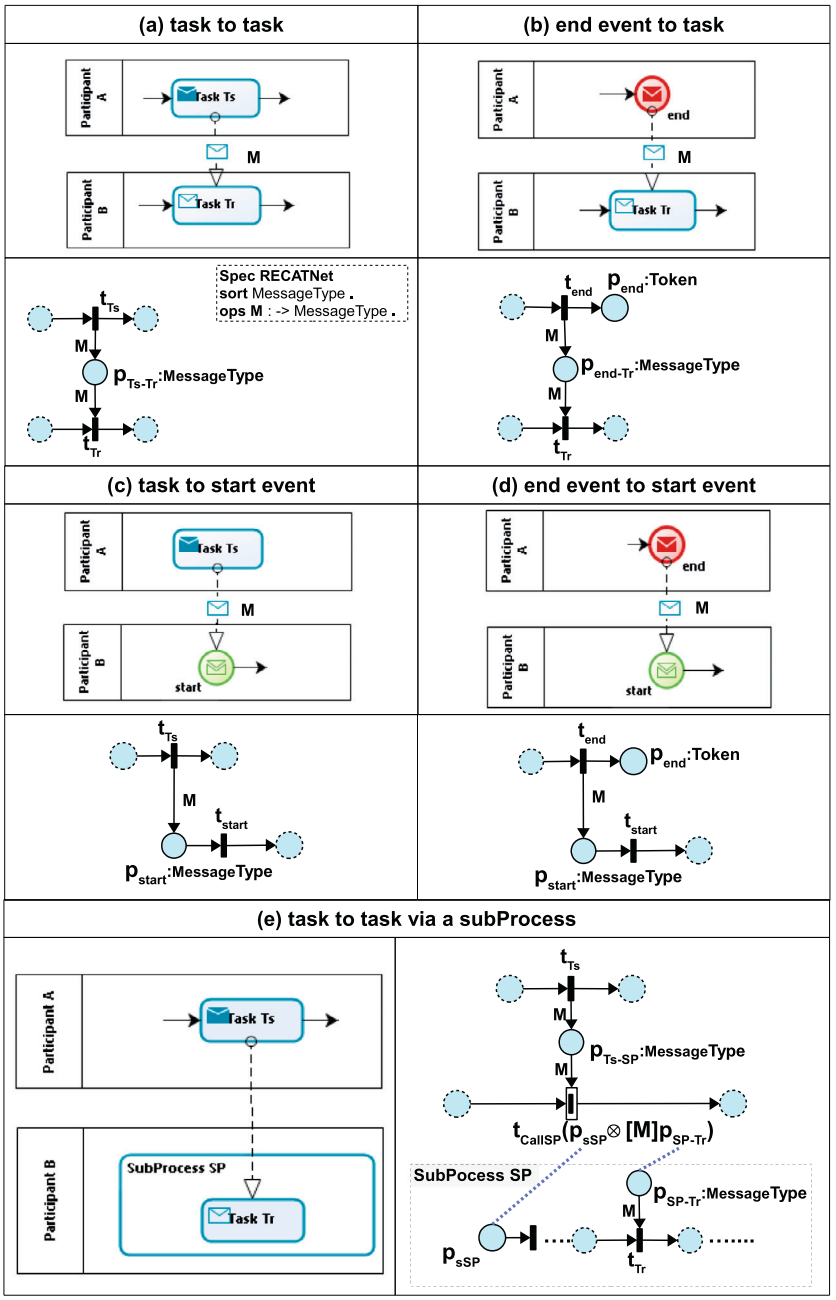


Fig. 5. Mapping BPMN message flows to RECATNets.

arcs. The arc labelled by the termination index $< 0 >$ means that the bookings process ends successfully. The arc labelled by the termination index $< 1 >$ *BookError* means that the bookings process ends with failure and the expected error *BookError* is returned. The arc labelled by the termination index $< 2 >$ *Cancelled* means that the Bookings process is cancelled when firing the elementary transition $t_{CancelBook}$ associated a term $K(t_{CancelBook}) = (t_{CallBook}, 2)$.

5.2. Discussion cycle process (multiple instance activity)

The BPMN diagram in Fig. 8(a) depicts the business process for *Discussion Cycle* adapted from [25]. The *Discussion Cycle* shows a cyclic process for resolving issues by email discussion. It starts with a task for the manager of the issue list to announce issues to working groups. Then, the manager assigns a chair to every working group to moderate the e-mail

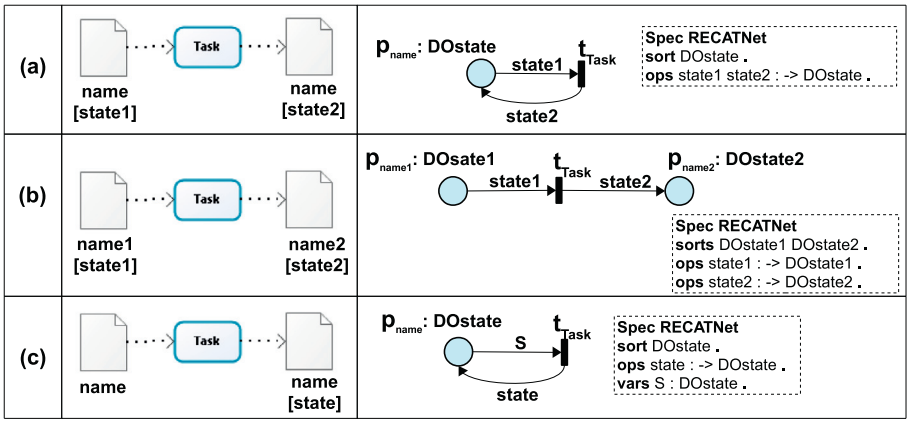
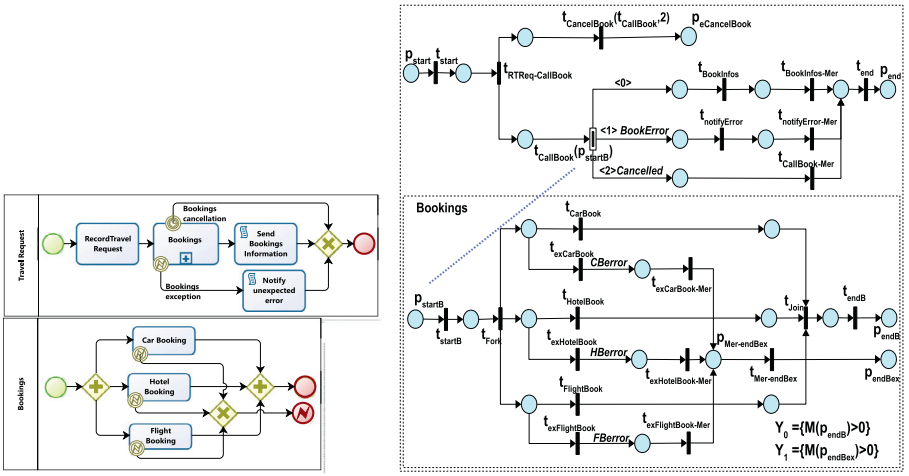
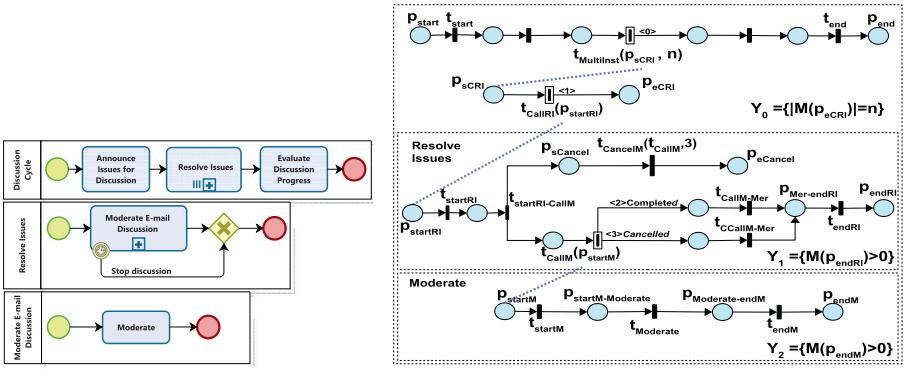


Fig. 6. Mapping BPMN data objects to RECATNets.



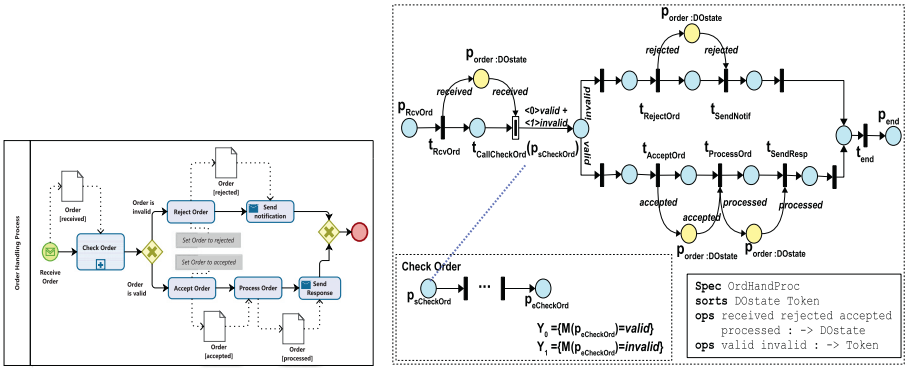
(a) Travel Request process (b) Derived RECATNet

Fig. 7. Travel request process and its derived RECATNet.



(a) Discussion Cycle process (b) Derived RECATNet

Fig. 8. Discussion cycle process and its derived RECATNet.



(a) Order Handling process (b) Derived RECATNet

Fig. 9. Order handling process and its derived RECATNet.

discussion, and the working groups start discussing the issues. There are (n) chairs in this organization. However, The discussion process may be interrupted when a specific deadline is expired.

The RECATNet derived from the Discussion Cycle process is depicted in Fig. 8(b). The firing of the abstract transition $t_{MultiInst}$ will create a sub-net, which starts by (n) tokens in place p_{sCRI}. The marking of this place makes the abstract transition t_{CallRI} enabled (n) times, where each firing of this transition will create an instance of resolve issues process. The abstract transition t_{CallM} will initialise a new instance of the discussion process. This abstract transition has two outgoing labelled arcs. The arc labelled by the termination index $< 2 >$ *Completed* means that the discussion process ends successfully. The arc labelled by the termination index $< 3 >$ *Cancelled* means that the discussion process is interrupted by firing the elementary transition $t_{CancelM}$, associated an interruption term $K(t_{CancelM}) = (t_{CallM}, 3)$. This type of BPMN process is not trivial to model by other types of Petri nets.

5.3. Order handling process (handling data)

A sample order handling process involving data objects is depicted in Fig. 9(a). In this process, the start message event occurs when an order is received. This message contains a data object *Order* in state *received*, indicated by the association from the start event to that data object. Then, the received order is checked by the subprocess *Check Order*. Two possibilities can be distinguished : if the order is *invalid*, the order is rejected and a message is sent. If the order is *valid*, the order is accepted then processed and a response is sent. Notice that the *Process Order* activity reads the order data object in state *accepted* and changes the state of that data object to *processed*.

The RECATNet derived from the Order Handling process is depicted in Fig. 9(b). In the RECATnet, all places labeled p_{order} models the data object *Order*. At any time, during the execution of the model, the place p_{order} may be marked by one of the possible states of the data object *Order*, which are specified by the set of atomic terms of sort *DOstate*. For instance, when firing the elementary transition t_{RcvOrd} (resp. $t_{RejectOrd}$), the place p_{order} will be marked by the state *received* (resp. *rejected*). Notice that the elementary transition $t_{ProcessOrd}$ may be enabled if the place p_{order} is marked by the state *accepted*. The firing of the elementary transition $t_{ProcessOrd}$ changes the marking of the place p_{order} by the state *processed*.

6. Formal semantics of mapping BPMN into RECATNet

In this section, we formally define the mapping of BPMN to RECATNet. In order to facilitate the definition, we introduce the following functions : The function $in(x) = \{y \in O | (y, x) \in F\}$ returns the input BPMN objects of a BPMN object x, the function $out(x) = \{y \in O | (x, y) \in F\}$ returns the output BPMN objects of a BPMN object x, the function *endsSubProc*(S) returns the set of end events in a BPMN subprocess associated to invocation activity S, the function *startSubProc*(S) returns the start event in a BPMN subprocess associated to invocation activity S and the function *getTermIndex*() returns a natural number, which represent the index of termination.

Definition 3. Let $B = \langle O, F, Me, Cond, \mathcal{X}, Ass \rangle$ be a core BPMN process. Without considering activities attributes, the communication between interacting processes and data objects, B can be mapped to a $RECATNet = \langle Spec, P, T, sort, Cap, IC, CT, TC, \Omega, I, \Upsilon, ICT, K \rangle$ where:

start event $start \in E^s$	$P = P \cup \{p_{start}\},\ T_{elt} = T_{elt} \cup \{t_{start}\},\ IC = IC \cup \{(p_{start}, t_{start}, token)\},\ CT = CT \cup \{(t_{start}, p_{(start,y)}, token) \mid y \in out(start)\}$
end event $end \in E^e$	$P = P \cup \{p_{end}\},\ T_{elt} = T_{elt} \cup \{t_{end}\},\ IC = IC \cup \{(p_{(x,end)}, t_{end}, token) \mid x \in in(end)\},\ CT = CT \cup \{(t_{end}, p_{end}, token)\},$ $\Upsilon = \Upsilon \cup \{(p_{end} > 0, getTermIndex()) \mid \exists_{x \in S} : end \in endsSubProc(x)\}$
sequence flow $(x, y) \in F$	$P = P \cup \{p_{(x,y)}\}$
fork $x \in G^F$	$T_{elt} = T_{elt} \cup \{t_x\},\ IC = IC \cup \{(p_{(x,y)}, t_y, token) \mid y \in in(x)\},\ CT = CT \cup \{(t_x, p_{(x,y)}, token) \mid y \in out(x)\}$
join $x \in G^J$	$T_{elt} = T_{elt} \cup \{t_x\},\ IC = IC \cup \{(p_{(x,y)}, t_y, token) \mid y \in in(x)\},\ CT = CT \cup \{(t_x, p_{(x,y)}, token) \mid y \in out(x)\}$
data-decision $x \in G^D$	$T_{elt} = T_{elt} \cup \{t_{(x,y)} \mid y \in out(x)\},\ TC = TC \cup \{(t_{(x,y),Cond(x,y))} \mid y \in out(x)\},\ IC = IC \cup \{(p_{(z,x)}, t_{(x,y)}, token) \mid y \in out(x) \wedge z \in in(x)\},$ $CT = CT \cup \{(t_{(x,y)}, p_{(x,y)}, token) \mid y \in out(x)\}$
event-decision $x \in G^E$	$T_{elt} = T_{elt} \cup \{t_{(x,y)} \mid y \in out(x)\},\ IC = IC \cup \{(p_{(z,x)}, t_{(x,y)}, token) \mid y \in out(x) \wedge z \in in(x)\},$ $CT = CT \cup \{(t_{(x,y)}, p_{(x,y)}, token) \mid y \in out(x)\}$
merge $x \in G^M$	$T_{elt} = T_{elt} \cup \{t_{(x,y)} \mid y \in in(x)\},\ IC = IC \cup \{(p_{(y,x)}, t_{(x,y)}, token) \mid y \in in(x)\},$ $CT = CT \cup \{(t_{(x,y)}, p_{(x,z)}, token) \mid y \in in(x) \wedge z \in out(x)\}$
intermediate event $x \in E^i$	$T_{elt} = T_{elt} \cup \{t_x \mid in(x) \neq \phi\},\ IC = \{(p_{(y,x)}, t_x, token) \mid y \in in(x)\},\ CT = CT \cup \{(t_x, p_{(x,y)}, token) \mid y \in out(x)\}$
task $x \in T$	$T_{elt} = T_{elt} \cup \{t_x\},\ IC = IC \cup \{(p_{(y,x)}, t_x, token) \mid y \in in(x)\},\ CT = CT \cup \{(t_x, p_{(x,y)}, token) \mid y \in out(x)\}$
subprocess $x \in S$	$T_{abs} = T_{abs} \cup \{t_x\},\ \Omega = \Omega \cup \{(t_x, p_{start}, token) \mid start = startSubProc(x)\},\ IC = IC \cup \{(p_{(y,x)}, t_x, token) \mid y \in in(x)\},$ $ICT = ICT \cup \{(t_x, p_{(x,y)}, index, token) \mid y \in out(x) \wedge \exists_{cs(p_{end},index) \in \Upsilon} end \in endsSubProc(x) \wedge end \in E^e_N\}$
exception@task $E^i_R@T$	$T_{elt} = T_{elt} \cup \{t_{excp} \mid excp \in E^i_R \wedge in(excp) = \phi \wedge \exists_{y \in T}\ y = \mathcal{X}(excp)\},$ $IC = IC \cup \{(p_{(x,y)}, t_{excp}, exception) \mid y \in T \wedge x \in in(y) \wedge excp \in E^i_R \wedge y = \mathcal{X}(excp)\},$ $CT = CT \cup \{(t_{excp}, p_{(excp,y)}, exception) \mid excp \in E^i_R \wedge \exists_{x \in T}\ x = \mathcal{X}(excp) \wedge y \in out(excp)\}$
exception@subprocess $E^i_R@S$	$ICT = ICT \cup \{(t_y, p_{(excp,z)}, index, exception) \mid y \in S \wedge excp \in E^i_R \wedge y = \mathcal{X}(excp) \wedge z \in out(excp) \wedge \exists_{cs(p_{end},index) \in \Upsilon} end \in$ $endsSubProc(y) \wedge end \in E^e_R\}$
cancellation@task $E^i_T@T$	$T_{elt} = T_{elt} \cup \{t_{cancel} \mid cancel \in E^i_T \wedge in(cancel) = \phi \wedge \exists_{y \in T}\ y = \mathcal{X}(cancel)\},$ $IC = IC \cup \{(p_{(x,y)}, t_{cancel}, cancelled) \mid y \in T \wedge x \in in(y) \wedge cancel \in E^i_T \wedge y = \mathcal{X}(cancel)\},$ $CT = CT \cup \{(t_{cancel}, p_{(cancel,y)}, cancelled) \mid cancel \in E^i_T \wedge \exists_{x \in T}\ x = \mathcal{X}(cancel) \wedge y \in out(cancel)\}$
cancellation@subprocess $E^i_T@S$	$P = P \cup \{p_{scancel}, p_{ecancel} \mid cancel \in E^i_T \wedge \exists_{y \in S}\ y = \mathcal{X}(cancel)\},$ $T_{elt} = T_{elt} \cup \{t_{cancel} \mid cancel \in E^i_T \wedge in(cancel) = \phi \wedge \exists_{y \in S}\ y = \mathcal{X}(cancel)\},$ $IC = IC \cup \{(p_{scancel}, t_{cancel}, cancelled) \mid cancel \in E^i_T \wedge \exists_{x \in S}\ x = \mathcal{X}(cancel)\},$ $CT = CT \cup \{(t_{cancel}, p_{ecancel}, cancelled) \mid cancel \in E^i_T \wedge \exists_{x \in S}\ x = \mathcal{X}(cancel)\},$ $\Upsilon = \Upsilon \cup \{(\phi_c, getTermIndex()) \mid \exists_{s \in S}\ \exists_{c \in E^i_T} s = \mathcal{X}(c)\},\ ICT = ICT \cup \{(t_y, p_{(cancel,z)}, index, cancelled) \mid y \in S \wedge cancel \in E^i_T \wedge y =$ $\mathcal{X}(cancel) \wedge z \in out(cancel) \wedge \exists_{cs(\phi_c,index) \in \Upsilon} c = cancel\},$ $K = K \cup \{(t_{cancel}, t_x, index) \mid cancel \in E^i_T \wedge x \in S \wedge x = \mathcal{X}(cancel) \wedge \exists_{cs(\phi_c,index) \in \Upsilon} c = cancel\}$

7. From RECATNet to rewriting logic

RECATNets benefit from their definition in terms of rewriting logic. A set of rewriting rules has been introduced in [2,3] to express the semantics of RECATNet in terms of rewriting logic. In fact, a RECATNet is defined as conditional rewrite theory as follows. The distributed state, i.e. the extended marking, of a RECATNet is described, in a recursive way, by the term $Th = [M, tabs, ThreadChilds]$ of sort *Thread* where M, of sort *Marking*, represents the internal marking of thread. The term *tabs* represents the name of the abstract transition, whose firing (in its thread father) gave birth to the thread *Th*. Note that, the root thread is not generated by any abstract transition, so the abstract transition which gave birth to it, is represented by the constant *nullTrans*. The term *ThreadChilds* represents a finite multiset of threads generated by the firing of abstract transitions in the thread *Th*. We denote by the constant *nullThread*, the empty thread. Moreover, each transition firing and cut step execution is expressed by conditional rewrite rule. Due to lack of space, we explain through the example of travel request process, how an abstract transition is expressed in a conditional rewriting rule. In Fig. 10, the rewriting rule in the lines 385 – 395 $rl[t - CallBook]$ describes the firing of the abstract transition $t - CallBook$. In fact, this rewriting rule requires that the left-hand side is an extended marking, where the place $p - RTReq - CallBook$ is marked and yields to an extended marking i.e. the right-hand side, where a new thread $[< p - startB; token >, t - CallBook, nullThread]$ is created and can begin its own token game with one token in place $p - startB$, which represents the *starting marking* attached to the abstract transition $t - CallBook$.

8. Prototype

In order to automate the approach presented in this paper, we developed a java tool named *BPMNChecker* on the basis of the definite meta-models for BPMNs and RECATNets, besides a set of mapping rules, that have been formally defined, previously. The tool uses standard file formats to keep it as open as possible. It is freely available at (URL: http://recatnets.cnam.fr/). In Fig. 11, we show the screenshot of our tool and the RECATNet graphical model generated from the BPMN model of travel request process of Fig. 7(a).

The developed prototype allows by (1) transforming a BPMN model file generated by the graphical editor eclipse BPMN2 Modeler into BPMN XMI files conforming to our proposed BPMN meta-model, (2) transforming the BPMN XMI files into RECATNet XMI files following the mathematical specification of the mapping defined in Section 6, by using the ATLAS Trans-

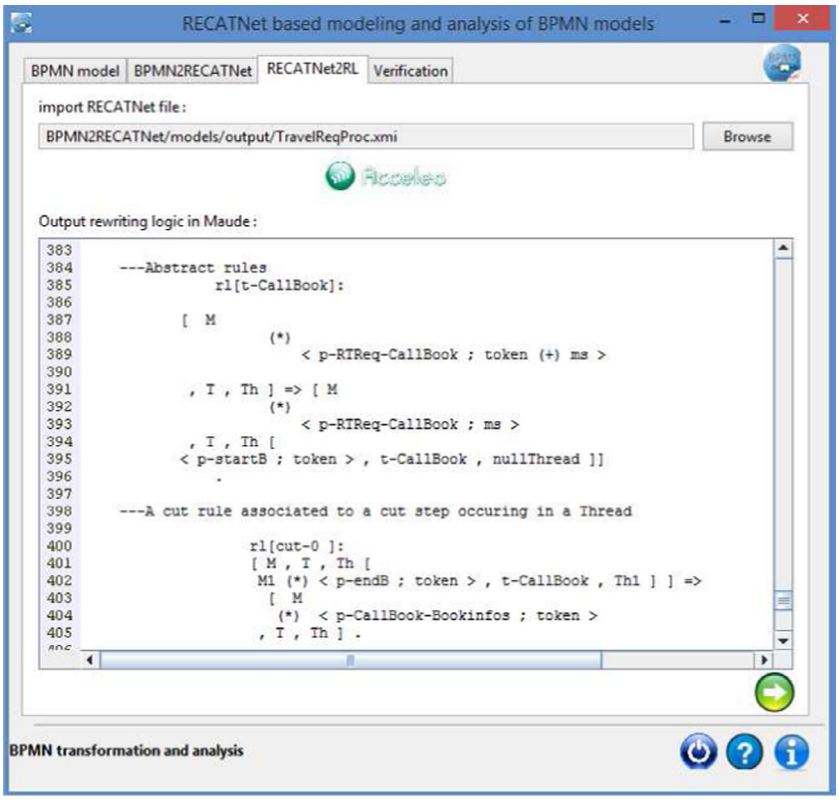


Fig. 10. The screenshot of the prototype showing the rewriting logic file generated from the BPMN model of travel request process of Fig. 7(a).

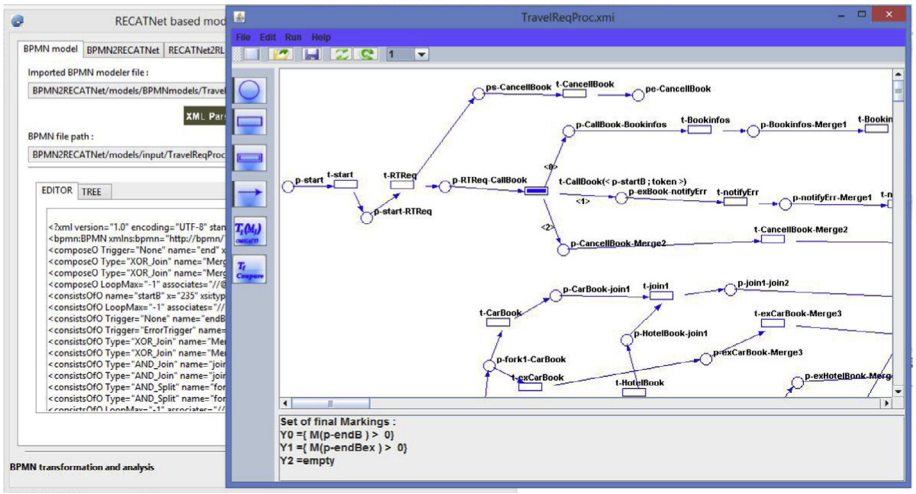


Fig. 11. The screenshot of the prototype shows the RECATNet graphical model generated from the BPMN model of travel request process of Fig. 7(a).

formation Language (ATL), (3) providing a graphical tool allowing to display, edit, modify and pre-analyse the obtained RECATNets (4) generating the rewriting logic description from the RECATNet XMI files following the set of mapping rules described in Section 7, by using the transformation tool Acceleo.

Since we obtain a rewriting logic description from RECATNets, we can benefit from the use of the Maude LTL model checker for verification of properties on the finite generated state space related to finite BPMN models.

```
Maude> in BPMN-RECATNet/MAIN.maude .

reduce in RECATNET-CHECK : modelCheck(initialState-TRP, <> finalState) .
rewrites: 1501546 in 3760ms cpu (3791ms real) (399322 rewrites/second)
result Bool: true
```

Fig. 12. Verification the *proper termination* property using Maude LTL model checker.

```
Maude> in BPMN-RECATNet/MAIN.maude .

reduce in RECATNET-CHECK : modelCheck(initialState-TRP, [](BookingInvoked =>
   <> (Booking(success) \/ Booking(exception)))) .
rewrites: 62 in 0ms cpu (0ms real) (~ rewrites/second)
result ModelCheckResult: counterexample({...
{[< p-start ; ems >(*)<  p-end ; ems >(*)< p-sCancelBook ; TReq2 >(*)< p-eCancelBook
; TReq1 >(*)<      p-CCallBook-Mer ; | TReq1 -> Cancelled | >,nullTrans,[< p-startB ;
TReq2 >,     t-CallBook,nullThread]],'t-CancelBook} {[< p-start ; ems >(*)< p-end ; ems
>(*)< p-eCancelBook ; TReq1 >(*)< p-eCancelBook ; TReq2 >(*)<  p-CCallBook-Mer ; |
TReq1  ->  Cancelled  |  >(*)<  p-CCallBook-Mer  ;  |  TReq2  ->  Cancelled  |
>,nullTrans,nullThread],'t-CCallBook-Mer} {[< p-start ; ems >(*)< p-end ; ems >(*)< p-
eCancelBook ; TReq1 >(*)< p-eCancelBook ; TReq2 >( *)< p-CCallBook-Mer ; | TReq2 -
> Cancelled | >(*)< p-Mer-end ; | TReq1 ->  Cancelled | >,nullTrans ,nullThread],'t-
end} {[< p-start ; ems >(*)< p-end ; | TReq1 -> Cancelled | >(*)< p-eCancelBook ;
TReq1 >(*)< p-eCancelBook ;  TReq2 >(*)< p-CCallBook-Mer ; | TReq2 -> Cancelled |
>,nullTrans,  nullThread],'t-CCallBook-Mer} {[< p-start ; ems >(*)< p-end ; | TReq1 ->
Cancelled | >(*)< p-eCancelBook ; TReq1 >(*)< p-eCancelBook ; TReq2 >(*)< p-Mer-
end ; | TReq2 -> Cancelled | > ,nullTrans , nullThread],'t-end}, {[< p-start ; ems >(*)<
p-end ; | TReq1 -> Cancelled |(+)| TReq2 -> Cancelled | >(*)< p-eCancelBook ; TReq1
>(*)< p-eCancelBook ; TReq2 >,nullTrans, nullThread],deadlock})
```

Fig. 13. Verification of travel request process properties using Maude LTL model checker.

9. Properties verification using the maude tool

In Fig. 12, we have checked the *proper termination* property for the travel request process. This property means that starting from an initial extended marking, every possible execution path properly terminates (eventually) i.e. reaches a final extended marking. This property is expressed in LTL by the following formula : $F finalState$ where the proposition *finalState* characterising the set of final states of the travel request process. The temporal operator F (Futur) is denoted by $< >$ in Maude notation. So, the proposition *finalState* is expressed in Maude as follows : $eq[< p \text{ - } end; ms >, nullTrans, nullThread] |= finalState = true$. As depicted in Fig. 12, this formula has been proven to be valid by using the LTL model checker of Maude.

In Fig. 13, we have checked the following property: *whenever we invoke the booking process (by firing the abstract transition $t_{CallBook}$), the process either terminates by success or by exception*. This property is expressed in LTL by the following formula : $[](Booking - Invoked => <> (Booking(success) \/ Booking(exception)))$. As depicted in Fig. 13, this formula has been proven to be not valid. Indeed, the model checker returns the expected *counterexample*, which corresponds to the cancellation of the booking process.

10. Conclusion and future work

The contribution of this work is a definition of a method allowing the transformation of a subset of BPMN to RECATNets. On the basis of mapping rules for which a formal semantics is established. With this mapping, we can cover a large set of BPMN features like cancellation, multiple instantiation of subprocesses and exception handling while taking into account the data flow aspect.

Subsequently, we showed the developed prototype based on model-driven approach to automate the proposed mapping rules and to generate rewriting logic specifications from the obtained RECATNets. The obtained rewriting logic files can be used by the Maude LTL model checker in order to verify several behavioral properties related to BPMN models. Finally, we showed how properties such as *proper termination* and LTL properties can be verified using the Maude LTL model checker through a, non-trivial, BPMN model.

In the future, we plan to complete this work by proposing a mapping rules for other BPMN constructs such as OR-Join gateways, transactions, compensation activities... etc. Other ongoing work aims at extending this work by considering time constraints [25] and shared resources [28] in BPMN models.

References

[1] W.M.P. van der Aalst, A.H.M. ter Hofstede, YAWL: yet another workflow language, Inf. Syst. 30 (4) (2005) 245–275.
[2] K. Barkaoui, H. Boucheneb, A. Hicheur, Modelling and analysis of time-constrained flexible workflows with time recursive ecatnets, in: R. Bruni, K. Wolf (Eds.), Web Services and Formal Methods, Lecture Notes in Computer Science, vol. 5387, Springer Berlin Heidelberg, 2009, pp. 19–36.
[3] K. Barkaoui, A. Hicheur, Towards analysis of flexible and collaborative workflow using recursive ecatnets, in: A. ter Hofstede, B. Benatallah, H.-Y. Paik (Eds.), Business Process Management Workshops, Lecture Notes in Computer Science, vol. 4928, Springer Berlin Heidelberg, 2008, pp. 232–244.
[4] J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, M. Weber, The Petri net markup language: concepts, technology, and tools, in: W. van der Aalst, E. Best (Eds.), Applications and Theory of Petri Nets 2003, Lecture Notes in Computer Science, vol. 2679, Springer Berlin Heidelberg, 2003, pp. 483–505.
[5] C. Ou-Yang, Y.D. Lin, BPMN-based business process model feasibility analysis: a Petri net approach, Int. J. Prod. Res. 46 (14) (2008) 3763–3781.
[6] R.M. Dijkman, M. Dumas, C. Ouyang, Semantics and analysis of business process models in BPMN, Inf. Softw. Technol. 50 (12) (2008) 1281–1294.
[7] C. Dufourd, A. Finkel, P. Schnoebelen, Reset nets between decidability and undecidability, in: 25th International Colloquium on Automata, Languages and Programming, ICALP '98, Springer-Verlag, 1998, pp. 103–115.
[8] S. Eker, J. Meseguer, A. Sridharanarayanan, The maude LTL model checker, Electron Notes Theor. Comput. Sci. 71 (2004) 162–187. WRLA 2002, Rewriting Logic and Its ApplicationsFabio Gadducci, Ugo Montanari.
[9] N. El-Saber, A. Boronat, BPMN formalization and verification using maude, in: Proceedings of the 2014 Workshop on Behaviour Modelling-Foundations and Applications, BM-FA '14, ACM, 2014, pp. 1:1–1:12.
[10] S. Haddad, D. Poitrenaud, Recursive Petri nets, Acta Inf. 44 (7–8) (2007) 463–508.
[11] A. Hicheur, Modélisation et Analyse des Processus Workflows Reconfigurables et Distribués par les Ecatnets Récursifs, CNAM 2009, Paris, 2009 Thèse de doctorat en Informatique.
[12] J. Rao, O. Chiotti, P. Villarreal, Specification of behavioral anti-patterns for the verification of block-structured collaborative business processes, Inf. Softw. Technol. 75 (2016) 148–170.
[13] F. Jouault, F. Allilaire, J. Bzivin, I. Kurtev, ATL: a model transformation tool, Sci. Comput. Program. 72 (12) (2008) 31–39. Special Issue on Second issue of Experimental Software and Toolkits (EST).
[14] A. Kheldoun, K. Barkaoui, M. Ioualalen, Specification and verification of complex business processes - a high-level Petri net-based approach, in: H.R. Motahari-Nezhad, J. Recker, M. Weidlich (Eds.), Business Process Management, Lecture Notes in Computer Science, vol. 9253, Springer International Publishing, 2015, pp. 55–71.
[15] C. Liu, F. Zhang, Petri net based modeling and correctness verification of collaborative emergency response processes, Cybern. Inf. Technol. 16 (3) (2016) 122–136.
[16] F. Lu, Q. Zeng, Y. Bao, H. Duan, Hierarchy modeling and formal verification of emergency treatment processes, Syst Man Cybern 44 (2) (2014) 220–234.
[17] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theor. Comput. Sci. 96 (1) (1992) 73–155.
[18] S. Morimoto, A survey of formal verification for business process modeling, in: M. Bubak, G. van Albada, J. Dongarra, P. Sloot (Eds.), Computational Science ICCS 2008, Lecture Notes in Computer Science, vol. 5102, Springer Berlin Heidelberg, 2008, pp. 514–522.
[19] O.M.G. (OMG), Business Process Model and Notation (BPMN) Version 2.0, Technical Report, 2011.
[20] C. Ouyang, E. Verbeek, W. van der Aalst, S. Breutel, M. Dumas, A. ter Hofstede, WofBPEL: a tool for automated analysis of BPEL processes, in: B. Benatallah, F. Casati, P. Traverso (Eds.), Service-Oriented Computing - ICSOC 2005, Lecture Notes in Computer Science, vol. 3826, Springer Berlin Heidelberg, 2005, pp. 484–489.
[21] F. Puhlmann, M. Weske, Investigations on soundness regarding lazy activities, in: S. Dustdar, J. Fiadeiro, A. Sheth (Eds.), Business Process Management, Lecture Notes in Computer Science, vol. 4102, Springer Berlin Heidelberg, 2006, pp. 145–160.
[22] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, N. Mulyar, Workflow Control-Flow Patterns: A Revised View, Technical Report, BPMcenter.org, 2006.
[23] E. Verbeek, W. van der Aalst, Woflan 2.0 a Petri-net-based workflow diagnosis tool, in: M. Nielsen, D. Simpson (Eds.), Application and Theory of Petri Nets 2000, Lecture Notes in Computer Science, vol. 1825, Springer Berlin Heidelberg, 2000, pp. 475–484.
[24] P.D. Villarreal, I. Lazarte, J. Roa, O. Chiotti, A Modeling Approach for Collaborative Business Processes Based on the UP-ColBPIP Language, Springer, Berlin, Heidelberg, 2010, pp. 318–329.
[25] K. Watahiki, F. Ishikawa, K. Hiraishi, Formal verification of business processes with temporal and resource constraints, in: Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on, 2011, pp. 1173–1180.
[26] P.Y. Wong, J. Gibbons, Formalisations and applications of BPMN, Sci. Comput. Program. 76 (8) (2011) 633–650. Special issue on the 7th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA08)
[27] J. Ye, S. Sun, W. Song, L. Wen, Formal semantics of BPMN process models using YAWL, in: Intelligent Information Technology Application, 2008. IITA '08. Second International Symposium on, volume 2, 2008, pp. 70–74.
[28] Q. Zeng, F. Lu, C. Liu, H. Duan, C. Zhou, Modeling and verification for cross-department collaborative business processes using extended Petri nets, Syst. Man Cybern 45 (2) (2015) 349–362.