High Level Petri Net Modelling and Analysis of Flexible Web Services Composition

Ahmed Kheldoun, Kamel Barkaoui, Malika Ioualalen and Djaouida Dahmani

Abstract In this paper we propose a model to deal with flexibility in complex Web services composition (WSC). In this context, we use a model based on high level Petri nets called RECATNets, where control and data flows are easily supported. Indeed, RECATNets combine the strengths of recursive Petri nets with the expressive power of abstract data types. Since RECATNets semantics is expressed in terms of the conditional rewriting logic, one can use the Maude LTL Model-Checker to investigate several behavioral properties of Web services composition.

1 Introduction

With the increasing complexity of business requirements, the distributed and flexible characteristics of Web services, the possibility of errors in Web service composition (WSC for short) is greatly increased. As a result, many researchers tried to propose formal methods, Finite State Machine [1], Pi calculus [2] or Petri nets [3, 4] to build the formal description and verification models of WSC. However, one of the weaknesses of these methods is their lack of support for managing flexible WSC which require dynamic adaptation of their structure. We refer to flexible WSC as the ability to create, modify, extend or suppress (sub)processes in a structured way, at the

A. Kheldoun (✉) · M. Ioualalen · D. Dahmani
MOVEP, Computer Science Department, USTHB, Algiers, Algeria
e-mail: ahmedkheldoun@yahoo.fr

M. Ioualalen
e-mail: mioualalen@usthb.dz

D. Dahmani
e-mail: ddahmani2000@yahoo.com

A. Kheldoun
Sciences and Technology Faculty, Yahia Fares University, Medea, Algeria

K. Barkaoui
CEDRIC-CNAM, 292 Rue Saint-Martin, 75141 Paris Cedex 03, France
e-mail: kamel.barkaoui@cnam.fr

© Springer International Publishing Switzerland 2016

R. Lee (ed.), *Software Engineering Research, Management and Applications*, Studies in Computational Intelligence 654,
DOI 10.1007/978-3-319-33903-0_12

occurrence of exceptions. In addition, such a composite Web services can potentially be very large, complex and cumbersome. In regard to the previous points, if we want to describe, faithfully, real-life WSC, we need an expressive modeling formalism that allows, in one hand, to specify their flexible and distributed features, and in other hand, to check the interaction (control-flow) correctness of these business processes while taking into account their data flow aspect. In this paper, a new model based on a kind of high level algebraic Petri nets combining the expressive power of abstract data types and Recursive Petri nets [5] called Recursive ECATNets (RECATNets for short) [6] is proposed in order to cope with the flexibility problem in complex WSC. The RECATNets model offers practical mechanisms for a direct and intuitive support of dynamic creation and suppression of processes. They are well-suited for handling the most advanced WSC patterns (involving cancellation and multiple instances). The proposed model is expressive enough to capture the semantics of complex service compositions and their respective specificities. Since RECATNets semantics is expressed in terms of the conditional rewriting logic [7], one can use the Maude LTL model-checker [8] to investigate several behavioral properties of Web services composition. The remainder of this paper is organized as follows. Section 2 gives a brief overview of related work. Section 3 presents the basic concepts of RECTANets. Web service modeling and specification using RECATNet are presented in Sect. 4. Section 5 is devoted to the algebra for composing Web services and its RECATNets-based formal semantics. A case study is presented in Sect. 6. Section 7 presents the analysis method and the verification process. Finally, Sect. 8 concludes and gives some further research directions.

2 Related Works

The composition of web services requires the modelling of different combinations of web services involved in this composition [9]. The modelling of web services composition is addressed in several papers. In this section, we briefly overview some approaches that are closely related to our work. In [10], the authors propose in their project e-flow to use workflow management system in order to compose web services. However, this approach lacks a formal model for specifying web services composition. In [11], the authors developed a Petri net based approach that uses several structural properties for identifying inconsistent dependency specification in a workflow. However, the proposed approach is restricted to acyclic workflows. In [3], the authors propose a Petri net-based algebra for composing web services. They provide a direct mapping from each composition operator to Petri nets. Their model is expressive enough; but data types cannot distinguish because they used elementary Petri nets. Contrary to our model, data types can be distinguished because the model used the expressive power of abstract data types. In [4], the authors used colored Petri nets [12] for modelling web services and their composition where data types can be distinguished. However, the author focalise in modeling, only, simple patterns. The author propose in [13] a model for composing web services based high

level Petri nets, called G-nets. In this model, the authors propose to compose web services via special places called instantiated switch places. For the analysis, the author need to tranform their G-net models into Predicate/Transition nets (PrT-nets). However, some useful patterns like multiple instance and cancellation of service are not addressed. In [14], the author present a review of forty-three patterns for modeling business process using Colored Petri-Net (CPN). However, we note that the pattern of multiple instantiation of a sub-process is difficult to implement when a particular instance of a sub-process initiates other sub-process instances or involves recursive calls to the one of these ancestors process. This is even more complex when the number of such instances is not known prior to the execution of the process or where such instances require synchronization on many levels. One of the weakness of the previous approach is their lack a support for modeling useful advanced patterns like multiple instance and cancellation of service. In order to address this issue, we present a modular and hierarchical formalism called RECATNet that allows composition via abstract transitions. The usefulness of our proposed model is: (1) offering a practical mechanisms for handling the most advanced flow patterns (dynamic) multiple instance and cancellation of Web service, (2) providing a hierarchical composition of web services, (3) its modular specification and its flexibility by adding/removing service's instances in a dynamic manner, (4) allowing distributed execution of web services composition and (5) its semantic may be defined in terms of conditional rewriting logic [7] therefore, the model-checker MAUDE [8] can be used to check its correctness.

3 Recursive ECATNet Review

Recursive ECATNets (abbreviated RECATNets) [6] are a kind of high level algebraic Petri nets combining the expressive power of abstract data types and Recursive Petri nets [5]. Each place in such a net is associated to a sort (i.e. a data type of the underlying algebraic specification associated to this net). The marking of a place is a multiset of algebraic terms (without variables) of the same sort of this place. Moreover, transitions in RECATNet are partitioned into two types (Fig. 1): elementary and abstract transitions. Each abstract transition is associated to a starting marking, denoted like a multi-set of places put inside bracket. A capacity associated to a place *p* specifies the number of algebraic terms which can be contained in this place for

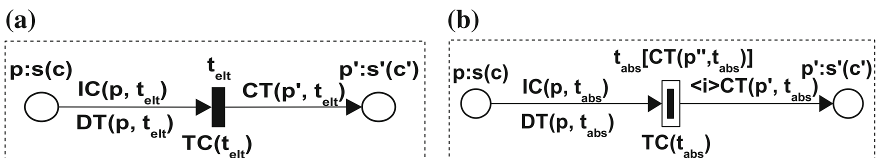

Fig. 1 Transition types in RECATNets. **a** Elementary transition. **b** Abstract transition

Table 1 Different forms of Input Condition IC(p, t)

IC(p, t)	Enabling condition
a^0	The marking of the place p must be equal to a (e.g. $IC(p,t) = \phi^0$ means the marking of p must be empty)
a^+	The marking of the place p must include a (e.g. $IC(p,t) = \phi^+$ means condition is always satisfied)
a^-	The marking of the place p must not include a, with $a \neq \phi$
$\alpha 1 \wedge \alpha 2$	Conditions $\alpha 1$ and $\alpha 2$ are both true
$\alpha 1 \vee \alpha 2$	$\alpha 1$ or $\alpha 2$ is true

each element of the sort associated to p. As shown in Fig. 1, the places p and p' are respectively associated to the sorts s and s' and to the capacity c and c'. An arc from an input place p to a transition t (elementary or abstract) is labelled by two algebraic expressions $IC(p,t)$ (Input Condition) and $DT(p,t)$ (Destroyed Tokens). The expression $IC(p,t)$ specifies the partial condition on the marking of the place p for the enabling of t (see Table 1). The expression $DT(p,t)$ specifies the multiset of terms to be removed from the marking of place p when t is fired. Also, each transition t may be labelled by a Boolean expression $TC(t)$ which specifies an additional enabling condition on the values taken by contextual variables of t (i.e. local variables of the expressions IC and DT labelling all the input arcs of t). When the condition $TC(t)$ is omitted, the default value is the term *True*. For an elementary transition t, an output arc (t,p') connecting this transition t to a place p' is labelled by the expression $CT(t,p')$ (*Created Tokens*). However, for an abstract transition t, an output arc (t,p') is labelled by the expression $\langle i \rangle CT(t,p')$ (*Indexed Created Tokens*). These two algebraic expressions specify the multiset of terms to produce in the output place p' when the transition t is fired. In the graphical representation of RECATNets, we note the capacity of a place regarding an element of its sort only if this number is finite. If $IC(p,t) =_{def} DT(p,t)$ on input arc (p,t) (e.g. $IC(p,t) = a^+$ and $DT(p,t) = a$), the expression $DT(p,t)$ is omitted on this arc. In what follows, we note $Spec = (\Sigma, E)$ an algebraic specification of an abstract data type associated to a RECATNet, where $\Sigma = (S, OP)$ is its multi-sort signature (S is a finite set of sort symbols and OP is a finite set operations, such $OP \cap S = \phi$). E is the set of equations associated to *Spec*. $X = (X_s)_{s \in S}$ is a set of disjoint variables associated to *Spec* where $OP \cap X = \phi$ and X_s is the set of variables of sort s. We denote by $T_{\Sigma,s}(X)$ the set of S-sorted S-terms with variables in the set $X.[T_{\Sigma}(X)]_{\oplus}$ denotes the set of the multisets of the Σ-terms $T_{\Sigma}(X)$ where the multiset union operator $(_{\oplus})$ is associative, commutative and admits the empty multiset ϕ as the identity element.

Definition 1 A recursive ECATNet is a tuple $RECATNet = \langle Spec; P, T, F; sort, Cap, IC, DT, CT, TC, I, \Upsilon, ICT, K \rangle$ where:

- $Spec = (\Sigma, E)$ is a many sorted algebra where the sorts domains are finite (with $\Sigma = (S, OP)$), and $X = (X_s)_{s \in S}$ is a set of S-sorted variables,

- $[P, T, F]$ is a net where ($T \cap P = \phi$) and $T = T_{elt} \cup T_{abs}$ is finite set of transitions partitioned into abstract and elementary ones. T_{abs} and T_{elt} denoted the set of abstract and elementary transitions,
- sort: $P \rightarrow S$, is a mapping called a sort assignment,
- *Cap*: is a P-vector on capacity places: p∈P, Cap(p): $T_{\Sigma}(\phi) \rightarrow \mathbb{N} \cup \{\infty\}$,
- $IC : P \times T \rightarrow [T_{\Sigma,sort(p)}(X)]^{*}_{\oplus}$ where $* \in \{0, +, -\}$ maps a multiset of terms for every input arc,
- $DT : P \times T \rightarrow [T_{\Sigma,sort(p)}(X)]_{\oplus}$ maps a multiset of terms for every input arc,
- $CT : P \times T \rightarrow [T_{\Sigma,sort(p)}(X)]_{\oplus}$ maps a multiset of terms for every output arc (p, t) where $t \in T_{elt}$ and a starting marking associated to $t \in T_{abs}$ according to place p,
- $TC : T \rightarrow [T_{\Sigma,bool}(X)]$ maps a boolean expression for each transition,
- $I = I_{cut} \cup I_{pre}$ is a finite set of indices, called termination indices, dedicated to cut steps and preemptions (interruptions) respectively,
- Υ is a family, indexed by I, of effective representation of semi-linear sets of final markings,
- $ICT : P \times T_{abs} \times I \rightarrow [T_{\Sigma,sort(p)}(X)]_{\oplus}$ maps a multiset of terms for every output arc (p, t, 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 trasitions, and their associated termination indexes, for every elementary transition.

Let's use the net presented in Fig. 2a to highlight RECATNet's graphical symbols and associated notations. (1) An elementary transition is represented by a filled rectangle; its name is possibly followed by a set of terms $t'\langle i \rangle \in T_{abs} \times I$. Each term specifies an abstract transition t' which is under the control of t, associated with a termination index to be used when aborting t' consequently to a firing of t. For instance t_{cancel} is an elementary transition where its firing preempts threads started by the firing of t_1 and the associated termination index is *1*. (2) An abstract transition t is represented by a double rectangles with the center filled; its name is followed by the starting marking $CT(t)$. For instance, t_1 is an abstract transition and $CT(t_1) = \langle p_5, Rq \rangle$ means that any thread created by firing of t_1 starts with one token i.e. one request Rq in place p_5. (3) Any termination set can be defined concisely based on place marking. For instance, Υ_0 specifies the final marking of threads such that the place p_6 is marked at least by one token. (4) The set I of termination indices is deduced from the indices used to subscript the termination sets and from the indices bound to elementary

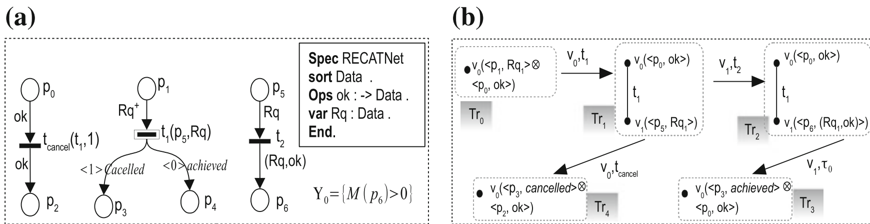

Fig. 2 Example of a RECATNet and two possible firing sequences

transitions i.e. interruption. From the example $I = \{0, 1\}$. Informally, a RECATNet generates during its execution a dynamical tree of marked threads called an extended marking, which reflects the global state of such net. This latter denotes the fatherhood relation between the generated threads (describing the inter-threads calls). Each of these threads has its own execution context.

Definition 2 (*Extended marking*) An extended marking of a RECATNet is a labelled rooted tree denoted $Tr = \langle V, M, E, A \rangle$ where:

- V is the set of nodes (i.e. threads),
- M is a Mapping $V \to [T_{\Sigma}(\phi)]_{\oplus}$ associating an ordinary marking with each node of the tree, such that $\forall v \in V, \forall p \in P, M(v)(p) \leq Cap(p)$,
- $E \subseteq V \times V$ is the set of edges,
- A is a mapping $E \to T_{abs}$ associating an abstract transition with each edge.

Note that contrary to ordinary nets, RECATNet are often disconnected since each connected component may be activated by the firing of abstract transitions.

Running example. Figure 2b highlights a possible firing sequences of the RECATNet represented in Fig. 2a. The graphical representation of any extended marking Tr is a tree where an arc $v_1(m_1) \to v_2(m_2)$ labeled by t_{abs} means that v_2 is a child of v_1 created by firing abstract transition t_{abs} and m_1 (resp. m_2) is the marking of v_1 (resp. v_2). Note that the initial extended marking Tr_0 is reduced to a single node v_0 whose marking is $\langle p_1, Rq_1 \rangle \otimes \langle p_0, ok \rangle$. From the initial extended marking Tr_0, the abstract transition t_1 is enabled; its firing leads to the extended marking Tr_1 which contains a fresh node v_1 marked by the starting marking $CT(t_1)$. Then, the firing of the elementary transition t_2 from 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 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 $ICT(t_1, 0) = (p_4, achieved)$. Also, another way to remove nodes in extended marking using elementary transition with associated preemption. For instance, from node v_0 in Tr_1, the elementary transition t_{cancel} with associated preemption $(t_1, 1)$ is enabled; its firing leads to an extended marking Tr_4 by removing the node v_1 and change the marking on its node father i.e. v_0 by adding $ICT(t_1, 1) = (p_3, cancelled)$. More details about RECATNets such as formal firing and fundamental properties are presented in [6, 15]. This paper shows the usefulness of using the formalism of RECATNets in the field of Web services composition.

4 Modeling Web Services Using RECATNet

We give now a formal definition of a Web service.

Definition 3 (*Web Service*) A Web service is a tuple [3] $S = \langle NameS, Desc, Loc, URL, CS, RECATNetS \rangle$ where:

- *NameS* is the name of the service used as its unique identifier,
- *Desc* is the description of the provided service. It summarizes what functionalities the service offers,
- *Loc* is the server in which the service is located,
- *URL* is the invocation of the Web service,
- *CS* is a set of the component services of the Web service, if $CS = \{NameS\}$ then S is a basic service, otherwise$S$ is a Composite service,
- $RECATNetS = \langle Spec; P, T, F; sort, Cap, IC, DT, CT, TC, I, \Upsilon, ICT, K\rangle$ is the RECATNet service modeling the dynamic behavior of the Web service.

We show in the next section how Web services can be incrementally composed.

5 Web Services Composition

The common control structure in Web services composition usually includes simple patterns like: sequential, choice, iteration, parallel and discriminator operators and complex patterns like multiple instance and cancellation service [14]. Suppose S_1, S_2 and S_3 are three different atomic Web services i.e. each service S_i perfoms an individual operation that cannot be split into sub-operations. The algebra operator descriptions of these control structure can be seen as: $S = \varepsilon \mid S_1 \bullet S_2 \mid S_1 + S_2 \mid \mu(S_1) \mid S_1 \parallel S_2 \mid (S_1 \mid S_2) \rhd S_3 \mid (S_1)^\star \mid S_1!$ where:

- ε stands for an empty service, i.e., a service performs no operation.
- $S_1 \bullet S_2$ performs the service S_1 followed by the service S_2, i.e., $\bullet$ is an operator of sequence.
- $S_1 + S_2$ can reproduce either the behavior S_1 or S_2, i.e., $+$ is an alternative operator.
- $\mu(S_1)$ represents a composite service where the behavior S_1 may be executed multiple times, i.e., μ is an iteration operator.
- $S_1 \parallel S_2$ performs concurrently the two services S_1 and S_2 i.e., $\parallel$ is a Parallel operator.
- $(S_1 \mid S_2) \rhd S_3$ waits for the execution of one service (among the S_1 and S_2) before activating the service S_3 i.e.$\rhd$ is a discriminator operator. Note that S_1 and S_2 are executed in parallel and independently,
- $(S_1)^\star$ repressents a composite service which allows creating multiple instances of a given Web service S_1. These instances are independent of each other and run concurrently,
- $S_1!$ represents a composite service which if the Web service S_1 has started, it is disabled and, where possible, the currently running instance is halted and removed.

In this section, we give a formal definition, in term of RECATNet, of the composition operators. Let specified, as defined in *Definition* 1, each atomic Web service by $S_i = \langle NameS_i, Desc_i, Loc_i, URL_i, CS_i, RECATNetS_i\rangle$ where $RECATNetS_i = \langle Spec_i; P_i, T_i, F_i; sort_i, Cap_i, IC_i, DT_i, CT_i, TC_i, I_i, \Upsilon_i, ICT_i, K_i\rangle$. Let's define a function *initMarking*(WS) that is used to return the start marking i.e. initial state of the

invoked Web service *WS*. The following notations are common to all the composition operators:

- *NameS* is the name of the new service,
- *Desc* is the description of the new service,
- *Loc* is the location of the new service,
- *URL* is the invocation of the new service.

5.1 Empty Service

The empty service ε is a service that performs no operation. It is used for technical and theoretical reasons.

Definition 4 The empty service is defined as $\varepsilon = \langle NameS, Desc, Loc, URL, CS, RECATNetS \rangle$ where:

- $NameS = Empty$
- $Desc =$ "*EmptyWebService*"
- $Loc = Null$ stating that there is no server for the service,
- $URL = Null$ stating that there is no *URL* for the service,
- $CS = \{Empty\}$ and
- $RECATNetS = \langle Spec; \{p\}, \phi, \phi; \phi, \phi, \phi, \phi, \phi \rangle$

In Fig. 3a, we show the graphic representation of the empty service ε.

5.2 Sequence

The sequence operator allows the construction of a service composed of two services executed one after the other. This is typically the case when a service should wait the execution result of another one before starting its execution.

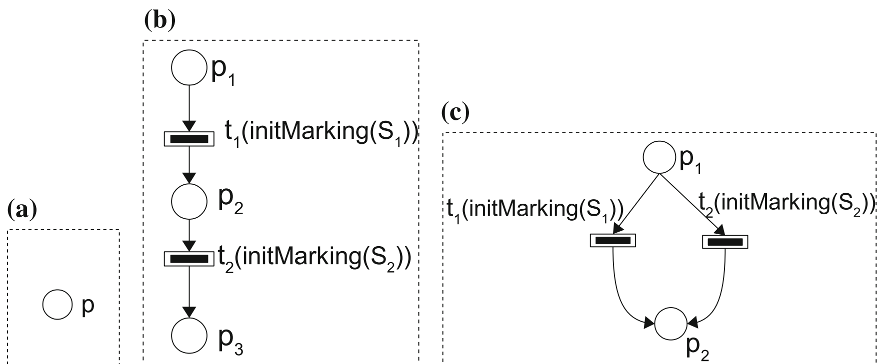

Fig. 3 Empty service (**a**), sequence service (**b**) and choice service (**c**)

Definition 5 The sequence operator $S_1 \bullet S_2$ is defined as $S = \langle NameS, Desc, Loc, URL, CS, RECATNetS \rangle$ where:

- $CS = CS_1 \cup CS_2$ and
- $RECATNetS = \langle Spec; P, T, F; ... \rangle$ where: $P = \{p_1 \cup p_2 \cup p_3\}$, $T = \{t_1 \cup t_2\}$, $F = \{(p_1, t_1), (t_1, p_2), (p_2, t_2), (t_2, p_3)\}$, $CT = \{(t_1, initMarking(S_1)), (t_2, initMarking(S_2))\}$.

Graphically, given two services S_1 and S_2, the composite service $S_1 \bullet S_2$ is represented by the RECATNet shown in Fig. 3b.

5.3 Choice

Given two services S_1 and S_2, the choice (alternative) operator allows modelling the execution of either S_1 or S_1, but not both.

Definition 6 The choice operator $S_1 + S_2$ is defined as $S = \langle NameS, Desc, Loc, URL, CS, RECATNetS \rangle$ where:

- $CS == CS_1 \cup CS_2$ and
- $RECATNetS = \langle Spec; P, T, F; ... \rangle$ where: $P = \{p_1 \cup p_2\}$, $T = \{t_1 \cup t_2\}$, $F = \{(p_1, t_1), (p_1, t_2), (t_1, p_2), (t_2, p_2)\}$, $CT = \{(t_1, initMarking(S_1)), (t_2, initMarking(S_2))\}$.

Graphically, given two services S_1 and S_2, the composite service $S_1 + S_2$ is represented by the RECATNet shown in Fig. 3c.

5.4 Iteration

The iteration operator allows a service S to be performed a certain number of times.

Definition 7 The iteration operator$\mu(S_1)$ is defined as $S = \langle NameS, Desc, Loc, URL, CS, RECATNetS \rangle$ where:

- $CS = CS_1$ and
- $RECATNetS = \langle Spec; P, T, F; ... \rangle$ where: $P = \{p_1 \cup p_2\}$, $T = \{t_1 \cup t_2\}$, $F = \{(p_1, t_1), (p_1, t_2), (t_2, p_2)\}$, $CT = \{(t_1, initMarking(S_1))\}$.

Graphically, if we consider the service S_1, the composite service $\mu(S_1)$ is represented by the RECATNet shown in Fig. 4a.

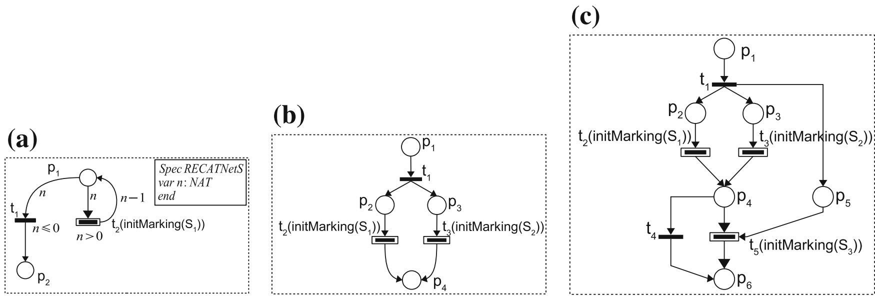

Fig. 4 Iteration (**a**), parallel (**b**) and discriminator service (**c**)

5.5 *Parallel*

Given two services S_1 and S_1, the parallel operator builds a composite service performing the two services in parallel and without interaction between them. The accomplishment of the resulting service is achieved when the two services are completed.

Definition 8 The parallel operator $S_1 \parallel S_2$ is defined as $S = \langle NameS, Desc, Loc, URL, CS, RECATNetS \rangle$ where:

- $CS = CS_1 \cup CS_2$ and
- $RECATNetS = \langle Spec; P, T, F; ... \rangle$ where: $P = \{p_1 \cup p_2 \cup p_3 \cup p_4\}$, $T = \{t_1 \cup t_2 \cup t_3\}$, $F = \{(p_1, t_1), (t_1, p_2), (t_1, p_3), (p_2, t_2), (p_3, t_3), (t_3, p_4), (t_3, p_4)\}$, $CT = \{(t_2, initMarking(S_1)), (t_3, initMarking(S_2))\}$.

Graphically, if we consider two services S_1 and S_2, the composite service $S_1 \parallel S_2$ is represented by the RECATNet shown in Fig. 4b.

5.6 *Discriminator*

Two or more equivalent services are invoked in parallel to achieve a given task but only one is required to finish before proceeding with the invocation of the next composed services of the composite service. It is presumed that these services are equivalent in terms of functionalities. The results of the first service to finish are used while the results of the remaining invoked services are ignored. At least one service of the invoked set of services must succeed for the composite service to succeed. The main goal of the discriminator operator is to increase reliability and delays of the services through the Web. For the customers, best services are those which respond in optimal time and are constantly available.

Definition 9 The discriminator operator $(S_1 \mid S_2) \rhd S_3$ is defined as $S = \langle NameS, Desc, Loc, URL, CS, RECATNetS \rangle$ where:

- $CS = CS_1 \cup CS_2 \cup CS_3$ and
- $RECATNetS = \langle Spec; P, T, F; ...\rangle$ where: $P = \{p_1 \cup p_2 \cup p_3 \cup p_4 \cup p_5 \cup p_6\}$, $T = \{t_1 \cup t_2 \cup t_3 \cup t_4 \cup t_5\}$, $F = \{(p_1, t_1), (t_1, p_2), (t_1, p_3), (t_1, p_5), (t_2, p_4), (t_3, p_4), (p_4, t_4), (p_4, t_5, (p_5, t_5), (t_4, p_6), (t_5, p_6))\}$, $CT = \{(t_2, initMarking(S_1)), (t_3, initMarking(S_2)), (t_5, initMarking(S_3))\}$.

Graphically, if we consider three Web services S_1, S_2 and S_3, the composite service $(S_1 \mid S_2) \rhd S_3$ is represented by the RECATNet shown in Fig. 4c.

5.7 Multiple Instance Service

Multiple instance operator allows for a given Web service to be instantiated multiple times in a business process. The number of instances is not known during the design or run time. These instances are run concurrently but, whilst they are running, new ones can be created.

Definition 10 Multiple instance service operator $(S_1)^\star$ is defined as $S = \langle NameS, Desc, Loc, URL, CS, RECATNetS\rangle$ where:

- $CS = CS_1$ and
- $RECATNetS = \langle Spec; P, T, F; ...\rangle$ where: $P = \{p_1 \cup p_2 \cup p_{create} \cup p_{stopcreate}\}$, $T = \{t_1 \cup t_{addIns} \cup t_{remove}\}$, $F = \{(p_1, t_1), (t_1, p_2), (p_1, t_{addIns}), (t_{addIns}, p_1), (p_{create}, t_{addIns}), (p_{create}, t_{remove}), (t_{remove}, p_{stopcreate})\}$, $CT = \{(t_1, initMarking(S_1))\}$.

Graphically, if we consider a Web service S_1, the composite service $(S_1)^\star$ is represented by the RECATNet shown in Fig. 5a.

5.8 Cancel Service

The cancel service operator provides the ability to stop a running instance of a Web service. For instance, the purchaser can cancel his buyonline's order at any time before it starts or during its running but not after the payment was done.

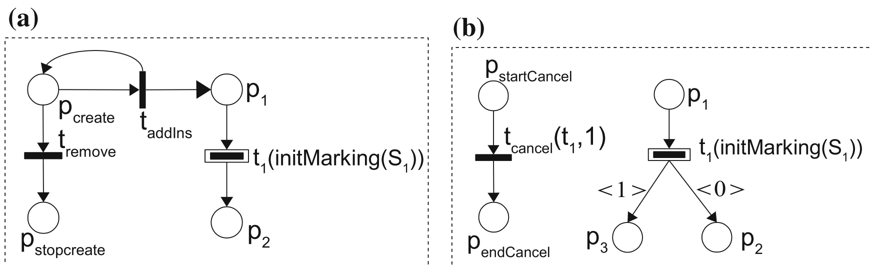

Fig. 5 Multiple instance service (**a**) and Cancel service (**b**)

Definition 11 Cancel service operator $S_1!$ is defined as $S = \langle NameS, Desc, Loc, URL, CS, RECATNetS \rangle$ where:

- $CS = CS_1$ and
- $RECATNetS = \langle Spec; P, T, F; ... \rangle$ where: $P = \{p_1 \cup p_2 \cup p_3 \cup p_{startCancel} \cup p_{endCancel}\}$, $T = \{t_1 \cup t_{cancel}\}$, $F = \{(p_1, t_1), (t_1, p_2, \langle 0 \rangle), (t_1, p_3, \langle 1 \rangle), (p_{startCancel}, t_{cancel}), (t_{cancel}, p_{endCancel})\}$,
  $CT = \{(t_1, initMarking(S_1))\}, K(t_{cancel}) = (t_1, 1)$.

Graphically, if we consider a Web service S_1, the composite service $S_1!$ is represented by the RECATNet shown in Fig. 5b.

6 A Case Study

Figure 6 shows an illustrative example of modelling a simplified *BuyOnline* service adapted from [16]. *BuyOnline* web service provides online book buying service which is composed of four atomic services: *LocateBook*, *SigIn*, *CreateAcct* and *Payement*. The composite web service may receive a list of *request*, sent by users, through the Place *StartBO*. Each *request* is represented by a token (ID, BN, SII, CAI, CCI) denotes repsectivelly *Identifier*, *BookName*, *SignInInfo*, *CreateAcctInfo*, *Credit CardInfo*. At the beginning, *BuyOnline* service starts by searching a book in web site according to the book name using service *LocateBook*. This operation is performed by firing the abstract transition *LocateBook* which, if this book can be found, returns its ISBN number. Then, the user can by this book but a valid register is required. If the user has a legal account, then finish loging using service *SignIn*; otherwise the user needs to create a new account using the service *CreateAcct*. In the last one, informations about the created account must be returned i.e. $CAO \neq \phi$. Finally, the service *payment* can finish the payment for the book according to *ISBN* number and credit card information *CCI* provided by user. Two cases are distinguished, if the credit card information are valid, the service *payment* will perform the payment by success; otherwise the service *payment* terminates by error i.e. *echecP*, and an error meaasge $CCI - not - valid$ is sent to user. Note that the user can cancel his/her online book buying service by firing the elementary transition *CancelBO*. Note that for each *request* sent by user, an instance of *BuyOnline* service is created. These instances are independent and may be executed in a distributed manner. In order to support dynamic creation instances of *BuyOnline* service, we need to update the model according to the pattern of Multiple instance operator shown in Fig. 5a. Here, and in order to perform analysis, we assume that our model is finite i.e. starts by a finite set of requests.

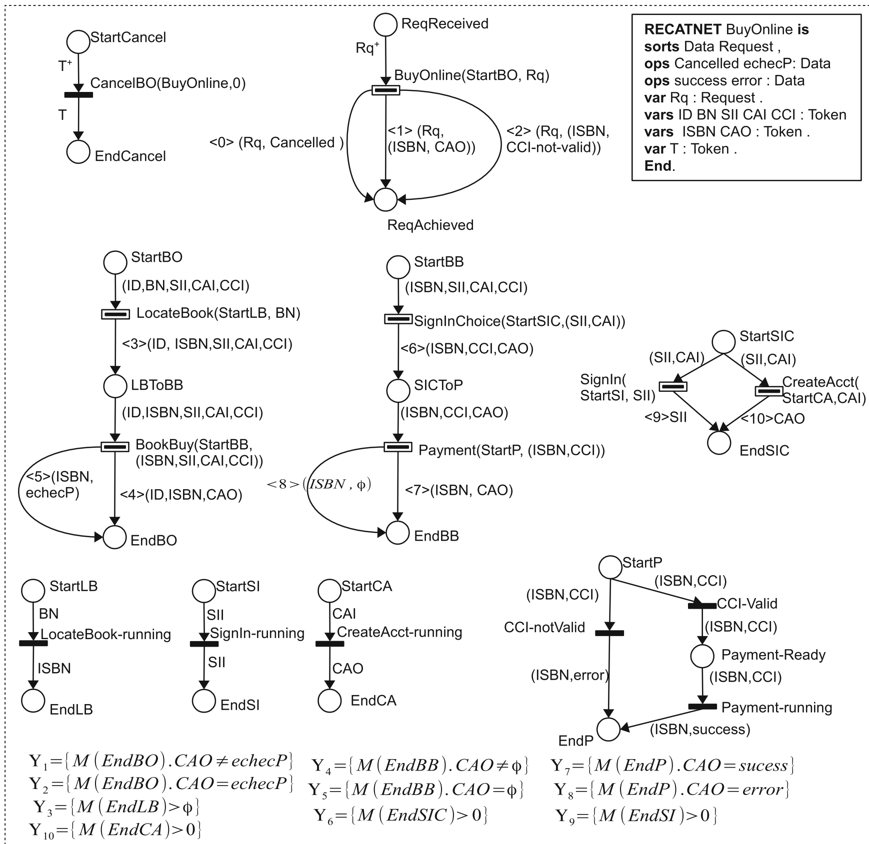

Fig. 6 *BuyOnline* book service

7 Verification of Web Services Composition

Our approach of verification can be described in Fig. 7. First, atomic services must be described in their associated RECATNets. Then, based on composition rules defined previously, generates the composite web services in terms of RECATNet. This operation may be insured by our java's tool *RECATNet-WSC* that is partially implemented. After that, from the obtained RECATNet, we generate in an automated manner its semantics in terms of rewriting logic [7] using the model-to-text (M2T) transformation tool Acceleo.[1] The rewriting logic files are used as an input of the model-checker Maude [8] to investigate several behavioral properties of Web services composition.

[1] http://www.eclipse.org/acceleo/.

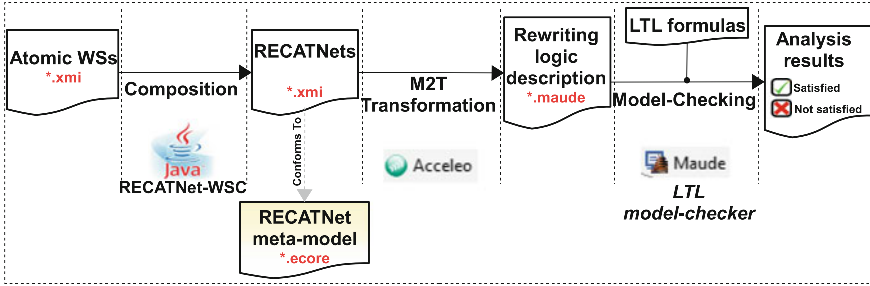

Fig. 7 Our approach

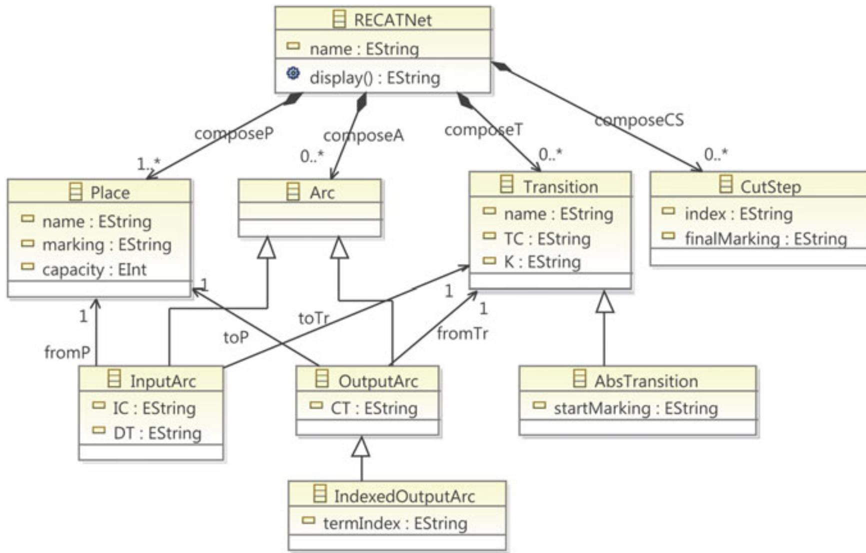

Fig. 8 RECATNets meta-model

7.1 RECATNets Meta-Modeling

In order to use M2T transformation using tool Acceleo, we need to define the meta-model of RECATNet. As shown in Fig. 8, we propose a general meta-model of our formalism using the UML class diagram model. Our proposed meta-model is composed mainly of the following classes.

- **RECATNet**: it builds the final model from a set of *Place*, *Arc*, *Transition* and *CutStep*.
- **Place**: it represents the RECATNet places. It has three attributes :*name*, *marking* and *capacity*.

- **Transition**: it represents the RECATNet transitions. It has three attributes: *name*, *TC* and *K*. One classe inherits from the super-class *Transition*: *AbsTransition* for abstract transition. This class contains one attribute *startMarking*.
- **Arc**: it represents the RECATNet arcs. It contains one attribute *inscription*. This class is a super-class of two classes. The first one is *InputArc* for arcs going from places to transitions. It contains two attributes *IC* for Input Condition and *DT* for Destroyed Token. The second is *OutputArc* for arcs going from transitions to places. It contains only the attribute *CT* for Created Token. In addition, the last class is a super-class of *IndexedOutputArc* for arcs going from abstract transition to places. It contains one attribute *index* to identify the set of indices of termination.
- **CutStep**: it represents the RECATNet cut steps. It contains two attributes *index* to identify the index of termination and *condition* to identify the condition for firing the cut step.

7.2 RECATNet Semantics in Terms of Rewriting Logic

RECATNet's semantics may be defined, easily, in terms of rewriting logic, therefore someone can use the LTL model-checker of MAUDE to investigate several behavioral properties of Web services compositions. A set of rewriting rules has been introduced in [6, 15] in order to express the semantics of RECATNet in terms of rewriting rules. In order to automate this approach, we have developed a model-to-text (M2T) transformation tool based Acceleo generator code. The transformation's rules have been inspired from rewriting rules proposed in [6]. For instance, if we consider the RECATNet of the atomic service *Payment* in Fig. 6, the generated Maude specification using M2T transformation is shown in Fig. 9. In fact, three rewriting rules are generated associated to the three elementary transitions in the RECATNet of the atomic service *Payment* in Fig. 6. For instance, the rewriting rule in line *4 rl[Payment-running]* describes the firing of the elementary transition *Payment-running*. So, this rewrite rule requires that the left-hand side is a marking where the place *Payment-Ready* is marked and yields to a marking i.e. the right-hand side, where the place *EndP* is marked.

```
1 mod PaymentS is
2 pr TYPE .
3 crl[CCI-Valid]: <StartP ; ISBN (+) CCI>=><Payment-Ready ; ISBN (+) CCI> if isValid(CCI) .
4 rl[Payment-running]: <Payment-Ready ; ISBN (+) CCI> => <EndP ; ISBN (+) success> .
5 crl[CCI-notValid]: <StartP ; ISBN (+) CCI>=><EndP ; ISBN (+) error> if not isValid(CCI) .
6 endm
```

Fig. 9 Generated Maude specification

7.3 Implementation Using the Maude Tool

An important property will be checked in Web service composition called *soundness* which concerns the correctness of internal control-flow of a composite Web service. The *soundness* of a Web services component is based on two criteria:

- *Proper termination*: This property called also *compatibility* of component Web services [17]. *Proper termination* 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* is valid in extended marking *Tr* if this latter is reduced to its root node with only terms in place *ReqAchieved*. The temporal operator *F* is denoted by $\langle\rangle$ in MAUDE notation. This formula has been proven to be *true* by MAUDE LTL-model checker in Fig. 10.
- *No dead service*: This property means that every atomic Web service must be invoked, at least, once. This requirement imposes that the Web services component should not contain Web services that can never be executed. In order to check this property, we define the proposition *isInvoked*(*ws*) which is valid in an extended marking *Tr*, if the specified Web service *ws* is invoked i.e. is running. Thus, to check that there is no dead service, we express the negation of this formula as the following LTL formula $\bigvee_{ws \in WS} G \neg isInvoked(ws)$ where *WS* is the set of atomic web services used during composition. Here, $WS = \{LocateBook, SigIn, CreateAcct, Payement\}$. If this formula is not valid, it means that the property *No dead service* is verified. The temporal operators *G* (Generally) and $\neg$ (not) are denoted, respectively, by [] and $\sim$ in MAUDE notation. In our case study, as we have a choice between the service *SigIn* and *CreateAcct*, this formula is expressed in LTL as following: $[] \sim isInvoked(LocateBook) \backslash/ [] \sim (isInvoked(SignIn) \backslash/ isInvoked(CreateAcct)) \backslash/ [] \sim isInvoked(Payment)$. This formula has been proven to be *not valid* by MAUDE LTL-model checker in Fig. 11. The model-checker returns the expected *counterexample*.

As the two properties *Proper termination* and *No dead service* are proved to be valid, therefore, the generated composite Web service is *sound*.

```
Maude> load BuyOnline/MAIN.maude .
==========================================
reduce in RECATNET-CHECK : modelCheck(initialState, <> finalState) .
rewrites: 324827 in 14437341288ms cpu (10079ms real) (0 rewrites/second)
result Bool: true
```

Fig. 10 Checking *Proper termination* property under Maude

```
Maude> load BuyOnline/MAIN.maude .
==========================================
reduce in RECATNET-CHECK : modelCheck(initialState, []~ isInvoked(Payment) \/ (
   []~ isInvoked(LocateBook) \/ []~ (isInvoked(SignIn) \/ isInvoked(
   CreateAcct)))) .
rewrites: 183 in 8811653550ms cpu (12ms real) (0 rewrites/second)
result ModelCheckResult: counterexample(
...
 {[ < ReqReceived ; ems >(*)< ReqAchieved ; 80 isbn2 CCI-not-valid empty empty
   >(*)< ListISBN ; ems >(*)< ListCAO ; ems >,nullTrans,[em,BuyOnline,[<
   SICToP ; 7(+)sii1 >,BookBuy,[< StartP ; isbn1(+)cci1 >,Payment,
   nullThread]]] [em,BuyOnline,[< SICToP ; 100(+)sii3 >,BookBuy,[< StartP ;
   isbn3(+)cci3 >,Payment,nullThread]]] [< EndBO ; 120(+)isbn4(+)echecP >,
   BuyOnline,nullThread]],'cut-2}, {[< ReqReceived ; ems >(*)< ReqAchieved ; (
   80 isbn2 CCI-not-valid empty empty)(+)(120 isbn4 CCI-not-valid empty empty)
   >(*)< ListISBN ; ems >(*)< ListCAO ; ems >,nullTrans,[em,BuyOnline,[<
   SICToP ; 7(+)sii1 >,BookBuy,[< StartP ; isbn1(+)cci1 >,Payment,
   nullThread]]] [em,BuyOnline,[< SICToP ; 100(+)sii3 >,BookBuy,[< StartP ;
   isbn3(+)cci3 >,Payment,nullThread]]]],deadlock})
```

Fig. 11 Checking *No dead service* property under Maude

8 Conclusion

In this paper, an effecient and flexible approach for Web services composition has been proposed. This approach takes fully advantage of modular, distributed execution aspects of RECATNets formalism. The formal semantic of the composition operators is expressed easily in terms of RECATNets by providing a direct transformation of each operator in terms of RECATNets. In fact, the model of RECATNets is particularly adequate for handling the most advanced flow patterns such as dynamic creation of processes and specifying exceptional behaviors in WSC at design time. Also, our method allows the verification of some properties using the LTL model-checker of the Maude system. In the future, we plan to complete this work by developing a tool capable of making automatic the mapping WSDL-descriptions into RECATNets.

References

1. Berardi, D., Calvanese, D., Giacomo, G., Lenzerini, M., & Mecella, M. (2003). Automatic composition of e-services that export their behavior. *Service-Oriented Computing—ICSOC 2003* (Vol. 2910, pp. 43–58), series Lecture Notes in Computer Science.
2. Lucchi, R., & Mazzara, M. (2007). A pi-calculus based semantics for ws-bpel. *The Journal of Logic and Algebraic Programming*, *70*(1), 96–118.

3. Hamadi, R., & Benatallah, B. (2003). A petri net-based model for web service composition. *Proceedings of the 14th Australasian Database Conference* (Vol. 17, pp. 191–200), series ADC '03.
4. Zhang, Z.-L., Hong, F., & Xiao, H.-J. (2008). A colored petri net-based model for web service composition. *Journal of Shanghai University (English Edition)*, *12*(4), 323–329.
5. Haddad, S., & Poitrenaud, D. (2007). Recursive petri nets: Theory and application to discrete event systems. *Acta Informatica*, *44*(7), 463–508.
6. Barkaoui, K., & Hicheur, A. (2008). Towards analysis of flexible and collaborative workflow using recursive ecatnets. In: A. Hofstede, B. Benatallah & H.-Y. Paik (Eds.), *Business Process Management Workshops* (vol. 4928, pp. 232–244), series Lecture Notes in Computer Science.
7. Bruni, R., & Meseguer, J. (2006). Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, *360*(1), 386–414.
8. Clavel, M. et al. (2007). Maude manual (version 2.3). http://maude.cs.uiuc.edu.
9. Srivastava, B., & Koehler, J. (2003). Web service composition—current solutions and open problems. *In: ICAPS 2003 Workshop on Planning for Web Services* (pp. 28–35).
10. Casati, F., Ilnicki, S., Jin, L.-J. & Shan, M.-C. (2000). *An Open, Flexible, and Configurable System for Service Composition* (pp. 125–132).
11. Adam, N. R., Atluri, V., & Huang, W.-K. (1998). Modeling and analysis of workflows using petri nets. *Journal of Intelligent Information Systems*, *10*(2), 131–158.
12. Jensen, K. (1990). Coloured petri nets: A high level language for system design and analysis. Technical Report.
13. Chemaa, S., Elmansouri, R., & Chaoui, A. (2013). Web services modeling and composition approach using object-oriented petri nets. *CoRR*, abs/1304.2080.
14. Russell, N., ter Hofstede, A., van der Aalst, W., & Mulyar, N. (2006). Workflow control-flow patterns: A revised view, BPM Center, Technical Report BPM-06-22.
15. Barkaoui, K., Boucheneb, H., & Hicheur, A. (2009). *Modelling and Analysis of Time-constrained Flexible Workflows with Time Recursive Ecatnets* (vol. 5387, pp. 19–36), series Lecture Notes in Computer Science. Berlin, Heidelberg: Springer.
16. Ding, Z., Wang, J., & Jiang, C. (2008). An approach for synthesis petri nets for modeling and verifying composite web service. *Journal of Information Science and Engineering* 1309–1328.
17. Li, X., Fan, Y., Sheng, Q., Maamar, Z., & Zhu, H. (2011). A petri net approach to analyzing behavioral compatibility and similarity of web services. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, *41*(3), 510–521.