A Coloured Petri Net Approach to Model and Analyse Stateful Workflows Based on WS-BPEL and WSRF

José Antonio Mateo, Valentín Valero, Hermenegilda Macià, and Gregorio Díaz(✉)

University of Castilla-La Mancha, Campus Universitario s/n, 02071 Albacete, Spain
{JoseAntonio.Mateo,Valentin.Valero,Hermenegilda.Macia, Gregorio.Diaz}@uclm.es

Abstract. Composite Web services technologies are widely used due to their ability to provide interoperability among services from different companies. Web services are usually *stateless*, which means that no state is stored from the clients viewpoint. However, some new applications and services require to capture the state of some resources after each computation. Thus, new standards to model Web services states have emerged e.g. Web Services Resource Framework (WSRF). In this paper, we present a formal model based on WS-BPEL and WSRF, and we provide a prioritised-timed coloured Petri net semantics for it. This semantics captures the main activities of WS-BPEL, but we also consider other important aspects, both from WS-BPEL and WSRF, such as fault handling, resource management, time-outs and a publish-subscribe system.

1 Introduction

The development of software systems is becoming more complex with the appearance of new computational paradigms such as Service-Oriented Computing (SOC), Grid Computing and Cloud Computing. These systems are characterized by a dynamic environment due to the heterogeneity and volatility of resources and, moreover, the service provider needs to ensure some levels of quality and privacy to the clients in a way that had never been considered. Formal models of concurrency have been widely used for the description and analysis of concurrent and distributed systems. It is then required to develop new techniques to benefit from the advantages of recent approaches such as Web service compositions. In this work, we use the language Web Services Business Process Execution Language (WS-BPEL) [1] to model this composition. In WS-BPEL, the behaviour of each participant (called orchestrator) is defined in terms of invocations to other services.

Although the Web service definition does not consider the notion of state, interfaces frequently provide the user with the ability to access and manipulate states, that is, data values that persist across, and evolve as a result of

Research partially supported by projects TIN2009-14312-C02-02 and TIN2012-36812-C02-02.

C. Canal and A. Idani (Eds.): SEFM 2014 Workshops, LNCS 8938, pp. 389–404, 2015.
DOI: 10.1007/978-3-319-15201-1_26

Web service interactions. The messages that the services send and receive imply (or encourage programmers to infer) the existence of an associated stateful resource. It is then desirable to define Web service conventions to enable the discovery of, introspection on, and interaction with stateful resources in standard and interoperable ways. To this end, a new standard, Web Services Resource Framework (WSRF) [2,5,10], was defined. In addition, it is required to provide notification mechanisms (e.g. publish-subscribe systems) so that each service can be notified about state changes.

The main motivation of this work is to provide a formal semantics for WS-BPEL+WSRF/WSN to manage stateful Web services workflows by using the existing machinery in distributed systems, and specifically a well-known formalism, such as coloured Petri nets extended with time and priorities, which are a graphical model, but they also provide us with the ability to simulate and analyse the modelled system. Notice that our aim is not to provide just another WS-BPEL semantics since WS-BPEL has been widely studied. Nevertheless, we have realised that it is more convenient to introduce a specific semantic model, which covers properly all the relevant aspects of WSRF/WSN (e.g. notifications and resource time-outs) instead of reusing some previous model.

WS-BPEL [1], for short BPEL, is an OASIS orchestration language for specifying actions within Web services business processes. WS-BPEL is therefore an orchestration language in the sense that it is used to define the composition of services from a local viewpoint, describing the individual behaviour of each participant. More specifically, WS-BPEL is a language for describing the behaviour of a business process based on interactions between the process and its partners. At the core of the WS-BPEL process model is the notion of peer-to-peer interaction between services described in Web Services Description Language (WSDL), both the process and its partners are exposed as WSDL services. Thus, a business process defines how to coordinate the interactions between a process instance and its partners through Web Service interfaces, whereas the structure of the relationship at the interface level is encapsulated in what is called a *partnerLink*. These are instances of typed connectors which specify the WSDL port types the process offers to and requires from a partner at the other end of the partner link.

In particular, we will define a web service composition as a set of orchestrators, described by BPEL+WSRF+WSN syntax, which exchange messages through some communication channels, *PartnerLinks*. Moreover, WS-BPEL processes use *variables* to temporarily store data. Variables are therefore declared on a process or on a scope within that process. In our case, there will be a single scope (*root*), so no nesting is considered in our framework. Besides, for simplicity again, we will only consider integer variables.

An orchestrator consists of a main activity, representing the normal behaviour of this participant. There are also fault activities, which are executed upon the occurrence of some unexpected events, or due to some execution failures, respectively. WS-BPEL activities can be *basic* or *structured*. *Basic activities* are those which describe the elemental steps of the process behaviour, such as the

assignment of variables (*assign*), empty action (*empty*), time delay (*wait*), invoke a service (*invoke*) and receive a message (*receive*), reply to a client (*reply*), and throw an exception (*throw*). We also have an action to *terminate* the process execution at any moment (*exit*). For technical reasons we have also included an additional activity $\overline{reply}$, which is used when a service invocation expects a reply, in order to implement the synchronization with the *reply* action from the server. On the contrary, *structured activities* encode control-flow logic in a nested way. The considered structured activities are the following: a *sequence* of activities, separated by a semicolon, the parallel composition, represented by two parallel bars (‖), the conditional repetitive behaviour (*while*), and a timed extension of the receive activity, which allows to receive different types of messages with a time-out associated (*pick*).

On the other hand, WSRF [2] is a resource specification language developed by OASIS and some of the most pioneering computer companies, This standard consists of a set of specifications that define the representation of a WS-Resource (web service + associated resource) in the terms that specify the messages exchanged and the related XML documents. These specifications allow the programmer to declare and implement the association between a service and one or more resources. It also includes mechanisms to describe the means to check the status and the service description of a resource, which together form the definition of a WS-Resource.

Here, we can see a WS-Resource as a collection of properties P identified by an address *EPR* with an associated *timeout*. This timeout represents the WS-Resource lifetime. Without loss of generality, we have reduced the resource properties set to only one allowing us to use the resource identifier *EPR* as the representative of this property. In addition, in WS-BPEL, we have taken into consideration the root scope only, thus avoiding any class of nesting among scopes, and we have considered fault handling, leaving the other handling types as future work.

Related Work. WS-BPEL has been extensively studied with many formalisms, such as Petri nets, Finite State Machines and process algebras, but there are only a few works considering WS-BPEL enriched with WSRF, and they only show a description of this union, without a formalization of the model. In [16] Slomiski uses BPEL4WS in Grid environments and discusses the benefits and challenges of extensibility in the particular case of OGSI workflows combined with WSRF-based Grids. Other two works centred around Grid environments are [8,12]. The first justifies the use of WS-BPEL extensibility to allow the combination of different GRIDs, whereas Ezenwoye et al. [8] share their experience on WS-BPEL to integrate, create and manage WS-Resources that implement the factory/instance pattern.

Table 1 shows the comparison of the related works where, the columns show the BPEL version considered, the coverage degree of the recovery framework, whether they use WSRF, the formalism they use, the focus area and if the work is supported by a tool.

Table 1. Related works comparison.

Author	BPEL	Rec.	WSRF	Formalism	Focus	Tool
Slomiski [16]	1.0	×	✓	–	Extensibility	×
Ezenwoye [8]	1.0	×	✓	–	Resource management	×
Ouyang [14]	1.0	Part	×	Petri nets	BPEL analysis	✓
Lohmann [7]	2.0	✓	×	Petri nets	BPEL analysis	✓
Dragoni [6]	2.0	✓	×	π-calculus	BPEL recovery framework	×
Qiu [15]	1.0	Part	×	Proc. Algebra	Fault and compensation	×
Farahbod [9]	1.0	Part	×	Finite State Machines	BPEL analysis	×
Busi [3]	1.0	Part	×	Proc. Algebra	Conformance Chor. vs Orch	×
Our work	2.0	Part	✓	Petri nets	Resource management	✓

2 Prioritised-Timed Coloured Petri Nets

Next, we introduce the specific model of prioritised-timed coloured Petri net considered for the translation. We use prioritised-timed coloured Petri nets, which are a prioritised-timed extension of coloured Petri nets, the well-known formalism supported by CPNTools [4]. In Definition 1, we recall the formal definition of coloured Petri nets presented in [11], whereas, in Definition 2, we define the precise model used in this work. We use the classical notation on Petri nets to denote the precondition and postcondition of both places and transitions:

$$\forall x \in P \cup T : {}^\bullet x = \{y \,|\, (y,x) \in A\} \qquad x^\bullet = \{y \,|\, (x,y) \in A\}$$

Definition 1. *A timed non-hierarchical Coloured Petri Net is a nine-tuple* $CPN_T = (P, T, A, \Sigma, V, C, G, E, I)$ *where:*

- P *is a finite set of places.*
- T *is a finite set of transitions such that* $P \cap T = \emptyset$.
- $A \subseteq (P \times T) \cup (T \times P)$ *is a set of directed arcs.*
- Σ *is a finite set of non-empty colour sets. Each colour set is either untimed or timed.*
- V *is a finite set of typed variables such that* $Type[v] \in \Sigma$ *for all variables* $v \in V$.
- $C : P \to \Sigma$ *is a colour set function that assigns a colour set to each place. A place p is timed if C(p) is timed, otherwise p is untimed.*
- $G : T \to EXPR_V$ *is a guard function that assigns a guard to each transition t such that* $Type[G(t)] = Bool$.
- $E : A \to EXPR_V$ *is an arc expression function that assigns an arc expression to each arc a such that*

- $Type[E(a)] = C(p)_{MS}$ *if* p *is untimed;*
- $Type[E(a)] = C(p)_{TMS}$ *if* p *is timed.*

Here, p *is the place connected to the arc* a*. Moreover, MS and TMS are untimed and timed colour sets in* Σ*, respectively.*

– $I : P \rightarrow EXPR_{\emptyset}$ *is an initialisation function that assigns an initialisation expression to each place* p *such that*
- $Type[I(p)] = C(p)_{MS}$ *if* p *is untimed;*
- $Type[I(p)] = C(p)_{TMS}$ *if* p *is timed.* □

In this work, we define a subclass of CPN_T, where three functions have been added. First, a *labelling* function is used to label places and transitions. Transitions can be labelled with either strings or nothing. Places are labelled as *entry places*, *output places*, *error places*, *exit places*, *internal places*, *variable places* and *resource places*, which, respectively, correspond to the following labels: $\{in, ok, er, ex, i, v, r\}$. Second, a *delay* function to assign a time interval to some transitions. This time interval is uniformly distributed. This is a shorthand for adding this time delay inscription to the time delay inscription of each output arc expression. Finally, the *priority* function assigns priorities to transitions, considering only two levels P_{LOW} and P_{NORMAL}(by default).

Definition 2. *We define a prioritised-timed coloured Petri net (PTCPN) as a tuple* (CPN_T, λ, D, π)*, where:*

– CPN_T *is a CPN according to Definition 1, with the restrictions indicated below.*
– λ *is the labelling function such that*
- $\lambda(p) = k$*, with* $k \in \{in, ok, er, ex, i, v, r\}$*, if* $p \in P$*.*
- $\lambda(t) = q$*, where* q *is a label with* $t \in T$*.*

– $D : T \longrightarrow \mathbb{N} \times \mathbb{N}$ *is the delay function.*
– $\pi : T \longrightarrow \{P_{LOW}, P_{NORMAL}\}$ *is the priority function.* □

In our specific model, a PTCPN will have an only *entry place* p_{in} with colour set *TUNIT* (*UNIT* colour set with time), such that ${}^{\bullet}p_{in} = \emptyset$, which will be initially marked with a single token. According to WS-BPEL and WSRF standards, we can distinguish between two kinds of termination: *normal and abnormal*. On the one hand, the *normal* mode corresponds to the execution of a workflow without faults or without executing any *exit* activity. Thus, in our net model, there is an *output place* p_{ok} with colour set *TUNIT*, such that $p_{ok}^{\bullet} = \emptyset$, which will be marked with one token when the workflow ends normally. On the other hand, a workflow can finish abnormally by means of the execution of an explicit activity (exit or throw) as well as the occurrence of an internal fault in the system. Each PTCPN has also a single *error place* p_{er} with colour set *TUNIT*, which will become marked with one token in the event of a failure, then starting the fault handling activity. In a similar way, the *exit place* (with colour set *TUNIT*) will be marked when the *exit* activity is performed by an orchestrator.

Variable places are denoted by p_v, to mean that they capture the value of variable v. They contain a single token, whose colour is the variable value. We

assume that the initial value of all variables is zero so that these tokens have initially value 0. For any resource r in the system we will have two complementary resource places, p_{r_i}, p_{r_a}. The first one will be marked with one token when the resource has not been instantiated or has been released (due to a time-out expiration), whereas the second one becomes marked when the resource is created, its token colour being a tuple representing the resource identifier (EPR), lifetime, and value. All the remaining places will be considered as *internal*. Markings of PTCPNs are defined in the same way as in [11]. The interested reader is referred to [11] for further information.

3 PTCPN Semantics for WSRF/BPEL/WSN

It is worth noting that we have previously presented an operational semantics for this language in a previous work [13].

3.1 Basic Activities

– *Throw, Empty, Assign, Exit* and *Wait* activities:
These are translated as indicated in Fig. 1, by means of a single transition labelled with the name of the corresponding activity linked with the corresponding terminating place. The time required to execute *assign*, *empty*, *throw* and *exit* is negligible, so that the corresponding transitions have a null delay associated. Notice that for the *assign* activity translation we use a self loop between the transition and the place associated with the variable (p_v) in order

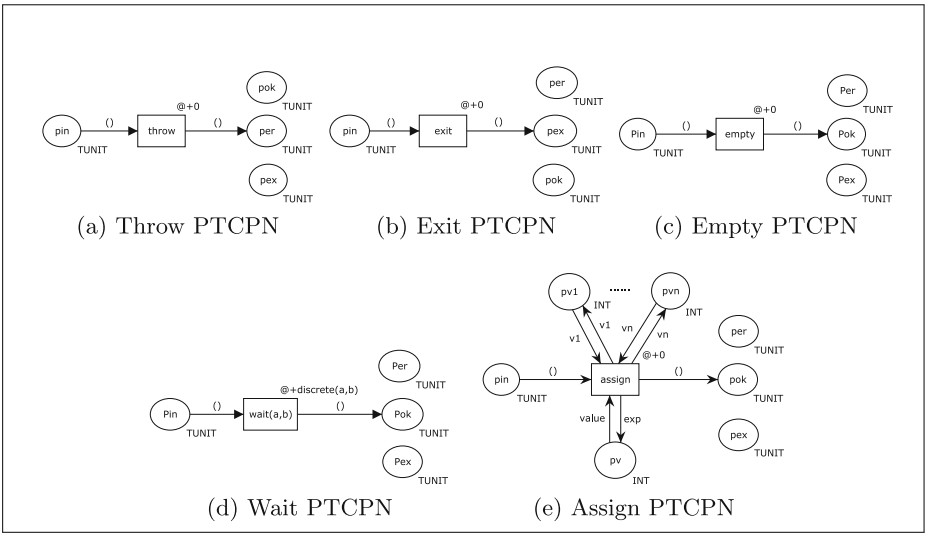
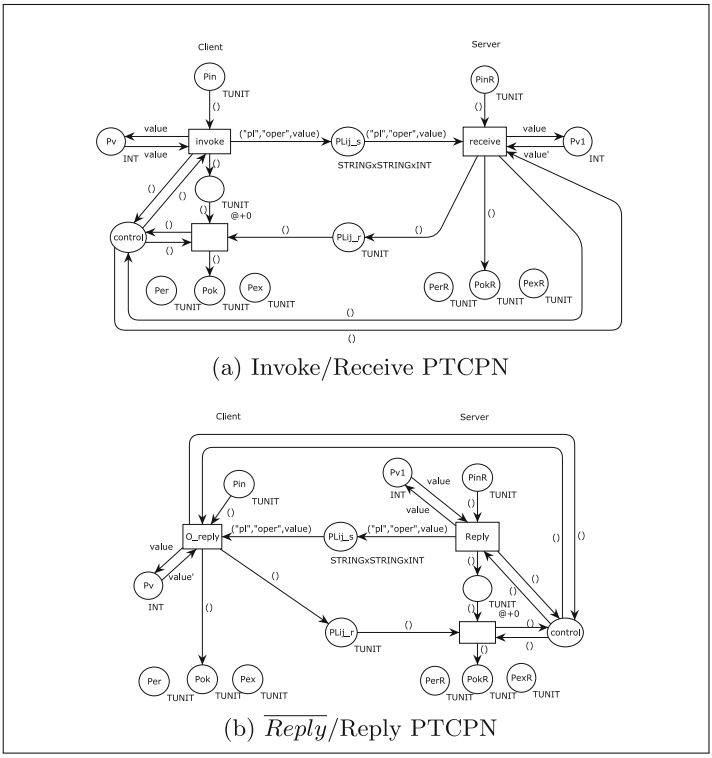


Fig. 1. Basic activities translation

to replace its previous value by the new one, being this new value obtained from an expression (exp) consisting of variables $p_{v1}, \ldots, p_{vn}$ and integers. For the *wait* activity, we have a time interval $[a, b]$ associated, so the delay is randomly selected inside this interval.
Notice the use of a "control" place, to abort all possible remaining activities in the system when either throw or exit are executed. Thus, the idea is that all transitions in the net must be connected with this place, as the different illustrations show.

– *Communication activities:* The model we use is based on the invoke and receive operations, as well as the reply activity that uses a server to reply to a client. We have also added a barred version of reply to synchronise with the response from the client. We have therefore introduced this last activity in our semantics to deal with the request-response operation mode, so the $\overline{reply}$ activity is optional in our syntax.



(a) Invoke/Receive PTCPN

(b) $\overline{Reply}$/Reply PTCPN

Fig. 2. Invoke/receive activities translation

Figure 2 shows the translation for both the invoke/receive and the reply/$\overline{reply}$ pairs of activities. Part Fig. 2a of the figure corresponds to the invoke/receive translation, in which the net of the invoke activity is depicted on the

left-hand-side part, whereas the receive activity is depicted on the right-hand-side part. There are two shared places, PL_{ij_s} and PL_{ij_r}, which are used to implement the synchronisation between the invocation and reception of services. Both places are associated to the partnerlink used for this communication, denoted here by (i, j), where i and j are the orchestrator identifiers performing those activities. Notice that the value of a single variable is transmitted, which is obtained from the corresponding variable place, p_v. In the same way, the receive activity stores this value in its own variable. The interpretation of Fig. 2b is analogous.

3.2 Ordering Structures

The set of structured activities in WS-BPEL is not intended to be minimal [1], so there are cases where the semantics of one activity can be represented using another activity. Nevertheless, in order to reduce the complexity of our translation, our approach omits many derived activities only dealing with the most important ones from the modelling viewpoint, such as sequence, parallel and choice. For all these cases we provide the translation by only considering two activities. However, the generalization to a greater number of activities is straightforward in all of them.

- *Parallel:* The translation for a parallel activity is depicted in Fig. 3, which includes two new transitions $t1$ and $t2$. The first to fork both parallel activities and the second to join them when correctly terminated. Transition $t1$ thus puts one token on the initial places of both PTCPNs, N_{A_1} and N_{A_2}, in order to activate them, and also puts one token on a new place, p_c, which is used to stop the execution of one branch when the other has failed or the exit activity is explicitly executed in one of them. This place is therefore a precondition of every transition in both PTCPNs, and it is also a postcondition of the non-failing transitions. However, in the event of a failure or an exit activity, the corresponding *throw* or *exit* transition will not put the token back on p_c, thus halting the other parallel activity.
  Notice also that the *error* places of N_{A_1} and N_{A_2} have been joined in a single error place (p_{er}), which becomes marked with one token on the firing of one *throw* transition. In this case, the other activity cannot execute any more actions (p_c is empty), so some dead tokens would remain permanently on some places in the PTCPN. However, these tokens cannot cause any damage, since the control flow has been transferred either to the fault handling activity of the PTCPN, once the place p_{er} has become marked, or the whole system has terminated once the place p_{ex} is marked.
- *Sequence:* A sequence of two activities $A_1; A_2$ (with PTCPNs N_{A_1} and N_{A_2}, respectively) is translated in a simple way by just collapsing in a single place (this will be an internal place of the new PTCPN) the *output* place P_{ok} of N_{A_1}, and the *entry* place of N_{A_2}. The *entry* place of the new PTCPN will be the *entry* place of N_{A_1}. The *output* place of the new PTCPN will be the *output* place of N_{A_2}, and we also collapse the *exit*, *error* and *control* places of both PTCPNs.

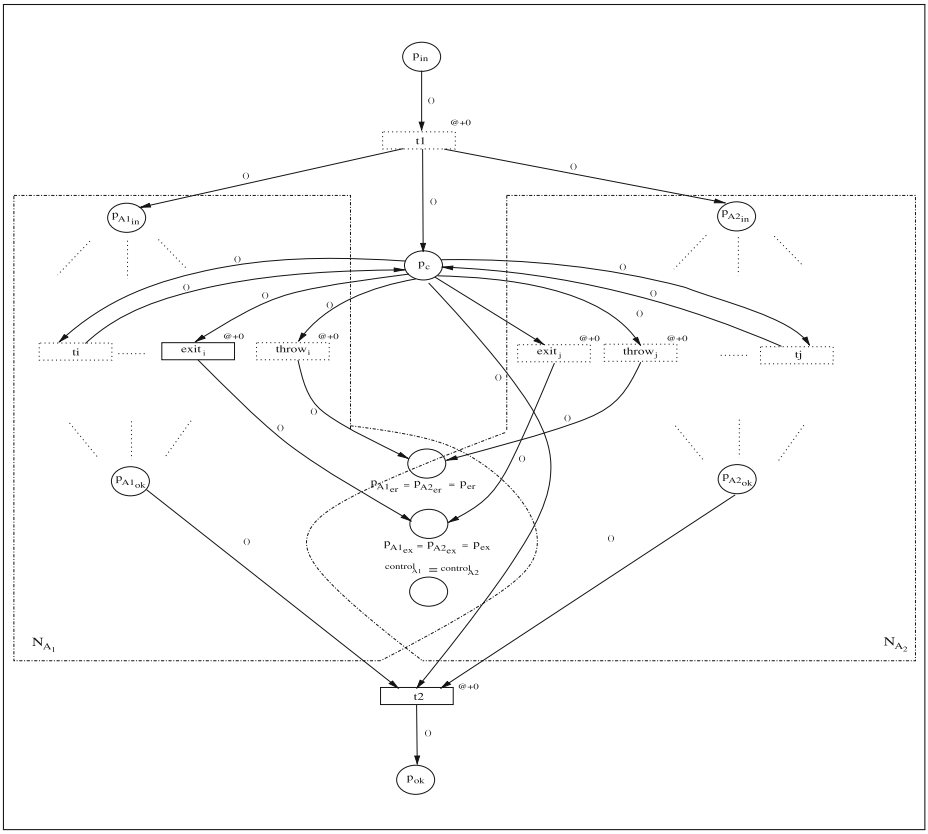


Fig. 3. Parallel activity translation.

– *Pick* $(\{(pl_i, op_i, v_i, A_i)\}_{i=1}^{n}, A, timeout)$: The <pick> activity waits for the occurrence of exactly one event from a set of events, also establishing a timeout for this selection. The translation is depicted in Fig. 4 where a timer is implemented on the place p_a in order to enforce the firing of transition ta when the timeout has elapsed, thus activating N_A. The colour set INT of the place p_a is timed. To illustrate how this construction works, we define the following example.

Example 1. In this example, there are three actors: two customers and a seller. The customers contact the seller in order to gather information about a specific product identified by id1 and id2, respectively. The seller checks the stock and sends the requested information to the customers. The seller has established a timeout of 24 h to receive requests. Let the orchestrations $O_{c1} = (A_{c1}, empty)$, $O_{c2} = (A_{c2}, empty)$ and $O_s = (A_s, empty)$, the BPEL-RF code for the primary activity of both participants is:

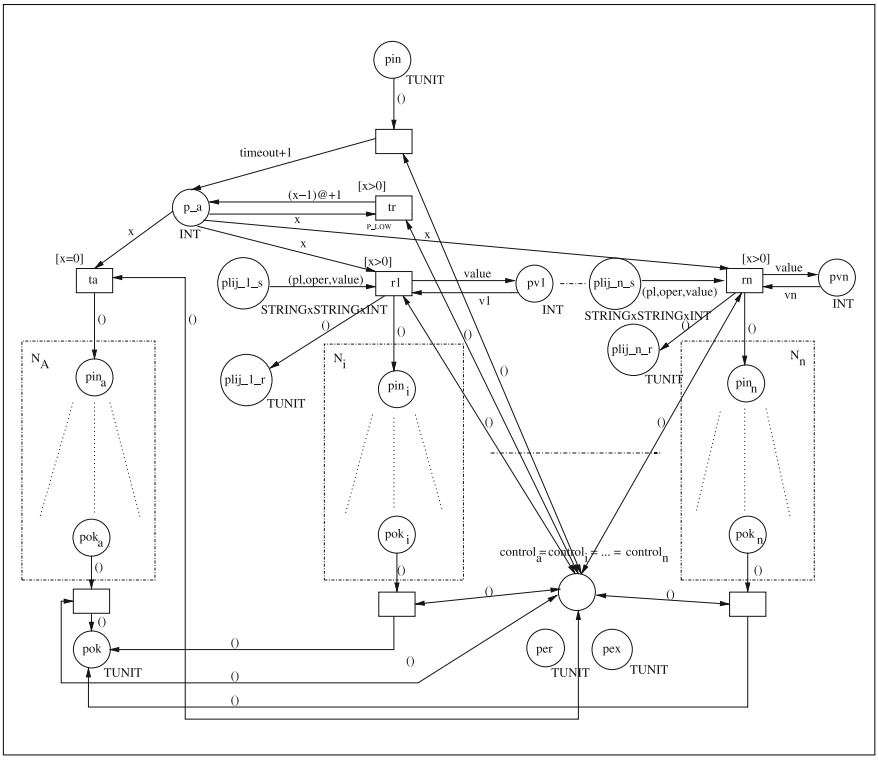


Fig. 4. Pick activity translation.

$$A_{c1} = invoke(pl_1, info, id1); receive(pl_1, inforec1, id_3)$$
$$A_{c2} = invoke(pl1, info, id1); receive(pl_2, inforec2, id_4)$$
$$A_s = pick(\{(pl_1, info, id_{s1}, reply(pl_1, inforec1, id_3), (pl_2, info, id_{s2}, reply(pl_2, inforec2, id_4))\}, empty, 24)$$

Looking at Fig. 4, it can be observed that when O_s executes the *pick* activity the input place, p_{in} of the net is marked. Next, transition t_{in} is fired in order to mark the place p_a with the value $timeout + 1$. This timeout is passed as a parameter in the activity and, in this case, its value is equal to 24. Once this place is marked, two possibilities can arise. On the one hand, one of the buyers runs its *i*nvoke activity before timeout expiration, putting a token in the corresponding input place, $plij_{i_s}$ of the transition r_i, $i \in 1, .., n$, and, then, the behaviour hereafter is the same as in the *receive* activity (Fig. 2). On the other hand, if none of the buyers executes an *invoke* activity, the current time must be increased by means of the transition t_r and the arc inscription @ + *1*. Thus, after *t*imeout time units without receiving any request, the alarm transition, t_a is fired executing the activity A passed as parameter. It is worthwhile to remark that variable x is used as a countdown timer since CPNTools does

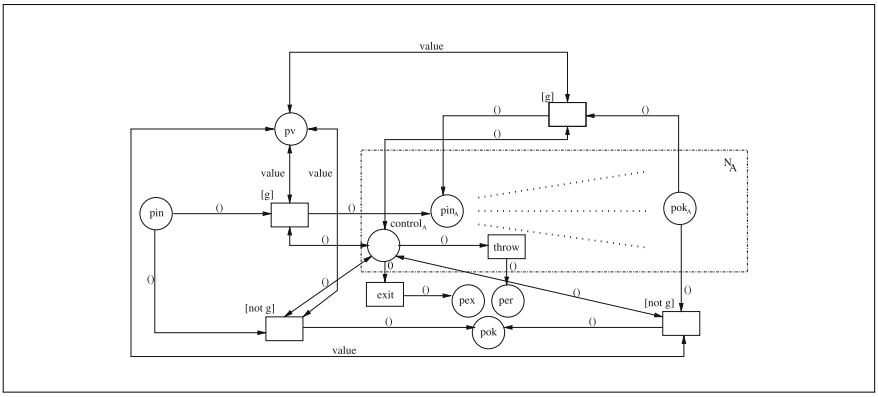


Fig. 5. While activity translation.

not allow to include the *time* function in guards since its inclusion could pose side-effects [4].

– *While* (cond,A): The machinery needed to model this construction is fairly straightforward since we only must check if the repetition condition holds or not in order to execute the contained activity or skip it. Figure 5 shows this translation.

3.3 WSRF-Compliant

Let us now see the WSRF activities, and their corresponding translations.

– *CreateResource (EPR,val,timeout,A)*: *EPR* is the resource identifier, for which we have two complementary places in Fig. 6, p_{r_i} and p_{r_a}, where the sub-index represents the state of the resource: i when it is inactive and a when it is active. The initial value is *val*, and A is the activity that must be executed when the time-out indicated as third parameter has elapsed.
We can see in Fig. 6 how the transition *createResource* removes the token from the *inactive* place, and puts a new token on the active place, whose colour contains the following information: resource identifier (EPR), its lifetime (max), and its value (val). Transition $t0$ is executed when the lifetime of the resource has expired, thus removing the token from the *active* place, marking again the *inactive* place, and activating N_A. We can also see that the *active* place is linked with a number of transitions, which correspond to the subscribers (we know in advance these possible subscribers from the WS-BPEL/WSRF document). These transitions can only become enabled if the corresponding places $subs_i$ are marked by performing the corresponding activity *subscribe*. The PTCPNs $Ncond_i$ are the nets for the activities passed as parameter in the invocation of a subscribe activity.
– Subscribe (EPR,cond′,A): In this case, an orchestrator subscribes to the resource EPR, with the associated condition $cond'$, upon which the activity

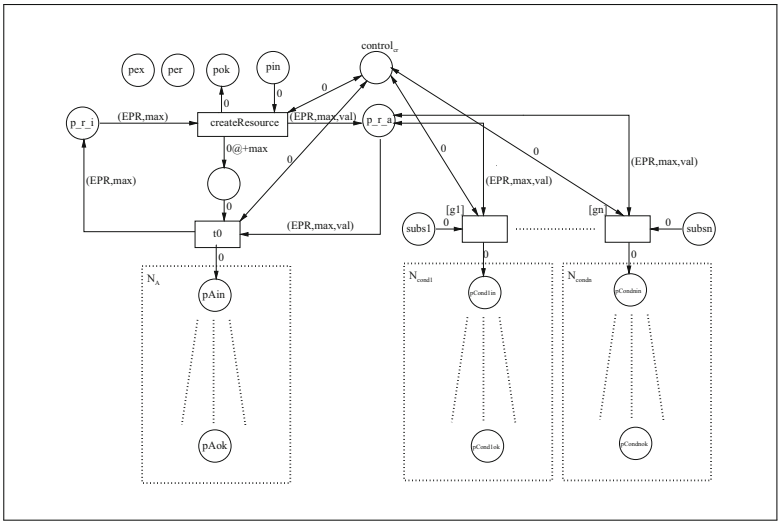


Fig. 6. CreateResource activity translation.

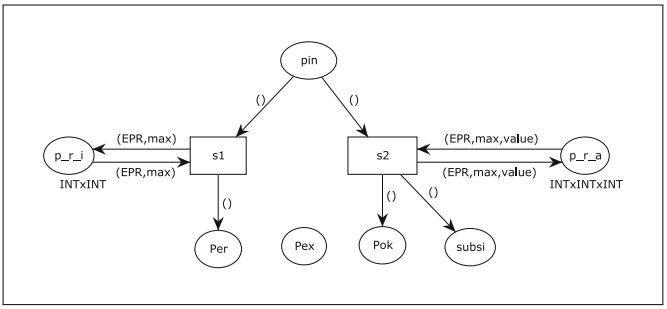


Fig. 7. Subscribe activity translation.

A must be performed. Figure 7 shows this translation, where we can observe that the associated place $subs_i$ is marked in order to allow the execution of the PTCPN for the activity A if the condition g_i holds. On the contrary, if the resource is not active, we will throw the fault handling activity.

– *SetProp (EPR,expr):* In Fig. 8 it can be observed how the new value is assigned to the resource. We omit the translation for the activities *getProp* and *SetLifeTime* since they are similar to this activity.

3.4 Orchestration Translation

Once we have defined the translation for the activities, we can now introduce the definition for the PTCPN at the orchestration level. Notice that all PTCPNs generated for the different orchestrators cooperate to form the entire system (choreography). Let us call N_A and N_f the PTCPNs that are obtained by applying

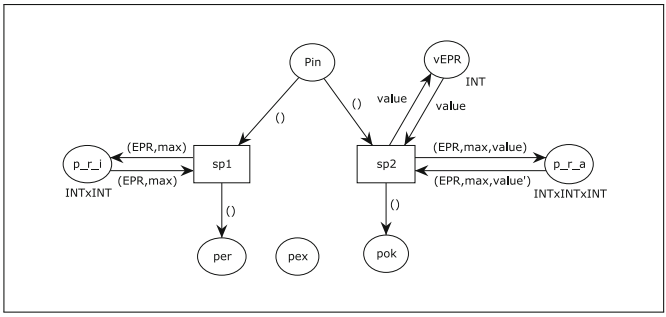


Fig. 8. SetProperty activity translation.

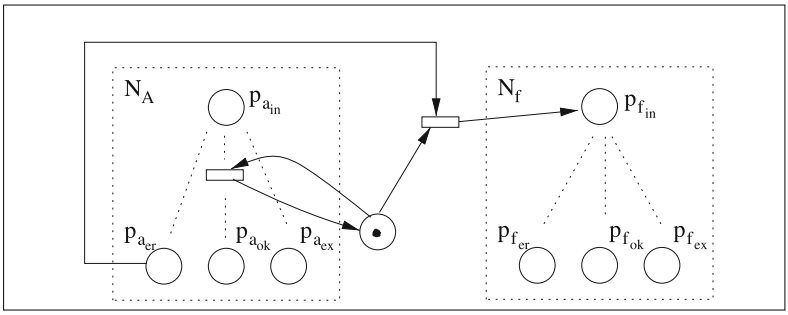


Fig. 9. Orchestration translation

the translation to each one of these activities A and A_f:

$$N_A = (P_a, T_a, A_a, \Sigma_a, V_a, G_a, E_a, \lambda_a, D_a, \pi_a) \qquad \text{(PTCPN for } A\text{)}$$
$$N_f = (P_f, T_f, A_f, \Sigma_f, V_f, G_f, E_f, \lambda_f, D_f, \pi_f) \qquad \text{(PTCPN for } A_f\text{)}$$

Let $p_{a_{in}}$ and $p_{f_{in}}$ be the initial places of N_A and N_f respectively; $p_{a_{ok}}$ and $p_{f_{ok}}$ their *correct* output places, $p_{a_{er}}$ and $p_{f_{er}}$ their *error* places and, finally, $p_{a_{ex}}$ and $p_{f_{ex}}$ their *exit* places. The PTCPN for the orchestrator is then constructed as indicated in Fig. 9. This PTCPN is then activated by putting one token 0 on $p_{a_{in}}$. However, we can have other marked places, for instance, those associated with integer variables or resources. The other places are initially unmarked.

4 Case Study: Automatic Management System for Stock Market Investments

The case study concerns a typical automatic management system for stock market investments, which consists of *n+1* participants: the online stock market system and n investors, A_i, $i = 1, \ldots, n$. The complete and detailed version of this case study can be obtained in http://www.dsi.uclm.es/retics/bpelrf/casestudies.htm. Here, the resource will be the stocks of a company that the investors want to

buy just in case the price falls below an established limit, which the investors fix previously by means of subscriptions, i.e., an investor subscribes to the resource (the stocks) with a certain guard (the value of the stocks he/she want to pay for it). The lifetime *lft* will be determined by the stock market system and the resource price will be fluctuating to simulate the rises/drops of the stock. Notice that we do not take into account the stock buy process since our aim is to model an investors' information system. Thus, the participants will be notified when their bids hold or the resource lifetime expires. Let us consider the choreography $C = (O_{sys}, O_1, \ldots, O_n)$, where $O_k = (A_k, A_{f_k})$, k=sys, 1,..., n; $Var_{sys} = \{at, vEPR\}, Var_i = \{v_i\}$, $A_{f_k=exit}$. Variable $vEPR$ serves to temporarily store the value of the resource property before being sent; v_i is the variable used for the interaction among participants, and, finally, *at* controls the period of time in which the auction is active. Note that the value x indicates the resource value at the beginning, *at0* is the time that the "auction" is active, and, finally, x_i is the value of the stocks that he/she wants to pay for. Suppose that the variables are initially 0:

$Asys = assign(x + 1, vEPR); assign(at0, at); CreateResource(EPR, lft, x, empty);$
$while(actualTime() <= at, Abid)$
$Abid = getProp(EPR, vEPR); assign(vEPR + bid(), vEPR); setProp(EPR, vEPR);$
$wait(1, 2)$
$A_i = wait(1, 2); subscribe(O_i, EPR, EPR < x_i, Acond_i);$
$pick((pl_i, buy, v_i, empty), empty, at0)$
$Acond_i = getProp(EPR, vEPR); invoke(pl_i, buy, vEPR)$

Here, the function *bid* is used to increase/decrease the stocks value simulating the fluctuation of the stocks price. Next, we present the analysis part.

CPNTools offers us two forms to check the correctness of our system: formal verification and simulation. First, the simulation helps designers to understand how the system exactly works and it is a mean to detect possible errors in early stages of the development process in order to refine the model according the clients' requirements. Besides, formal verification through state space analysis could be done in order to ensure that our system achieves some formal properties such as liveness, deadlock-freeness and so on. In this way, Table 2 shows the results obtained considering 1, 2, 3, 4 or 5 investors. Note that we have considered the following assumptions:

- The "auction" time *at0* is limited to 10 time units.
- The resource is active during 15 time units (*lft=15*).
- The resource value x is 100 money units.
- The value of subscription of each investor i, x_i, is $x - (9 + i)$, that is, if the system has only one investor its subscription guard will be $x < 90$, whereas with 5 investors, the last investor will have a subscription guard of $x < 86$.
- The function *bid* will fluctuate the stocks price between -2 and 1 in order to simulate that the price only can rise 1 and drop 2 at most each time unit.

We will focus on deadlock-freeness to ensure that the system never gets stuck while the participants have activities to do in their workflow. We have leveraged

Table 2. State space analysis results

Properties	Number of investors				
	1	2	3	4	5
State Space Nodes	3561	7569	16983	50350	89879
State Space Arcs	5203	12843	33271	112101	262215
Time (s)	2	7	23	146	1140
Dead Markings	124	244	454	1108	874

the functions offered by CPNTools to demonstrate that in all dead markings of the system the final place is marked, which leads us to conclude the system has finished correctly. Let us suppose that the final place of this Petri net is called *Pokfinal0* and this final place is marked by a transition when all the participants have finished their execution. For the sake of clarity, we have not drawn this place in each figure. Thus, the next SML code checks when this situation occurs: fun DesiredTerminal n =((Mark.PetriNet'Pokfinal0 1 n) == 1'true), which returns *true* if the place *Pokfinal0* is marked. In addition to this, it is necessary to evaluate the predicate: PredAllNodes DesiredTerminal=ListDeadMarkings(), to check that the list of dead marking contains the marking of the *Pokfinal0* place.

By using CPNTools, we checked that all dead markings hold the predicate *DesiredTerminal*, and, therefore, when the system reaches a dead marking is because system has terminated, which demonstrates the absence of deadlocks in our case study.

5 Conclusions and Future Work

In this paper, we have integrated two complementary approaches in order to improve the definition of business processes models on BPEL by adding the capability of storing their state. We have thus transformed *stateless* business processes into *stateful* business processes. To this end, we have defined a prioritised-timed coloured Petri net model and presented its corresponding semantics to represent the constructions of WS-BPEL and the standard selected for the definition of resources, namely WSRF. Apart from including the notion of state in business processes, our work also includes a publish-subscribe notification system based on WS-BaseNotification, presenting a PTCPN model and its semantics. Thus, an orchestrator can show interest of being notified when a condition holds, e.g., the load of a server exceeds a certain limit. Our approach is based on the one used in CPNTools, allowing us to take advantage of its capability of analysis and verification systems. Moreover, our work in progress is the development of a tool (a beta version can be accessed at: http://www.dsi.uclm.es/retics/bpelrf/) to transform automatically WS-BPEL and WSRF specifications into CPNTools nets. As future work, we plan to study some interesting properties such as safeness, soundness and so on. In addition, it is interesting to define a complete semantics of WS-BPEL and WSRF. Finally, as commented above, we

defined an operational semantics in a previous work, so we will demonstrate in a future work the equivalence between both semantics.

References

1. Andrews, T., et al.: BPEL4WS - Business Process Execution Language for Web Services, Version 1.1 (2003). http://www.ibm.com/developerworks/library/specification/ws-bpel/
2. Banks, T.: Web Services Resource Framework (WSRF) - Primer, OASIS (2006)
3. Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and orchestration: a synergic approach for system design. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 228–240. Springer, Heidelberg (2005)
4. CPNTools website. http://cpntools.org/
5. Czajkowski, K., Ferguson, D., Foster, I., Frey, J., Graham, S., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W.: The WS-Resource Framework Version 1.0 (2004). http://www.globus.org/wsrf/specs/ws-wsrf.pdf
6. Dragoni, N., Mazzara, M.: A formal semantics for the WS-BPEL recovery framework. In: Laneve, C., Su, J. (eds.) WS-FM 2009. LNCS, vol. 6194, pp. 92–109. Springer, Heidelberg (2010)
7. Lohmann, N.: A feature-complete petri net semantics for WS-BPEL 2.0. In: Dumas, M., Heckel, R. (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 77–91. Springer, Heidelberg (2008)
8. Ezenwoye, O., Sadjadi, S.M., Cary, A., Robinson, M.: Orchestrating WSRF-based GridServices. Technical report FIU-SCIS-2007-04-01 (2007)
9. Farahbod, R., Glässer, U., Vajihollahi, M.; A formal semantics for the business process execution language for Web services. In: Joint Workshop on Web Services and Model-Driven Enterprise Information Services (WSMDEIS), pp. 122–133 (2005)
10. Foster, I., Frey, J., Graham, S., Tuecke, S., Czajkowski, K., Ferguson, D., Leymann, F., Nally, M., Storey, T., Weerawaranna, S.: Modeling Stateful Resources with Web Services, Globus Alliance (2004)
11. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer, Heidelberg (2009)
12. Leyman, F.: Choreography for the grid: towards fitting BPEL to the resource framework. J. Concurrency Comput. Pract. Exp. **18**(10), 1201–1217 (2006)
13. Mateo, J.A., Valero, V., Díaz, G.: An operational semantics of BPEL orchestrations integrating Web services resource framework. In: Carbone, M., Petit, J.-M. (eds.) WS-FM 2011. LNCS, vol. 7176, pp. 79–94. Springer, Heidelberg (2012)
14. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: Formal semantics and analysis of control flow in WS-BPEL. Sci. Comput. Program. **67**(2–3), 162–198 (2007)
15. Qiu, Z., Wang, S.-L., Pu, G., Zhao, X.: Semantics of BPEL4WS-like fault and compensation handling. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 350–365. Springer, Heidelberg (2005)
16. Slomiski, A.: On using BPEL extensibility to implement OGSI and WSRF grid workflows. J. Concurrency Comput. Pract. Exp. **18**, 1229–1241 (2006)