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*

Escuela Superior de Ingeniería Informática, Universidad de 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 its 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 have emerged, which require to capture the state of some resources. Thus, new standards to model Web services states have appeared, such as Open Grid Services Infrastructure (OGSI), which became 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 BPEL, but we also consider other important aspects, both from BPEL and WSRF, such as the fault and event handlers, time-outs and the WSRF publish-subscribe system.

Keywords: Web service composition, orchestration, stateful workflow, WSRF, WS-BPEL, timed coloured Petri nets.

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. In these systems, the service provider needs to ensure some levels of quality and privacy to the final user in a way that had never been raised. It is therefore necessary to develop new techniques to benefit from the advantages of recent approaches, as Web service compositions. Formal models of concurrency have been widely used for the description and analysis of concurrent and distributed systems. Grid/Cloud environments are characterized by a dynamic environment due to the heterogeneity and volatility of resources. There are two complementary views to composite Web

* This work received financial support from the Spanish Government, Secretaría de Estado de Investigación (cofinanced by FEDER funds) through the TIN2009-14312-C02-02 Project, and the JCCLM Regional Project PEII09-0232-7745.

services: Choreography and Orchestration. The choreography view describes the observable interactions among services and can be defined by using specific languages such as Web Services Choreography Description Language (WS-CDL) or by using more general languages like UML Messages Sequence Charts (MSC). On the other hand, orchestration concerns the internal behaviour of a Web service in terms of invocations to other services. Web Services Business Process Execution Language (WS-BPEL) [1] is normally used to describe Web service orchestrations, so this is considered the de-facto standard language for describing Web services workflows in terms of Web service compositions.

To facilitate additional interoperability among services, more standardization is required to deal with distributed resources. In January of 2004, several members of the *Globus Alliance* organization and the computer multinational *IBM* with the help of experts from companies such as *HP, SAP, Akamai, etc.* defined the basis architecture and the initial specification documents of a new standard for that purpose, Web Services Resource Framework (WSRF) [9]. 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 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 [4].

The main motivation of this work is to provide a formal semantics for WS-BPEL+WSRF to manage stateful Web services workflows by using the existing machinery in distributed systems, and specifically a well-known formalism, such as prioritised-timed Petri nets, which are a graphical model, but they also provide us with the ability to simulate and analyse the modelled system.

Thus, our aim is not to provide just another WS-BPEL semantics. In order to deal with the integration of BPEL plus WSRF in a proper way, we have realized that it is more convenient to introduce a specific semantic model, which covers properly all the relevant aspects of WSRF such as notifications and resource time-outs. The integration of WS-BPEL and WSRF is not new; in the literature, there are a bundle of works defining this integration, but none of these works define a formal semantics in terms of Petri nets. In the next Section we present an overview of these works.

2 Background and Related Work

In this Section, we provide an overview of WS-BPEL and WSRF, and we also review some related works.

2.1 Overview of BPEL/WSRF

WS-BPEL [1], for short BPEL, is an OASIS orchestration language for specifying actions within Web service business processes. 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. Thus, we will consider a composite Web service consisting of a set of orchestrators, described by BPEL+WSRF syntax, which exchange messages through some communication channels, which are called *PartnerLinks* in BPEL terminology. 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.

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 event and fault activities, which are executed upon the occurrence of some events, or due to some execution failures, respectively. 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 a barred form of *reply* action, which is used when a service invocation expects a reply, in order to implement the synchronization with the *reply* action from the server.

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*).

WSRF [2] is a resource specification language developed by OASIS and some of the most pioneering computer companies, whose purpose is to define a generic framework for modelling Web services with stateful resources, as well as the relationships among these services in a Grid/Cloud environment. This approach consists of a set of specifications that define the representation of a WS-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. We now describe the WSRF elements that are considered in the BPEL+WSRF framework:

– **WS-ResourceProperties**: There is a precise specification to define WS-Resource properties, based on a Resource Properties Document (RPD), which defines the properties of the associated resource (disk size, processor capacity, ...). Nevertheless, for simplicity, we only consider a single property for each resource, which is an integer value. Resources are identified by their EPRs (End-Point References), so we will also use this mechanism for

identification purposes, but, for simplicity, we will consider these references as static, instead of assuming a dynamic mechanism to assign them. As a shorthand notation, EPRs will also be used to denote the resource property values.

Furthermore, a WSDL file must be provided in order to facilitate the allowed resource operations. Among the operations allowed by the standard are *GetResourceProperty* and *SetResourceProperty*, which are used to manipulate the resource property values.

- **WS-ResourceLifetime**: The WSRF specification does not provide a standard way to create resources. However, resources have an associated lifetime, which means that once this time has elapsed, the resource is considered to be destroyed, and the subscribers are correspondingly notified. We have then included, for completeness, an operation to create resources, *createResource*, in which the initial value of the resource, its lifetime and the activity that must be launched upon its destruction are indicated. We also have an operation in order to modify the current resource lifetime, *setTimeout*.
- **WS-Notification**: Clients can subscribe to WSRF resources in order to be notified about some topics (resource conditions). We therefore include the *subscribe* operator for a customer to subscribe to a resource, indicating the condition under which it must be notified, and the activity that must be executed upon that event.

In WSRF there are some additional technical elements to increase the modelling power that due to its technical nature are not considered in our framework. Among them, we have the so-called *WS-Basefaults*, which define a standard format for delivering error messages. *WS-ServiceGroup* is a tool to create "Service groups", which can be created with the purpose of sharing a common set of properties, i.e. it is a mechanism for grouping together different Web services with similar behaviour. Finally, *WS-BrokeredNotification* allows us to define *NotificationBrokers*, which are intermediary elements who, among other things, allow interactions between one or more *NotificationPublishers* and one or more *NotificationConsumers*.

2.2 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 [15] 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 [11] and [7]. The first justifies the use of BPEL extensibility to allow the combination of different GRIDs, whereas Ezenwoye et al. [7] share their experience on BPEL to integrate, create and manage WS-Resources that implement the factory/instance pattern.

On the other hand, Ouyang et al. [13] define the necessary elements for translating BPEL processes into Petri nets. Thus, they cover all the important aspects in the standard such as exception handling, dead path elimination and so on. The model they consider differs from ours in that we formalize the whole system as a composition of orchestrators with resources associated, whereas they describe the system as a general scope with nested sub-scopes leaving aside the possibility of administering resources. Furthermore, we have also formalized the event handling and notification mechanisms. Another extensive semantics for BPEL 2.0 is presented in [6] by Dumas et al, which introduces two new interesting improvements. They define several patterns to simplify some huge nets and introduce the semantics for the WS-BPEL 2.0 new patterns. Related to π-calculus semantics, Dragoni and Mazzara [5] propose a theoretical scheme focused on dependable composition for the WS-BPEL recovery framework. In this approach, the recovery framework is simplified and analysed via a conservative extension of π-calculus. The aim of this approach clearly differs from ours, but it helps us to have a better understanding of the WS-BPEL recovery framework. Other work focused on the BPEL recovery framework is [14]. Although this is more focused in the compensation handler, they describe the corresponding rules that manage a Web service composition. Our work is therefore quite complete as we define rules for nearly all possible activities. In addition, we also consider time constraints. Finally, we would like to highlight the works of Farahbod et al. [8] and Busi et al. [3]. In the first one, the authors extract an abstract operational semantics for BPEL based on abstract state machines (ASM) defining the framework BPEL_{AM} to manage the agents who perform the workflow activities. In this approach time constraints are considered, but they do not formalize the timed model. On the other hand, the goal of the latter one is fairly similar to ours. They also define a π-calculus operational semantics for BPEL and describe a conformance notion. They present all the machinery to model Web service compositions (choreographies and orchestrations). The main difference with our work is that we deal with distributed resources.

For further details about the formalization of service oriented languages we would like to encourage the reader to review the works presented at the SENSORIA project in [16]. Here, an extensive work is presented from different international research groups aimed by the common goal of providing a rigorous software engineering view point for service-oriented system using as a cornerstone the formal specification of Web Services and WS-BPEL in particular.

3 Prioritised-Timed Coloured Petri Nets

In this section we introduce the specific model of prioritised-timed coloured Petri net that we consider for the translation.

4 Prioritised-Timed Coloured Petri Net Semantics for WS-BPEL+WSRF

4.1 Prioritised-Timed Coloured Petri Nets

We use prioritised-timed coloured Petri nets, which are a prioritised-timed extension of coloured Petri nets [10], the well-known model supported by CPN Tools[1].

Definition 1. (Prioritised-Timed Coloured Petri Nets)
We define a prioritised-timed coloured Petri net (PTCPN) as a tuple $(P, T, A, V, G, E, \lambda, D, \pi)$, where[2]:

- P is a finite set of *coloured places*. Colours used in this semantics will be introduced progressively, as we define the PTCPNs corresponding to each activity. We will use timed and untimed coloured tokens, so timed tokens will have associated a time stamp, according to the CPNTools interpretation [10].
- T is a finite set of *transitions* ($P \cap T = \emptyset$).
- $A \subseteq (P \times T) \cup (T \times P)$ is a set of directed *arcs*.
- V is a finite set of *integer variables* i.e. $Type(v) \in \mathbb{Z}$, for all $v \in V$. We will assume that all variables have 0 as initial value.
- $G : T \longrightarrow EXPR_V$ is the *guard function*, which assigns a Boolean expression to each transition, i.e. $Type(G(t)) = Bool$. $EXPR_V$ denotes the expressions constructed using the variables in V, with the same syntax admitted by CPN Tools.
- $E : A \longrightarrow EXPR_V$ is the *arc expression function*, which assigns an expression to each arc.
- λ is the *labelling function*, defined both on places and transitions. Transitions can be labelled with either activity names or $\emptyset$. 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: $\{in, ok, er, ex, i, v, r\}$. In our specific model, a PTCPN will have an only *entry place* p_{in}, such that ${}^\bullet p_{in} = \emptyset$, which will be initially marked with a single token, whose colour value will be 0. According to WS-BPEL and WSRF standards, we can distinguish between two kind 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}, such that $p_{ok}^\bullet = \emptyset$, which will be marked with one token of colour 0 when the workflow ends normally. On the other hand, a workflow can finish abnormally by means of the execution of

[1] Official web page: http://cpntools.org/

[2] We use the classical notation on Petri nets to denote the precondition and postcondition of both places and transitions:

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

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}, which will become marked with one token of colour 0 in the event of a failure, then starting the fault handling activity. In a similar way, the *exit place* will be marked when the *exit* is executed 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. 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, value, list of subscribers and activity to be executed upon the time-out expiration. All the remaining places will be considered as *internal*.

- $D : T \longrightarrow \mathbb{N} \times \mathbb{N}$ *(delay function)*, which associates a time interval to each transition. For $D(t) = [d_1, d_2]$, this means that the time delay associated to t can be any value in this interval, all of them with the same probability.
- $\pi : T \longrightarrow \mathbb{N}$ is the *priority function*, which assigns a priority level to each transition. □

Markings of PTCPNs are defined in the same way supported by CPNTools, as well as the semantics of PTCPNs, so due to the lack of space we omit the formal definitions. The interested reader may see for instance [10].

4.2 PTCPN Semantics for WSRF/BPEL

Before introducing the PTCPN semantics, we define the formal model that captures the integration of BPEL and WSRF.

A system for our purposes consists of a set of orchestrators that run in parallel using a set of distributed resources. Orchestrators relate with one another by invoking the services they respectively provide. This set of orchestrators and resources is here called a *choreography*. We use the following notation: *ORCH* is the set of orchestrators in the system, *Var* is the set of integer variable names, *PL* is the set of partnerlinks, *OPS* is the set of operation names that can be performed, *EPRS* is the set of resource identifiers, and A is the set of basic or structured activities that can form the body of a process.

An orchestrator O is defined as a tuple $O = (PL, Vars, A, A_f, \mathcal{A}_e)$, where *PL* are the partnerlinks this orchestrator uses to communicate with others, *Vars* is the set of local variables of this orchestrator, A and A_f are activities of WS-BPEL and WSRF, and $\mathcal{A}_e$ is a set of activities. Specifically, A represents the normal workflow, A_f is the orchestrator fault handling activity and $\mathcal{A}_e = \{A_{e_i}\}_{i=1}^{m}$ are the event handling activities.

Activities in BPEL-WSRF follow the syntax defined by the following BNF expression (see Table 1 for the equivalence with the XML syntax of BPEL and WSRF):

WS-BPEL/WSRF Syntax	Model
<process ...> <partnerLinks> ... </partnerLinks>? <Variables> ... </Variables>? <faultHandlers> ... </faultHandlers>? <eventHandlers> ... </eventHandlers>? (activities)* </process>	$(PL,Var,A,A_f,\mathcal{A}_e)$
<throw/>/any fault	throw
<receive partnerLink="pl" operation="op" variable="v" createInstance="no"> </receive>	receive(pl,op,v)
<reply partnerLink="pl" variable="v"> </reply>	reply(pl,v)
<invoke partnerLink="pl" operation="op" inputVariable="v_1" outputVariable="v_2"?> </invoke>	invoke(pl,op,v_1); [$\overline{reply}$(pl,op,v_2)]
<empty> ... </empty>	empty
<exit> ... </exit>	exit
<assign><copy><from>expr</from><to>v_1</to></copy></assign>	assign(expr,v_1)
<wait><from>a</from><to>b</to> </wait>	wait(a,b)
<sequence> $activity_1$ $activity_2$ </sequence> <flow> $activity_1$ $activity_2$ </flow>	A_1 ; A_2 $A_1 \parallel A_2$
<while><condition>cond</condition>$activity_1$</while>	while(cond,A)
<pick createInstance="no"> <onMessage partnerLink="pl" operation="op"variable="v"> $activity_1$ </onMessage> <onAlarm><for>timeout</for>$activity_1$</onAlarm> </pick>	pick($\{(pl_i, op_i, v_i, A_i)\}_{i=1}^{n}$, A,timeout)
<invoke partnerLink="Factory"operation="CreateResource" inputVariable="val,timeout" outputVariable="EPR"> </invoke><assign><copy><from variable="EPR">part="ref" query="/test:CreateOut/wsa:endpointreference"</from> <to> partnerlink="Factory"</to></copy></assign>	createResource(EPR,val,timeout,A)
<wsrp:GetResourceProperty> <wsa:Address>EPR< /wsa:Address> variable_identifier </wsrp:GetResourceProperty>	getProp(EPR,v)
<wsrp:SetResourceProperties> <wsa:Address>EPR< /wsa:Address> <wsrp:Update>expression</wsrp:Update> < /wsrp:SetResourceProperties>	setProp(EPR,expr)
<wsrl:SetTerminationTime> <wsa:Address>EPR< /wsa:Address> <wsrl:RequestedTerminationTime> timeout </wsrl:RequestedTerminationTime> </wsrl:SetTerminationTime>	setTimeout(EPR,timeout)
<wsnt:Subscribe> <wsnt:ProducerReference>EPR</wsnt: ProducerReference> <wsnt:Precondition>cond′</Precondition> </wsnt:Subscribe>	subscribe(EPR,cond′,A)
<wsnt:Notify> <wsnt:NotificationMessage> <wsnt:ProducerReference>EPR</wsnt:ProducerReference> <wsnt:Message> ... </wsnt:Message> </wsnt:NotificationMessage> </wsnt:Notify>	Executes the associated event handler activity

Table 1: Conversion table

$$
\begin{aligned}
&A ::= \mathit{throw} \mid \underline{\mathit{receive}}(pl, op, v) \mid \mathit{invoke}(pl, op, v_1) \mid \mathit{exit} \mid \\
&\ \mathit{reply}(pl, v) \mid \overline{\mathit{reply}}(pl, op, v_2) \mid \mathit{assign}(expr, v_1) \mid \mathit{empty} \mid \\
&\ A\,;A \ \mid\ A \,\|\, A \mid \mathit{while}(cond, A) \mid \mathit{wait}(a, b) \mid \\
&\mathit{pick}(\{(pl_i, op_i, v_i, A_i)\}_{i=1}^{n}, A, \mathit{timeout}) \mid \mathit{getProp}(EPR, v) \mid \\
&\mathit{createResource}(EPR, val, \mathit{timeout}, A) \mid \\
&\mathit{setProp}(EPR, expr) \mid \mathit{setTimeout}(EPR, \mathit{timeout}) \mid \\
&\mathit{subscribe}(EPR, cond', A)
\end{aligned}
$$

where $O \in ORCH, EPR \in EPRS, pl, pl_i \in PL, op, op_i \in OPS, a, b \in \mathbb{N}, a \leq b$, *expr* is an arithmetic expression constructed by using the variables in *Var* and integers; v, v_1, v_2, v_i range over *Var*, and $val \in \mathbb{Z}$. A condition *cond* is a predicate constructed by using conjunctions, disjunctions, and negations over the set of variables *Var* and integers, whereas *cond′* is a predicate constructed by using the corresponding *EPR* (as the resource value) and integers. Notice that *setProp* and *getProp* do not contain the property name since, for simplicity, we are only considering a single property for each resource. We therefore use its EPR as representative of this property. It is worth noting that we have previously presented an operational semantics for this language in the previous work [12].

Let us call N_A, N_f and N_{e_i} the PTCPNs that are obtained by applying the translation to each one of these activities A, A_f, A_{e_i}, with $i \in \{1, m\}$:

$$
\begin{aligned}
N_A &= (P_a, T_a, A_a, V_a, G_a, E_a, \lambda_a, D_a) && \text{(PTCPN for } A\text{)} \\
N_f &= (P_f, T_f, A_f, V_f, G_f, E_f, \lambda_f, D_f) && \text{(PTCPN for } A_f\text{)} \\
N_{e_i} &= (P_{e_i}, T_{e_i}, A_{e_i}, V_{e_i}, G_{e_i}, E_{e_i}, \lambda_{e_i}, D_{e_i}) && \text{(PTCPN for } A_{e_i}\text{)}
\end{aligned}
$$

Let $p_{a_{in}}$, $p_{f_{in}}$ and $p_{e_{i\,in}}$ be the initial places of N_A, N_f and N_{e_i}respectively; $p_{a_{ok}}$, $p_{f_{ok}}$ and $p_{e_{i\,ok}}$ their *correct* output places, $p_{a_{er}}$, $p_{f_{er}}$ and $p_{e_{i\,er}}$ their *error* places and, finally, $p_{a_{ex}}$, $p_{f_{ex}}$ and $p_{e_{i\,ex}}$ their *exit* places. The PTCPN for the orchestrator is then constructed as indicated in Fig. 1. 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. The other places are initially unmarked.

Variables and resources: There is one place for each variable, whose token value is the current variable value. As regards resources, there are two places associated to each resource, p_{r_i}, p_{r_a}. For any resource r, p_{r_a} becomes marked when the orchestrator executes the *createResource* activity, whereas the second one, p_{r_i}, is marked as far as the orchestrator does not execute the *createResource* activity. When the resource lifetime terminates, the resource is released, passing the token from p_{r_a} to p_{r_i}. Observe that we can know in advance the number of resources in the system by reading the WS-BPEL/WSRF document.

4.3 Basic activities

– *Throw, Empty, Assign, Exit* and *Wait* activities:
These are translated as indicated in Fig. 2, 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*,

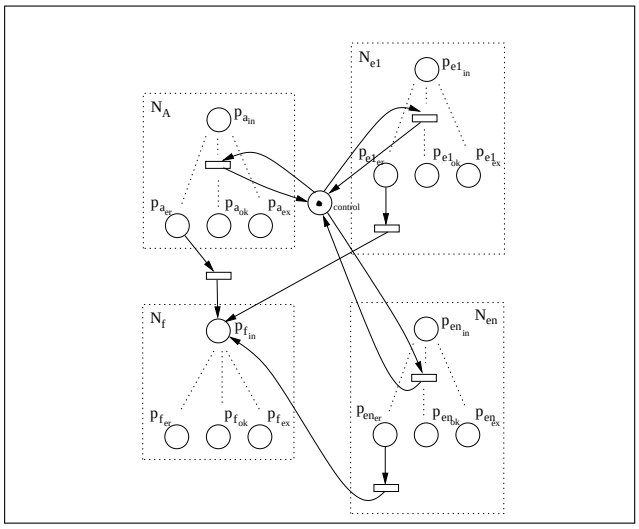
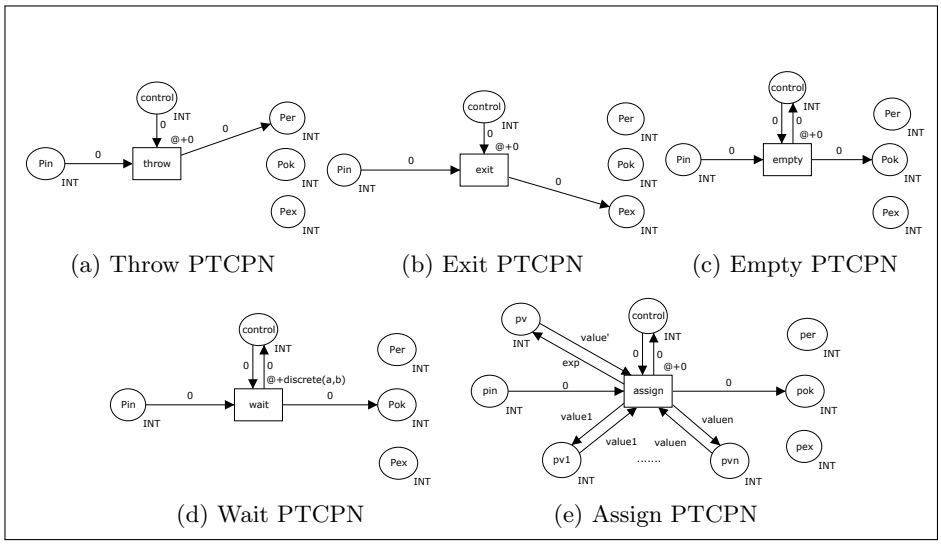


Fig. 1: Orchestration Translation

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 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 arrest 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.

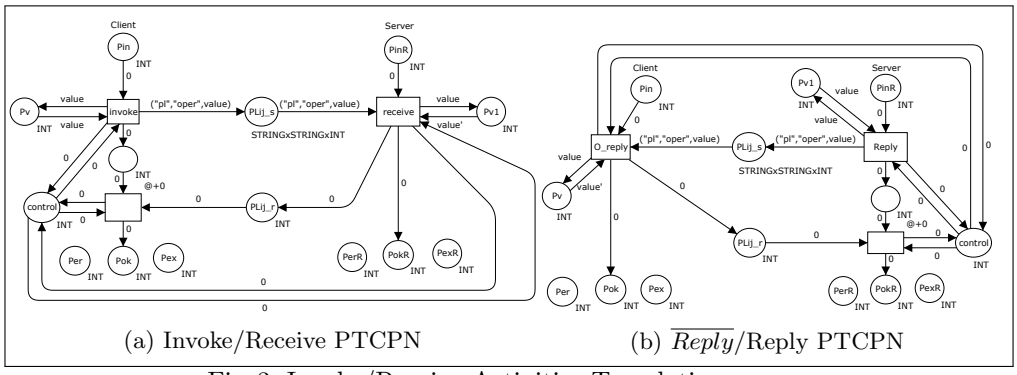


(a) Throw PTCPN (b) Exit PTCPN (c) Empty PTCPN

(d) Wait PTCPN (e) Assign PTCPN

Fig. 2: Basic Activities Translation

– *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 synchronous or asynchronous nature of invoke activity (one-way or request-response operation, respectively), so the $\overline{reply}$ activity is optional in the syntax depicted in Table 1.



(a) Invoke/Receive PTCPN (b) $\overline{Reply}$/Reply PTCPN

Fig. 3: Invoke/Receive Activities Translation

Fig. 3 shows the translation for both the invoke/receive and the reply/$\overline{reply}$ pairs of activities. Part 3a 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. 3b is analogous.

4.4 Ordering structures

WS-BPEL defines structured activities for various control-flow patterns:

– Sequential control between activities is provided by <sequence>, <if>, <while>, <repeatUntil>, and the serial variant of <forEach>.
– Concurrency and synchronization between activities is provided by <flow> and the parallel variant of <forEach>.
– Deferred choice controlled by external and internal events is provided by <pick>.

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.

– *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 (Fig. 4), by just collapsing in a single place (this will be an internal place of the new PTCPN) the *output* place *Pok* 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.

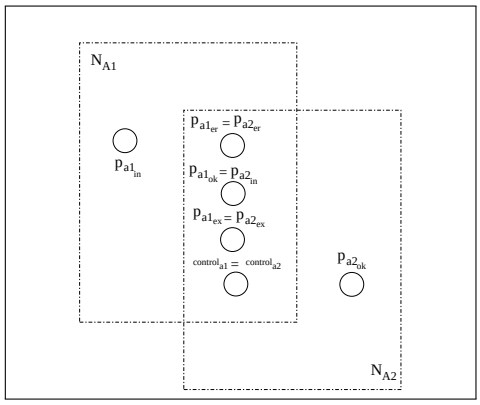


Fig. 4: Sequence Translation

– *Parallel*: The translation for a parallel activity is depicted in Fig. 5, 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 arresting 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.

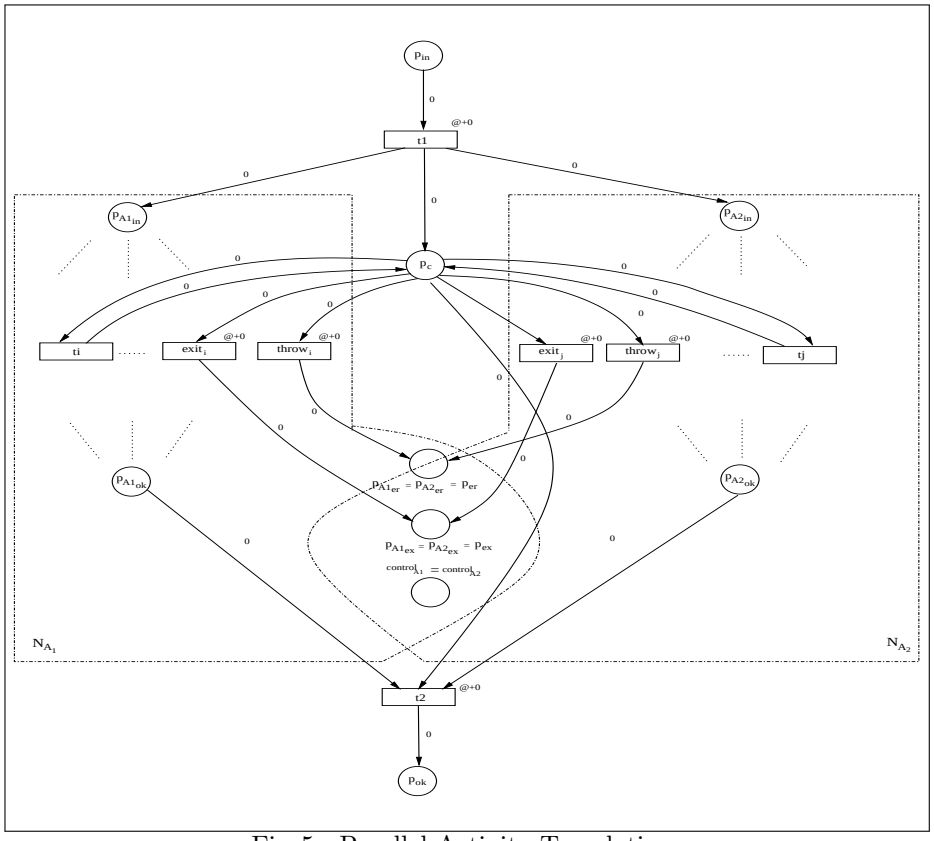


Fig. 5: Parallel Activity Translation.

- *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 time-out for this selection. The translation is depicted in Fig. 6 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. Notice also the use of both timed and untimed places in this figure, respectively called *INT* and *UINT*.
- *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. Fig. 7 shows this translation.

4.5 WSRF-compliant

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

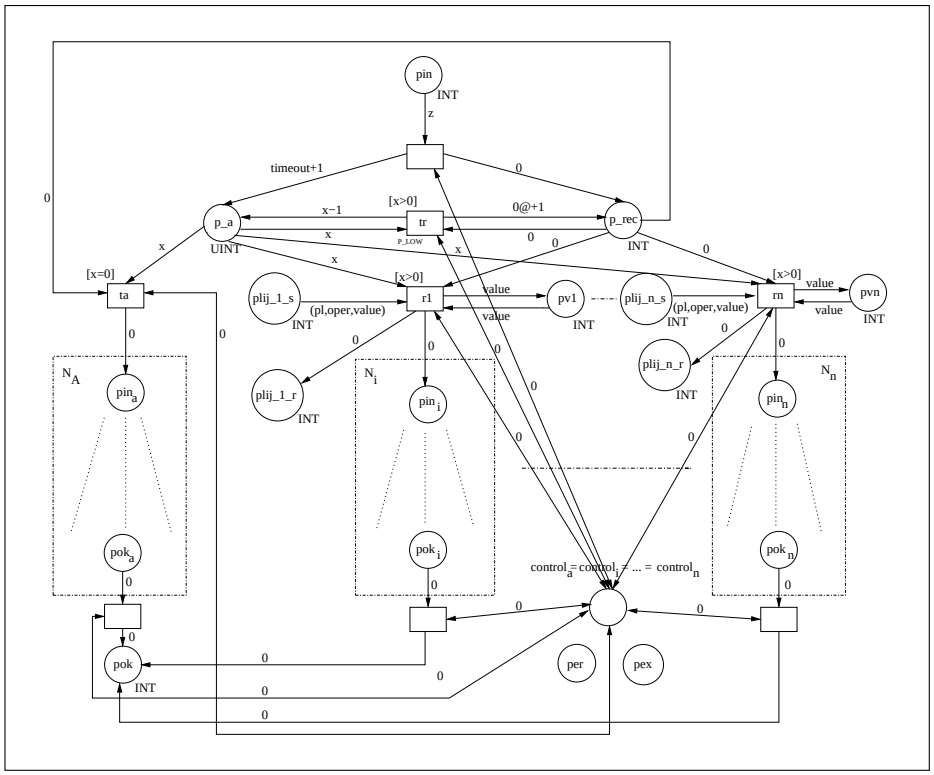


Fig. 6: Pick Activity Translation.

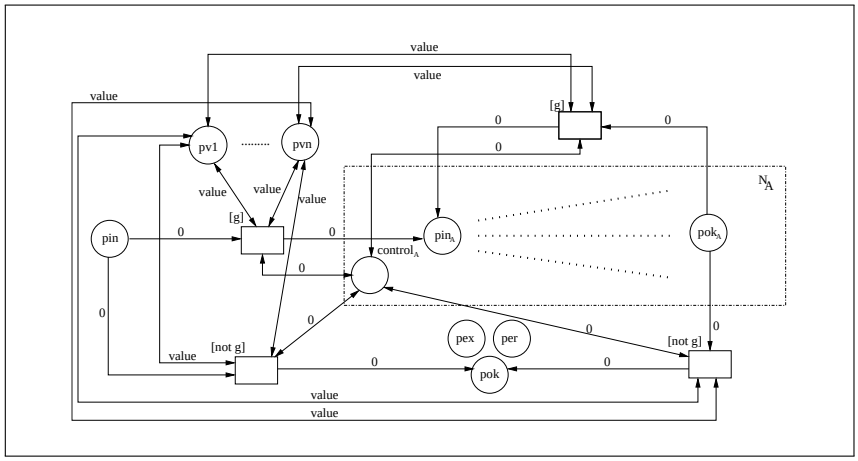


Fig. 7: While Activity Translation.

– *CreateResource (EPR,val,timeout,A)*: *EPR* is the resource identifier, for which we have two complementary places in Fig. 8, 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. 8 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

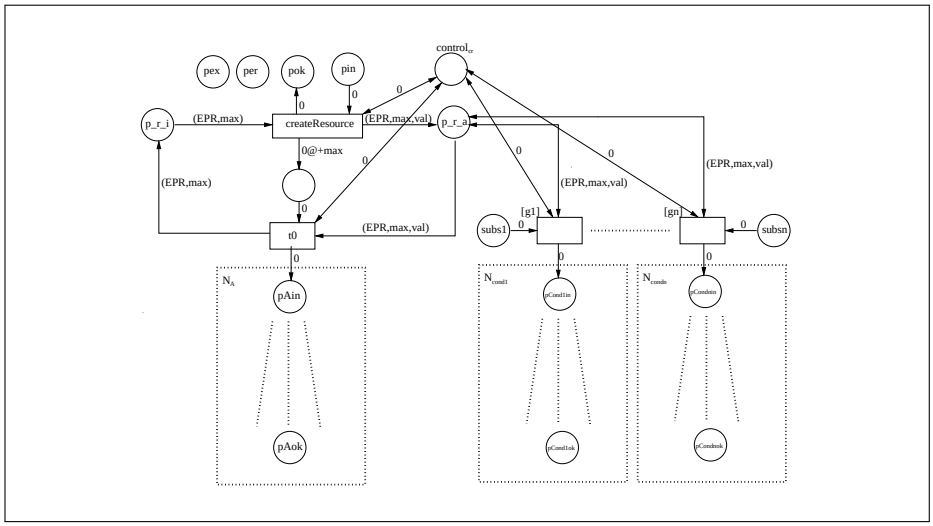


Fig. 8: CreateResource Activity Translation.

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 A must be performed. Fig. 9 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.

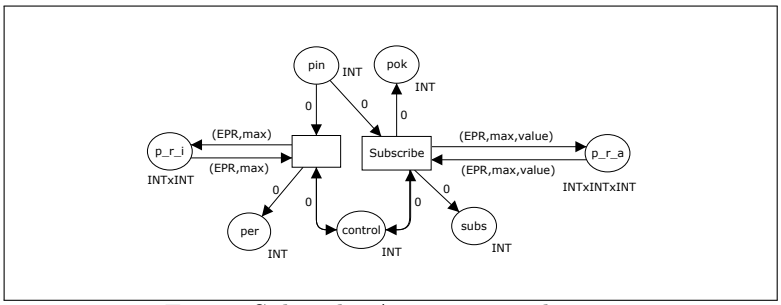


Fig. 9: Subscribe Activity Translation.

– GetProp (EPR,v) and SetProp (EPR,expr): These are easily translated, as shown in Figs. 10 and 11, where the resource value is obtained and assigned to variable v (GetProp), or a new value is assigned to the resource (SetProp).

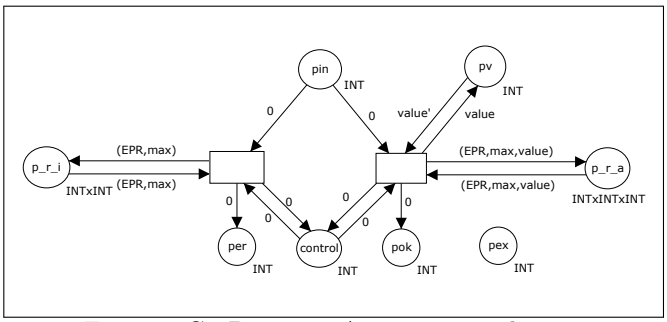


Fig. 10: GetProperty Activity Translation.

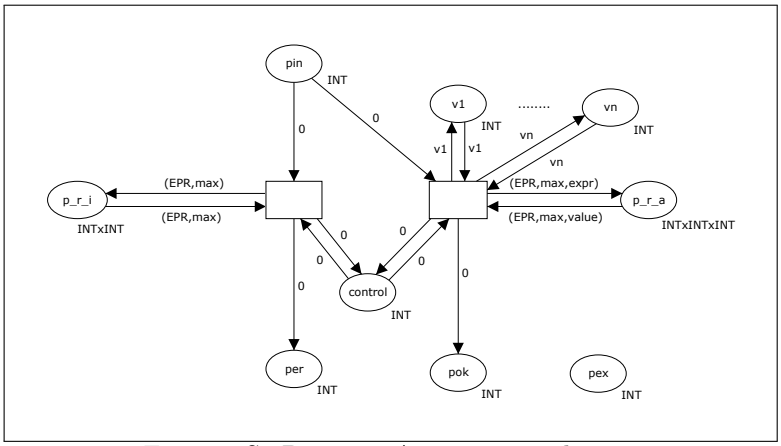


Fig. 11: SetProperty Activity Translation.

– SetTimeout (EPR,timeout): This activity is analogous to *SetProp* activity. In this case, the resource lifetime is updated with a new value. Fig. 11 shows this translation.

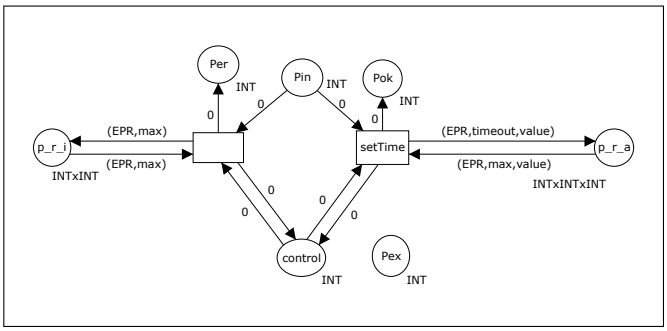


Fig. 12: SetTimeout Activity Translation.

5 Case study: Online auction service

The case study concerns a typical online auction process, which consists of three participants: the online auction system and two buyers, A_1 and A_2. A seller owes a good that wants to sell to the highest possible price. Therefore, he introduces the product in an auction system for a certain time. Then, buyers (or bidders) may place bids for the product and, when time runs out, the highest bid wins. In our case, we suppose the resource is the product for auction, the value of the resource property is the current price (only the auction system can modify it), the resource subscribers will be the buyers, their subscription conditions hold when the current product value is higher than their bid, and the resource lifetime will be the time in which the auction is active. Finally, when the lifetime has expired, the auction system sends a notification to the buyers with the result of the process (the identifier of the winner, v_w) and, after that, all the processes finish. Let us consider the choreography $C = (O_{sys}, O_1, O_2)$, where $O_i = (PL_i, Var_i, A_i, A_{f_i}, \mathcal{A}_{e_i})$, i=1,2, $Var_{sys} = \{v_w, v_1, v_2, v_{EPR}, at, t\}$, $Var_1 = \{at_1, v_1, v_{w_1}\}$, $Var_2 = \{at_2, v_2, v_{w_2}\}$, $A_{f_1} = exit$, and $A_{f_2} = exit$. Variable v_{EPR} serves to temporarily store the value of the resource property before being sent; v_1, v_2, v_w, v_{w_1}, v_{w_2} are variables used for the interaction among participants, and, finally, at, at_1 and at_2 are used to control the period of time in which the auction is active. In this example, we consider a period of 10 time units. Suppose $s_{0_{sys}}, s_{0_1}$ and s_{0_2} are the initial states of O_{sys}, O_1 and O_2, respectively, and all the variables are initially 0:

$A_{sys} = assign(10, at); createResource(EPR, 25, 11, A_{not});$
$\quad while(actualTime() <= at, A_{bid})$
$A_1 = wait(1, 1); subscribe(O_1, EPR, EPR >= 0, A_{cond_1});$
$\quad invoke(pl1, auction\_time_1, at1); \overline{reply}(pl1, auction\_time_1, at1);$
$\quad while(actualTime() <= at1, A_{bid_1}); receive(pl3, bid\_finish_1, vw_1, empty)$
$A_2 = wait(1, 1); subscribe(O_2, EPR, EPR >= 0, A_{cond_2});$
$\quad invoke(pl2, auction\_time_2, at2); \overline{reply}(pl2, auction\_time_2, at2);$
$\quad while(actualTime() <= at2, A_{bid_2}); receive(pl4, bid\_finish_2, vw_2, empty)$
$A_{not} = ((invoke(pl_3, bid\_finish_1, v_w)||invoke(pl_4, bid\_finish_2, v_w))$
$A_{bid} = getprop(EPR, v_{EPR}); pick($
$\quad (pl1, auction\_time1, t, reply(pl1, auction\_time1, at)),$
$\quad (pl2, auction\_time2, t, reply(pl2, auction\_time2, at)),$

$$
\begin{aligned}
&(pl_1, cmp, v_1, while(v_1 > v_{EPR}, assign(v1, v_{EPR});\\
&\quad setProp(EPR, v_{EPR}); assign(1, vw))),\\
&(pl_2, cmp, v_2, while(v_2 > v_{EPR}, assign(v2, v_{EPR});\\
&\quad setProp(EPR, v_{EPR}); assign(2, vw))), empty, 1)\\
A_{cond_1} &= getProp(EPR, v_{EPR}); invoke(pl_1, bid\_up_1, v_{EPR})\\
A_{cond_2} &= getProp(EPR, v_{EPR}); invoke(pl_2, bid\_up_2, v_{EPR})\\
A_{bid_1} &= receive(pl_1, bid\_up_1, v_1); assign(v_1 + random(), v_1);\\
&\quad invoke(pl_1, cmp, v_1); subscribe(O_1, EPR, EPR > v_1, A_{cond_1}); wait(1, 1)\\
A_{bid_2} &= receive(pl_2, bid\_up_2, v_2); assign(v_2 + random(), v_2);\\
&\quad invoke(pl_2, cmp, v_2); subscribe(O_2, EPR, EPR > v_2, A_{cond_2}); wait(1, 1)
\end{aligned}
$$

Regarding to the operations *auction_time*1 and *auction_time*2 inform buyers about the period of time in which the auction is active via variables at, $at1$ and $at2$, which are used in the while structures to control this period. The operations bid_up_1 and bid_up_2 are used to increase the current bid by adding a random amount to the corresponding variable v_i. The operation *cmp* is an auction system operation that receives as parameter a variable of the buyers, v_i. If the value of this variable is greater than the current value of v_{EPR}, then v_{EPR} is modified with this new value, that is, the new bid exceeds the current bid. After that, by means of the activity $setProp(EPR, v_{EPR})$, we can update the value of the resource property with the new bid. Finally, the operations bid_finish_1, bid_finish_2 update the value of v_w to inform the buyers who is the winner once the auction has expired.

In Fig. 13, we depict a simplified version of the PTCPN for the online auction system. The complete model can be accessed at the following web address: `http://www.dsi.uclm.es/retics/PetriNets2012/`. Here, we have constructed a hierarchical net relying on the notions of substitution transitions, sockets and ports offered by CPNTools [10]. We have then simulated and analysed the system, and we have concluded that the system finalizes successfully, that is, the output place of the system (p_ok) is reached in all the simulations. To check the consistency of the model, we have simulated the possibility of reaching an error place. For instance, if we delete the $wait(1, 1)$ sentences from activities A_1 and A_2, then it would imply that the buyers could access to the resource, that is the bid, even before the resource has been created. This possibility would trigger the expected error. Furthermore, we have analysed the data output from an experiment consisting of 5000 simulations. From the analysis of these data, we observe that the system is fair, from the point of view of the buyers, since they have equal right to place a bid. Indeed, the average of placed bids from each buyer is similar. Other information gathered from these data shows that buyers can evenly place higher bids than their competitors.

6 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

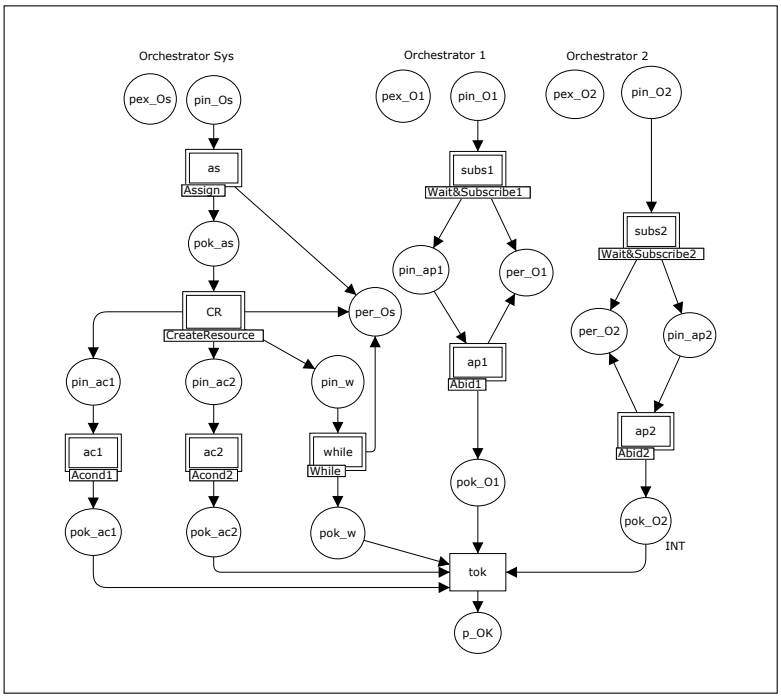


Fig. 13: A simplified PTCPN for the online auction system.

capability of storing their state. We have thus transformed *stateless* business processes into *stateful* business processes. To this end, we have defined a 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 TCPN 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[3] 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. As well, 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.

[3] The beta version can be accessed at: `http://www.dsi.uclm.es/retics/Petrinets2012/`

References

1. T. Andrews et. al. BPEL4WS – Business Process Execution Language for Web Services, Version 1.1, 2003. *http://www.ibm.com/developerworks/library/specification/ws-bpel/.*
2. T. Banks, *Web Services Resource Framework (WSRF) - Primer*, OASIS, 2006.
3. N. Busi, R. Gorrieri, C. Guidi, R. Lucchi and G. Zavattaro, *Choreography and Orchestration: A Synergic Approach for System Design*. In International Conference of Service Oriented Computing (ICSOC), Lecture Notes in Computer Science, vol. 3826, pp. 228-240, 2005.
4. K. Czajkowski, D. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke and W. Vambenepe, *The WS-Resource Framework Version 1.0*, http://www.globus.org/wsrf/specs/ws-wsrf.pdf, 2004.
5. N. Dragoni and M. Mazzara, *A formal Semantics for the WS-BPEL Recovery Framework - The pi-Calculus Way*. In International Workshop on Web Services and Formal Methods (WS-FM). Lecture Notes in Computer Science, vol. 6194, pp. 92-109, 2009.
6. M. Dumas, R. Heckel and N. Lohmann, *A Feature-Complete Petri Net Semantics for WS-BPEL 2.0*. In International Workshop on Web Services and Formal Methods (WS-FM). Lecture Notes in Computer Science, vol. 4937, pp. 77-91, 2008.
7. O. Ezenwoye, S.M. Sadjadi, A. Cary, and M. Robinson, *Orchestrating WSRF-based GridServices*. Technical Report FIU-SCIS-2007-04-01, 2007.
8. R. Farahbod, U. Glässer and M. Vajihollahi, *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.
9. I. Foster, J. Frey, S. Graham, S. Tuecke, K. Czajkowski, D. Ferguson, F. Leymann, M. Nally, T. Storey and S. Weerawaranna, *Modeling Stateful Resources with Web Services*, Globus Alliance, 2004.
10. K. Jensen and L. M. Kristensen, *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*, Springer, 2009.
11. F. Leyman. *Choreography for the Grid: towards fitting BPEL to the resource framework*. Journal of Concurrency and Computation : Practice & Experience, vol. 18, issue 10, pp. 1201-1217, 2006.
12. J.A. Mateo, V. Valero and G. Diaz. *An Operational Semantics of BPEL Orchestrations Integrating Web Services Resource Framework*. In International Workshop on Web Services and Formal Methods (WS-FM), 2011.
13. C. Ouyang, E. Verbeek, W.M.P. van der Aalst, S. Breutel, M. Dumas and A.H.M. ter Hofstede. *Formal semantics and analysis of control flow in WS-BPEL*. Science of Computing Programming, vol. 67, issue 2-3, pp. 162-198, 2007.
14. Z. Qiu, S. Wang, G. Pu and X. Zhao. *Semantics of BPEL4WS-Like Fault and Compensation Handling*. World Congress on Formal Methods (FM), pp. 350-365, 2005.
15. A. Slomiski. *On using BPEL extensibility to implement OGSI and WSRF Grid workflows*. Journal of Concurrency and Computation : Practice & Experience, vol. 18, pp. 1229-1241, 2006.
16. M. Wirsing and M. Holzl (Eds.), *Rigorous Software Engineering for Service-Oriented Systems*, Lecture Notes in Computer Science, Vol. 6582. Springer-Verlag, 2011.