A Coloured Petri Net Approach to Model and Analyse Stateful Workflows Based on WS-BPEL and WSRF/WSN[☆]

José Antonio Mateo, Valentín Valero, Hermenegilda Macià, Gregorio Díaz

Universidad de Castilla-La Mancha, Instituto de Investigación en Informática, 02071, Albacete, Spain.

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 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/WSN, and we provide a prioritised-timed coloured Petri net semantics for it. This semantics captures the main activities of WS-BPEL considering also other important aspects, both from WS-BPEL and WSRF/WSN, such as fault and event handlers, time-outs and the WS-Notification 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. In these systems, the service provider needs to ensure some levels of quality and privacy to the clients in a way that had never been considered. It is therefore necessary to develop new techniques to benefit from the advantages of recent approaches, such 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

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

Email addresses: joseantonio.mateo@uclm.es (José Antonio Mateo), valentin.valero@uclm.es (Valentín Valero), hermenegilda.macia@uclm.es (Hermenegilda Macià), gregorio.diaz@uclm.es (Gregorio Díaz).

the heterogeneity and volatility of resources. There are two complementary views to composite Web 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) [2, 5, 11]. 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. In addition, the WS-Notification (WSN) family of specifications defines the Web Services standard approach to send and receive notifications. Thus, WSN can be used in conjunction with WSRF to model a publish/subscribe architecture, thus allowing clients to subscribe to distributed WS-Resources and being notified in the case that this subscription condition holds or the state of that resource 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.

Thus, our aim is not to provide just another WS-BPEL semantics. In order to deal with the integration of WS-BPEL plus WSRF/WSN in a proper way, we have realised 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/WSN is not new; in the literature, there are a bundle of works defining this integration, although none of these works define a formal semantics in terms of Petri nets. In the next section we present an overview of these works. However, we have considered that the integration of WSRF/WSN, and specifically the time constraints that are inherent in this service language, would be captured in a better way in a specific

language, in which the main activity constructions of WS-BPEL are considered, as well as a model for event and fault handling.

2. Background and Related Work

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

2.1. Overview of WS-BPEL/WSRF/WSN

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. A port type defines the connection point to a web service defining a named set of abstract operations and the abstract messages involved.

In particular, we will consider a composite Web service consisting of a set of orchestrators, described by BPEL+WSRFN 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 event and fault activities, which are executed upon the occurrence of some 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.

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 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, which has been proposed to fully fit the OpenGrid Services Infrastructure (OGSI) into the Web services stack, thus allowing Web service middleware to be the base for Grid environments. The two main concepts behind the resource framework are WS-Resources and the implied resource pattern [13].

On the one hand, 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. On the other hand, the implied resource pattern specifies how to identify this resource. It is based on the WS-Addressing [19] specification and describes how a data structure (EPR) is used for the identification of a WS-Resource. Next, we describe the WSRF elements that are considered in the BPEL+WSRFN 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, ...). Resources are identified by their EPRs, 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 could be 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, *setLifeTime*.

- **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/WSN 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 [17] 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 [13] and [9]. The first justifies the use of WS-BPEL extensibility to allow the combination of different GRIDs, whereas Ezenwoye et al. [9] share their experience on WS-BPEL to integrate, create and manage WS-Resources that implement the factory/instance pattern.

On the other hand, Ouyang et al. [15] define the necessary elements for translating WS-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 WS-BPEL 2.0 is presented in [8] by Lohmann, 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 [7] 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 WS-BPEL recovery framework is [16]. Although this is more focused in the compensation handler, they describe the

Author	WS-BPEL	Rec.	WSRF	Formalism	Focus	Tool
Slomiski[17]	1.0	×	✓	–	Extensibility	×
Ezenwoye[9]	1.0	×	✓	–	Resource Management	×
Ouyang[15]	1.0	Part	×	Petri nets	BPEL analysis	✓
Lohmann[8]	2.0	✓	×	Petri nets	BPEL analysis	✓
Dragoni[7]	2.0	✓	×	π-calculus	FCT	×
Qiu[16]	1.0	Part	×	Proc. Algebra	FC	×
Farahbod[10]	1.0	Part	×	Finite State Machines	Analysis	×
Busi[3]	1.0	Part	×	Proc. Algebra	Conformance Chor. vs Orch.	×
Our work	2.0	Part	✓	Petri nets	Resource Management	✓

Table 1: Bibliography comparison.

corresponding rules that manage a Web service composition. Our work is therefore more general as we define rules for nearly all possible activities instead of focusing only on the recovery framework. In addition, we also consider time constraints. Finally, we would like to highlight the works of Farahbod et al. [10] and Busi et al. [3]. In the first one, the authors extract an abstract operational semantics for WS-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 WS-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 [18]. Here, an extensive work is presented from different international research groups aimed by the common goal of providing a rigorous software engineering viewpoint for service-oriented system using as a cornerstone the formal specification of Web Services and WS-BPEL in particular.

Table 1 shows a brief comparison among all the works where, the columns show the WS-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.

3. 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 3.1, we recall the formal definition of coloured Petri nets presented in [12], whereas, in Definition 3.2, we define the precise model used in this work.

Definition 3.1. *A timed non-hierarchical Coloured Petri Net is a nine-tuple CPN_T = (P, T, A, Σ ,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.
- *$I : P \to EXPR_\emptyset$ 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 consider a class 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 3.2. *We define a prioritised-timed coloured Petri net (PTCPN) as a tuple (CPN_T, λ, D, π), where*[1]*:*

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

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

- *CPN_T is a CPN according to Definition 3.1, 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 string or nothing (by default), if $t \in T$.*
- *$D : T \longrightarrow \mathbb{N} \times \mathbb{N}$ 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/WSN standards, we can distinguish between two types 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 [12]. The interested reader is referred to [12] for further information.

4. PTCPN Semantics for BPEL+WSRFN

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

A system 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 all 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 (Vars $\subseteq$ Var), 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+WSRFN follow the syntax defined by the following BNF expression (see Table 2 for the equivalence with the XML syntax of BPEL and WSRF/WSN):

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

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 a previous work [14].

Now, let us introduce the translation for the different elements of a BPEL+WSRFN description:

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 just inspecting the WS-BPEL+WSRFN specification.

In the following subsections we complete the translation of a BPEL+WSRFN specification into the corresponding PTCPN. First, we show the translation at the activity level, and, after, at the orchestration level. Let us note here that all the places in the figures that have not attached the corresponding colour set are considered TUNIT, whereas TINT represents the timed version of the colour

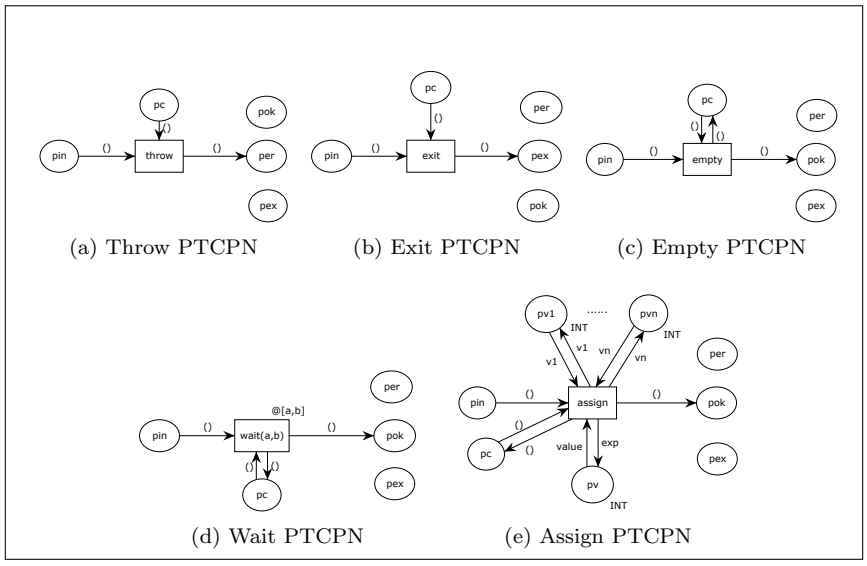
WS-BPEL/WSRF/WSN Syntax	Model
<process ...> <partnerLinks> ... </partnerLinks>? <Variables> ... </Variables>? <faultHandlers> ... </faultHandlers>? <eventHandlers> ... </eventHandlers>? (activities)* </process>	(PL,Var,A,A_f,$\mathcal{A}_e$)
<throw/>/any fault	throw
<receive partnerLink="pl" operation="op" 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>	$\frac{A_1 \; ; \; A_2}{A_1 \parallel A_2}$
<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>	setLifeTime(EPR,timeout)
<wsnt:Subscribe> <wsnt:ProducerReference>EPR</wsnt: ProducerReference> <wsnt:Precondition>cond′</Precondition> </wsnt:Subscribe>	subscribe(EPR,cond′,A_{e_i})
<wsnt:Notify> <wsnt:NotificationMessage> <wsnt:ProducerReference>EPR</wsnt:ProducerReference> <wsnt:Message> ... </wsnt:Message> </wsnt:NotificationMessage> </wsnt:Notify>	Executes the associated event handler activity A_{e_i}

Table 2: Conversion table

set INT.

4.1. 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 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.

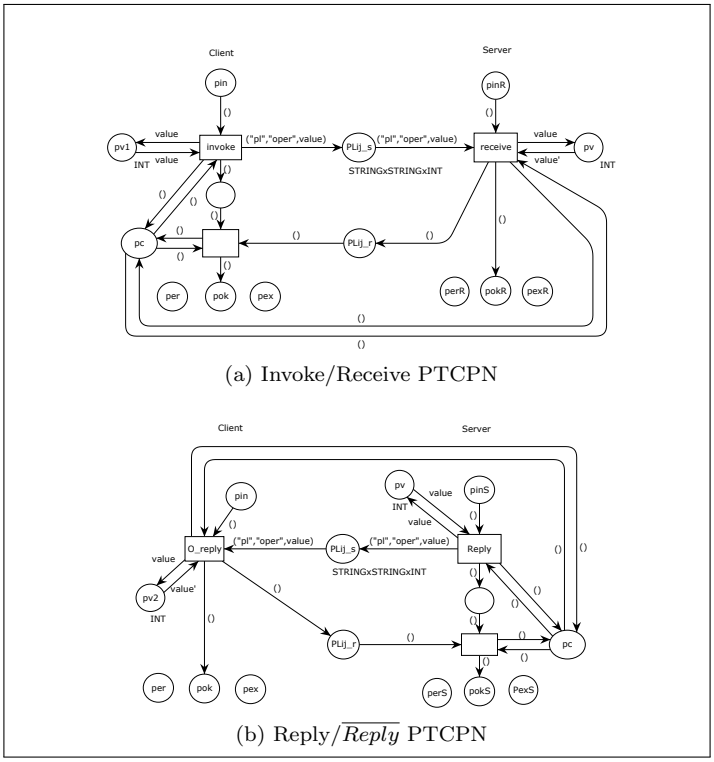


(a) Throw PTCPN (b) Exit PTCPN (c) Empty PTCPN

(d) Wait PTCPN (e) Assign PTCPN

Figure 1: 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 server. We have therefore introduced this last activity in our semantics to deal with the synchronous or asynchronous nature of the invoke activity (one-way or request-response operation, respectively), so the $\overline{reply}$ activity is optional in the syntax depicted in Table 2.



(a) Invoke/Receive PTCPN

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

Figure 2: Invoke/Receive Activities Translation

Fig. 2 shows the translation for both the invoke/receive and the reply/$\overline{reply}$ pairs of activities. Part 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.

4.2. 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. 3), 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.

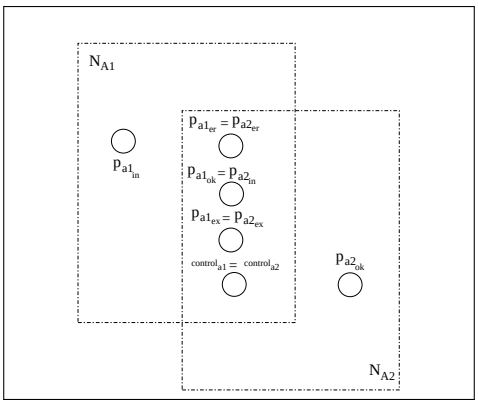


Figure 3: Sequence Translation

- *Parallel*: The translation for a parallel activity is depicted in Fig. 4, 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.

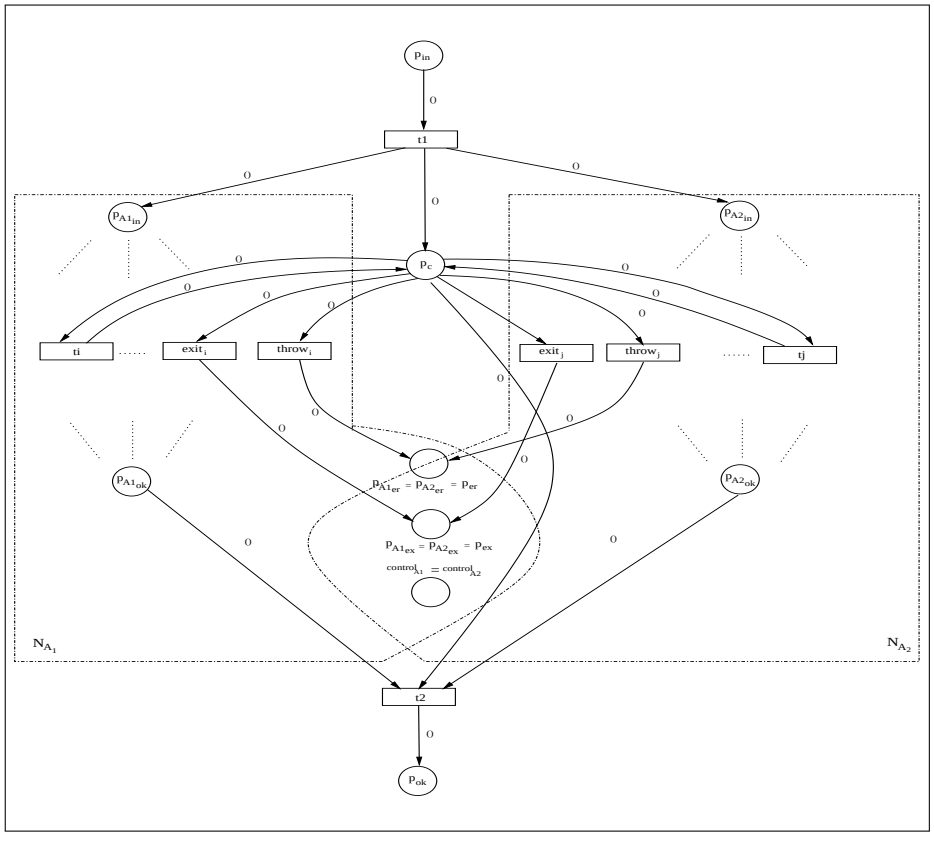


Figure 4: Parallel Activity Translation.

- *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. 5 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 *p_a* is timed. To illustrate how this construction works, we define the following example.

Example 4.1. *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 send the requested information to the customers. The seller has established a timeout of 24 hours to receive requests. Let the orchestrations* $O_{c1} = (pl_1, id_1, id_3, A_{c1}, empty)$, $O_{c2} = (pl_2, id_2, id_4, A_{c2}, empty)$ *and* $O_s = (pl_1, pl_2, id_{s1}, id_{s2}, inf_{s1}, inf_{s2}, A_s, empty)$ *, the BPEL+ WSRFN code for the primary activity of both participants is:*

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

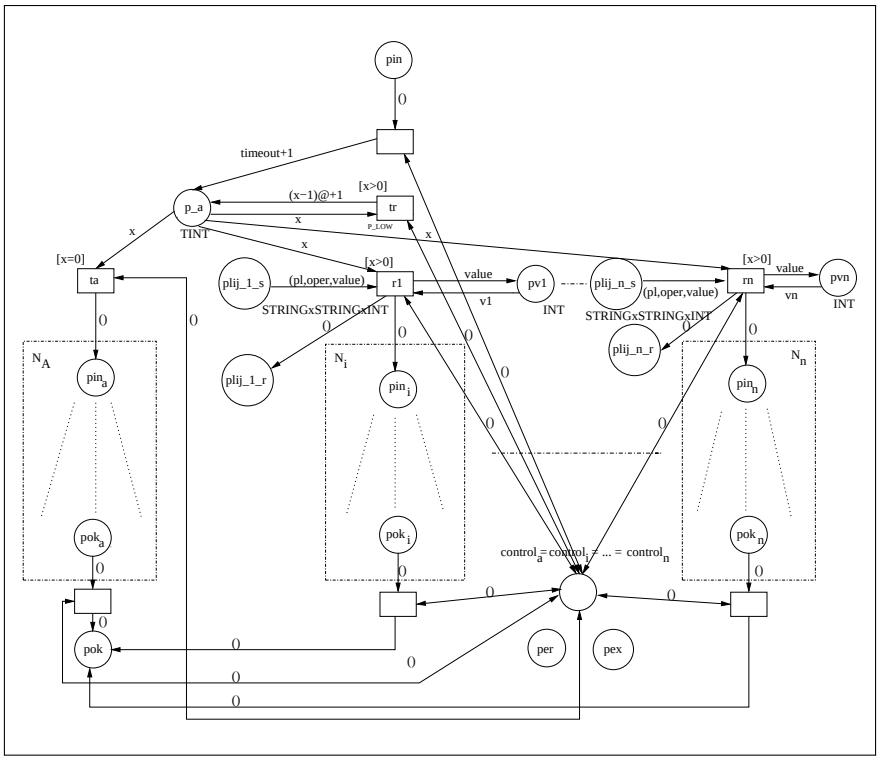


Figure 5: Pick Activity Translation.

Looking at Fig. 5, 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 $timeout + 1$. Two situations can then occur. One of the buyers may perform its *i*nvoke activity before the 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 is increased by firing the transition t_r. This transition is enabled until the timeout is reached, that is, the value of x is equal to 0. In that case, the PTCPN corresponding to activity A is performed. We have used variable x as a countdown timer due to a restriction of CPNTools, which does 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. Fig. 6 shows this translation.

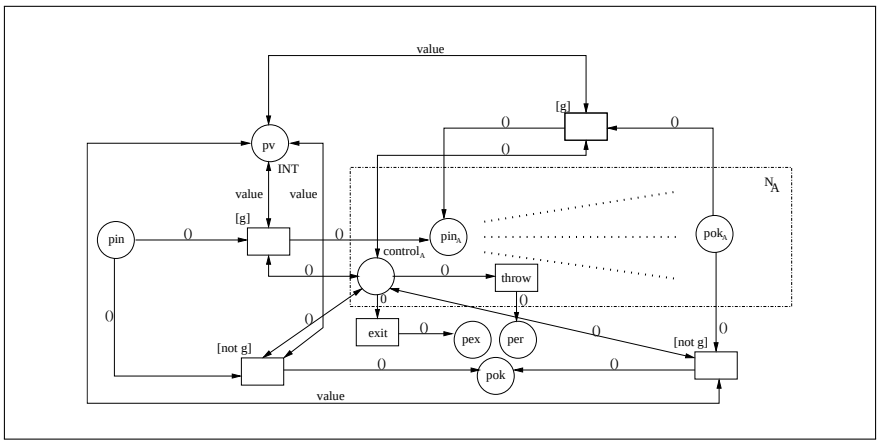


Figure 6: While Activity Translation.

4.3. WSRF/WSN compliant

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

- *CreateResource(EPR,val,timeout,A)*: *EPR* is the resource identifier, for which we have two complementary places in Fig. 7, 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. 7 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

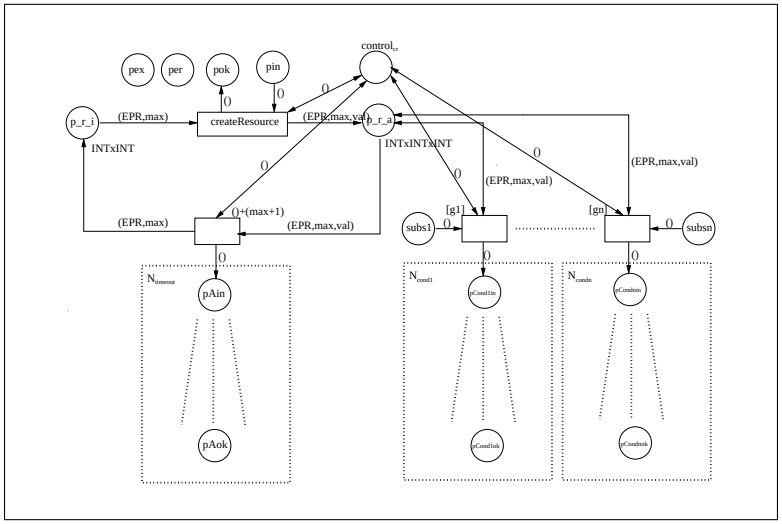


Figure 7: CreateResource Activity Translation.

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 BPEL+WSRFN 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. 8 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.

- *GetProp(EPR,v) and SetProp(EPR,expr): These are easily translated, as shown in Figs. 9 and 10, where the resource value is obtained and assigned to variable v (GetProp), or a new value is assigned to the resource (SetProp).*

- *SetLifeTime(EPR,timeout): This activity is analogous to* SetProp *activity. In this case, the resource lifetime is updated with a new value. Fig. 10 shows this translation.*

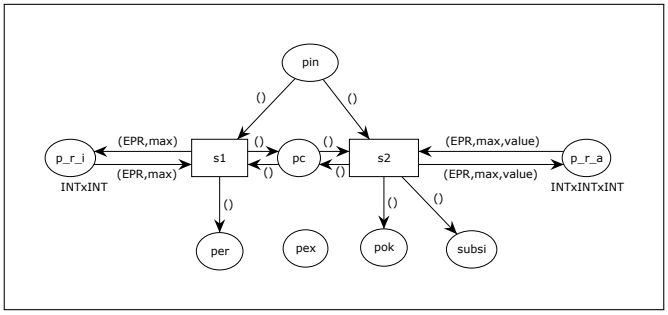


Figure 8: Subscribe Activity Translation.

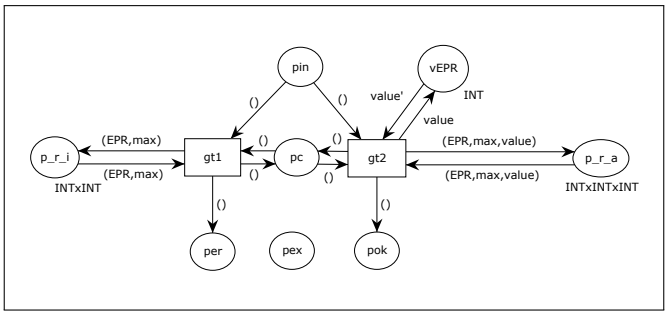


Figure 9: GetProperty Activity Translation.

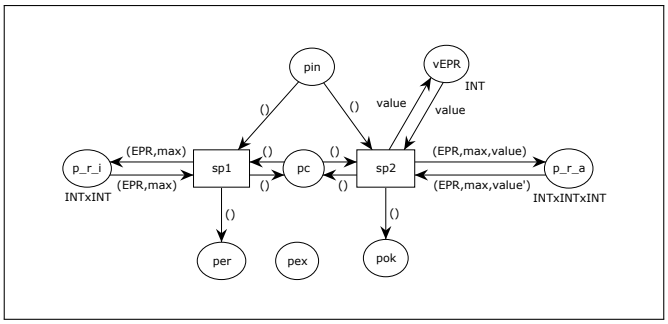


Figure 10: SetProperty Activity Translation.

4.4. 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.

Let us call N_A, N_f and N_{e_i} the PTCPNs that are obtained by applying the

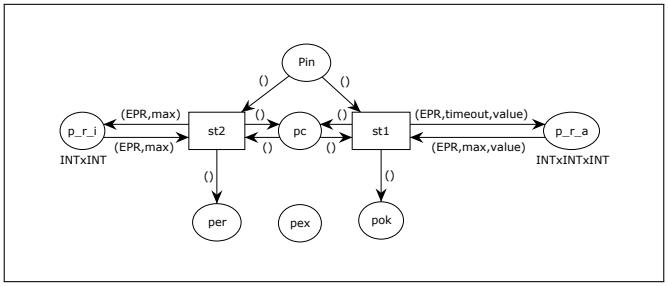


Figure 11: SetLifeTime Activity Translation.

translation to each one of these activities A, A_f, A_{e_i}, with $i \in \{1, m\}$:

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

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. 12. 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.

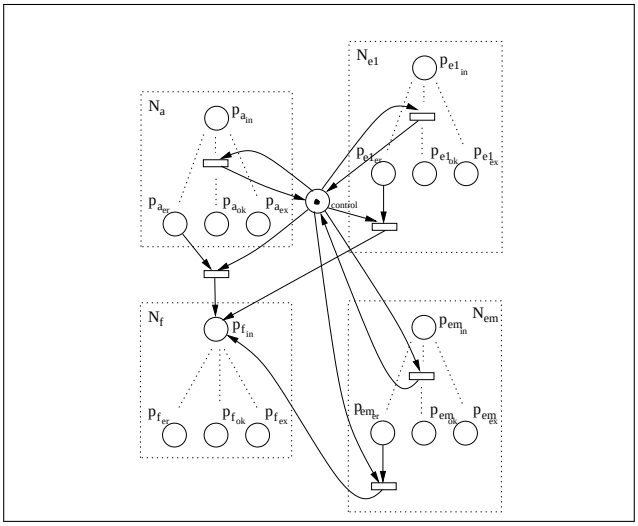


Figure 12: Orchestration Translation

5. 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, $i = 1, \ldots, n$. 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 $O_k = (PL_k, Var_k, A_k, A_{f_k}, \mathcal{A}_{e_k})$, 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. We assume that all variable values are initially 0:

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

Here, the function *bid* is used to increase/decrease the stocks value simulating the fluctuation of the stocks price.

In Figs. 13 and 14, the PTCPNs for one buyer and for the system are depicted. These figures have been obtained automatically by using our tool [6]. A beta version of our tool is available at `http://www.dsi.uclm.es/retics/bpelrf/`.

5.1. Analysis

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 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 3 shows the results

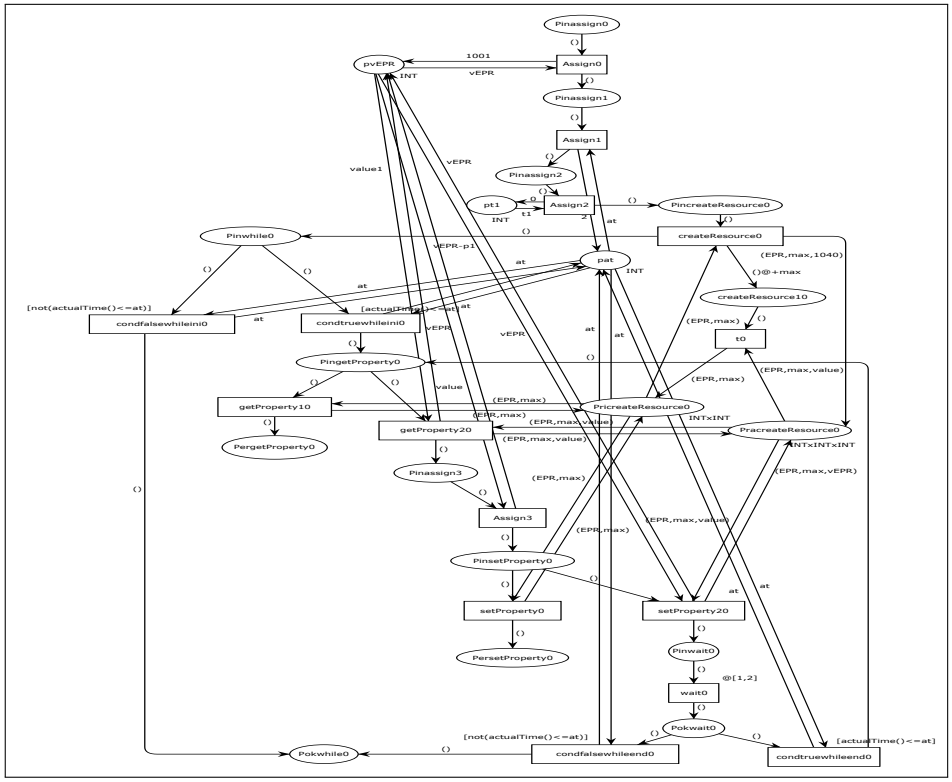


Figure 13: PTCPN of the online stock market.

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

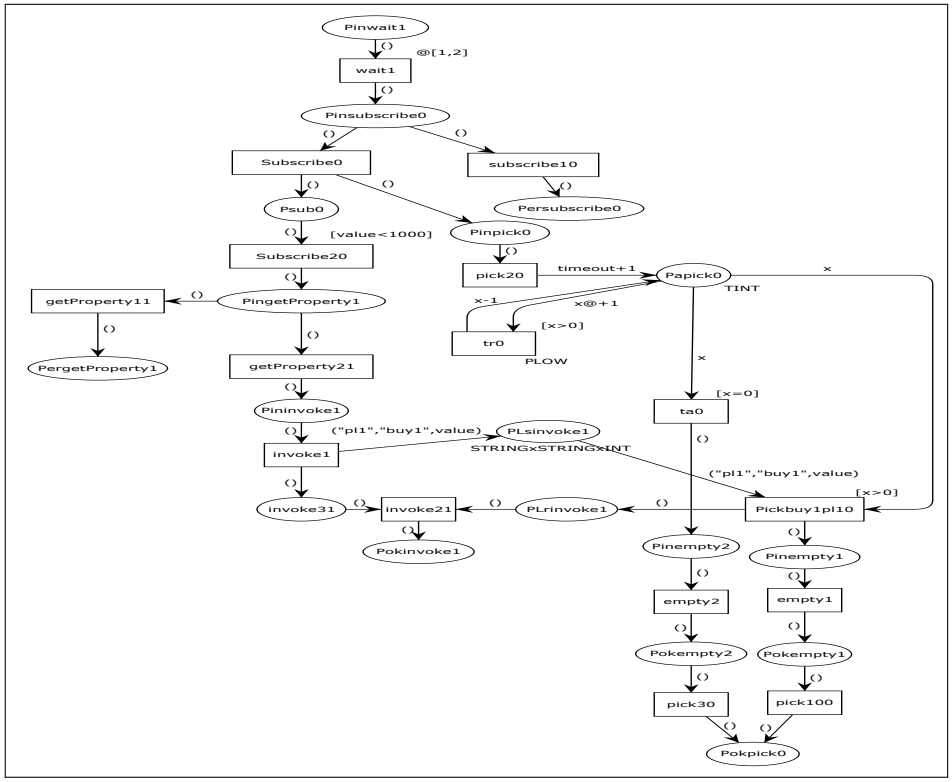


Figure 14: PTCPN of one buyer.

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. This final place, *Pokfinal0*, is marked by a transition when all the participants have finished their workflow. 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 following predicate to check that the list of dead marking contains the marking of the *Pokfinal0* place:

PredAllNodes DesiredTerminal=ListDeadMarkings().

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

Table 3: State space analysis results

In Fig. 15, we show the results offered by CPNTools to our queries for the case of **three** investors. Here, it can be appreciated that all dead markings hold the predicate *DesiredTerminal*, and, therefore, when the system reaches a dead marking is because it has terminated correctly, which demonstrates the absence of deadlocks in our case study.

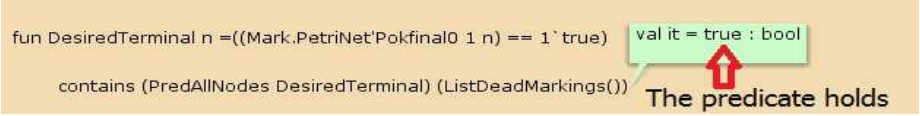


Figure 15: Result of the queries in CPNTools.

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 capability of storing their state. We have thus transformed *stateless* business processes into *stateful* business processes. To this end, we have defined a Prioritized-Timed Coloured Petri net model and presented its corresponding semantics to represent the constructions of WS-BPEL and the standards selected for the definition of resources and notifications, namely WSRF/WSN. 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. As future work, we plan to study some interesting formal properties such as safeness, soundness and so on. Furthermore, it will be also of interest to complete the semantics of WS-BPEL and WSRF/WSN by adding new features such as compensation and termination handling, and brokered notification in WSN. As regards WSRF, it seems that it could be also interesting to consider multiple properties from different data domains, in order to have a more realistic and usable model.

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] CPNTools website, `http://cpntools.org/`.

[5] 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.

[6] M. Díaz, V. Valero, H. Macia, J. A. Mateo and G. Díaz, *BPEL-RF Tool: An Automatic Translation from WS-BPEL/WSRF Specifications to Petri Nets*. In Seventh International Conference on Software Engineering Advances (ICSEA), pp. 325-330, 2012.

[7] 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.

[8] 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.

[9] O. Ezenwoye, S.M. Sadjadi, A. Cary, and M. Robinson, *Orchestrating WSRF-based GridServices*. Technical Report FIU-SCIS-2007-04-01, 2007.

[10] 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 (WS-MDEIS), pp. 122-133, 2005.

[11] 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.

[12] K. Jensen and L. M. Kristensen, *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*, Springer, 2009.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] M. Wirsing and M. Holzl (Eds.), *Rigorous Software Engineering for Service-Oriented Systems*, Lecture Notes in Computer Science, Vol. 6582. Springer-Verlag, 2011.

[19] Web Services Addressing (WS-Addressing), *http://www.w3.org/Submission/ws-addressing/*.