

# Formalizing WS-BPEL in Binding Bigraphs<sup>\*</sup>

Mikkel Bundgaard, Arne John Glenstrup, Thomas Hildebrandt, and Espen Højsgaard

IT University of Copenhagen, Denmark  
{mikkelbu, panic, hilde, espen}@itu.dk

**Abstract.** We provide a bigraphical formalization of a non-trivial subset of WS-BPEL, including concurrent flows, shared variables, nested scopes, dynamic manipulation of links between processes, inter-process communication, invocation and abnormal termination. Compared to formalizations based on process calculi and Petri Net a particular feature of the bigraphical formalization is that processes can be represented as graphs very similar to the XML syntax of WS-BPEL, thereby reducing the usual gap between the language specification and the formalization. The prize to pay is the use of a set of *language specific* reaction rules, resulting in a behavioral theory for which no tools for automatic verification yet exists. However, we indicate for future work how the uniformity of bigraphical reactive systems opens up for a study of formal relationships *within* the framework of bigraphical reactive systems, between very concrete semantics as the one presented in the present paper and more abstract semantics, in particular the many existing formalizations based on variants of the  $\pi$ -calculus and Petri Net.

## 1 Introduction

In the present paper we provide a formalization of a non-trivial subset of WS-BPEL [17] based on Milner’s bigraphical reactive systems [9] and implemented in the BPL-tool [21] developed as part of the Bigraphical Programming Languages (BPL) project at ITU. The main reason why we think it is interesting to apply bigraphs to formalize WS-BPEL is that it is a *compositional* and *extensible* model with a uniform behavioral theory. By compositional we refer to the fact that the model of bigraphs comes with an algebraic theory allowing complex bigraphs to be composed from basic bigraphs, which facilitates inductive definitions of maps between different bigraphical reactive systems and even automatic derivation of labelled bisimulation congruences [9]. By extensible we refer to the fact that an instance of a bigraphical reactive system is defined by its signature (defining the syntax) and its reaction rules (defining the semantics), which can be chosen to fit a particular language and its semantics. Since a bigraph consists of a place graph and a link graph which correspond closely to respectively the

---

<sup>\*</sup> This work was funded in part by the Danish Research Agency (grant no.: 2106-07-0019, no.: 274-06-0415 and no.: 2059-03-0031) and the IT University of Copenhagen (the TrustCare, CosmoBiz and BPL projects).

nested element structure and sharing of attribute values in the XML data model, we are able to represent the state of WS-BPEL-processes in our formalization as graphs very similar to the XML syntax of WS-BPEL. The extensibility allows the bigraphical reactive system to be adapted by changing the signature and reaction rules according to e.g. changes in, or extensions to, the language specification. Moreover, it has already been shown by Milner and co-authors that bigraphs allow for very direct formalizations of the  $\pi$ -calculus [9] and (1-safe) Petri Nets [12]. This uniformity of bigraphical reactive systems opens up for a study of formal relationships *within* the framework of bigraphical reactive systems, between very concrete semantics as the one presented in the present paper and more abstract semantics. We aim to explore this possibility in future work by providing (possibly a chain of) formal maps between our concrete semantics and more abstract semantics, e.g. based on different variants of the  $\pi$ -calculus and Petri Net. Besides the obvious use for relating different formalizations, this could bridge the usual gap between the language specification and the abstract semantics, ultimately supported by tools such as the BPL-tool used in the present paper. Also, the existing verification tools for the  $\pi$ -calculus and Petri Net could make up for the lack of automatic verification tools for general bigraphical reactive systems and provide grounds for effective implementations based on the formal semantics.

The close correspondence between bigraphs and XML was already explored in our previous work on formalizing WS-BPEL as bigraphs [7, 6], on which the present work builds. However, the formalization in [7, 6] was obtained at the cost of introducing so-called higher-order reaction rules, for which the relationship to the existing notions of bigraphs and theory of behavioral congruences remain to be developed. In addition to covering a larger subset of WS-BPEL, the present formalization stays within the standard format for binding bigraphs described in [9]. Thus, the general theory, techniques and tools developed for binding bigraphs remain applicable to our formalization. In particular, we were able to use the BPL Tool to implement the bigraphical reactive system (as standard data types within Standard ML) constituting our WS-BPEL formalization and use the web interface of the tool to visualize and interactively simulate the execution of processes. As described in a companion paper [2], we have employed the extensibility of bigraphical reactive systems to extend the language and formalization to allow for *mobile embedded sub-processes*. The result is a higher order business process language allowing to describe processes that manipulate, exchange and manage processes as nested sub-processes.

WS-BPEL has been the target for several formalizations (see [22, 18] for overviews) accompanying the official informal specification. Many of the prior formalizations have been based on versions of Petri Nets [20, 8, 13, 14, 18] following the tradition of formal workflow models. Other authors have been promoting the use of process calculi, notably (variants of) the  $\pi$ -calculus [19, 15] or as in [11] using domain specific calculi abstracting key features from WS-BPEL which are then subsequently compiled into the  $\pi$ -calculus. This diversion between Petri Net and process calculi can partly be explained by the fact that WS-

BPEL is a convergence and development of two radically different approaches to web service orchestration proposed back in 2001: The IBM Web Services Flow Language (WSFL) and the Microsoft XLANG specification. While WSFL was based on flow graphs which are characteristic to the Petri Net model and most workflow languages, XLANG was based on the notion of message exchange behavior which is characteristic to the  $\pi$ -calculus.

We do by no means claim our formalization to be more complete nor superior to all of the existing formalizations. Indeed, we focus on the XLANG subset of WS-BPEL, in particular the control flow, scope structure, message passing and dynamic manipulation of partner links (akin to name-passing in the  $\pi$ -calculus). Moreover, compared to the formalizations based on the  $\pi$ -calculus or Petri Net, there are so far no tool support for automatic verification of bigraphical reactive systems. However, since bigraphical reactive systems have been shown to faithfully represent both the  $\pi$ -calculus and Petri Nets, we believe the model is a promising candidate for a unified study of the different formalizations as suggested above, and for providing at the same time a faithful representation of both the WSFL and the XLANG features of WS-BPEL. Already for the present subset, we crucially exploit the nesting structure of bigraphs to give a very succinct semantics of "abnormal" termination caused by the WS-BPEL `exit` activity, which is not as easy to formalize in Petri Net nor the  $\pi$ -calculus. Indeed the formalization in [15] employs a non-trivial extension of the  $\pi$ -calculus with *transactions* to deal with abnormal termination.

The paper is structured as follows. In Sec. 2 we present the subset of WS-BPEL we consider, briefly present the model of binding bigraphs using the term language for bigraphs used in the BPL Tool and sketch the representation of the  $\pi$ -calculus and 1-safe Petri Nets in binding bigraphs. In Sec. 3 we then present our formal bigraphical semantics of a subset of WS-BPEL implemented using the BPL Tool. We conclude and propose directions for future work in Sec. 4.

## 2 WS-BPEL and Binding Bigraphs

In this section we present the subset of WS-BPEL considered in the paper, briefly review the binding bigraphs of Milner and Jensen [9], and introduce the syntactical representation of binding bigraphs as implemented in the BPL Tool. For a complete introduction to bigraphs we refer to [9].

**WS-BPEL.** We consider a subset of the WS-BPEL syntax given by the grammar in Tab. 1. For brevity we do not use XML notation or an XML schema and omit attributes in the grammar. We use `?` and `*` to indicate that an element can appear at most once and any number of times respectively. We also assume that sequence elements always contain exactly two actions. Note that in regard to data flow we only consider the constant values given by XPath expressions `true()` and `false()` and references to variables, assuming that  $x$  range over strings. We let **BPEL** refer to the set of terms defined by the grammar.

**Binding Bigraphs and the BPL Tool Term Language.** A bigraph is a pair of graphs: a *place graph* and a *link graph*. The place graph is an  $n$ -tuple of finite,

<i>proc</i>	::= Process( <i>scopecontent</i> )
<i>scopecontent</i>	::= <i>partnerlinks</i>   <i>vars</i>   <i>act</i>
<i>partnerlinks</i>	::= PartnerLinks(PartnerLink*)
<i>vars</i>	::= Variables(Variable*( <i>value?</i> ))
<i>act</i>	::= <i>scope</i>   <i>seq</i>   <i>flow</i>   <i>while</i>   <i>if</i>   <i>assign</i>   Invoke   Receive   Reply   Exit
<i>scope</i>	::= Scope( <i>scopecontent</i> )
<i>seq</i>	::= Sequence( <i>act act</i> )
<i>flow</i>	::= Flow( <i>act*</i> )
<i>while</i>	::= While(Condition( <i>expr</i> )   <i>act?</i> )
<i>if</i>	::= If(Condition( <i>expr</i> )   Then( <i>act?</i> )   Else( <i>act?</i> ))
<i>assign</i>	::= Assign(Copy(From   To))
<i>value</i>	::= <i>true</i> ()   <i>false</i> ()
<i>expr</i>	::= <i>value</i>   $\$x$

**Table 1.** Grammar for WS-BPEL processes.

unordered trees. Except for roots, every node is labelled by a *control*, which has a finite ordered set of (free) ports, and (in binding bigraphs) also possibly a finite ordered set of *binding* ports. The link graph is a hypergraph connecting every free port of the nodes in the place graph to either a closed link, a binding port, or a name in a finite set  $X$  of names. Jointly with a collection of pairwise disjoint sets  $X_i \subseteq X$  of local names, one for each root in the bigraph, the set  $X$  defines the (outer) *interface* of the link graph. The so-called *scope condition* enforces that any binding port and any name in a set  $X_i$  is only connected to ports nested strictly inside the node of the binding port and root  $i$  respectively.

In general bigraphs are (multi-hole) *contexts* that can be composed: The place graph has a finite ordered set of *holes* (referred to as *sites* in the usual bigraph terminology), each associated as a child of a node. The link graph has a set of *local names*  $Y_i$  for each hole. As for the outer interface, the sets  $Y_i$  are pairwise disjoint and contained in a finite set of names  $Y$  which jointly with the sets  $Y_i$  forms the *inner* interface. As the free ports, the names in  $Y$  are connected to either a closed link, a binding port or a name in the outer interface.

Outer (resp. inner) interfaces of binding bigraphs are thus triples  $\langle n, \vec{X}, X \rangle$ , where the *width*  $n$  is a finite ordinal representing the number of roots (resp. holes),  $X$  is a finite set of names, and  $\vec{X}$  is an  $n$ -tuple of pairwise disjoint subsets of  $X$  which declares some of the names in  $X$  as *local* to specific roots (resp. holes). If  $x \notin \vec{X}$  then  $x$  is said to be *global*, else it is *local*; if an interface  $I$  has no global names  $x$ , it is a *local* interface. We write  $G : I \rightarrow I'$  for the bigraph  $G$  with inner interface  $I$  and outer interface  $I'$ . The composition  $H \circ G : I \rightarrow J$  of bigraphs  $G : I \rightarrow I'$  and  $H : I' \rightarrow J$  is obtained by making the children of the  $i$ th root of  $G$  children of the node to which the  $i$ th hole of  $H$  is associated, discarding the roots of  $G$  and holes of  $H$ , and by coalescing links as prescribed by the correspondence of  $H$ 's inner and  $G$ 's outer names.

Binding bigraphs are often visualized graphically, but can also intuitively be thought of as an ordered tuple of terms (or contexts if the bigraph contains holes) of a process calculus up to structural congruence. This intuition is captured by

$$\begin{aligned}
P &::= P \circ P \mid P \parallel P \mid P \mid P \mid C \\
C &::= c \mid c[N^?] \mid c[N^?][NS^?] \mid -//[N^?] \mid n//[N^?] \mid [N^?] \mid \langle - \rangle \\
N^? &::= \epsilon \mid N \quad N ::= n \mid n, N \quad NS^? ::= \epsilon \mid NS \quad NS ::= [N^?] \mid [N^?], NS
\end{aligned}$$

**Table 2.** Grammar for BPL Binding Bigraphs.

the axiomatisation of binding bigraphs given in [4] (extending the axiomatisation of bigraphs given in [16]). The axiomatisation is exploited in the BPL Tool to provide a term language allowing compact and compositional textual descriptions of binding bigraphs and their reaction rules. It is also used in the underlying formalization and implementation of matching used for the execution of reaction rules in the tool [5]. The subset of the term language employed in this paper is defined by the grammar in Tab. 2, where  $n$  ranges over strings representing names and  $c$  over strings representing controls<sup>1</sup>. The double bars  $\parallel$  and the single bar  $\mid$  allow for *horizontal* composition of bigraphs.  $P \parallel P'$  places the roots of the two bigraphs  $P$  and  $P'$  next to each other (and merges non-local names and require local names to be disjoint in the interfaces).  $P \mid P'$  merges the roots of the two bigraphs, i.e. making the top level nodes of the two bigraphs siblings. The symbol  $\circ$  denotes *vertical* composition of a matching inner and outer interface of two bigraphs as defined above (implicitly extending with links mapping additional names from the inner interface to the outer interface if necessary).

The terms  $c$ ,  $c[N^?]$  and  $c[N^?][NS^?]$  denote so-called *ions*, which correspond to the prefixes in process calculi, i.e. they are bigraphs consisting of a single root with a single node as child, having control  $c$  and a single hole inside (except if the control is declared to be atomic as described below). For instance, an ion with name  $c$  and  $i$  free ports and  $j$  binding ports is written  $c[n_1, \dots, n_i][NS_1, \dots, NS_j]$  where  $n_i$  is the name of the  $i$ th free port and  $NS_k$  is the set of names bound to the  $k$ 'th binding port. The terms  $-//[N^?]$  and  $n//[N^?]$  denote a bigraph with an empty place graph (i.e. no roots) and a link graph mapping the names in the list  $N^?$  to respectively each their closed link and to the name  $n$ . The term  $[\ ]$  denotes a hole and the term  $[n_1, \dots, n_k]$  denotes a hole with local names  $n_1, \dots, n_k$ . Finally, the term  $\langle - \rangle$  denotes a bigraph just consisting of a single root with no children.

As an example, we may define binding bigraphs in the BPL-tool (representing a redex and a simple process in the asynchronous  $\pi$ -calculus described below) as

```

val PiRedex = Sum o (Send[x,z] | []) | Sum o (Get[x][[y]] o [y] | [])
val PiProcess = Sum o Send[x,z] | Sum o Get[x][[y]] o Sum o Send[y,v]

```

The holes in a bigraph term are ordered from left to right indexed from 0, i.e. the hole next to the `Send[x,z]` control in the `PiRedex` has index 0, the hole within

<sup>1</sup> The tool requires a few additional back quotes since the terms are written in SML.

the  $\text{Get}[x][y]$  control has index 1 and a local name  $y$ , and the last hole has index 2. Thus, the inner and outer interfaces of  $\text{PiRedex}$  are  $\langle 3, \langle \emptyset, \{y\}, \emptyset \rangle, \{y\} \rangle$  and  $\langle 1, \langle \emptyset \rangle, \{x, y\} \rangle$  respectively.

**Bigraphical Reactive Systems.** A binding bigraphical reactive system is defined with respect to a *signature*, which declares the set of possible controls labelling nodes of the bigraph and for each control  $K$  the number of binding and free ports of nodes in the bigraph labelled with  $K$ . The signature also declares each control as either atomic, active or passive. Only nodes with non-atomic controls can have children, and reactions (as defined below) can only occur in sub-bigraphs nested solely within active controls.

As an example, consider the asynchronous  $\pi$ -calculus (with summations) given by the following grammar

$$P ::= \sum_{i \in n} T_i \mid P_1 \mid P_1 \mid \nu x P \mid 0 \qquad T ::= \bar{x}(y) \mid x(y).P$$

As shown in [9] the asynchronous  $\pi$ -calculus can be represented as a bigraphical reactive system using the passive controls  $\text{Sum}$  (with no ports),  $\text{Send}$  (atomic and with two free ports) and  $\text{Get}$  (with one free and one binding port) by the following compositionally defined translation (using the BPL-tool term language)

$$\begin{aligned} \llbracket \sum_{i \in n} T_i \rrbracket &= \text{Sum} \circ (\llbracket T_1 \rrbracket \mid \dots \mid \llbracket T_n \rrbracket) \\ \llbracket P_1 \mid P_2 \rrbracket &= \llbracket P_1 \rrbracket \mid \llbracket P_2 \rrbracket \\ \llbracket \nu x P \rrbracket &= -//[x] \circ \llbracket P \rrbracket \\ \llbracket \bar{x}(z) \rrbracket &= \text{Send}[x, z] \\ \llbracket x(y).P \rrbracket &= \text{Get}[x][y] \circ \llbracket P \rrbracket \\ \llbracket 0 \rrbracket &= \langle - \rangle \end{aligned}$$

Note how the use of single and double square brackets of the control  $\text{Get}$  specify that  $x$  is a free port and  $y$  is a binding port, whereas  $\text{Send}[x, z]$  has two normal ports.

The dynamics of bigraphical reactive systems is defined in terms of a reaction relation generated from a set of reaction rules  $\mathcal{R}$ . Such rules are generally parametric, and may discard and also duplicate their parameters.

A rule, written in the BPL-tool as "**rule name**"  $::: R \text{ --}\bar{\varrho}\text{--} \mid \rangle R'$ , consists of two bigraphs: the *redex*  $R : I \rightarrow J$  and the *reactum*  $R' : I' \rightarrow J$ , where both  $I$  and  $I'$  are local interfaces, and a parameter mapping  $\bar{\varrho}$  which indicates for each hole in the reactum from which hole in the redex the parameter is copied.

For the asynchronous  $\pi$ -calculus, the reaction,  $\bar{x}(z) + T \mid x(y).P + T' \rightarrow \{z/y\}P$ , is modelled by the bigraphical reaction

$$\text{"pisync"} \quad ::: \text{PiRedex} \text{ --}[0 \mid \rightarrow 1] \text{ --} \mid \rangle \quad z//[y] \circ [y] \mid x//[] \quad .$$

Note that the contents of the hole 0 and hole 2 in the redex are discarded and the local name  $y$  is fused with the free name  $z$ . The link  $x//[]$  is only there to ensure that the name  $x$  is in the outer interface of the reactum (it must be identical to the outerface of the redex), it is not connected to anything.

A rule matches an agent  $a$  if  $a = C \circ R \circ d$  for some active context  $C$  (i.e., no site of  $C$  is nested within a passive node), and parameter bigraph  $d$  (without

any closed links); In this case the reaction produces a new agent  $\mathbf{a}' = \mathbf{C} \circ \mathbf{R}' \circ \mathbf{d}'$ , where  $\mathbf{d}'$  is computed from  $\mathbf{d}$  as prescribed by  $\bar{\rho}$ . (Note that the composition will usually implicitly extend the interfaces of the redex with an identity linking connecting all non-local names in the outerface of  $\mathbf{d}$  to  $\mathbf{C}$ .)

In general parameters (possibly with local names) may be copied. Consider for instance a rule for replicated input in the  $\pi$ -calculus  $\bar{x}(z) + T \mid !x(y).P \rightarrow \{z/y\}P \mid !x(y).P$ , which could be modelled by the bigraphical reaction

```
pirepsync ::= Sum o (Send[x,z] | []) | RGet[x][[y]] o [y]
            --[0&[y1] |-> 1[y], 1&[y2] |-> 1[y]]--|>
            z/y1 o [y1] | RGet[x][[y2]] o [y2] .
```

Note that the mapping  $\bar{\rho}$  of parameters defines how the local names of a parameter is mapped to local names in the holes of the reactum.

When duplicating parts of the agent (by letting  $\bar{\rho}$  map several reactum sites to a single redex site as above), *bound* links in  $\mathbf{d}$  are *copied* for each copy in  $\mathbf{d}'$ , while *free* links are *shared* between the copies. Since the format for reaction rules does not allow  $\mathbf{d}$  to contain closed links (local, but not bound names) it means that one must take care in defining rules that copy parts of the bigraph containing a closed link: The closed links will be converted to free links in the parameter (and closed in the context of the redex), thus the different copies will share the same local name. (For this reason, the `pirepsync` rule suggested above is in fact only correct if  $Q$  has no local names. If  $Q$  has a local name all replicas will share the same name. We refer to [9] for a solution to this problem.) In our formalization of WS-BPEL the use of binding ports are crucial to create *fresh* id and scope links when new instances or scopes are created, allowing us e.g. to distinguish between variables in different instances created from the same process.

As a token of the uniformity of bigraphical reactive systems, we briefly show how to encode 1-safe Petri Nets as binding bigraphs using the BPL-tool language. A 1-safe Petri Net  $(P, T, \mathcal{F}, M_0)$  of places  $P$ , transitions  $T$ , flow relation  $\mathcal{F}$  and initial marking  $M_0$  can be modelled in the BPL Tool by using atomic controls  $\mathbf{m}$  (marked),  $\mathbf{u}$  (unmarked) and  $\mathbf{th}_k$  for transitions with input/output arity  $h/k$  [12]. For each transition  $t \in T$  with its flows  $p_{i_1} \rightarrow t, \dots, p_{i_h} \rightarrow t, t \rightarrow p_{j_1}, \dots, t \rightarrow p_{j_k} \in \mathcal{F}$ , a node  $\mathbf{th}_k[\mathbf{p}_{i_1}, \dots, \mathbf{p}_{i_h}, \mathbf{p}_{j_1}, \dots, \mathbf{p}_{j_k}]$  is created, and for each initial mapping  $p_i \mapsto \mathbf{m}$  or  $p_i \mapsto \mathbf{u}$  in  $M_0$ , a node  $\mathbf{m}[\mathbf{p}_i]$  or  $\mathbf{u}[\mathbf{p}_i]$  is created, respectively. The initial state is obtained by putting all nodes in parallel with the  $|$ -operator (and closing the links between places and transitions), and for each transition arity  $h/k$  a rule is created:

```
 $\mathbf{m}[\mathbf{y}_1] \mid \dots \mid \mathbf{m}[\mathbf{y}_h] \mid \mathbf{th}_k[\mathbf{y}_1, \dots, \mathbf{y}_h, \mathbf{z}_1, \dots, \mathbf{z}_k] \mid \mathbf{u}[\mathbf{z}_1] \mid \dots \mid \mathbf{u}[\mathbf{z}_k]$ 
----|>
```

```
 $\mathbf{u}[\mathbf{y}_1] \mid \dots \mid \mathbf{u}[\mathbf{y}_h] \mid \mathbf{th}_k[\mathbf{y}_1, \dots, \mathbf{y}_h, \mathbf{z}_1, \dots, \mathbf{z}_k] \mid \mathbf{m}[\mathbf{z}_1] \mid \dots \mid \mathbf{m}[\mathbf{z}_k]$  .
```

expressing that if all the pre-places of a transition is marked and all its post-places are unmarked then the transition can fire and change the pre-places to unmarked and the post-places to marked.

Below we will show that bigraphs indeed also allow a very direct representation of a far more concrete language, that is, the WS-BPEL-language.

### 3 Formalizing WS-BPEL with Binding Bigraphs

We define our bigraphical representation of WS-BPEL in the BPL Tool with respect to the signature given in Tab. 3. As explained in the previous section, the signature determines the allowed controls for labeling nodes, and for each control the number of binding and free ports of nodes labeled with this control. For instance, we write `Reply =: 0 --> 6` for a control called `Reply` with binding arity 0 and free arity 6, which can be abbreviated to `Reply -: 6`. A control having zero binding and free arity is declared by just writing the control name, e.g. `Next`. The controls listed in the upper part of the signature correspond directly

<i>Active controls</i>	<i>Passive controls</i>	<i>Atomic controls</i>
PartnerLinks	Process =: 1 --> 1	To -: 2
Variables	Scope =: 1 --> 1	From -: 2
If -: 1	Variable -: 2	ToPLink -: 2
Condition	While -: 1	FromPLink -: 2
Sequence -: 1	Then	Invoke -: 8
Flow -: 1	Else	Receive -: 6
	Assign -: 1	Reply -: 6
	Copy	Exit -: 1
	PartnerLink -: 2	True
		False
		VariableRef -: 3
		CreateInstance -: 1
		GetReply -: 6
		ReplyTo -: 2
		Link -: 1
		Running -: 1
Instance -: 2	Next	Invoked -: 1
ActiveScope -: 2	Message -: 1	Stopped -: 1

**Table 3.** Signature for WS-BPEL

to constructs in WS-BPEL<sup>2</sup> and allow us to give a very direct representation of WS-BPEL processes, while the controls listed in the lower part are introduced to facilitate the formalization of the execution semantics.

Below we describe the bigraphical formalization of terms in the grammar in Tab. 1 by examples and refer to the full version of the paper for an inductive map defined over the grammar.

As an example, consider the process in Fig. 1(a). The process is represented in the BPL Tool as shown in Fig. 1(b). The place graph (nesting of controls) and the link graph correspond almost directly to the nesting of elements and the sharing of attributes of the XML representation respectively. But we need to introduce some additional structure. The main differences are:

<sup>2</sup> WS-BPEL allows several forms of the from and to elements. We have chosen to formalize two of these, namely those for variables (`From`, `To`) and partner links (`FromPLink`, `ToPLink`).

```

<process name="echo_process">
  <partnerLinks><partnerLink name="echo_client" /></partnerLinks>
  <variables><variable name="x" /></variables>
  <sequence>
    <receive partnerLink="echo_client" operation="echo"
      createInstance="yes" variable="x" />
    <reply partnerLink="echo_client" operation="echo" variable="x" />
  </sequence>
</process>

```

(a) Example WS-BPEL process.

```

val echo_process =
Process[echo_process][[echo_id]]
o ( PartnerLinks o PartnerLink[echo_client, echo_id] o CreateInstance[echo]
  | Variables o Variable[x, echo_id] o <->
  | Sequence[echo_id] o (
    Receive[echo_client, echo_id, echo, x, echo_id, echo_id]
  | Next o (
    Reply[echo_client, echo_id, echo, x, echo_id, echo_id]))

```

(b) BPL Tool representation of example process.

**Fig. 1.** Example process.

1. Since the children of a node in a bigraph are unordered, and children of an XML element are ordered, the second activity of the sequence construct (which we assumed to be binary) is enclosed in a node labeled by the `Next` control.
2. To be able to identify the scope of partner links and variables we have added an explicit link from the `PartnerLink` and `Variable` nodes to a binding port (called `echo_id` in the example) of the `Process` and `Scope` nodes. This will be explained below when we describe the semantics of assignment and scopes. For similar reasons, we also link expressions and activities to the binding port of the enclosing `Process` node.
3. We insert `CreateInstance` nodes in each `PartnerLink` to be able to identify at top level the receive activities with the `createInstance="yes"` attribute using that partner link.

We represent the XPath expressions `true()` and `false()` by nodes with the controls `True` and `False`, respectively. Variable references (e.g. `$x`) are represented by `VariableRef` nodes.

A key feature of the formalization is that active process instances are represented almost as the processes, the main difference is that they are nested within an (active) `Instance` control instead of a (passive) `Process` control. Fig. 2(b) is an example of an instance. It exemplifies the case, where the echo process has been invoked resulting in a new instance of that process, which has performed the initial receive activity.

One might use the close correspondence between bigraphs and XML to translate the representation of instances into XML as shown in Fig. 2(a). This illustrates that the run-time execution format is very close to the process specification

```

val echo_instance =
Instance[echo_process, echo_id]
o ( Running[echo_id]
  | PartnerLinks o PartnerLink[echo_client, echo_id]
    o (Link[client_id] | ReplyTo[echo, client_id])
  | Variables o Variable[x, echo_id] o True
  | Sequence[echo_id] o
    (Reply[echo_client, echo_id, echo, x, echo_id, echo_id]))

```

(a) BPL Tool representation of example instance.

```

<instance name="echo_process" id="echo_id">
  <running id="echo_id" />
  <partnerLinks>
    <partnerLink name="echo_client" />
    <link id="client_id" /><replyTo operation="echo" id="client_id" />
  </partnerLink>
</partnerLinks>
<variables><variable name="x">True</variable></variables>
<sequence>
  <reply partnerLink="echo_client" operation="echo" variable="x" />
</sequence>
</instance>

```

(b) Example WS-BPEL instance.

**Fig. 2.** Example instance.

format. However, notice that we have also added a **Running** node in the instance; we call this the *status* node of the instance. The status node can be one of the three nodes **Running**, **Invoked** or **Stopped** and it will be explained in more detail below when we describe some of the central reaction rules.

### 3.1 Reaction Rules

In this section we present some of the reaction rules used in the formalization of WS-BPEL. For the full set of reaction rules see App. A.

*Abnormal termination:* The **Exit** activity in WS-BPEL allows processes to be abnormally terminated. Its semantics is given by two rules: The first rule **exit stop instance** changes the status of the instance from running to stopped by replacing the **Running** node inside the instance with a **Stopped** node.

```

"exit stop instance" :::
  Exit[id] || Running[id] ----|> <-> || Stopped[id]

```

The second rule **exit remove inst** removes the instance together with all its remaining content. This is simply done by replacing the **Instance** node with the empty root bigraph **<->** and discarding the parameter in hole 0.

```

"exit remove inst" :::
Instance[n, id] o (Stopped[id] | [ ]) ----|> <-> || n//[ ] || id//[ ]

```

(The “idle” links `n//[ ] || id//[ ]` in the reactum are simply there to ensure that the redex and reactum have the same outer interface.)

One may think that the above semantics could be defined as a single rewrite rule, with a redex matching an instance containing an active `Exit` activity, and a reactum that simply replaces this instance with the empty bigraph `<->`. However, the `Exit` node may be nested arbitrarily deep (e.g. inside `Flow` nodes) within the `Instance` node. The format for parametric rules of binding bigraphs described in the previous section does not allow to match on the `Instance` node and test for the existence of an `Exit` node arbitrarily nested inside it<sup>3</sup>. Therefore we first match on the status node `Running` and the `Exit` node and change the status to `Stopped`. As the status node is a child of the `Instance` node, we can write a rule which matches instances which are stopped and discard them. All other rules, except for the rule `exit remove inst`, checks for the presence of the `Running` node, so the two reaction rules will always be applied consecutively.

For similar reasons, we also change the status temporarily to `Invoked` when creating a new instance, that is, when we execute a receive activity with the `createInstance="yes"` attribute.

*Conditionals and Iteration:* We give semantics to a while-loop in the traditional manner, by unfolding the loop once and using an if-then-else statement with the loop condition. In the syntax of the BPL Tool the rule `while unfold` looks as follows.

```
"while unfold" ::= While[id] o ( Condition o [ ] | [ ] ) || Running[id]
  --[0 |-> 0, 1 |-> 1, 2 |-> 0, 3 |-> 1]--|>
  If[id] o ( Condition o [ ] | Then o Sequence[id] o ( [ ] | Next o
    While[id] o ( Condition o [ ] | [ ] ) )
  | Else o <-> ) || Running[id]
```

Note how the parameter map `[0 |-> 0, 1 |-> 1, 2 |-> 0, 3 |-> 1]` on the arrow of the rule describes that the parameter in hole 0 (the condition expression) is copied and placed in both hole 0 and hole 2 of the reactum. Also, the parameter in hole 1 (the body of the while loop) is copied and placed in both hole 1 and hole 3 of the reactum. One may also note that the empty process, to be executed in the `Else` branch, is represented by the bigraph with a single empty root. As explained above, the `Running` node linked to the `While` node via the name `inst_id` is used to guarantee that the instance containing the while-loop is indeed running.

*Assignment and dynamic manipulation of partner links:* Of the many variants of the “Assign” activity in WS-BPEL, we cover in our formalization only the four allowing for copying the content of a `Variable` or `PartnerLink` to a `Variable` or `PartnerLink`. Each are covered by a *single* rule in the formalization. Below we show the case of the rule `assign copy plink2var` which copies the content of the `PartnerLink` node referenced to by the `FromPLink` node to the `Variable` node referenced to by the `To` node.

<sup>3</sup> This is possible using the higher-order reaction rules introduced in [7, 6].

```

"assign copy plink2var" :::
  Assign[id] o Copy o ( FromPLink[f, s1] | To[t, s2])
|| PartnerLink[f, s1] o [ ] || Variable[t, s2] o [ ] || Running[id]
--[0 |-> 0, 1 |-> 0]--|>
  <->
|| PartnerLink[f, s1] o [ ] || Variable[t, s2] o [ ] || Running[id]

```

The parameter link describes that the parameter of hole 0 is copied to both hole 0 and 1 in the reactum, and that the content of hole 1 is discarded. The  $f$  and  $t$  links determine the name of the partner link and the variable respectively. However, the name alone may not uniquely determine a variable/partner link. Since variables/partner links may be defined within nested scopes, several variables/partner links may have the same name. In this case the WS-BPEL specification states that the closest variable/partner link should be fetched. We represent this in the formalization by letting the  $s1$  and  $s2$  links connect respectively the `FromPLink` and `To` nodes to the closest partner link and variable with the correct name.

*Scopes:* The scope elements of WS-BPEL are represented by a corresponding node with a scope control in the formalization. As the other nodes above, the scope control has a port linked to the id port of the enclosing process/instance. However, as indicated above, scope controls also need an additional port to which every variable and partner link having it as closest scope are linked.

To make sure that scopes of different instances created from the same process gets a unique such link, this port needs to be a *binding* port. Again, a technical limitation of the format for reaction rules in binding bigraphs forces us to make the `Scope` control passive and have an explicit rule for *activating* a scope defined below. The rule replaces the passive `Scope` node with an active `ActiveScope` node, where the binding port is replaced by a normal (free) port, and the bound link by an local link (representing the local scope id) connected to that port. This is safe because the `Running` control ensures that activation is done *after* the instance has been created, thus scopes within two different instances created from the same process will not get the same scope id.

```

"scope activation" :::
  Scope[id][[s]] o [s] || Running[id]
--[0 |-> 0]--|>
  -//[s] o (ActiveScope[id, s] o [s]) || Running[id]

```

Note that the link map `-//[s]` closes the link connected to the scope port in the reactum, and that the name  $s$  is local to the hole 0 in both redex and reactum.

*Communication:* The content of partner links is used in the rules for invoke and reply activities to determine the target instance for communication. Thus, the ability of copying between partner links and variables makes it possible to send partner links as messages and dynamically assign instances as target for communication.

This is illustrated by the rule `invoke_instance` below, which allows two active instances to communicate. It synchronizes an active `Invoke` activity in one instance with a corresponding `Receive` activity in another instance, replacing the `Invoke` with a `GetReply` activity and removing the `Receive`. The parameter map ensures that the content of the input variable `invar` (hole 1) is copied to the appropriate variable of the receiving instance (hole 3 in the reactum).

```
"invoke_instance" :::
  Invoke[partner_link_invoker, partner_link_scope_invoker, oper,
         invar, invar_scope, outvar, outvar_scope, inst_id_invoker]
|| PartnerLink[partner_link_invoker, partner_link_scope_invoker]
  o (Link[inst_id_invoked] | [ ])
|| Variable[invar, invar_scope] o [ ]
|| Running[inst_id_invoker]
|| Receive[partner_link_invoked, partner_link_scope_invoked, oper,
           var, var_scope, inst_id_invoked]
|| PartnerLink[partner_link_invoked, partner_link_scope_invoked] o [ ]
|| Variable[var, var_scope] o [ ]
|| Running[inst_id_invoked]

--[0 |-> 0, 1 |-> 1, 2 |-> 2, 3 |-> 1]--|>

  GetReply[partner_link_invoker, partner_link_scope_invoker, oper,
           outvar, outvar_scope, inst_id_invoker]
|| PartnerLink[partner_link_invoker, partner_link_scope_invoker]
  o (Link[inst_id_invoked] | [ ])
|| Variable[invar, invar_scope] o [ ]
|| Running[inst_id_invoker]
|| <->
|| PartnerLink[partner_link_invoked, partner_link_scope_invoked]
  o ([ ] | ReplyTo[oper, inst_id_invoker])
|| Variable[var, var_scope] o [ ]
|| Running[inst_id_invoked]
```

A similar rule allows the `GetReply` activity to synchronize with the corresponding reply activity in the invoked instance.

## 4 Conclusion and Future Work

We have formalized a non-trivial subset of WS-BPEL as a binding bigraphical reactive system. As in our previous work [6] we have utilized the close correspondence between bigraphs and XML to provide a formalization close to the WS-BPEL syntax, thereby reducing the gap between the language specification and the formalization. The prize payed for providing a formalization close to the concrete syntax of WS-BPEL is the use of a set of *language specific* reaction rules, resulting in a behavioral theory for which no tools for automatic verification yet exists. However, the uniformity of bigraphical reactive systems opens up for a future study of formal relationships *within* the framework of bigraphical reactive systems, between very concrete semantics as the one presented in the present paper and more abstract semantics, e.g. based on the  $\pi$ -calculus and Petri Net.

We do not claim to give a feature complete formalization of WS-BPEL, yet several new non-trivial aspects of WS-BPEL have been formalized compared to [6], including support for nested scopes, termination (exit), and dynamic assignment and communication of partner links. As a technical, but important point, we avoided higher-order reaction rules as used in [6]. This means that the general theory, techniques and tools developed for standard, binding bigraphs remain applicable to our formalization. In particular, we have described how the formalization can be defined within the BPL Tool [21], which allows for compositional definition, graphical visualization, and interactive simulation of binding bigraphical reactive systems.

We aim as future work to extend the formalization to a larger subset of WS-BPEL and provide formal maps between formalizations at different abstraction levels, and in particular the existing formalizations based on different approaches such as process calculi and Petri Net as outlined in the introduction. In formalizing the data flow it would be useful if the BPL Tool was extended to a notion of high-level bigraphs allowing e.g. built-in XML and/or ML datatypes and transformations in the reaction rules, analogous to the built-in ML datatypes and functions found in Coloured Petri Nets and the CPN Tool. This was already partly explored in the ReactiveXML implementation of pure bigraphical reactive systems described in [6]. A notion of *prioritized* reactions could also be interesting to explore as an alternative to the use of the status control in the present paper.

As mentioned in the introduction, we have already in [2] initiated work on extensions to the WS-BPEL language and formalization to allow for *mobile embedded sub-processes*. Future work will also include the study of type systems, e.g. relations to the work on formalizations of WSDL types, contracts and session types [10, 1, 3], and types for controlling mobility and adaptability in the higher-order WS-BPEL language proposed in [2]. Finally, it would be interesting to investigate the use of the general theory of bisimulation congruences available for bigraphical reactive systems in the setting of WS-BPEL.

## References

- [1] M. Bravetti and G. Zavattaro. Contract based multi-party service composition. In *Proceedings of FSEN 2007*, volume 4767 of *LNCS*, pages 207–222. Springer Verlag, 2007.
- [2] M. Bundgaard, A. J. Glenstrup, T. Hildebrandt, E. Højsgaard, and H. Niss. Formalizing higher-order mobile embedded business processes with binding bigraphs. In *Proceedings of COORDINATION 2008*, LNCS. Springer Verlag, 2008.
- [3] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *Proceedings of ESOP 2007*, volume 4421 of *LNCS*, pages 2–17. Springer Verlag, 2007.
- [4] T. C. Damgaard and L. Birkedal. Axiomatizing binding bigraphs. *Nordic Journal of Computing*, 13(1–2):58–77, 2006.
- [5] A. J. Glenstrup, T. C. Damgaard, L. Birkedal, and E. Højsgaard. An implementation of bigraph matching. *submitted*, 2008.

- [6] T. Hildebrandt, H. Niss, and M. Olsen. Formalising business process execution with bigraphs and Reactive XML. In *Proceedings of COORDINATION 2006*, volume 4038 of *LNCS*, pages 113–129. Springer Verlag, 2006.
- [7] T. Hildebrandt, H. Niss, M. Olsen, and J. W. Winther. Distributed Reactive XML. In *Proceedings of MTCoord 2005*, volume 150 of *ENTCS*, pages 61–80, 2006.
- [8] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to petri nets. In *Proceedings of BPM 2005*, volume 3649 of *LNCS*, pages 220–235. Springer Verlag, 2005.
- [9] O. H. Jensen and R. Milner. Bigraphs and mobile processes (revised). Technical Report UCAM-CL-TR-580, University of Cambridge – Computer Laboratory, 2004.
- [10] A. Lapadula, R. Pugliese, and F. Tiezzi. A WSDL-based type system for WS-BPEL. In *Proceedings of COORDINATION 2006*, volume 4038 of *LNCS*, pages 145–163. Springer Verlag, 2006.
- [11] A. Lapadula, R. Pugliese, and F. Tiezzi. A formal account of WS-BPEL. In *Proceedings of COORDINATION 2008*, volume 5052 of *LNCS*, pages 199–215. Springer Verlag, 2008.
- [12] J. J. Leifer and R. Milner. Transition systems, link graphs and Petri nets. *Journal of Mathematical Structures in Computer Science*, 16(6):989–1047, 2006.
- [13] N. Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0. In *Proceedings of WS-FM 2007*, volume 4937 of *LNCS*, pages 77–91. Springer Verlag, 2007.
- [14] N. Lohmann, H. Verbeek, C. Ouyang, C. Stahl, and W. M. P. van der Aalst. Comparing and evaluating Petri net semantics for BPEL. Computer Science Report 07/23, Eindhoven University of Technology, 2007.
- [15] R. Lucchi and M. Mazzara. A pi-calculus based semantics for ws-bpel. *Logic and Algebraic Programming*, 70:96–118, 2007.
- [16] R. Milner. Axioms for bigraphical structure. *Journal of Mathematical Structures in Computer Science*, 15(6):1005–1032, 2005.
- [17] OASIS WSBPEL Technical Committee. Web Services Business Process Execution Language, version 2.0, 2007. (<http://docs.oasis-open.org/wsbpel/2.0/0S/wsbpel-v2.0-0S.pdf>).
- [18] C. Ouyang, E. Verbeek, W. M. van der Aalst, S. Breutel, M. Dumas, and A. H. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL (revised version). BPM center report, BPMcenter.org, 2005.
- [19] F. Puhlmann and M. Weske. Using the pi-calculus for formalizing workflow patterns. In *Proceedings of BPM 2005*, volume 3649 of *LNCS*, pages 153–168. Springer Verlag, 2005.
- [20] C. Stahl. A Petri net semantics for BPEL. Informatik-Berichte 188, Humboldt-Universität zu Berlin, 2005.
- [21] The Bigraphical Programming Languages Group. The BPL Tool. ([http://www.itu.dk/research/pls/wiki/index.php/BPL\\_Tool](http://www.itu.dk/research/pls/wiki/index.php/BPL_Tool)), 2007. IT University of Copenhagen.
- [22] F. van Breugel and M. Koshkina. Models and verification of BPEL. Draft., 2006.

## A Formalizing WS-BPEL in the BPL Tool

This section contains all the reaction rules (with comments) of our formalization of WS-BPEL in the BPL Tool. In conjunction with the signature from Table 3

these rules allow one to simulate and visualize WS-BPEL processes. Both are available in electronic form at the BPLweb site:  
<http://tiger.itu.dk:8080/bplweb/index/18>.

*Structural activities:* When a **Flow** is completed (i.e. there are no more instructions in the flow to be executed) we garbage collect the flow. In the same manner, we garbage collect a **Sequence** if the current instruction is completed. We then make the following instruction the next to be executed.

```
"flow completed" :::
```

```
    Flow[inst_id] o <->
|| Running[inst_id]
    ----|>
    <->
|| Running[inst_id]
```

```
"sequence completed" :::
```

```
    Sequence[inst_id] o Next o '[]'
|| Running[inst_id]
    --[0 |-> 0]--|>
    '[]'
|| Running[inst_id]
```

The rules for evaluating an if-then-else statement is as expected. If the condition is **True** we execute the then-branch, otherwise we execute the else-branch.

```
"if true" :::
```

```
    If[inst_id] o (    Condition o True
                    ' | ' Then o '[]'
                    ' | ' Else o '[]' )
|| Running[inst_id]
    --[0 |-> 0]--|>
    '[]'
|| Running[inst_id]
```

```
"if false" :::
```

```
    If[inst_id] o (    Condition o False
                    ' | ' Then o '[]'
                    ' | ' Else o '[]' )
|| Running[inst_id]
    --[0 |-> 1]--|>
    '[]'
|| Running[inst_id]
```

We give semantics to a while-loop in the traditional manner, by unfolding the loop once and using an if-then-else construct with the loop condition.

```
"while unfold" :::

  While[inst_id] o (Condition o '[' ' '| ' '[' ')
|| Running[inst_id]

--[0 |-> 0, 1 |-> 1, 2 |-> 0, 3 |-> 1]--|>

  If[inst_id] o (
    Condition o '['
    '| ' Then o Sequence[inst_id] o (
      '['
      '| ' Next o
        While[inst_id]
        o (Condition o '[' ' '| ' '[' ')
    '| ' Else o <->)
  || Running[inst_id]
```

*Expression evaluation:* Our current formalization only supports one type of expressions, namely variable references. But one can easily extend the semantics to more expression types, simply by adding rules describing how to evaluate them – without having to alter the current rules.

A variable reference is evaluated by locating the referenced variable, using its name and “scope”-link, and then replacing the `VariableRef` node by the current content of the variable.

```
"variable reference" :::

  VariableRef[var, var_scope, inst_id]
|| Variable[var, var_scope] o '['
|| Running[inst_id]

--[0 |-> 0, 1 |-> 0]--|>

  '['
|| Variable[var, var_scope] o '['
|| Running[inst_id]
```

*Assignment:* The “Assign copy” activity copies the content of a `Variable/PartnerLink` to a `Variable/PartnerLink`. We only explain one of the cases (rule `assign copy var2plink`) which copies the content of the `Variable` node referenced to by the `From` node to the `PartnerLink` node referenced by the `ToPLink` node. More kinds of assignments can be supported by adding rules describing their semantics.

```
"assign copy var2var" :::

  Assign[inst_id] o Copy o (
    From[f, scope1]
    '| ' To[t, scope2])
|| Variable[f, scope1] o '['
```

```

|| Variable[t, scope2] o '[]'
|| Running[inst_id]

--[0 |-> 0, 1 |-> 0]--|>

<->
|| Variable[f, scope1] o '[]'
|| Variable[t, scope2] o '[]'
|| Running[inst_id]

"assign copy var2plink" :::

    Assign[inst_id] o Copy o (    From[f, scope1]
                                '| ' ToPLink[t, scope2])
|| Variable[f, scope1] o '[]'
|| PartnerLink[t, scope2] o '[]'
|| Running[inst_id]

--[0 |-> 0, 1 |-> 0]--|>

<->
|| Variable[f, scope1] o '[]'
|| PartnerLink[t, scope2] o '[]'
|| Running[inst_id]

"assign copy plink2var" :::

    Assign[inst_id] o Copy o (    FromPLink[f, scope1]
                                '| ' To[t, scope2])
|| PartnerLink[f, scope1] o '[]'
|| Variable[t, scope2] o '[]'
|| Running[inst_id]

--[0 |-> 0, 1 |-> 0]--|>

<->
|| PartnerLink[f, scope1] o '[]'
|| Variable[t, scope2] o '[]'
|| Running[inst_id]

"assign copy plink2plink" :::

    Assign[inst_id] o Copy o (    FromPLink[f, scope1]
                                '| ' ToPLink[t, scope2])
|| PartnerLink[f, scope1] o '[]'
|| PartnerLink[t, scope2] o '[]'
|| Running[inst_id]

--[0 |-> 0, 1 |-> 0]--|>

```

```

<->
|| PartnerLink[f, scope1] o '[]'
|| PartnerLink[t, scope2] o '[]'
|| Running[inst_id]

```

*Scope:* When we “execute” a scope we replace the passive `Scope` node with an active node of control `ActiveScope` and we replace the binding port with an edge (which we call `scope`).

```

"scope activation" :::

  Scope[inst_id][[scope]] o '[scope]'
|| Running[inst_id]
  --[0 |-> 0]--|>
  -//[scope] o (ActiveScope[inst_id, scope] o '[scope]')
|| Running[inst_id]

```

In a similar manner we need to be able to activate scopes in newly created instances, while their status is still `Invoked`.

```

"scope activation 2" :::

  Scope[inst_id][[scope]] o '[scope]'
|| Invoked[inst_id]
  --[0 |-> 0]--|>
  -//[scope] o (ActiveScope[inst_id, scope] o '[scope]')
|| Invoked[inst_id];

```

When we are finished executing the body of the scope we remove the scope, including its variables, partner links, and its associated “scope”-edge.

```

"scope completed" :::

  ActiveScope[inst_id, scope]
  o (Variables o '[]' '|' PartnerLinks o '[]')
|| Running[inst_id]
  ----|>
  <-> || scope//[[]]
|| Running[inst_id]

```

*Process termination:* Processes can terminate in two ways: 1) normally, i.e. when no more activities remain, or 2) abnormally by executing an `Exit` activity. In the first case, we simply remove the instance in the same way as for scopes.

```

"inst completed" :::

Instance[proc_name, inst_id]
o (Variables o '[]' '|' PartnerLinks o '[]' '|' Running[inst_id])
  ----|>
<-> || proc_name//[[]] || inst_id//[[]]

```

In the case of an **Exit** activity we change the status of the instance from running to stopped by replacing the **Running** node inside the instance with a **Stopped** node. This prevents other rules from being used, except for the rule "exit remove instance" below, effectively stopping any other activity from proceeding.

```
"exit stop instance" :::
```

```
    Exit[inst_id]
|| Running[inst_id]
----|>
    <->
|| Stopped[inst_id]
```

Once an **Instance** node contains a **Stopped** node we garbage collect the instance together with all its remaining content.

```
"exit remove inst" :::
```

```
Instance[proc_name, inst_id]
o (Stopped[inst_id] '[]')
----|>
<-> || proc_name//[] || inst_id//[]
```

*Process communication:* Our formalization includes synchronous request-response communication, which is achieved in WS-BPEL using, in order, the invoke, receive, and reply activities. There are two cases: the receive can either 1) be an activity of a running instance, or 2) it can create a new instance of a process.

The first case is implemented by the "invoke instance" rule which handles both the invoke and receive in one step, while the second is modeled by two rules: "invoke" and "receive".

The invoke rule represents the case where an **Invoke** activity is executed inside a running instance and we have a process with the appropriate operation available and marked as being able to create new instances. The reactum 1) replaces the **Invoke** activity in the calling instance with a **GetReply** activity, which is used to represent that the instance is waiting for the reply, and 2) creates a new instance with the body of the process definition and the value of the input variable in a **Message** node within the relevant **PartnerLink** node. The partner links are updated to reflect the connection between the two instances: A **Link** node is inserted into the **PartnerLink** nodes of the instances, with a connection to the scope link of the other instance.

```
"invoke" :::
```

```
    Invoke[partner_link_invoker, partner_link_scope_invoker, oper,
           invar, invar_scope, outvar, outvar_scope, inst_id_invoker]
|| PartnerLink[partner_link_invoker, partner_link_scope_invoker] o <->
|| Variable[invar, invar_scope] o '[]'
|| Running[inst_id_invoker]
```

```

|| Process[proc_name][[scope]]
  o ( PartnerLinks
    o ( PartnerLink[partner_link, scope]
      o ( CreateInstance[oper] '| ' [] '
        '| ' scope//[scope1] o '[scope1]'
        '| ' scope//[scope2] o '[scope2]'

--[0 |-> 0, 1 |-> 1, 2 |-> 2, 3 |-> 3,
  4 |-> 0, 5&[inst_id_invoked1] |--> 2&[scope1],
  6&[inst_id_invoked2] |--> 3&[scope2]]--|>

-//[inst_id_invoked]
o ( GetReply[partner_link_invoker, partner_link_scope_invoker, oper,
  outvar, outvar_scope, inst_id_invoker]
  || PartnerLink[partner_link_invoker, partner_link_scope_invoker]
  o Link[inst_id_invoked]
  || Variable[invar, invar_scope] o ' [] '
  || Running[inst_id_invoker]
  || (Process[proc_name][[scope]]
    o ( PartnerLinks
      o ( PartnerLink[partner_link, scope]
        o ( CreateInstance[oper] '| ' [] '
          '| ' scope//[scope1] o '[scope1]'
          '| ' scope//[scope2] o '[scope2]'
        '| ' Instance[proc_name, inst_id_invoked]
          o ( PartnerLinks
            o ( PartnerLink[partner_link, inst_id_invoked]
              o ( Link[inst_id_invoker]
                '| ' Message[oper] o ' [] '
                '| ' ReplyTo[oper, inst_id_invoker])
                '| ' inst_id_invoked//[inst_id_invoked1]
                  o '[inst_id_invoked1]'
                '| ' Invoked[inst_id_invoked]
                '| ' inst_id_invoked//[inst_id_invoked2]
                  o '[inst_id_invoked2]'))))

```

The receive rule takes care of activating the instance, by removing a receive node associated to the partner link and the operation (indicated by the link of the `Message`), copying the content of the `Message` in the `PartnerLink` to the proper input variable, and changing the `Invoked` node to a `Running` node.

```
"receive" :::
```

```

  Receive[partner_link, partner_link_scope, oper, var, var_scope, inst_id]
  || PartnerLink[partner_link, partner_link_scope]
  o (' [] ' '| ' Message[oper] o ' [] '
  || Variable[var, var_scope] o ' [] '
  || Invoked[inst_id]

--[0 |-> 0, 1 |-> 1]--|>

```

```

<-> || oper//[[]
|| PartnerLink[partner_link, partner_link_scope]
  o '[]'
|| Variable[var, var_scope] o '[]'
|| Running[inst_id]

```

The `invoke` instance rule executes an `Invoke` activity in one instance simultaneously with a corresponding `Receive` activity in another instance, replacing the `Invoke` with a `GetReply` activity and removing the `Receive`. The invoking process' `PartnerLink` is used to identify the receiving instance. The content of the input variable is copied to the appropriate variable of the receiving instance.

```
"invoke_instance" :::
```

```

  Invoke[partner_link_invoker, partner_link_scope_invoker, oper,
    invar, invar_scope, outvar, outvar_scope, inst_id_invoker]
|| PartnerLink[partner_link_invoker, partner_link_scope_invoker]
  o (Link[inst_id_invoked] '| ' '[]')
|| Variable[invar, invar_scope] o '[]'
|| Running[inst_id_invoker]
|| Receive[partner_link_invoked, partner_link_scope_invoked, oper,
  var, var_scope, inst_id_invoked]
|| PartnerLink[partner_link_invoked, partner_link_scope_invoked] o '[]'
|| Variable[var, var_scope] o '[]'
|| Running[inst_id_invoked]

```

```
--[0 |-> 0, 1 |-> 1, 2 |-> 2, 3 |-> 1]--|>
```

```

  GetReply[partner_link_invoker, partner_link_scope_invoker, oper,
    outvar, outvar_scope, inst_id_invoker]
|| PartnerLink[partner_link_invoker, partner_link_scope_invoker]
  o (Link[inst_id_invoked] '| ' '[]')
|| Variable[invar, invar_scope] o '[]'
|| Running[inst_id_invoker]
|| <->
|| PartnerLink[partner_link_invoked, partner_link_scope_invoked]
  o ('[]' '| ReplyTo[oper, inst_id_invoker])
|| Variable[var, var_scope] o '[]'
|| Running[inst_id_invoked]

```

The `Reply` activity inside one instance can synchronize together with a `GetReply` activity inside another instance, thereby copying the content from variable `var` to variable `outvar`.

```
"reply" :::
```

```

  Reply[partner_link_invoked, partner_link_scope_invoked, oper,
    var, var_scope, inst_id_invoked]
|| PartnerLink[partner_link_invoked, partner_link_scope_invoked]

```

```
o (ReplyTo[oper, inst_id_invoker] '[]')
|| Variable[var, var_scope] o '[]'
|| Running[inst_id_invoked]
|| GetReply[partner_link_invoker, partner_link_scope_invoker, oper,
            outvar, outvar_scope, inst_id_invoker]
|| Variable[outvar, outvar_scope] o '[]'
|| Running[inst_id_invoker]

--[0 |-> 0, 1 |-> 1, 2 |-> 1]--|>

<-> || oper//[ ]
|| PartnerLink[partner_link_invoked, partner_link_scope_invoked] o '[]'
|| Variable[var, var_scope] o '[]'
|| Running[inst_id_invoked]
|| <-> || partner_link_invoker//[ ] || partner_link_scope_invoker//[ ]
|| Variable[outvar, outvar_scope] o '[]'
|| Running[inst_id_invoker]
```