Formalizing WS-BPEL and Higher Order Mobile Embedded Business Processes in the Bigraphical Programming Languages (BPL) Tool

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

IT University Technical Report Series TR-2008-103

ISSN 1600–6100 May 2008

ISSN 1600–6100

ISBN 978-87-7949-162-5

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S
Denmark

Telephone:	+45 72 18 50 00
Telefax:	+45 72 18 50 01
Web	www.itu.dk

Formalizing WS-BPEL and Higher Order Mobile Embedded Business Processes in the Bigraphical Programming Languages (BPL) Tool[*]

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

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

Abstract

Bigraphical Reactive Systems (BRSs) have been proposed as a formal meta-model for global ubiquitous computing that encompasses process calculi for mobility, notably the π-calculus and the Mobile Ambients calculus, as well as graphical models for concurrency such as Petri Nets. In this paper we demonstrate that BRSs also allow natural formalizations of programming languages used in practice. We do so by providing a direct and extensible formalization of a subset of WS-BPEL as a binding bigraphical reactive system using the BPL Tool developed in the Bigraphical Programming Languages (BPL) project. The tool allows for compositional definition, visualization and simulation of the execution of bigraphical reactive systems. The formalization exploits the close correspondence between bigraphs and XML to provide a formalized run-time format very close to standard WS-BPEL syntax.

The formalization is the starting point of an endeavor to provide a completely formalized and extensible business process engine within the Computer Supported Mobile Adaptive Business Processes (CosmoBiz) research project at the IT University of Copenhagen. Building upon the formalization of WS-BPEL we propose and formalize HomeBPEL, a higher-order WS-BPEL-like business process execution language where processes are *first-class values* that can be stored in variables, passed as messages, and activated as embedded *sub-instances*. A sub-instance is similar to a WS-BPEL scope, except that it can be dynamically *frozen* and stored as a process in a variable, and then subsequently be *thawed* when reactivated as a sub-instance. We motivate HomeBPEL by an example of pervasive health care where treatment guidelines are dynamically deployed as sub processes that may be delegated dynamically to other workflow engines and in particular stay available for disconnected operation on mobile devices.

[*]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).

Contents

1 Introduction

Services implemented and orchestrated by processes written in languages such as WS-BPEL are being put forward as a means to achieve loosely coupled and highly flexible computer supported business and work processes. In the current architectures, services are deployed and managed on web servers by meta-level tools and cannot be replaced or moved during the life-time of a session with an instance of the service. In the present paper we propose and formalize a higher-order WS-BPEL-like language, called HomeBPEL, where processes are values that can be stored in variables and dynamically instantiated as embedded sub-instances. A sub-instance is similar to a WS-BPEL scope, except that it can be dynamically frozen during a session and stored as a process in a variable. When frozen in a variable, the process instance can be sent to remote services as any other content of variables and dynamically re-instantiated as a local sub-instance continuing its execution.

We envisage a use of HomeBPEL where the necessary services or even active instances can be dynamically moved to a local process engine running on a mobile device and thereby allow for disconnected operation. We exemplify this use by an example of pervasive health care, where treatment workflows are moved between and executed locally on mobile devices belonging to either the doctor or the patient, depending on whether the guideline requires actions by the doctor or it prescribes actions carried out as self-treatment by the patient.

As a first step towards the formalization of HomeBPEL we provide a formalization of a non-trivial subset of the Web Service Business Process Execution Language, WS-BPEL [35], which is being promoted by major industrial players including IBM, SAP, BEA, Oracle, and Microsoft as the future standard for orchestrating web-services as business processes. The formalization exploits the close correspondence between bigraphs and XML to provide a small step rewrite semantics of the behavior of WS-BPEL, and the formalization uses a representation of the state of active process instances which is very close to the XML syntax of WS-BPEL processes. Building upon this formalization we provide a bigraphical formalization of a WS-BPEL-like business process language supporting higher-order primitives.

The investigation is part of the Computer Supported Mobile Adaptive Business Processes (CosmoBiz) project [24], which aims to provide a fully formalized runtime engine for a business process language extended to allow for mobile and adaptive processes. Our primary goals of the formalization is 1) to be able to guarantee that the implemented engine actually conforms to the semantics and 2) to form a basis for the development of type systems that can be used to statically guarantee safe and reliable behavior. To achieve the first goal a main concern is to limit the gap between the source language, its formalization, and the implementation. A key element to achieve the second goal is to strive for a *compositional* formalization supporting subsequent formalization of type rules for the individual parts. We want to stress that it is *not* a main concern at this point to provide techniques or principles for verification of processes, which has been the main concern of most WS-BPEL formalizations so far. However, we do hope that future reasoning techniques developed for bigraphs can be employed also to support formal verification.

The work on HomeBPEL is inspired and guided by our previous work on the Homer process calculus of Higher-order mobile embedded resources [21, 19, 8], and in particular its formalization as a bigraphical reactive system [7]. Not surprisingly, the new features add to the complexity of the language and its formalization. Yet, the formal approach ensures that they are completely unambiguously specified. Also, the close relationship to semantics of process calculi such as Homer and the Mobile Ambients gives a very succinct formalization of sub-process mobility. Indeed, the serialized representation of a mobile process is just a process description. In particular, this means that a future implementation could use the standard XML format for serialized process instances.

The theory of *Bigraphical reactive systems* [26] provides a framework in which process models for concurrent and ubiquitous computing can be uniformly defined and formally analyzed. In particular, the π-calculus [33], Mobile Ambients calculus [10] and (1-safe) Petri Nets [29] have been represented as instances of bigraphical reactive systems [26]. Bigraphical reactive systems can be seen as a specialized kind of graph rewriting systems, in which processes are represented as two graphs (hence the name *bi*graphs): The place graph and the link graph respectively. The *place graph* is a collection of node labeled trees generalizing the nesting of process constructors in process calculi. The *link graph* is a hyper graph specifying links between ports associated to the nodes of the place graph and a set of external names, generalizing the link structure characteristic of the π-calculus. The dynamics is specified by a set of parametric reaction rules, generalizing the rule formats used in e.g. the π-calculus and the calculus of Mobile Ambients. The general theory of bigraphical reactive systems provides a general notion of contexts and composition. This allows for *compositional* description of processes as characteristic to process calculi. Together with a notion of "minimal" contexts, it forms the basis for an *automatic* derivation of a labeled transition bisimulation congruence from the reaction rules supporting compositional reasoning about the behavior of systems.

There are several reasons for why it is interesting to apply bigraphs to formalize an evolving standard such as WS-BPEL. Firstly, we thereby demonstrate that bigraphical reactive systems can not only be used as a meta-format for process calculi, but also be used to formalize programming languages used in practice. Secondly, the model of bigraphical reactive systems is *extensible*: An instance of a bigraphical reactive system is defined by its signature (the possible labels and ports of nodes) and its reaction rules, which can be chosen to fit a particular language and its semantics. By extending the signature and the set of reaction rules, a bigraphical reactive system can be adapted according to e.g. changes in the language specification or incremental extensions of the language. We exploit the extensibility of bigraphical reactive systems to extend the language and formalization of WS-BPEL with primitives for mobile, embedded sub-processes. Furthermore, the place and link graph of bigraphs correspond closely to respectively the nested element structure and sharing of attribute values in the XML data model. Since WS-BPEL is equipped with an XML syntax, we are able to provide a small step rewrite semantics in the model of bigraphical reactive systems using a representation of the current state of processes which is very close to the WS-BPEL syntax. Indeed, the close correspondence between bigraphs and XML was explored in our previous work on formalizing WS-BPEL as bigraphs [23, 22], on which the present work builds. However, the formalization in [23, 22] 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 been developed. In addition to covering a larger subset of WS-BPEL, the present formalization stays within the standard format for binding bigraphs described in [26]. Thus, the general theory, techniques and tools developed for binding bigraphs remain applicable to our formalization. In particular, we describe how the formalization can be explored within the BPL Tool [3] developed in the Bigraphical Programming Languages (BPL) project at the IT University of Copenhagen. The tool allows compositional definition of bigraphical reactive systems within Standard ML. It is also equipped with a web interface supporting visualization and interactive simulation of the execution of binding bigraphical reactive systems based on the formal inference of rule matching described in [1, 2].

The present paper combines the work presented in the two papers: [5] and [6].

Related work. WS-BPEL has been the target for several formalizations [40] accompanying the official informal specification [35]. Generally, any formalization requires a compilation of a BPEL process to a representation in the formal model. Clearly, the usefulness of a formalization depends on the availability of tools and reasoning techniques for the formal model, but also on how easy it is to relate the formal representation to the original BPEL process description.

Many of the prior formalizations have been based on versions of Petri Nets [38, 32], following the tradition of formal workflow models. Other authors have been promoting the use of process calculi, notably the π-calculus [39, 36]. This diversion can be partly 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 π-calculus. A third line of formalizations are based on abstract state machines (ASMs) [15, 16, 14, 13]. These seek to represent the informal specification *as is*, i.e. they aim at using the same terminology and level of abstraction in their formalizations, thereby hoping to minimize the gap to the informal specification. This goal is shared by our approach, though our method focuses on keeping the formalization close to the BPEL language itself and not its informal description.

In this paper 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 π-calculus). However, since bigraphical reactive systems have been shown to faithfully represent both the π-calculus and Petri Nets, we believe the model is a good candidate 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 the π-calculus.

Our formalization of the core WS-BPEL subset relates to the WS-BPEL process calculus given in [28]. An advantage to our approach is that we can reuse the general theory developed for bigraphical reactive systems, instead of redeveloping an entire theory of a new process calculus. As in [28], we hope to be able to equip our formalization with WSDL-like (or even richer) type systems.

Sub-processes have been proposed by IBM and SAP in [27] as an extension to WS-BPEL (called BPEL-SPE) to allow for modularization and reuse of process fragments to ease the burden of designing large business processes. As argued in [27] one could simulate some of the behavior of sub-processes by invoking another process instead of invoking a sub-process. However, this makes it impossible to establish any coupling between the life-cycles of the two process instance, e.g. enforcing that a sub-process exits if the super process exits prematurely. The sub-processes we propose in this paper extend the proposal in [27] in several aspects. First and foremost, BPEL-SPE requires that the *sole* interaction of a sub-process is an initial receive activity, and a last reply activity, basically making the sub-process act as a method or function call. We allow that the sub-process can communicate unrestrictedly with the parent process (and vice versa) using invoke-receive. Furthermore, we add facilities for "freezing" and "thawing" sub-processes as well as (sub-)process mobility.

Higher order workflow models applied to health care processes have been considered in the context of Higher-Order (Petri) Nets [25], allowing sub-processes (nets) as values (tokens), which may be dynamically composed. The approach in [25] differs from ours in several ways: Firstly, the approach in [25] is based on Petri Nets as opposed to process calculi, and has no direct relationship to WS-BPEL nor service orchestration. Another central difference is that we execute sub-processes as sub-threads, whereas in [25] a sub-process is executed step-by-step by the super process — and sub-processes can not contain sub-processes themselves. Finally, the model in [25] allows for dynamic modification and composition of sub-processes, which is not yet supported in our setting.

As described above, our proposal of higher order mobile sub-processes relates to our work on the higher-order process calculus Homer. The Homer calculus is related to the process calculus of Mobile Ambients [10] and the Seal calculus [11]. Indeed, HomeBPEL shares with Seal the combination of name (link) and process passing. We leave for future work to explore the relationship between HomeBPEL and these process calculi. Again, we hope that the many type systems proposed for process calculi for mobility can guide us to equip our formalization with useful type systems for controlling the mobility.

In [20] a notion of *mobile* business process is defined by processes in which a) the place of execution of an activity can change, or can be different for different instances, b) the change is caused by external factors, and c) cooperation with external resources is needed. The mobile sub-instances we propose meet all these criteria.

Structure of the paper. In Sec. 2 we introduce the meta-model of binding bigraphs, and in Sec. 3 we utilize binding bigraphs to give a formal semantics of a subset of WS-BPEL and visualize it using the BPL Tool. In Sec. 4 we motivate the need for higher-order constructs with an example of computer supported pervasive health care, and we present the resulting language HomeBPEL — a WS-BPEL-like language where processes are values that can be stored in variables and dynamically instantiated as embedded sub-instances. In Sec. 5 we formalize HomeBPEL using the BPL Tool. We conclude and propose directions for future work in Sec. 6.

2 Binding Bigraphs and BPL Tool

In this section we briefly review the binding bigraphs of Milner and Jensen [26] and introduce the syntactical representation of binding bigraphs as implemented in the BPL Tool. For a complete introduction to bigraphs we refer to [26].

2.1 Binding Bigraphs

A binding bigraph is a pair of graphs: a *place graph* and a *link graph*. The place graph is an n-tuple of finite, unordered trees. Except for roots, every node is labelled by a *control* and has two finite ordered sets of respectively *free* and *binding ports*. The link graph is essentially 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.

What we just described above is known as *ground* binding bigraphs. Intuitively, one may think of a ground binding bigraph as an ordered tuple of terms of a process calculus up to structural congruence: Sibling nodes in the place graph represent processes combined by an associative and commutative parallel operator. Each node is a prefix, and each control denotes a distinct prefix operation (e.g. send or receive in the π-calculus) with free and binding ports representing names and name binders of the particular operation (e.g. for the π-calculus, any node labelled by a send control would have 2 free ports, while nodes labelled by a receive control would have one free and one binding port). The link graph then maps each name in a prefix to either a local name (closed link), a binder (i.e. a binding port) or a name in the interface.

A ground bigraph with a single root is also similar to the data model for XML, with controls playing the role of the names of XML elements, ports playing the role of attributes and the linking of ports playing the role of attribute values. As we will see below, we exploit this similarity to give a bigraphical semantics to HomeBPEL resembling closely the XML syntax.

A central ingredient of the theory of bigraphs is that bigraphs in general 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. sites), 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. sites). 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 $G : I \to J$ for the bigraph G with inner interface I and outer interface J. The composition $H \circ G : I \to J$ of bigraphs $G : I \to I'$ and $H : I' \to J$ with compatible interfaces is obtained by making the children of the ith root of G children of the (parent) node of the ith site of H, discarding the roots of G and sites of H, and by coalescing links as prescribed by the correspondence of H's inner and G's outer names.

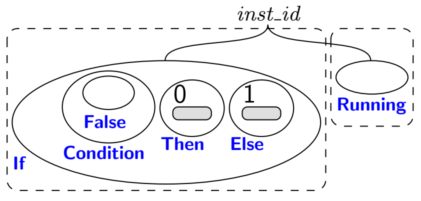
A binding bigraphical reactive system is defined with respect to a *signature*, which declare the set of possible controls labelling nodes of the bigraph and for each control K the number of binding and free ports of nodes in the bigraph labelled with K. The signature also declares each control as either 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, i.e. the active controls determine evaluation contexts.

2.2 BPL Tool Term Language

Binding bigraphs are often visualized graphically. However, binding bigraphs also admit a representation via a term language based on the axiomatization of binding bigraphs [12]. This representation is exploited in the BPL Tool to allow compact and compositional textual descriptions of binding bigraphs and their reaction rules[1].

In the present paper we will use the syntax of the term language as used in the BPL Tool. The language consists of Standard ML constructs which allows the user to write the terms directly in SML, at the cost of introducing a few additional back quotes. (Future versions of the BPL Tool will also support a clean input language stripped of SML artifacts.) The employed subset of the language can be defined by the following

[1]The representation is also exploited in the underlying formalization and implementation of matching used for the execution of reaction rules as described in [18].



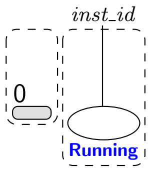


Figure 1: Example bigraphs `R` $: I \to J$ and `R'` $: I' \to J$, where $I = \langle 2, [\{\}, \{\}], \{\}\rangle, I' = \langle 1, [\{\}], \{\}\rangle, J = \langle 2, [\{\}, \{\}], \{inst\_id\}\rangle$

grammar.

$$P ::= P \texttt{ o } P \mid P \texttt{ || } P \mid P \texttt{ `|` } P \mid C$$
$$C ::= c \mid c\texttt{[}N^?\texttt{]} \mid c\texttt{[}N^?\texttt{][}NS^?\texttt{]} \mid \texttt{-//[}N^?\texttt{]} \mid n\texttt{//[}N^?\texttt{]} \mid \texttt{`[}N^?\texttt{]`} \mid \texttt{<->}$$
$$N^? ::= \epsilon \mid N \quad N ::= n \mid n\texttt{,}N \quad NS^? ::= \epsilon \mid NS \quad NS ::= \texttt{[}N^?\texttt{]} \mid \texttt{[}N^?\texttt{],}NS$$

where n ranges over strings representing names and c over strings representing controls. C describes so-called *ions* which are bigraphs consisting of a single root with a single node as child, having a control as defined in the signature. If the control is non-atomic the ion has a single hole inside. For instance, an ion with name c and i free ports and j binding ports is written $c\texttt{[}n_1, \ldots, n_i\texttt{][}NS_1, \ldots, NS_j\texttt{]}$ where the NS_k is the set of names bound to the kth binding port.

We use the double bars `||` to separate roots in the place graph and the single bar `` `|` `` as a separator between sibling nodes. The symbol `o` denotes composition as defined above (the tool checks that the interfaces of the bigraphs match). The terms $\texttt{-//[}N^?\texttt{]}$ and $n\texttt{//[}N^?\texttt{]}$ 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 $\texttt{`[}n_1, \ldots, n_k\texttt{]`}$ denotes a hole with local names $n_1, \ldots, n_k$. Finally, the term `<->` denotes a bigraph just consisting of a single empty root. As an example, we may define two binding bigraphs as follows (and depict the graphical representation of them in Fig. 1).

```
val R = If[inst_id] o (Condition o False `|` Then o `[]` `|` Else o `[]`)
     || Running[inst_id]
val R' = `[]` || Running[inst_id]
```

The bigraphs `R` and `R'` both have two roots. The first root of `R` has a single node as child with the control `If` and a single free port linked to the name `inst_id`. The node has three nodes as children, labelled respectively with the controls `Condition`, `Then` and `Else`. The first node has a single node as child labelled with the atomic control `False`. The two latter nodes both have a hole as a child. The holes in a bigraph term are ordered from left to right, i.e. the hole below the `Then` is indexed 0 and the hole below the `Else` is indexed 1. The second root of `R` has a single node as child labelled with the atomic control `Running` and a single free port linked to the name `inst_id`. The bigraph `R'` has simply a hole below its first root and the atomic `Running` control below its second root. The two bigraphs in fact form respectively the redex and reactum of a reaction rule, as defined below, defining the meaning of an if-then-else construct in the case where the condition has been evaluated to false.

2.3 Parametric Reaction Rules

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 `"rule name" ::: R --`$\bar{\varrho}$`--|> R'`, consists of two bigraphs: the *redex* `R` : $I \to J$ and the *reactum* `R'` : $I' \to J$, where both I and I' are local interfaces, and a parameter mapping $\bar{\varrho}$. The mapping $\bar{\varrho}$ indicates for each site in the reactum from which site in the redex the parameter is copied.

The expression `"if false" ::: R --[0 |-> 1]--|> R'` is a reaction rule for executing an `If` activity with a false condition. During a reaction, the first tree of `R` (the if-then-else construct) is replaced by the first tree the reactum `R'`. Since the second tree of `R` and `R'` are identical it simply means that a node with the `Running` control (and the correct id link) must be present in the context—this is used to ensure that rewrites are only performed on running instances which are ready to execute a step. The mapping `[0 |-> 1]` specifies that the hole in the reactum (site 0) should contain the contents of the hole in the `Else`-branch (site 1), while the contents of the hole in the `Then`-branch is discarded as it is not mentioned in the mapping.

In general parameters may have local names, thus the mapping $\bar{\varrho}$ must also define how the local names of a parameter is mapped to local names in the hole of the reactum. For instance, `[0&[x1] |--> 0&[x], 1&[x2] |--> 0&[x]]` is a mapping which (a) maps site 0 of the reactum and its local name `x1` to site 0 of the redex and its local name `x`, and (b) also maps site 1 of the reactum and its local name `x2` to site 0 of the redex and its local name `x`.

A rule matches an agent `a` if `a` = `C o (id`$_Z$` || R) o d` for some identity linking `id`$_Z$ and active context `C` (i.e., no site of `C` is nested within a passive node); the linking `id`$_Z$ connects all non-local names in the outerface of `d` to `C`. In this case reaction produces a new agent `a' = C o (id`$_Z$` || R') o d'`, where `d'` is computed from `d` as prescribed by $\bar{\varrho}$. When duplicating parts of the agent (by letting $\bar{\varrho}$ map several reactum sites to a single redex site), *local* links in `d` are *copied* to each copy in `d'`, while *free* links are *shared* between the copies. Binding ports thus enforce a notion of scope and locality on a bigraph's links, resembling the usual notion of binders in the λ- and the π-calculus. This feature of binding bigraphs is crucial in our formalization of WS-BPEL to create *fresh* id and scope links when new instances or scopes are created.

3 Formalizing WS-BPEL in the BPL Tool

In this section we present the subset of WS-BPEL considered in this report and its formalization in the BPL Tool. First we present the WS-BPEL subset in Sec. 3.1. Second we present the static representation in Sec. 3.2, i.e. the representation of processes and

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

Table 1: Grammar for WS-BPEL processes.

instances. Third we present the reaction rules capturing the dynamic behavior of WS-BPEL in Sec. 3.3.

3.1 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. For technical reasons, which will be explained in Sec. 3.3.6, we also assume that receive elements with the `createInstance="yes"` attribute refer to a partner link defined in the outermost scope. We write attributes as sets following the element name (e.g. `Process` {`name=echo`}) and let A range over such sets. If an element has no attributes, we leave out the set of attributes. Note that in regard to data flow we only consider the constant values given by the XPath expressions `true`() and `false`() and references to variables, assuming that x ranges over strings. We let **BPEL** refer to the set of terms defined by the grammar.

3.2 The Static Representation

We define our bigraphical representation of WS-BPEL in the BPL Tool with respect to the signature given in Tab. 2. 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 to the subset of WS-BPEL elements we are considering[2] (cf. Tab. 1) and allow us to give a very direct representation of WS-BPEL processes, while the controls listed in the lower part are

[2]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`).

Active controls	*Passive controls*	*Atomic controls*
`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 2: Signature for WS-BPEL

introduced to facilitate the formalization of the execution semantics. We will call the bigraphical reactive system for $\mathbf{BRS}^{\texttt{BPEL}}$.

As an example, consider the process in Fig. 2(a). The process is represented in the BPL Tool as shown in Fig. 2(b). The graphical representation, generated by the BPL Tool, is shown in Fig. 2(c) (the link to the binding port of the `Process` node has been colored green to improve readability). 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:

1. Since the children of a node in a bigraph are unordered, and children of an XML element are ordered, we represent the sequence construct as a nesting of binary sequence constructs, in which the second activity 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 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. To be able to identify the initial receive actions we insert `CreateInstance` nodes in each `PartnerLink` which identify an operation for which there is a receive activity with the `createInstance="yes"` attribute using that partner link.

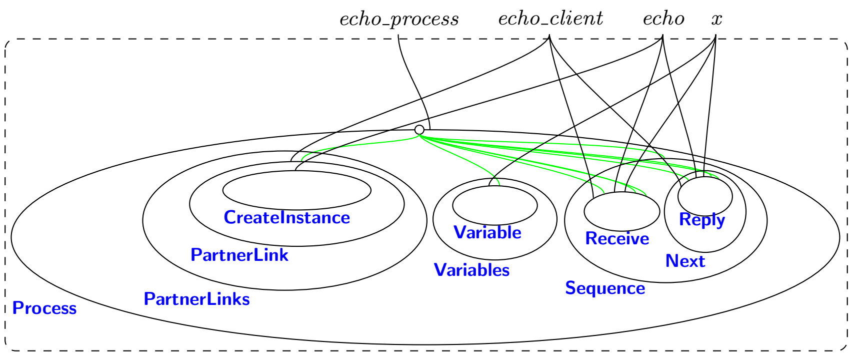
4. We require `PartnerLinks` and `Variables` nodes in each scope (including process). This allows for fewer and simpler reaction rules. This is a technicality as the absence of e.g. a `variables` declaration in a WS-BPEL process is equivalent to an empty `variables` declaration, so we just make them explicit in the representation.

```
<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.



(c) BPL Tool visualization of example process.

Figure 2: Example process.

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.

We give the formal definition of the map $\mathcal{C}[\![-]\!] : \mathbf{BPEL} \to \mathbf{BRS}^{\text{BPEL}}$ in Tab. 3. The map is defined using a map $[\![-]\!]^{x\_id,s\_id}_{\tilde{x},\sigma,\phi} : \mathbf{BPEL} \to \mathbf{BRS}^{\text{BPEL}}$ on sub-terms indexed by names x_id and s_id, indicating the name connected to the bound link identifying the process and the scope respectively, a set $\tilde{x}$ of names constituting the current names of bound links, a scope map σ mapping every defined partnerlink name or variable name to its scope link, and a map $\phi : \mathbf{Name} \to \mathbf{BRS}^{\text{BPEL}}$ mapping partnerlink names to bigraph terms of the form `CreateInstance`[op_1] `` `|` ``$\ldots$`` `|` ``CreateInstance[op_n] indicating for a given partnerlink name n that the process contains terms `Receive` A_i where $\{$`partnerLink=`n`, operation=`op_i`, createInstance=yes`$\} \subseteq A_i$, $1 \le i \le n$. We use the map $\mathcal{I}[\![-]\!] : \mathbf{BPEL} \to \mathbf{Name} \to \mathbf{BRS}^{\text{BPEL}}$ defined in Tab. 4 to find ϕ from the body of the process. For a set of partnerlink and variable declarations *pls* and *vars* we will write $\sigma[pls \cup vars \mapsto x\_id]$ for the update of the scope map σ mapping every partnerlink and

$$
\begin{aligned}
\mathcal{C}\llbracket\texttt{Process }\{\texttt{name=}n\}(pls\ vars\ act)\rrbracket &= \texttt{Process[}n\texttt{][[}n\_id\texttt{]]}\circ\\
&\quad(\llbracket pls\rrbracket^{n\_id,n\_id}_{\{n\_id\},[],\mathcal{I}\llbracket act\rrbracket}\ \texttt{`|`}\ \llbracket vars\rrbracket^{n\_id,n\_id}_{\{n\_id\},[],\phi_0}\ \texttt{`|`}\ \llbracket act\rrbracket^{n\_id,n\_id}_{\{n\_id\},[pls\cup vars\mapsto n\_id],\phi_0})
\end{aligned}
$$

$$
\begin{aligned}
\llbracket\texttt{PartnerLink }\{\texttt{name=}n\}\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi} &= \texttt{PartnerLink[}n\texttt{,}s\_id\texttt{]}\circ\phi(n)\\
\llbracket\texttt{Variable }\{\texttt{name=}n\}(p)\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi} &= \texttt{Variable[}n\texttt{,}s\_id\texttt{]}\circ\llbracket p\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi}\\
\llbracket\texttt{Scope}(pls\ vars\ act)\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi} &= \texttt{Scope[}x\_id\texttt{][[}y\_id\texttt{]]}\circ\\
&\quad(\llbracket pls\rrbracket^{x\_id,y\_id}_{\tilde{x},\sigma,\phi}\ \texttt{`|`}\ \llbracket vars\rrbracket^{x\_id,y\_id}_{\tilde{x}y\_id,\sigma,\phi}\ \texttt{`|`}\ \llbracket act\rrbracket^{x\_id,y\_id}_{\tilde{x}y\_id,\sigma[pls\cup vars\mapsto y\_id],\phi})\\
\llbracket\texttt{Sequence}(act\ act')\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi} &= \texttt{Sequence[}x\_id\texttt{]}\circ(\llbracket act\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi}\ \texttt{`|`}\ \texttt{Next}\circ\llbracket act'\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi})\\
\llbracket\texttt{From }\{\texttt{var=}n\}\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi} &= \texttt{From[}n\texttt{,}\sigma(n)\texttt{]}\\
\llbracket\texttt{To }\{\texttt{var=}n\}\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi} &= \texttt{To[}n\texttt{,}\sigma(n)\texttt{]}\\
\llbracket\texttt{From }\{\texttt{partnerLink=}n\}\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi} &= \texttt{FromPLink[}n\texttt{,}\sigma(n)\texttt{]}\\
\llbracket\texttt{To }\{\texttt{partnerLink=}n\}\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi} &= \texttt{ToPLink[}n\texttt{,}\sigma(n)\texttt{]}\\
\llbracket\texttt{Receive }\texttt{A}\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi} &= \texttt{Receive[}n\texttt{,}\sigma(n)\texttt{,}op\texttt{,}x\texttt{,}\sigma(x)\texttt{,}x\_id\texttt{]},\\
&\quad\text{if } A\supseteq\{\texttt{partnerLink=}n\texttt{,operation=}op\texttt{,variable=}x\}\\
\llbracket\texttt{Invoke }\texttt{A}\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi} &= \texttt{Invoke[}n\texttt{,}\sigma(n)\texttt{,}op\texttt{,}ix\texttt{,}\sigma(ix)\texttt{,}ox\texttt{,}\sigma(ox)\texttt{,}x\_id\texttt{]},\\
&\quad\text{if } A=\{\texttt{partnerLink=}n\texttt{,operation=}op\texttt{,}\\
&\qquad\qquad\texttt{inputVariable=}ix\texttt{,outputVariable=}ox\}\\
\llbracket\texttt{Reply }\texttt{A}\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi} &= \texttt{Reply[}n\texttt{,}\sigma(n)\texttt{,}op\texttt{,}x\texttt{,}\sigma(x)\texttt{,}x\_id\texttt{]},\\
&\quad\text{if } A=\{\texttt{partnerLink=}n\texttt{,operation=}op\texttt{,variable=}x\}\\
\llbracket E(p)\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi} &= E\circ\llbracket p\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi}\\
&\quad\text{where } E\in\{\texttt{PartnerLinks},\texttt{Variables},\texttt{Then},\texttt{Else},\texttt{Condition},\texttt{Copy}\}\\
\llbracket Eid(p)\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi} &= Eid\texttt{[}x\_id\texttt{]}\circ\llbracket p\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi}\\
&\quad\text{where } Eid\in\{\texttt{Flow},\texttt{If},\texttt{While},\texttt{Assign},\texttt{Exit}\}\\
\llbracket\texttt{true}()\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi} &= \texttt{True}\\
\llbracket\texttt{false}()\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi} &= \texttt{False}\\
\llbracket\$x\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi} &= \texttt{VariableRef[}x\texttt{,}\sigma(x)\texttt{,}x\_id\texttt{]}\\
\llbracket p\ p'\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi} &= \llbracket p\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi}\ \texttt{`|`}\ \llbracket p'\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi}\\
\llbracket\varepsilon\rrbracket^{x\_id,s\_id}_{\tilde{x},\sigma,\phi} &= \texttt{<->}
\end{aligned}
$$

Table 3: Translating **BPEL** into **BRS**$^{\text{BPEL}}$.

$$\mathcal{I}[\![\texttt{Receive } A]\!] = \begin{cases} \phi_0[n \mapsto \texttt{CreateInstance[}op\texttt{]}] & \text{if } ci = \texttt{yes} \\ \phi_0 & \text{otherwise} \end{cases}$$

$$\text{where } A \supseteq \{\texttt{partnerLink=}n\texttt{,operation=}op\texttt{,createInstance=}ci\}$$

$$\mathcal{I}[\![E\ A(p)]\!] = \phi_0$$

$$\text{where } E \in \{\texttt{Condition}, \texttt{Assign}, \texttt{Invoke}, \texttt{Reply}, \texttt{Exit}, \texttt{Variables}, \texttt{PartnerLinks}\}$$

$$\mathcal{I}[\![E\ A(p)]\!] = \mathcal{I}[\![p]\!]$$

$$\text{where } E \in \{\texttt{Scope}, \texttt{Flow}, \texttt{Sequence}, \texttt{While}, \texttt{If}, \texttt{Then}, \texttt{Else}\}$$

$$\mathcal{I}[\![p\ p']\!] = \mathcal{I}[\![p]\!] \mid \mathcal{I}[\![p']\!]$$

$$\mathcal{I}[\![\varepsilon]\!] = \phi_0$$

$$\phi_0(n) = \texttt{<->}$$

$$(\phi \mid \phi')(n) = \phi(n)\ \text{`}\texttt{|}\text{`}\ \phi'(n)$$

Table 4: Finding `Receive` terms with `createInstance=yes`.

variable name in *pls* and *vars* to the name x_id and every other name n in the domain of σ to $\sigma(n)$. We write $\tilde{x}y$ for the disjoint union of $\tilde{x}$ and $\{y\}$ (i.e. implying $y \notin \tilde{x}$), which is used in generating fresh bound names of scopes. Note that we assume a partition of the set of names into two disjoint sets, one ranged over by strings with the suffix *_id* (e.g. x_id), which is used for scope identifiers, and one ranged over by strings without the suffix, which corresponds to XML attribute values. This prevents clashes between scope identifiers introduced in the translation and attribute values. We let p range over any sub term of **BPEL**, including the empty term ε.

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. 3(b) is an example of an instance, which is visualized in Fig. 3(c) (again some of the links have been colored to improve readability). 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. We have left out the calling instance from the figure to keep the example simple — the edges "client id edge" and "echo id edge", is connected to respectively the id port and a port of a link node in the PartnerLink (used for getting the reply) in the calling instance.

One might use the close correspondence between bigraphs and XML to translate the representation of instances into XML as shown in Fig. 3(a). This illustrates that the run-time execution format is very close to the process specification format.

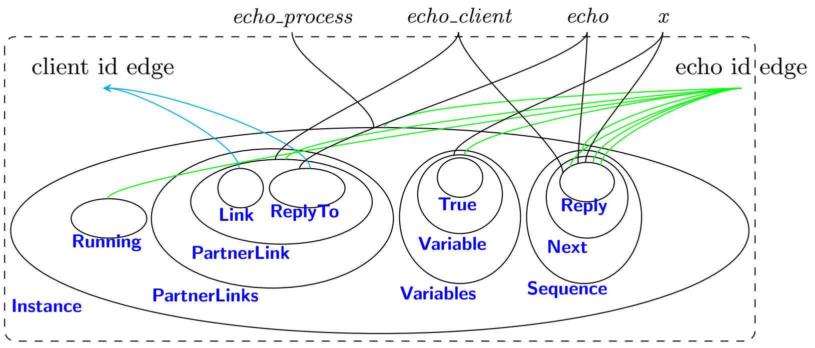
However, notice that we have also added a `Running` node in the instance; we call this the *status* node of the instance. The purpose of the status node is twofold and somewhat intricate, and will be explained below when we describe the reaction rules.

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

(a) Example WS-BPEL instance.

```
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 (
       Next o (
         Reply[echo_client, echo_id, echo, x, echo_id, echo_id])))
```

(b) BPL Tool representation of example instance.



(c) BPL Tool visualization of example instance.

Figure 3: Example instance.

3.3 Reaction Rules

In this section we present the reaction rules used in the formalization of WS-BPEL. The reaction rules (in BPL Tool syntax) is also available via the on-line tool[3].

3.3.1 Structural Activities

The rules for structural activities covers completion of flows and sequences, conditionals (if-then-else) and iteration (while-loop).

[3]See `http://tiger.itu.dk:8080/bplweb/index/18`

Completion of Activities: When a Flow is completed (i.e. there are no more instructions in the flow to be executed) we garbage collect the flow, by replacing the `Flow` node with an empty bigraph (denoted by `<->`).

```
"flow completed" :::

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

In the same manner, we garbage collect a Sequence if the current instruction is completed (i.e. if there is no current instruction). We then make the following instruction the next to be executed by replacing the `Sequence` node with the content of the `Next` node.

```
"sequence completed" :::

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

Conditionals: 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. One of the two rules for evaluating an if-then-else statement was already given in Sec. 2 (rule `if false`), so we only present the rule for when the condition is true in this section. The rule is similar to the rule given in Sec. 2 except for the value of the condition and the instantiation.

```
"if true" :::

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

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 (emphasizing the order of the holes using Standard ML comments), the rule `while unfold` for unfolding looks as follows.

```
"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];
```

Note how the instantiation `[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 barren root. As explained above, the `Running` node linked to the `While` node via the name `inst_id` is used to guarantee that the instance which the while-loop is part of is indeed running.

3.3.2 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];
```

3.3.3 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. The remaining 3 rules are quite similar.

```
"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];
```

The instantiation 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 (or partner link). Since variables (or partner links) may be defined within nested scopes, several variables (or 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 `scope1` and `scope2` links connect respectively the `FromPLink` and `To` nodes to the closest partner link and variable with the correct name.

3.3.4 Scopes

The formalization of nested scopes makes crucial use of *bound* links. In WS-BPEL, local scopes may be defined within a while loop. As an example consider the while loop shown below.

```
<while>
  <condition>true()</condition>
  <scope>
    <variables><variable name="x" /></variables>
    <assign>
       <copy><from partnerlink="echo_client" /><to variable="x" /></copy>
    </assign>
  </scope>
</while>
```

Every iteration of the while loop, i.e. every unfolding of the loop, must create a *new* scope, containing a new copy of the variable. If we (naively) used a normal, i.e. free, link within the scope to connect the reference to x in the assignment, then the two copies created by the rule `while unfold` would share the *same* scope link. The consequence would be that the assignment rules would not tell the two scopes apart, and thus the variable from the wrong scope could be used in the assignment. For this reason we let the scope links be connected to a *binding* port of the `Scope` control. This ensures that each copy of the scope gets its own bound link. However, this introduces another problem: In the rule `assign copy plink2var` above we *must* place the assignment, partner link and variable controls below different roots, since they could all potentially be located in different scopes. However, interfaces of binding bigraphs do not allow a bound link to be shared between two nodes located below two different roots in the place graph.

To cope with this problem, we make the `Scope` control passive, and introduce a rule `scope activation` as 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 edge connected to that port. The rule is defined as follows in the BPL Tool syntax.[4]

[4]An alternative solution to this problem would be to use so-called local bigraphs, which exactly allow the more general interfaces where names can be bound to several roots. Alas, local bigraphs are not supported by the BPL Tool.

```
"scope activation" :::

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

Note how the use of single and double square brackets of the control `Scope` specify that `inst_id` is a normal port and `scope` is a binding port, whereas `ActiveScope[inst_id, scope]` has two normal ports. Note also that the link map `-//[scope]` closes the link connected to the scope port in the reactum, and that the name `scope` is local to the hole 0 in both redex and reactum. 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];
```

3.3.5 Process Termination

Processes can terminate in two different ways: 1) normally, i.e. when no more activities remain, or 2) abnormally by executing an `Exit` activity.

Normal Termination: In the first case, we simply remove the instance in the same way as for scopes. The precondition for the reaction rule is that there are no activities remaining in the instance, and we then replace the instance with an empty bigraph. As redex and the reactum are required to have the same outer face we add the "idle" link `proc_name` and `inst_id` using a wiring `proc_name//[] || inst_id//[]`.

```
"inst completed" :::

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

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[inst_id]
|| Running[inst_id]
  ----|>
   <->
|| Stopped[inst_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[proc_name, inst_id]
o (Stopped[inst_id] `|` `[]`)
  ----|>
<-> || proc_name//[] || inst_id//[];
```

(Again the "idle" links `proc_name//[] || inst_id//[]` in the reactum is simply there to ensure that the redex and reactum have the same outer face.)

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. This cannot be captured in the format of parametric rules of binding bigraphs.[5] 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.

3.3.6 Communication

The 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 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

[5]This however is possible using the higher-order reaction rules introduced in [23, 22].

between partner links and variables makes it possible to send partner links as messages and dynamically assign instances as target for communication.

The rule `invoke instance` below 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 instantiation 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, 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]
|| PartnerLink[partner_link_invoker, partner_link_scope_invoker]
   o (Link[inst_id_invoked] `|` `[]`)
|| Variable[outvar, outvar_scope] o `[]`
|| Running[inst_id_invoker]

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

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

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.

Note that the `PartnerLink` in the invoked process must be defined in the outermost scope. This is essentially the same issue as with `Exit` (cf. Sec. 3.3.5), namely that binding bigraph contexts cannot express arbitrary nesting depth: it is impossible to capture the whole process in the rule while also matching an arbitrarily nested partner link withing the process. With hindsight, we could probably have sidestepped this limitation by placing the `CreateInstance` nodes at a fixed location under the `Process` node, thus decoupling them from the partner links.

```
"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 status from a `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];
```

4 Motivating HomeBPEL

In this section we motivate the use of HomeBPEL with a simplified example of workflow management for pervasive health care. Each doctor is assumed to run a workflow process, which is initiated when he/she is hired. Every new treatment of a patient causes a new workflow process to be initiated, describing the clinical guideline to be followed for the particular treatment of the patient. In a centralized solution, this process would be running as a separate workflow on the workflow server and only be available when connected to the network. In HomeBPEL business processes can be manipulated as first class values, so we can let the doctor's workflow process execute the treatment process as a *sub-process.* By assuming that the doctor carries a mobile device running its own HomeBPEL engine the treatment process can be executed independently of the network. Moreover, if each patient is equipped with a mobile device running a self-treatment workflow process, the doctor may *delegate* the treatment process (or parts of it) by sending a sub-process to the patient's workflow process.

A sequence diagram illustrating a simple example of this scenario is shown in Fig. 4. The two large boxes represent the patient's and the doctor's PDA respectively. The dotted continuation of the "life-line" of the sub-process `guideline` indicates that it is the same process continuing its execution at the patient's PDA. The BPMN diagram in Fig. 5 gives a more detailed view of the patient process, with a group of guideline sub-processes indicated in the dashed box in the middle. Fig. 6 shows the corresponding HomeBPEL process for the patient. We have left out details related to the data-flow which

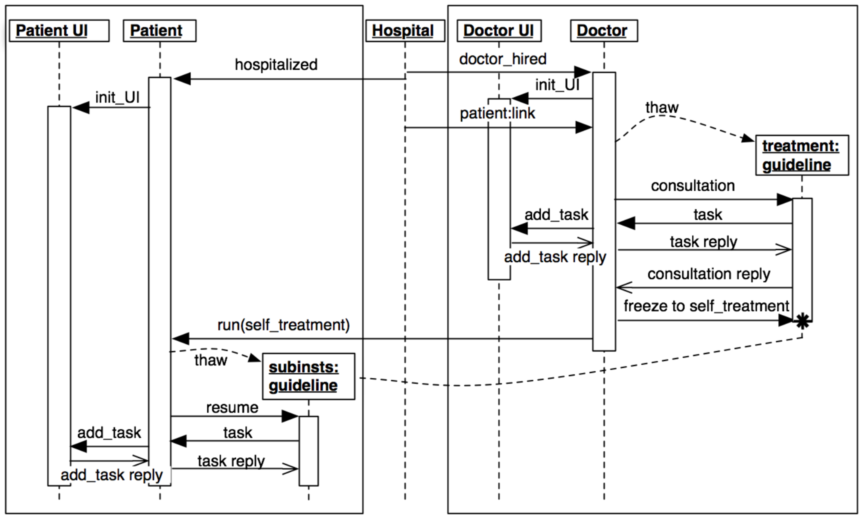


Figure 4: Sequence diagram for the pervasive health care scenario.

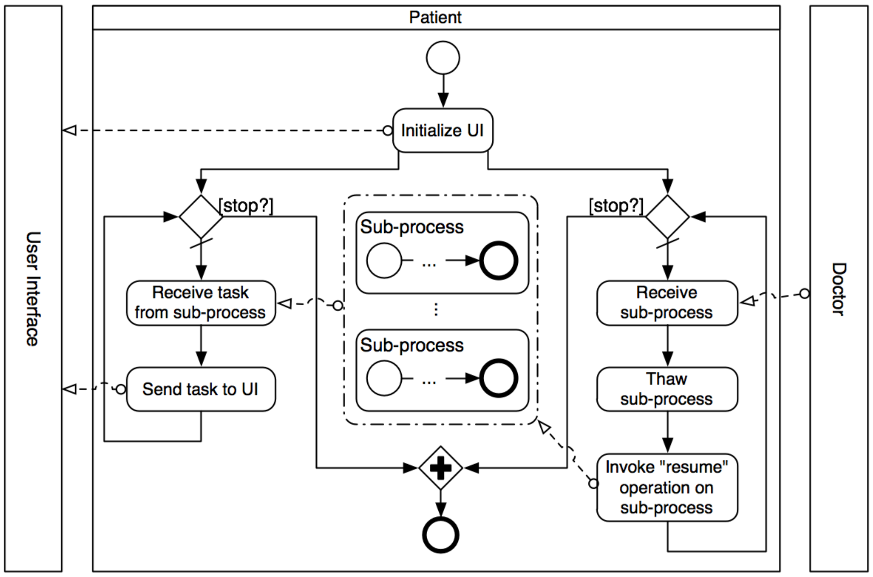


Figure 5: BPMN diagram of the patient workflow process.

```
<process name="patient">
  <partnerLinks>
    <partnerLink name="patient_client" />
    <partnerLink name="task_list_UI" />
  </partnerLinks>
  <subLinks>
    <subLink name="subinsts" />
  </subLinks>
  <variables>
    <variable name="guideline" />
    <variable name="task" />
    <variable name="reply" />
    ...
  </variables>
  <sequence>
    <receive partnerLink="patient_client" operation="hospitalized"
             createInstance="yes" ... /><reply operation="hospitalized" ... />
    <invoke  partnerLink="task_list_UI" operation="init_UI" ... />
    <flow>
      <!-- Thaw-loop: Continually receives and executes sub-instances -->
      <while>
        <condition>...</condition>
        <sequence>
          <receive   partnerLink="patient_client" operation="run"
                     variable="guideline" /><reply operation="run" ... />
          <thaw      subLink="subinsts" variable="guideline" />
          <invokeSub subLink="subinsts" operation="resume" ... />
        </sequence>
      </while>
      <!-- UI-loop: Continually receives tasks from sub-instances and
           pass them on to the UI service -->
      <while>
        <condition>...</condition>
        <sequence>
          <receiveSub subLink="subinsts" operation="task" variable="task" />
          <invoke     partnerLink="task_list_UI" operation="add_task"
                      inputVariable="task" outputVariable="reply" />
          <replySub   subLink="subinsts" operation="task" variable="reply" />
        </sequence>
      </while>
    </flow>
  </sequence>
</process>
```

Figure 6: Patient workflow process.

are not relevant for this example. The initial `receive` on the `hospitalized` operation is used to invoke the patient process, as indicated by the `createInstance` attribute. We have only formalized synchronous communication, so most receive operations are immediately followed by a ”dummy” reply. As also shown in the sequence and BPMN diagrams, the following invoke instantiates a local user interface process running on the patient's PDA which we assume takes care of handling the task list of the patient. It is followed by a WS-BPEL `flow`, which contains two while-loops executing in parallel. The first while-loop (corresponding to the right-hand loop in the BPMN diagram) allows for arbitrarily many self-treatment sub-processes to be received and instantiated: The `receive` on the `run` operation waits for a message containing a process and stores it in the input variable `guideline`. The following activity `thaw` is part of the new features introduced in HomeBPEL and it is used to create an instance of a process stored in a variable (in the example named `guideline`) and execute it as a sub-instance within the scope of the corresponding `subLink` (in the example named `subinsts`) of the current running instance. The second while-loop (corresponding to the left-hand loop in the BPMN diagram) forwards messages received from the guideline sub-processes by the HomeBPEL `receiveSub` activity to the user interface, and in turn forwards the answer back to the sub-process by the HomeBPEL `replySub` activity.

The doctor's workflow process shown in Fig. 7 also invokes a user interface process, and contains an identical loop for forwarding messages from treatment workflows to the user interface process (which we have omitted from the example code to save space). However, different from the patient workflow, the first step of the main loop of the doctor workflow is to receive a link (on the `patient` operation) which is then dynamically assigned to the `patient` partner link by the `copy` operation. Thereby the doctor workflow process can be dynamically linked to different patient workflow processes during its lifetime. The following `thaw` activity instantiates a treatment guideline as a sub-process from the variable named `guideline`. Fig. 8 shows an outline of the treatment process consisting of two phases: A *consultation phase* invoked explicitly by the doctor and carried out within the doctor's workflow, and a *self-treatment* phase carried out within the patient's workflow. To initiate the first part of the treatment, the operation `consultation` is invoked from the doctor workflow by the action `invokeSub`. The reply of this operation signals that the consultation is finished, and the treatment process is ready to be frozen (by the `freeze` action) and sent to the patient's workflow process.

Note that we have not specified the specific tasks for each phase in the treatment, which in general could be part of an arbitrarily complex workflow. However, we have illustrated how tasks in each phase can be scheduled at the user interface of the *current* super workflow by invoking the `task` operation by the `invokeSup` action. This shows how *context-dependent communication* is elegantly facilitated in HomeBPEL. One could easily imagine that the treatment processes could also access local information, e.g. special expertise of the doctor or relevant characteristics of the patient.

We claim that the use of higher-order processes in this example is much more flexible than a workflow simply based on a fixed *configuration* of a self-treatment process at the patient engine: Configuration is limited to a pre-defined set of parameters — in contrast to the treatment template that can be an arbitrary process which could be received by the doctor process from a central server of clinical guidelines.

The above example is of course still highly simplified. One would most likely want more control over the behavior of sub-processes, i.e. to disallow malicious processes from entering ones mobile device, to only allow processes from known, trusted sources, etc. It would also be relevant to allow reflection, combination and adaption of sub-processes on the fly. In the health care scenario, this could be used to avoid repeating a blood pressure measurement in two concurrent treatments, or more important, that the same pill is not commanded to be taken twice. We expect to address these questions in future work. A necessary first step is a formal semantics of the execution which will be provided in the following sections.

5 Formalizing HomeBPEL

In this section we present the formalization of HomeBPEL in the BPL Tool. Using basically the same approach as in Sec. 3 we first present the static representation in

```
<process name="doctor">
  <partnerLinks>
    <partnerLink name="hospital" />
    <partnerLink name="patient" />
    <partnerLink name="task_list_UI" />
  </partnerLinks>
  <subLinks><subLink name="treatment" /></subLinks>
  <variables>
    <variable name="guideline"><process name="guideline">...</process></variable>
    <variable name="link" /><variable name="self_treatment" /> ...
  </variables>
  <sequence>
    <receive partnerLink="hospital" operation="doctor_hired"
             createInstance="yes" ... /><reply operation="doctor_hired" ... />
    <invoke  partnerLink="task_list_UI" operation="init_UI" ... />
    <flow>
      <while>
        <condition>...</condition>
        <sequence>
          <receive   partnerLink="hospital" operation="patient"
                     variable="link" /><reply operation="patient" ... />
          <assign><copy>
              <from variable="link" /><to partnerLink="patient" />
          </copy></assign>
          <thaw      subLink="treatment" variable="guideline" />
          <invokeSub subLink="treatment" operation="consultation" ... />
          <freeze    subLink="treatment" variable="self_treatment" />
          <invoke    partnerLink="patient" operation="run"
                     inputVariable="self_treatment" ... />
        </sequence>
      </while>
      <!-- while-loop forwarding tasks to the local user interface -->
    </flow>
  </sequence>
</process>
```

Figure 7: Doctor workflow process.

Sec. 5.1 and then the reaction rules in Sec. 5.3, with a brief discourse on the semantics of sub-links in Sec. 5.2.

5.1 The Static Representation

The formalization of HomeBPEL as a binding bigraphical reactive system in the BPL Tool is given by a signature, determining the allowed controls and the ports for each type of control, and a set of reaction rules, determining the run-time semantics. As described previously, we utilize the extensibility of bigraphs to extend and adapt the previous formalization of WS-BPEL given in Sec. 3.

Table 5 shows the signature of HomeBPEL. The controls listed in the upper part of the signature correspond directly to elements in WS-BPEL, while the controls listed in the lower part are introduced to facilitate the formalization of the execution semantics. The underlined controls are the controls introduced (or adapted) in order to support higher order mobile embedded sub-processes.

Not all bigraphs of the given signature will correspond to valid processes and instances. The grammar in Table 6 shows the valid nesting of elements.[6] (For brevity we have

[6]This restriction can be represented in the theory of bigraphs using the notion of *sorting*.

```
<process name="guideline">
  ...
  <sequence>
    <!-- Doctor initializes treatment -->
    <receiveSup operation="consultation" ... />
    <!-- Instruct doctor on how to perform consultation -->
    <invokeSup  operation="task" ... />
    <replySup   operation="consultation" ... />
    <!-- Ready to be moved to patient -->
    <receiveSup operation="resume" ... /><replySup operation="resume" ... />
    <!-- Instruct patient how to perform self-treatment -->
    <invokeSup  operation="task" ... />
  </sequence>
</process>
```

Figure 8: Treatment guideline process.

Active controls	*Passive controls*	*Atomic controls*	
PartnerLinks	Process =: 1 --> 1	To -: 2	
Variables	Scope =: 1 --> 1	From -: 2	VariableRef -: 3
If -: 1	Variable -: 2	ToPLink -: 2	ReplyTo -: 2
Condition	While -: 1	FromPLink -: 2	Link -: 1
Sequence -: 1	Then	Invoke -: 8	CreateInstance -: 1
Flow -: 1	Else	Receive -: 6	SubTransition -: 1
	Assign -: 1	Reply -: 6	InvokeSub -: 8
	Copy	Exit -: 1	InvokeSup -: 6
	PartnerLink -: 2		ReceiveSub -: 6
			ReceiveSup -: 4
		True	ReplySub -: 6
		False	ReplySup -: 4
		GetReply -: 6	GetReplySub -: 7
Instance -: 3		Running -: 3	GetReplySup -: 4
ActiveScope -: 2		Invoked -: 3	Freeze -: 5
TopInstance	Next	Stopped -: 3	FreezingSub -: 5
Instances	Message -: 1	Freezing -: 3	Thaw -: 5
SubLinks	SubLink -: 2	TopRunning -: 1	FrozenSupLink -: 2

Table 5: Signature for HomeBPEL

abstracted away from the ports of nodes in the grammar. The arity of each control is provided in the signature, where e.g. `Process =: 1 --> 1` means that the `Process` control has one normal port and one binding port). We let i range over the set $\{0, 1\}$ which we use to index some of the productions to keep the presentation succinct. We write *prod*? for indicating that the terminal or non-terminal is optional and we write `Link`$*$ to denote that there can be 0 or more `Link` terminals. Currently the formalization only supports one type of expressions, namely variable references. But one can easily extend the semantics to more expression types (e.g. XPath expressions), simply by adding rules describing how to evaluate them — without having to alter the current rules. Similarly, values (i.e. *value*) are currently restricted to be either the constants `True` and `False`, processes (higher-order values), or the content of a `PartnerLink` (akin to name passing in the π-calculus). One could exploit the correspondence between XML and bigraphs to represent any kind of XML-data.

As mentioned in the introduction, the key idea of the formalization is that a process is represented by a bigraph very similar to the XML syntax for WS-BPEL processes. Also, an active *instance* is represented almost exactly as the process, except it has an

system	::=	*procs* ‘|‘ *state*
procs	::=	*proc* ‘|‘ ... ‘|‘ *proc*
state	::=	*topinst* ‘|‘ ... ‘|‘ *topinst*
proc	::=	`Process`($scopecontent_0$)
partnerlinks	::=	`PartnerLinks`(*partnerlink* ‘|‘ ... ‘|‘ *partnerlink*)
partnerlink	::=	`PartnerLink`(*partnerlinkcontent*)
partnerlinkcontent	::=	`CreateInstance`? ‘|‘ *link*?
link	::=	`Link` ‘|‘ *message*?
message	::=	`Message`(*value*)
sublinks	::=	`SubLinks(SubLink(Link*)` ‘|‘ ... ‘|‘ `SubLink(Link*))`
vars	::=	`Variables(Variable`(*value*) ‘|‘ ... ‘|‘ `Variable`(*value*))
topinst	::=	`TopInstance`(*inst* ‘|‘ *topinststatus*)
topinststatus	::=	`TopRunning` | `SubTransition`
insts	::=	`Instances`(*inst* ‘|‘ ... ‘|‘ *inst*)
inst	::=	`Instance`(*status* ‘|‘ $scopecontent_1$)
status	::=	`Invoked` | `Running` | `Freezing` | `Stopped`
act_i	::=	$scope_i$ | seq_i | $flow_i$ | $while_i$ | if_i | *assign* | `Invoke`
	|	`Receive` | `Reply` | `GetReply` | `Exit` | `InvokeSub` | `InvokeSup`
	|	`ReceiveSub` | `ReceiveSup` | `ReplySub` | `ReplySup` | `Thaw`
	|	`GetReplySub` | `GetReplySup` | `Freeze` | `FreezingSub`
$scope_0$	::=	`Scope`($scopecontent_0$)
$scope_1$	::=	`ActiveScope`($scopecontent_1$) | `Scope`($scopecontent_0$)
$scopecontent_i$	::=	*partnerlinks* ‘|‘ *sublinks* ‘|‘ *insts* ‘|‘ *vars* ‘|‘ act_i?
seq_i	::=	`Sequence`(act_i? ‘|‘ `Next`(act_i?))
$flow_i$	::=	`Flow`(act_i? ‘|‘ ... ‘|‘ act_i?)
$while_i$	::=	`While(Condition`(*expr*) ‘|‘ act_i?)
if_i	::=	`If(Condition`(*expr*) ‘|‘ `Then`(act_i?) ‘|‘ `Else`(act_i?))
assign	::=	`Assign(Copy`(*from* ‘|‘ *to*))
from	::=	`From` | `FromPLink`
to	::=	`To` | `ToPLink`
value	::=	`True` | `False` | *proc* | *partnerlinkcontent*
expr	::=	`True` | `False` | `VariableRef`

Table 6: Grammar for HomeBPEL

outermost node labeled by an `Instance` control. Instances keep the current content of variables inside the variable node, and are executed as in process calculi by rewriting the bigraph according to the set of reaction rules to be described in the following section.

As an example, the process `patient` from Sec. 4 is represented as a binding bigraph in the BPL Tool as shown in Fig. 9(a) – (b). (To shorten the example we have not fully specified the task loop. The full representation is available at the BPL Tool web page). Note that the compositionality of bigraphs allow us to separate the process into several parts. Fig. 10 shows the graphical representation provided by the BPL Tool of the `thaw_loop` bigraph in Fig. 9(a). To keep the figure clear we have abstracted away from the identity of the patient.

Looking at the graphical representation, it should be clear that the place graph corresponds closely to the nesting of elements in the XML syntax, the ports of controls correspond to attributes, and the link graph corresponds to shared values of attributes. However, already for the formalization of the subset of WS-BPEL given in Sec. 3 we needed to introduce some additional structure. For instance, a `Next` control is embedded in `Sequence` controls to cope with the fact that children nodes in bigraph place graphs are unordered while children nodes in XML are ordered (which is exploited in the sequence construct of WS-BPEL). To facilitate the definition of reaction rules in the semantics we needed to add links representing instance and scope identities. More intricately, we

```
val thaw_loop_body =
    Sequence[patient_id] o (
      Receive[patient_client, patient_id, run, x, patient_id, patient_id]
`|` Next o Sequence[patient_id] o (
      Thaw[subinsts, patient_id, x, patient_id, patient_id]
`|` Next o
      Reply[patient_client, patient_id, run, y, patient_id, patient_id]));

val thaw_loop =
While[patient_id] o (    Condition o VariableRef[y, patient_id, patient_id]
                    `|` thaw_loop_body);

val task_loop =
While[patient_id] o ( ...... );

val patient_body = Flow[patient_id] o (thaw_loop `|` task_loop);
```

(a) `patient_body`

```
val patient_process =
Process[patient][[patient_id]] o (
    PartnerLinks o (
        PartnerLink[patient_client, patient_id] o CreateInstance[start]
    `|` PartnerLink[task_list_UI, patient_id]  o <->)
`|` SubLinks o SubLink[subinsts, patient_id] o <->
`|` Variables o (
        Variable[x, patient_id] o <->
    `|` Variable[y, patient_id] o True)
`|` Instances o <->
`|` Sequence[patient_id] o (
      Receive[patient_client, patient_id, start, x, patient_id, patient_id]
`|` Next o Sequence[patient_id] o (
      Reply[patient_client, patient_id, start, y, patient_id, patient_id]
`|` Next o patient_body)));
```

(b) `patient_process`

Figure 9: BPL Tool representation of the patient process.

also needed to introduce a node within each instance with a *status* control being either `Invoked`, `Running`, or `Stopped`. Partly, this is needed because the semantics of `Invoke` and `Exit` activities requires two consecutive reactions. The extension with mobile sub-instances made it necessary to add an additional status control, `Freezing`, since freezing an instance into a process in a variable cannot be done atomically either. Also, we needed at top level to introduce a status control indicating if the top instance or any of its (arbitrarily nested) sub-instances are allowed to perform normal activities (by the control `TopRunning`) or if one of them are performing a sub-transition (control `SubTransition`) as part of a non-atomic activity. These aspects could most likely have been dealt with more elegantly if bigraphical reactive systems had a notion of priority on the reaction rules. We leave it for future work to study this.

5.2 Sub-links

In this section we take a closer look at the semantics of sub-links and the associated operations, using the patient process in Fig. 6 as example. Note how each iteration of the thaw loop thaws a new sub-instance which is bound to the sub-link `subinsts` and

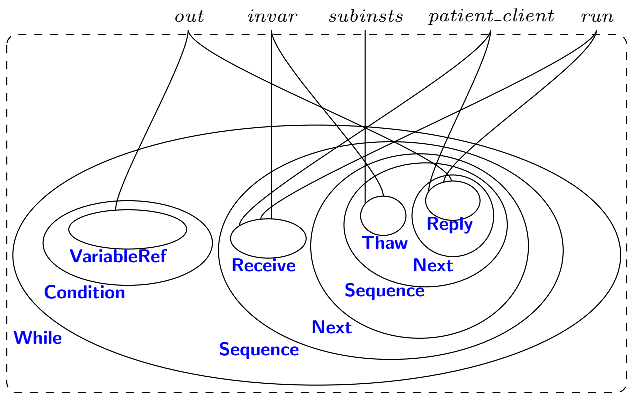


Figure 10: BPL Tool visualization of `thaw_loop`.

then invokes the operation `resume` on `subinsts`. What happens when a sub-instance is already bound to the `subinsts` sub-link? Is this an error or should it be allowed, and in that case which sub-instance(s) should be bound to the sub-link after the execution of `thaw`?

As one of the goals of the CosmoBiz project is to integrate process management into the process language itself, it seems reasonable to provide easy management of collections of processes at the language level. One simple way to achieve this feature, and the one we have chosen, is to allow multiple sub-instances to be bound to the same sub-link simultaneusly. This choice abstracts the implementation of collections of processes away from the programmer, building the concept of a process collection into the semantics of the activies which use sub-links: `thaw`, `freeze`, `invokeSub`, and `receiveSub`. The latter three activities are only intended to affect one sub-instance, but several of the sub-instances connected to the sub-link might be a suitable target for the activity. In this case, we choose an arbitrary process, based on the assumption that if the programmer wanted to distinguish two processes, she would place them in different collections. This might not hold true, and will be the subject of future work. One could imagine, for instance, that in the case of `freeze`, that the programmer wants to freeze a particular sub-instance or a sub-instance which is ready to be frozen.

5.3 Reaction Rules

In this section we present the reaction rules used in the formalization of HomeBPEL, focusing on the new rules for freezing and thawing and for communication between parent and child processes. The full set of reaction rules (in BPL Tool syntax) is available via the on-line tool[7].

[7]See `http://tiger.itu.dk:8080/bplweb/index/20`

5.3.1 Changes to the Representation

We have extended the representation in a smaller degree in order to facilitate the representation of higher-order primitives in the formalization. An `Instance` node now have an additional port which should be connected state node of the parent instance if a parent instance exists. We have introduced a node `Instances` to group together sub-instances of an instance, similar to the effect of the `Variables` node. As mentioned above we also introduce a status node in the top-level instance to track whether the top instance or any of its (arbitrarily nested) sub-instances are allowed to perform normal activities (by the control `TopRunning`) or if one of them are performing a sub-transition (control `SubTransition`) as part of a non-atomic activity. Also as mentioned above we add the status `Freezing`. As a technicality we also introduce nodes of control `TopInstance` to encapsulate top-level instances together with their associated top-level status node.

5.3.2 Augmenting the Existing Rules

Most of the reaction rules of the formalization remains unchanged from Sec. 3, except that the rules also need to make sure that the status of the top-level instance is `TopRunning`. However for the rule `scope completed`, the rule responsible for removing scopes that have been executed, we now also need to make sure that there are no running sub-instance inside the scope before removing the scope. The case is similar for the rule `inst completed` just for instances instead of scopes. We also need one additional rule to remove the new `TopInstance` nodes when removing a top-level instance. The added `TopInstance` node also add a bit to the complexity of the invoke rule.

```
"top instance completed" :::

TopInstance o (-//[inst_id_top] o TopRunning[inst_id_top])
  ----|>
<->;
```

We have added some new reaction rules to the formalization to implement the added primitives for higher-order processes. Below we present the new reaction rules.

5.3.3 Communication Between Parent and Child

The rule `invoke sub` takes care of an instance invoking a method in a subinstance. The parent instance performs the `InvokeSub` activity in parallel with the `ReceiveSup` of the subinstance. Both instances are required to be running as well as the top-level instance. The result is that the content from the variable `invar` is copied to variable `var`. Besides these changes the rule resembles the rule `invoke instance` (described below) which is responsible for communication between two top-level instances.

```
"invoke sub" :::

   InvokeSub[sub_link, sub_link_scope, oper, invar, invar_scope,
             outvar, outvar_scope, inst_id_sup]
|| SubLink[sub_link, sub_link_scope] o (Link[inst_id_sub] `|` `[]`)
|| Variable[invar, invar_scope] o `[]`
|| Running[inst_id_sup, active_scopes_sup, inst_id_top]
|| ReceiveSup[oper, var, var_scope, inst_id_sub]
|| Variable[var, var_scope]
|| Running[inst_id_sub, active_scopes_sub, inst_id_top]
|| TopRunning[inst_id_top]

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

   GetReplySub[sub_link, sub_link_scope, inst_id_sub, oper,
               outvar, outvar_scope, inst_id_sup]
|| SubLink[sub_link, sub_link_scope] o (Link[inst_id_sub] `|` `[]`)
|| Variable[invar, invar_scope] o `[]`
|| Running[inst_id_sup, active_scopes_sup, inst_id_top]
|| <->
|| Variable[var, var_scope] o `[]`
|| Running[inst_id_sub, active_scopes_sub, inst_id_top]
|| TopRunning[inst_id_top];
```

In the rule `reply sup` the `ReplySup` activity inside an instance can synchronize together with a `GetReplySub` activity inside the parent instance, thereby copying the content from variable `var` to variable `outvar`.

```
"reply sup" :::

   ReplySup[oper, var, var_scope, inst_id_sub]
|| Variable[var, var_scope] o `[]`
|| Running[inst_id_sub, active_scopes_sub, inst_id_top]
|| GetReplySub[sub_link, sub_link_scope, inst_id_sub, oper,
               outvar, outvar_scope, inst_id_sup]
|| SubLink[sub_link, sub_link_scope] o (Link[inst_id_sub] `|` `[]`)
|| Variable[outvar, outvar_scope] o `[]`
|| Running[inst_id_sup, active_scopes_sup, inst_id_top]
|| TopRunning[inst_id_top]

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

   <-> || oper//[]
|| Variable[var, var_scope] o `[]`
|| Running[inst_id_sub, active_scopes_sub, inst_id_top]
|| <->
|| SubLink[sub_link, sub_link_scope] o (Link[inst_id_sub] `|` `[]`)
|| Variable[outvar, outvar_scope] o `[]`
|| Running[inst_id_sup, active_scopes_sup, inst_id_top]
|| TopRunning[inst_id_top];
```

Rule `invoke sup` is similar to the rule `invoke sub`, except that it is the subinstance which invokes a method in the parent.

```
"invoke sup" :::

   InvokeSup[oper, invar, invar_scope, outvar, outvar_scope, inst_id_sub]
|| Variable[invar, invar_scope] o `[]`
|| Running[inst_id_sub, active_scopes_sub, inst_id_top]
|| ReceiveSub[sub_link, sub_link_scope, oper, var, var_scope, inst_id_sup]
|| SubLink[sub_link, sub_link_scope] o (Link[inst_id_sub] `|` `[]`)
|| Variable[var, var_scope]
|| Running[inst_id_sup, active_scopes_sup, inst_id_top]
|| TopRunning[inst_id_top]

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

   GetReplySup[oper, outvar, outvar_scope, inst_id_sub]
|| Variable[invar, invar_scope] o `[]`
|| Running[inst_id_sub, active_scopes_sub, inst_id_top]
|| <->
|| SubLink[sub_link, sub_link_scope] o (Link[inst_id_sub] `|` `[]`)
|| Variable[var, var_scope] o `[]`
|| Running[inst_id_sup, active_scopes_sup, inst_id_top]
|| TopRunning[inst_id_top];
```

The rule `reply sub` is similar to the rule `reply sup`, except for the direction of the communication.

```
"reply sub" :::

   ReplySub[sub_link, sub_link_scope, oper, var, var_scope, inst_id_sup]
|| SubLink[sub_link, sub_link_scope] o (Link[inst_id_sub] `|` `[]`)
|| Variable[var, var_scope] o `[]`
|| Running[inst_id_sup, active_scopes_sup, inst_id_top]
|| GetReplySup[oper, outvar, outvar_scope, inst_id_sub]
|| Variable[outvar, outvar_scope] o `[]`
|| Running[inst_id_sub, active_scopes_sub, inst_id_top]
|| TopRunning[inst_id_top]

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

   <-> || oper//[]
|| SubLink[sub_link, sub_link_scope] o (Link[inst_id_sub] `|` `[]`)
|| Variable[var, var_scope] o `[]`
|| Running[inst_id_sup, active_scopes_sup, inst_id_top]
|| <->
|| Variable[outvar, outvar_scope] o `[]`
|| Running[inst_id_sub, active_scopes_sub, inst_id_top]
|| TopRunning[inst_id_top];
```

5.3.4 Freezing Processes

Freezing a sub-instance requires several transitions, initiated by a `Freeze` activity. The `Freeze` activity references a running subinstance through its `SubLinks` and changes the status of the instance from `Running` to `Freezing` (thus ensuring that the subinstance will not execute anymore), at the same time the `Freeze` activity is replaced by a `FreezingSub` activity, and the top-level status is changed from `TopRunning` to `SubTransition` to indicate that we have started a multistep reaction.

```
"freeze sub" :::

   Freeze[sub_link, sub_link_scope, var, var_scope, inst_id_sup]
|| (    SubLinks o (    SubLink[sub_link, sub_link_scope]
                        o (Link[inst_id_sub] `|` `[]`)
                    `|` `[]`)
    `|` Instances
        o (    Instance[sub_name, inst_id_sub, active_scopes_sup]
               o (Running[inst_id_sub, active_scopes_sub, inst_id_top] `|` `[]`)
           `|` `[]`))
|| Running[inst_id_sup, active_scopes_sup, inst_id_top]
|| TopRunning[inst_id_top]

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

   FreezingSub[sub_link, sub_link_scope, var, var_scope, inst_id_sup]
|| (    SubLinks o (    SubLink[sub_link, sub_link_scope]
                        o (Link[inst_id_sub] `|` `[]`)
                    `|` `[]`)
    `|` Instances
        o (    Instance[sub_name, inst_id_sub, active_scopes_sup]
               o (Freezing[inst_id_sub, active_scopes_sub, inst_id_top] `|` `[]`)
          `|` `[]`))
|| Running[inst_id_sup, active_scopes_sup, inst_id_top]
|| SubTransition[inst_id_top];
```

Inside a freezing subinstance an active scope can be frozen when all nested scopes and sub-instances have been frozen. This is ensured by requiring that the content of the scope does not refer to the active-scopes link of the enclosing sub-instance. We then change the `ActiveScope` to a `Scope` and bind the free edge denoted by "scope".

```
"freeze scope" :::

-//[active_scopes]
o (   Freezing[inst_id, active_scopes, inst_id_top]
   || -//[scope] o (ActiveScope[active_scopes, scope] o `[scope]`)
   || `[active_scopes]`)

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

-//[active_scopes]
o (   Freezing[inst_id, active_scopes, inst_id_top]
   || Scope[inst_id][[scope]] o `[scope]`
   || `[active_scopes]`);
```

Sub-instances of a sub-instance which is being frozen, are frozen by propagating the freezing state, which again allows its scopes and subinstances to be frozen. This is done by changing the status of the nested sub-instance from `Running` to `Freezing`.

```
"freeze sub instance" :::

   Freezing[inst_id, active_scopes, inst_id_top]
|| Instance[sub_name, inst_id_sub, active_scopes]
   o (    Running[inst_id_sub, active_scopes_sub, inst_id_top]
      `|` `[]`)

  --[0 |-> 0]--|>

   Freezing[inst_id, active_scopes, inst_id_top]
|| Instance[sub_name, inst_id_sub, active_scopes]
   o (    Freezing[inst_id_sub, active_scopes_sub, inst_id_top]
      `|` `[]`);
```

When all those are frozen, ie. the "active_scopes" link of the sub-sub-instance is only connected to the state node, the sub-sub-instance is frozen (remaining at the same location) and a `FrozenSupLink` is inserted in the frozen instance to remember which `SubLink` it was connected to.

```
"freeze sub instance2" :::

-//[inst_id_sub]
o (   Freezing[inst_id_sup, active_scopes, inst_id_top]
   || (    SubLinks o (    SubLink[sub_link, sub_link_scope]
                           o (Link[inst_id_sub] `|` `[]`)
                       `|` `[]`)
       `|` Instances
           o (    Instance[sub_name, inst_id_sub, active_scopes]
                  o (    -//[active_scopes_sub]
                         o Freezing[inst_id_sub, active_scopes_sub, inst_id_top]
                      `|` `[inst_id_sub]`)
               `|` `[]`)))
```

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

   Freezing[inst_id_sup, active_scopes, inst_id_top]
|| (    SubLinks o (SubLink[sub_link, sub_link_scope] o `[]` `|` `[]`)
    `|` Instances
        o (    Process[sub_name][[inst_id_sub]]
               o (    FrozenSupLink[sub_link, sub_link_scope]
                  `|` `[inst_id_sub]`)
           `|` `[]`));
```

When no more sub-instances and scopes are connected to the "active_scopes" link of the sub-instance being frozen, it can itself be frozen and placed into the proper variable denoted by `var`. To indicate that the multistep reaction is completed we change the top-level status from `SubTransition` and back to `TopRunning`.

```
"freeze complete" :::

-//[inst_id_sub]
o (   FreezingSub[sub_link, sub_link_scope, var, var_scope, inst_id_sup]
   || Variable[var, var_scope] o `[]`
   || SubLink[sub_link, sub_link_scope] o (Link[inst_id_sub] `|` `[]`)
   || Running[inst_id_sup, active_scopes_sup, inst_id_top]
   || SubTransition[inst_id_top]
   || Instance[sub_name, inst_id_sub, active_scopes_sup]
      o (    -//[active_scopes_sub]
             o Freezing[inst_id_sub, active_scopes_sub, inst_id_top]
         `|` `[inst_id_sub]`))

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

   <->
|| Variable[var, var_scope]
   o Process[sub_name][[inst_id_sub]] o `[inst_id_sub]`
|| SubLink[sub_link, sub_link_scope] o `[]`
|| Running[inst_id_sup, active_scopes_sup, inst_id_top]
|| TopRunning[inst_id_top]
|| <->;
```

5.3.5 Thawing Processes

Using the of new `Thaw` activity one can thaw a sub-process stored in a variable and instantiating it as a sub-instance. The `Thaw` activity in the redex refers via its third port to the process inside the variable `var`. In the reactum the `Thaw` activity has been removed (indicating it has been executed) and a new running sub-instance has been inserted within the `Instances` control. The last part (`4&[inst_id_sub] |--> 0&[sub_scope]]`) of the instantiation map on the arrow from the redex to the reactum ensures that the process body (contained in hole 0 in the redex) is copied and used as body of the new sub-instance (hole 4 in the reactum). It also ensures that the local bound link `sub_scope` of the process body is renamed to `inst_id_sub` in the new copy. Note also that we insert the status node `Running` in the new sub-instance. Finally, the rule also insert a `Link` control within the `SubLinks` control. The `Link` control points to the new sub-instance via its link `inst_id_sub`.

```
"thaw sub" :::

   Thaw[sub_link, sub_link_scope, var, var_scope, inst_id_sup]
```

```
|| Variable[var, var_scope]
   o Process[sub_name][[sub_scope]] o `[sub_scope]`
|| (    SubLinks o (SubLink[sub_link, sub_link_scope] o `[]` `|` `[]`)
    `|` Instances o `[]`)
|| Running[inst_id_sup, active_scopes_sup, inst_id_top]
|| TopRunning[inst_id_top]

  --[0 |-> 0, 1 |-> 1, 2 |-> 2, 3 |-> 3,
     4&[inst_id_sub] |--> 0&[sub_scope]]--|>

   <->
|| Variable[var, var_scope]
   o Process[sub_name][[sub_scope]] o `[sub_scope]`
|| -//[inst_id_sub]
   o (    SubLinks o (    SubLink[sub_link, sub_link_scope]
                          o (Link[inst_id_sub] `|` `[]`)
                      `|` `[]`)
      `|` Instances
          o (    `[]`
             `|` Instance[sub_name, inst_id_sub, active_scopes_sup]
                 o (    -//[active_scopes_sub]
                        o Running[inst_id_sub, active_scopes_sub, inst_id_top]
                    `|` `[inst_id_sub]`)))
|| Running[inst_id_sup, active_scopes_sup, inst_id_top]
|| TopRunning[inst_id_top];
```

In general, when we thaw a process it may itself contain frozen sub-instances frozen "in place", i.e. within the `Instances` control. An additional reaction rule (`thaw sub instance`) is thus included for thawing frozen sub-instances. The rule replaces the `Process` node with a `Instance` node and restores the `SubLinks` using the information represented by the `FrozenSupLink` node. Finally the rule sets the status of the instance to `Running`. Note that this rule is the inverse of rule `freeze sub instance2`.

```
"thaw sub instance" :::

   (    SubLinks o (SubLink[sub_link, sub_link_scope] o `[]` `|` `[]`)
    `|` Instances
        o (    Process[sub_name][[inst_id_sub]]
               o (    FrozenSupLink[sub_link, sub_link_scope]
                  `|` `[inst_id_sub]`)
           `|` `[]`))
|| Running[inst_id_sup, active_scopes_sup, inst_id_top]
|| TopRunning[inst_id_top]

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

-//[inst_id_sub]
o (   (    SubLinks o (    SubLink[sub_link, sub_link_scope]
                           o (Link[inst_id_sub] `|` `[]`)
                       `|` `[]`)
       `|` Instances
           o (    Instance[sub_name, inst_id_sub, active_scopes_sup]
                  o (    -//[active_scopes_sub]
                         o Running[inst_id_sub, active_scopes_sub, inst_id_top]
                     `|` `[inst_id_sub]`)
              `|` `[]`))
   || Running[inst_id_sup, active_scopes_sup, inst_id_top]
   || TopRunning[inst_id_top]);
```

6 Conclusion and Future Work

We have formalized a subset of WS-BPEL as a binding bigraphical reactive system. As in our previous work described in [22] we have utilized the close correspondence between bigraphs and XML to provide a formalization close to the original WS-BPEL syntax and yet stays within the existing format for binding bigraphs [26].

Several new non-trivial aspects of WS-BPEL have been formalized compared to [22], 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 [22]. 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 implemented and explored within the BPL Tool [3] developed in the Bigraphical Programming Languages project at the IT University of Copenhagen. The tool allows compositional definition of binding bigraphs and reaction rules, as well as graphical visualization and interactive simulation of the execution of binding bigraphical reactive systems based on the formal inference of rule matching described in [1, 2].

We have utilised the extensibility of bigraphical reactive systems to extend the formalization of WS-BPEL to a formalization of a higher-order WS-BPEL-like language called HomeBPEL. The extensibility of bigraphical reactive systems enables us to directly reuse most of the existing formalization. In HomeBPEL processes are first-class values that can be stored in variables, passed as messages, and activated as embedded sub-instances. We have formalized HomeBPEL in the BPL Tool. We have motivated HomeBPEL by an example of pervasive health care where treatment guidelines are dynamically deployed as sub processes that may be delegated dynamically to other workflow engines and in particular stay available for disconnected operation on mobile devices. The added features of HomeBPEL allow us — among other — to define business processes for business process management within HomeBPEL as opposed to relying on meta-level tools for deployment and process administration.

Future Work. It is important to stress that we do not in this paper claim to give a feature complete formalization of WS-BPEL, as e.g. provided in [31]. We leave as future work to compare our formalization to the work in [31] and to provide more complete semantics and simulation of WS-BPEL. To do this, it is likely to be helpful if the BPL Tool was extended with 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 [22].

A notion of *prioritized* reactions could be interesting to explore as an alternative to the explicit encoding of transitions and sub-transitions used in the present paper. It will also be interesting to investigate the use of the general theory of bisimulation congruences available for bigraphical reactive systems in the setting of WS-BPEL.

An interesting path for future research into HomeBPEL will be to examine different primitives for management and manipulation of processes. Currently, we can copy and discard processes (by copying and overwriting the content of variables) and we can —

to some extend — combine processes, but we are currently examining more expressive primitives, such as sub-process reflection and general manipulation, e.g. editing or joining of frozen sub-processes. This relates to the work on Higher-Order (Petri) Nets and applications to workflow studied in [25].

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 [28, 4, 9]. The addition of mobile embedded sub-instances also opens for a study of type systems that can guarantee safe process mobility and manipulation. We plan to explore this in the CosmoBiz research project. In particular, we plan to examine the approaches done in Boxed Ambients [17] and in the higher-order π-calculus [34] on the safe integration of higher-order mobility and sessions.

Another relevant direction of work is a detailed and complete study of the expressiveness of HomeBPEL in relation to workflow patterns (e.g. [37]). We will also study the language primitives and expressiveness in relation to process calculi for mobility such as Ambients, Seal and Homer. In particular, we expect to examine a notion of subjective mobility as in Safe Ambients [30] by introducing a co-freeze activity to be carried out by the sub-instance, allowing it to decide whether (and when) it can be frozen.

Acknowledgements. Many thanks to the anonymous referees for their suggestions and comments from which this paper has benefited greatly.

References

[1] Lars Birkedal, Troels Christoffer Damgaard, Arne John Glenstrup, and Robin Milner. Matching of bigraphs. In Arend Rensink, Reiko Heckel, and Barbara König, editors, *Proceedings of the Graph Transformation for Verification and Concurrency workshop (GT-VC'06)*, volume 175 of *Electronic Notes in Theoretical Computer Science*, pages 3–19. Elsevier, 2006.

[2] Lars Birkedal, Troels Christoffer Damgaard, Arne John Glenstrup, and Robin Milner. An inductive characterisation of matching in binding bigraphs. *to appear*, 2008.

[3] The Bigraphical Programming Languages Group. The BPL Tool. `http://www.itu.dk/research/pls/wiki/index.php/BPL_Tool`, 2007.

[4] Mario Bravetti and Gianluigi Zavattaro. Contract based multi-party service composition. In Farhad Arbab and Marjan Sirjani, editors, *Proceedings of the IPM International Symposium on Fundamentals of Software Engineering (FSEN'07)*, volume 4767 of *Lecture Notes in Computer Science*, pages 207–222. Springer Verlag, 2007.

[5] Mikkel Bundgaard, Arne John Glenstrup, Thomas Hildebrandt, and Espen Højsgaard. An extensible formalization of WS-BPEL in binding bigraphs. Draft, 2008.

[6] Mikkel Bundgaard, Arne John Glenstrup, Thomas Hildebrandt, Espen Højsgaard, and Henning Niss. Formalizing higher-order mobile embedded business processes

with binding bigraphs. In *Proceedings of the 10th international conference on Coordination Models and Languages (COORDINATION'08)*, Lecture Notes in Computer Science, pages 83–99. Springer Verlag, 2008.

[7] Mikkel Bundgaard and Thomas Hildebrandt. Bigraphical semantics of higher-order mobile embedded resources with local names. In Arend Rensink, Reiko Heckel, and Barbara König, editors, *Proceedings of the Graph Transformation for Verification and Concurrency workshop (GT-VC'05)*, volume 154 of *Electronic Notes in Theoretical Computer Science*, pages 7–29. Elsevier, 2006.

[8] Mikkel Bundgaard, Thomas Hildebrandt, and Jens Chr. Godskesen. Modelling the security of smart cards by hard and soft types for higher-order mobile embedded resources. In Daniele Gorla and Catuscia Palamidessi, editors, *Proceedings of the 5th International Workshop on Security Issues in Concurrency (SecCo'07)*, volume 194 of *Electronic Notes in Theoretical Computer Science*, pages 23–38. Elsevier, 2007.

[9] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In Rocco De Nicola, editor, *Proceedings of the 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer Verlag, 2007.

[10] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[11] Giuseppe Castagna, Jan Vitek, and Fracesco Zappa Nardelli. The Seal calculus. *Journal of Information and Computation*, 201(1):1–54, 2005.

[12] Troels Christoffer Damgaard and Lars Birkedal. Axiomatizing binding bigraphs. *Nordic Journal of Computing*, 13(1–2):58–77, 2006.

[13] Dirk Fahland. Complete Abstract Operational Semantics for the Web Service Business Process Execution Language. Technical Report 190, Humboldt-Universität zu Berlin, 2005.

[14] Dirk Fahland and Wolfgang Reisig. ASM-based semantics for BPEL: The negative Control Flow. In Danièle Beauquier, Egon Börger, and Anatol Slissenko, editors, *Proceedings of the 12th International Workshop on Abstract State Machines (ASM'05)*, pages 131–151. Paris XII, March 2005.

[15] Roozbeh Farahbod, Uwe Glässer, and Mona Vajihollahi. Specification and validation of the business process execution language for web services. In *Abstract State Machines 2004. Advances in Theory and Practice*, volume 3052 of *Lecture Notes in Computer Science*, pages 78–94. Springer Verlag, 2004.

[16] Roozbeh Farahbod, Uwe Glässer, and Mona Vajihollahi. An abstract machine architecture for web service based business process management. In Christoph Bussler and Armin Haller, editors, *Business Process Management Workshops*, volume 3812 of *Lecture Notes in Computer Science*, pages 144–157. Springer Verlag, 2006.

[17] Pablo Garralda, Adriana B. Compagnoni, and Mariangiola Dezani-Ciancaglini. BASS: Boxed ambients with safe sessions. In *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pages 61–72. ACM Press, 2006.

[18] Arne John Glenstrup, Troels Christoffer Damgaard, Lars Birkedal, and Espen Højsgaard. An implementation of bigraph matching. *submitted*, 2008.

[19] Jens Chr. Godskesen and Thomas Hildebrandt. Extending Howe's method to early bisimulations for typed mobile embedded resources with local names. In *Proceedings of the 25th Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS'05)*, volume 3821 of *Lecture Notes in Computer Science*, pages 140–151. Springer Verlag, 2005.

[20] Volker Gruhn and André Köhler. Effects of mobile business processes on the software process. In *Proceedings of the 5th International Workshop on Software Process Simulation and Modeling (ProSim'04)*, pages 228–231. IEEE Computer Society Press, 2004.

[21] Thomas Hildebrandt, Jens Chr. Godskesen, and Mikkel Bundgaard. Bisimulation congruences for Homer — a calculus of higher order mobile embedded resources. Technical Report TR-2004-52, IT University of Copenhagen, 2004.

[22] Thomas Hildebrandt, Henning Niss, and Martin Olsen. Formalising business process execution with bigraphs and Reactive XML. In Paolo Ciancarini and Herbert Wiklicky, editors, *Proceedings of the 8th international conference on Coordination Models and Languages (COORDINATION'06)*, volume 4038 of *Lecture Notes in Computer Science*, pages 113–129. Springer Verlag, 2006.

[23] Thomas Hildebrandt, Henning Niss, Martin Olsen, and Jacob W. Winther. Distributed Reactive XML. In Lubos Brim and Isabelle Linden, editors, *Proceedings of the 1st International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord'05)*, volume 150 of *Electronic Notes in Theoretical Computer Science*, pages 61–80, 2006.

[24] Thomas Hildebrandt (principal investigator). Computer supported mobile adaptive business processes (CosmoBiz) research project. Webpage, 2007. `http://www.cosmobiz.org/`.

[25] Kathrin Hoffmann and Till Mossakowski. Algebraic higher-order nets: Graphs and petri nets as tokens. In Martin Wirsing, Dirk Pattinson, and Rolf Hennicker, editors, *Proceedings of the 16th International Workshop on Recent Trends in Algebraic Development Techniques (WADT'02)*, volume 2755 of *Lecture Notes in Computer Science*, pages 253–267. Springer Verlag, 2003.

[26] Ole Høgh Jensen and Robin Milner. Bigraphs and mobile processes (revised). Technical Report UCAM-CL-TR-580, University of Cambridge – Computer Laboratory, 2004.

[27] Matthias Kloppmann, Dieter Koenig, Frank Leymann, Gerhard Pfau, Alan Rickayzen, Claus von Reigen, Patrick Schmidt, and Ivana Trickovic. WS-BPEL extension for sub-processes: BPEL-SPE. Technical report, IBM and SAP, 2005.

[28] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A WSDL-based type system for WS-BPEL. In Paolo Ciancarini and Herbert Wiklicky, editors, *Proceedings of the 8th international conference on Coordination Models and Languages (COORDINATION'06)*, volume 4038 of *Lecture Notes in Computer Science*, pages 145–163. Springer Verlag, 2006.

[29] James J. Leifer and Robin Milner. Transition systems, link graphs and Petri nets. *Journal of Mathematical Structures in Computer Science*, 16(6):989–1047, 2006.

[30] Francesca Levi and Davide Sangiorgi. Mobile safe ambients. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(1):1–69, 2003.

[31] Niels Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0. In Marlon Dumas and Reiko Heckel, editors, *Proceedings of the 4th International Workshop on Web Services and Formal Methods (WS-FM'07)*, volume 4937 of *Lecture Notes in Computer Science*, pages 77–91. Springer Verlag, 2007.

[32] Niels Lohmann, H.M.W. Verbeek, Chun Ouyang, Christian Stahl, and Wil M. P. van der Aalst. Comparing and evaluating Petri net semantics for BPEL. Computer Science Report 07/23, Eindhoven University of Technology, 2007.

[33] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Journal of Information and Computation*, 100:1–40 and 41–77, 1992.

[34] Dimitris Mostrous and Nobuko Yoshida. Two session typing systems for higher-order mobile processes. In Simona Ronchi and Della Rocca, editors, *Proceedings of the 8th International Conference on Typed Lambda Calculi and Applications (TLCA'07)*, volume 4583 of *Lecture Notes in Computer Science*, pages 321–335. Springer Verlag, 2007.

[35] OASIS WSBPEL Technical Committee. Web Services Business Process Execution Language, version 2.0, 2007. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf`.

[36] Frank Puhlmann and Mathias Weske. Using the pi-calculus for formalizing workflow patterns. In Wil M. P. van der Aalst, Boualem Benatallah, Fabio Casati, and Francisco Curbera, editors, *Proceedings of the 3rd International Conference on Business Process Management (BPM'05)*, volume 3649 of *Lecture Notes in Computer Science*, pages 153–168. Springer Verlag, 2005.

[37] Nick Russell, Arthur H.M. ter Hofstede, Will M.P. van der Aalst, and Nataliya Mulyar. Workflow control-flow patterns: A revised view. BPM Center Report BPM-06-22, BPMcenter.org, 2006.

[38] Christian Stahl. A Petri net semantics for BPEL. Informatik-Berichte 188, Humboldt-Universität zu Berlin, 2005.

[39] Christian Stefansen. A declarative framework for enterprise information systems. Master's thesis, Dept. of Computer Science, University of Copenhagen, 2005.

[40] Franck van Breugel and Maria Koshkina. Models and verification of BPEL. Draft., 2006.