Universität Stuttgart

Fakultät Informatik, Elektrotechnik und Informationstechnik

Generating WS-BPEL 2.0 Processes from a Grounded BPEL4Chor Choreography

Peter Reimann, Oliver Kopp,
Gero Decker, Frank Leymann

2008/07
(slightly updated on 2014-04-01)

Institut für Architektur von Anwendungssystemen

Universitätsstraße 38
70569 Stuttgart
Germany

CR: H.4.1, K.1

Contents

1. Introduction

BPEL4Chor [DKLW07, DKLW09] is a choreography extension for WS-BPEL (BPEL for short, [Org07]). BPEL4Chor consists of a participant topology, participant behavior descriptions, and participant groundings. The participant behavior descriptions are abstract BPEL processes and are not executable by themselves. To be portable across multiple enterprises, the participant behavior descriptions do not contain any WSDL specific information. This information is added at the participant grounding, which is specific to the "deployment" of the choreography. Using the information of the participant grounding, each participant behavior description can be mapped to an abstract BPEL process following the "abstract process profile for observable behavior". Such a process can then serve as basis for an "executable completion" [Org07], where additional constructs are added to enable the process to be executed on a workflow engine. Figure 1 summarizes the necessary steps to get from a BPEL4Chor choreography description to executable BPEL processes. This report deals with the third step and describes how to automatically generate abstract BPEL processes containing the WSDL information of the grounding.

The main issues of the transformation are:

- multiple port types for a service when creating partner links (cf. Section 2),
- declaring partner links in the right scopes (cf. Section 2),
- mapping transmitted references to variables or partner links (cf. Section 5.1),
- declaring newly created message variables in the right scopes (cf. Section 5.1).

To generate BPEL Abstract Processes from a BPEL4Chor choreography, we convert the single participant behavior descriptions following the *Abstract Process Profile for Participant Behavior Descriptions* into BPEL processes following the *Abstract Process Profile for Observable Behavior*. Since the *Abstract Process Profile for Participant Behavior Descriptions* forbids the usage of *partnerLink*, *portType*, and *operation* attributes at the BPEL constructs used for communication, these attributes and the declarations

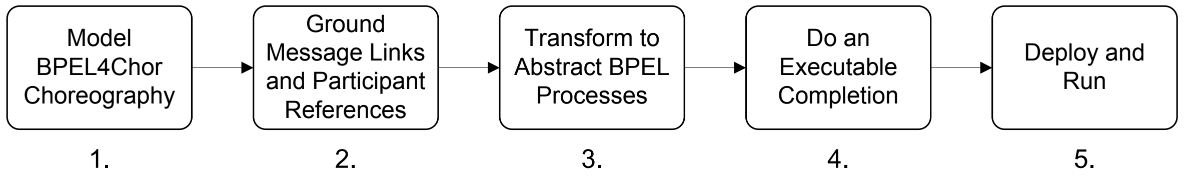
Model BPEL4Chor Choreography
Ground Message Links and Participant References
Transform to Abstract BPEL Processes
Do an Executable Completion
Deploy and Run
1.
2.
3.
4.
5.

Figure 1.: From a BPEL4Chor choreography to executable BPEL processes

of partner links have to be generated during the transformation. This is described in Section 2.

In correlation sets the NCNames of the correlation properties have to be replaced by the QNames of the corresponding references to WSDL properties. This is described in Section 3.

In BPEL4Chor a <forEach> activity may iterate over a set of participant references. Since pure BPEL does not support of the concept of participant references, such a <forEach> has to be converted into a <forEach> activity iterating over a variable holding an endpoint reference. This is described in Section 4.

Section 5 presents how the WSDL definitions of partner link types are generated automatically. They need to be generated since we create completely new partner link declarations and partner link types (cf. Section 2). As we do not have any information about the data types of the correlation properties which are referenced in the participant groundings, we cannot generate their WSDL definitions automatically. They need to be defined before starting the automated transformation. The definitions of port types and operations, WSDL messages, properties and property aliases for endpoint references and property aliases for correlation properties might be generated automatically, too. This is out of scope for this work. We assume that these definitions have been generated before starting the automated transformation.

Link passing mobility is accomplished in BPEL4Chor by forwarding participant references over message links. In BPEL processes we need to pass on endpoint references. The transmitted endpoint reference needs to be copied to the *partnerRole* of the partner link between the receiver of the reference and the transmitted participant reference if they want to communicate. So, we have to add an <assign> activity at the receiver side. We do not integrate this part of the transformation into the automated transformation. Therefore, we assume that in our BPEL4Chor choreography no participant references are transmitted over message links. Section 5.1 presents conceptionally how the conversion of transmitted participant references to endpoint preferences can be realized. The integration into the automated transformation is subject to future work.

Finally, Section 5 presents how certain WSDL definitions may be generated or extended automatically. These definitions are the partner link types, port types and operations. The definitions of partner link types will be created newly. If the definitions of port types and operations have been generated before starting the automated transformation, they will be extended by definitions which are not included in them, but which are referenced in the participant groundings of the BPEL4Chor description. Otherwise, they will be created newly, too. As we do not have any information about the data types or definitions of property aliases of the correlation properties or endpoint references which are referenced in the participant groundings, we cannot generate their WSDL definitions automatically. They need to be defined before starting the automated transformation.

The underlying procedure of the transformation is subdivided into four steps as presented below. As input it needs a completely grounded BPEL4Chor choreography (a participant topology, a PBD for each defined participant type and complete participant groundings) and WSDL definitions of port types and operations as well as WSDL messages. Additionally, the input includes WSDL definitions of properties and property aliases for correlation sets and endpoint references, if used. As output it produces one BPEL process following the *Abstract Process Profile for Observable Behavior* for each PBD. Furthermore, it produces WSDL definitions of partner link types. Figure 2 presents an overview of what is needed as input for the procedure and what is created as output.

Four steps from BPEL4Chor to BPEL Abstract Processes:

1. Analyze the participant topology and store all relevant data needed to execute the transformation. The data are amongst others the set of newly created partner link declarations and their mapping to message constructs (BPEL constructs used for communication) or the mapping of WSDL port types and operations to message constructs.

2. Analyze the participant groundings and extend the stored data.

3. Convert the single PBDs to BPEL Abstract Processes following the *Abstract Process Profile for Observable Behavior* as stated above (e. g. add declarations of partner links). The missing information can be derived from the data stored in steps 1 and 2.

4. Use the data stored in steps 1 and 2 to create WSDL definitions of partner link types.

To describe the data stored in steps 1 and 2, we use definitions of mathematical sets and functions and relations on this sets. Some of these definitions have been taken or derived from [KML08]. This technical report describes an abstract syntax of a WS-BPEL 2.0 process. It is developed by the Institute of Architecture of Application Systems at the University of Stuttgart. A summary of all definitions are summarized in Appendix A.

In the following, we use the notion “participant reference” for both a single participant reference and a set of participant references. Furthermore, we mean by saying that a participant reference sends a message to another participant reference, that the participant bound to the sending participant reference sends a message to the participant bound to the receiving participant reference.

We assume that there is exactly one name space prefix for each name space and vice versa in the participant topology and the participant groundings. It is clear that there is exactly one name space for each name space prefix, but it may happen that several prefixes refer to the same name space. If this occurs, one of the prefixes has to be chosen and the others have to be replaced by this one. Furthermore, we assume that there is

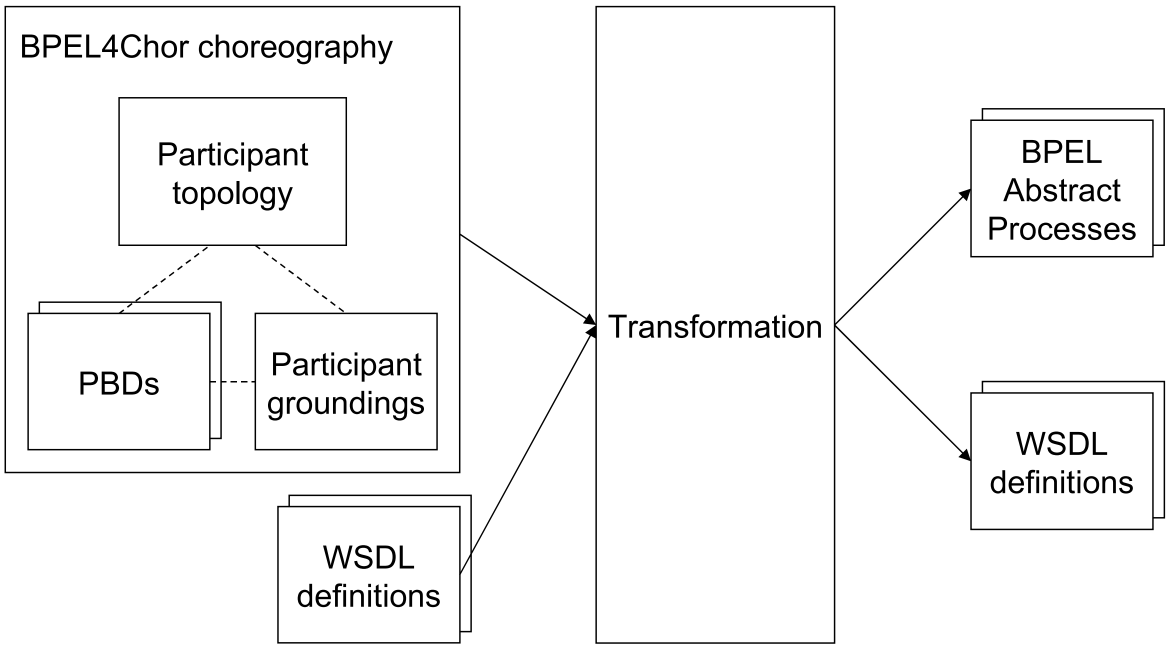

Figure 2.: Input and output of the transformation

no name space prefix which is defined in the participant topology or the participant groundings and which is named by *topologoyNS*. Otherwise, it has to be renamed. This prefix is used as a designated prefix referencing to the target name space of the participant topology.

In the course of the transformation, we will add new name space declarations to the PBDs. We assume that there are no conflicts with existing name space declarations and the newly created ones. A conflict might e. g. be that that an existing name space declaration defines the same name space prefix as a newly created one, but they refer to different name spaces. Then, the existing declaration is overwritten. If any conflicts occur, a renaming of name space prefixes has to be done. This is out of scope for this work.

Furthermore, we assume that there are no naming conflicts with existing variable declarations and newly created ones. Otherwise, a renaming of variables has to be done. Again, this is out of scope for this work.

2. Partner Links, Port Types, and Operations

The following definitions will be used to generate partner link declarations and add *partnerLink*, *portType*, and *operation* attributes to the BPEL constructs used for communication. Definitions 1 to 16 represent the data which is stored in step 1 of our procedure and definitions 17 to 31 the data of step 2.

Partner link types and partner link declarations are created while analyzing the message link declarations of the participant topology and the participant groundings. For each new combination of communicating participant references we will create exactly one partner link type if each reference is not realized by multiple port types. Otherwise, we need more than one partner link type for the combination. Let n be the number of port types by which the first participant reference(s) of the combination is realized, and let m be the number of port types for the second participant reference(s). Then, we need exactly $max(n, m)$ partner link types for the combination of participant references. For each partner link type we need to create two partner link declarations, one for the first participant reference(s) of the combination and one for the second participant reference(s).

A participant reference may be limited to a scope of any PBD by associating it with this scope in its <participant> or <participantSet> declaration by a *scope* attribute. In BPEL4Chor it is possible to associate a participant reference to a <forEach> activity of any PBD. If the reference is a set of participant references, the <forEach> activity will iterate over this set. If it is a single participant reference, it will represent the current participant in each of the branches of the <forEach> activity. The latter is limited to the scope nested in the <forEach> activity. The consequence of limiting a participant reference to a scope is that other participant references may send messages to this reference or receive messages from it only within the corresponding scope. It also means that every time the scope is entered, the participant reference may be bound to a different concrete participant. A partner link may be limited to one scope if the participant reference with which another participant reference communicates over it is limited to this scope, too. In such a case, we need to declare the partner link within the corresponding scope. This guarantees that every time the scope is entered, a new instance of the partner link is created.

Assume the code snippet of a message link declaration presented in listing 2.1. This message link specifies several senders. The actual sender is bound to the participant reference `z`. If the partner link declaration at receiver side needs to be declared within a scope, we will commit the modeler to limit the participant reference `z` to this scope. This goes for each message link having a *senders* attribute assigned. It makes our procedure easier when associating a partner link declaration with a scope. Otherwise, the partner link declaration needs to be associated with a set of scopes, if the participant references `s1`, `s2` and `s3` are limited to different scopes. Then, the partner link needs to be declared within the right of these scopes. This is out of scope for this work. Since a set of participant references might only be specified as sender of a message link by using a *senders* attribute, we simply ignore the fact that it may be limited to a scope by a *scope* attribute in its <participantSet> declaration. So, only single participant references are associated with a scope during our procedure. For the automated transformation we furthermore assume that a participant reference is limited to at most one scope.

```
<messageLink name="severalSenders"
  senders="s1 s2 s3")
  receiver="r"
  bindSenderTo="z"
  messageName="severalSenders"
  ...
/>
```

Listing 2.1: Message link specifying several senders

There is an alternative way to create partner link types and partner link declarations. In that way, they are created for each pair of communicating participant types instead of combinations of participant references. A short discussion why we must not use this alternative way is placed at Section 2.4.

2.1. Analyzing the Participant Topology

Definition 1 (set of name spaces).

The set *NS* is defined as the set containing all name spaces which are used in the participant topology and the participant groundings of the BPEL4Chor choreography and which are referenced by a name space prefix including the target name space of the participant topology.

Definition 2 (set of name space prefixes).

The set *NSPrefix* is defined as the set containing all name space prefixes which are used in the participant topology and the participant groundings of the BPEL4Chor choreography and which refer to a name space contained in NS including a designated element $topologoyNS$.

The designated element $topologoyNS \in NSPrefix$ refers to the target name space of the participant topology.

Definition 3 (the function assigning a name space to its name space prefix).

The function $prefix_{NS} : NSPrefix \rightarrow NS$ is defined as the function that assigns a name space to the corresponding name space prefix. The target name space of the participant topology is assigned to the designated element $topologoyNS$.

BPEL constructs may be associated with the same *wsu:id*. If they are declared in the same name space, they will be associated with the same partner link, port type and operation (in the case of message constructs), or the same partner link declarations will be added to them (in the case of <scope> or <forEach> activities). In the case of a <forEach> activity, these partner link declarations are declared within the <scope> activitiy nested directly in the <forEach> activity. If the constructs are declared in different name spaces, they might be associated with different partner links, port types or operations, or they might get different partner link declarations added. Therefore, we need the sets *NS* and *NSPrefix* and the function $prefix_{NS}$. They will be used later to identify scopes and <forEach> activities which have a *wsu:id* attribute assigned and message constructs within their processes and to add the proper *partnerLink*, *portType* and *operation* attributes or partner link declarations to them (cf. Section 2.3). The target name space of the participant topology will be used to create a new name space for the WSDL definitions of partner link types (cf. Section 5). The name spaces and name space prefixes which are used in the participant groundings will be used to add name space declarations for port types and operations to the PBDs (cf. Section 2.3).

The identifiers of the sets *NS* and *NSPrefix* are the names of the corresponding name spaces or name space prefixes. Their elements and the function $prefix_{NS}$ can be derived from the declarations of name space references in the <topology> tag of the participant topology (in step 1), from the <grounding> tag of the participant groundings (in step 2) and from the *targetNamespace* attribute of the participant topology (in step 1).

Since we assume that there is exactly one name space prefix for each name space and vice versa in the participant topology and the participant groundings, the function $prefix_{NS}$ is bijective.

Example 3.1 (name spaces and name space prefixes of the participant topology). In our participant topology we have the following name space references and target name space.

```
targetNamespace="example"
xmlns:buyer="example:buyer"
xmlns:seller="example:seller"
```

We get the set $\mathit{NS} = \{\mathit{example}, \mathit{example} : \mathit{buyer}, \mathit{example} : \mathit{seller}\}$ and the set $\mathit{NSPrefix} = \{\mathit{topologoyNS}, \mathit{buyer}, \mathit{seller}\}$ and the function $\mathit{prefix}_{\mathit{NS}}$ with $\mathit{prefix}_{\mathit{NS}}(\mathit{topologoyNS}) = \mathit{example}$, $\mathit{prefix}_{\mathit{NS}}(\mathit{buyer}) = \mathit{example} : \mathit{buyer}$ and $\mathit{prefix}_{\mathit{NS}}(\mathit{seller}) = \mathit{example} : \mathit{seller}$.

Definition 4 (set of participant types).

The set PaType is defined as the set containing all participant types of the BPEL4Chor choreography.

Definition 5 (set of processes).

The set $\mathit{Process}$ is defined as the set containing all BPEL processes (PBDs) of the BPEL4Chor choreography.

Definition 6 (the function assigning a PBD to each participant type).

The function $\mathit{process}_{\mathit{PaType}} : \mathit{PaType} \rightarrow \mathit{Process}$ is defined as the function that assigns a PBD to each participant type.

The function $\mathit{process}_{\mathit{PaType}}$ is bijective since there is exactly one PBD for each participant type and vice versa.

Definition 7 (the function assigning a name space prefix to each PBD).

The function $\mathit{nsprefix}_{\mathit{Process}} : \mathit{Process} \rightarrow \mathit{NSPrefix}$ is defined as the function that assigns a name space prefix to each PBD of which the target name space is associated with this name space prefix.

The sets and functions defined in definitions 4 to 7 can be derived from the <participantTypes> declaration of the participant topology. The identifier of an element of *PaType* is the NCName of the *name* attribute of the appropriate <participantType> declaration. There has to be exactly one element of *PaType* for each <participantType> declaration.

The value of the *participantBehaviorDescription* attribute of each <participantType> declaration is a QName consisting of two colon-separated NCNames. The first NCName is the name space prefix which refers to the target name space of the process of the corresponding participant type. The second NCName is the local name of that process. As identifier of an element of *Process* we use this QName. So, we have named an element of *Process* uniquely since every BPEL process needs to be named uniquely within its name space. There has to be exactly one element of *Process* for each PBD. The function $\mathit{nsprefix}_{\mathit{Process}}$ is set to the first NCName of the identifier of the corresponding element of *Process*.

Let $patype \in PaType$ be a participant type. Then, $process_{PaType}(patype) \in Process$ represents the PBD referenced by the *participantBehaviorDescription* attribute of the <participantType> declaration which is represented by *patype*.

Example 3.2 (participant types and processes). The following code snippet of a participant topology defines two participant types and associates each of them with one PBD. The sets and functions of definition 1 to 3 are the same as in example 3.1.

```
<participantTypes>
  <participantType name="Buyer"
    participantBehaviorDescription="buyer:buyer" />
  <participantType name="Seller"
    participantBehaviorDescription="seller:seller" />
</participantTypes>
```

We get the sets $PaType = \{Buyer,\ Seller\}$ and $Process = \{buyer : buyer,\ seller : seller\}$. Furthermore, we get $process_{PaType}(Buyer) = buyer : buyer$, $process_{PaType}(Seller) = seller : seller$, $nsprefix_{Process}(buyer : buyer) = buyer$ and $nsprefix_{Process}(seller : seller) = seller$.

Definition 8 (set of participant references).

The set *Pa* is defined as the set containing all participant references of the BPEL4Chor choreography. This set interprets sets of participant references and single participant references equally. There is no differentiation between them.

Definition 9 (the function assigning a participant type to each participant reference).

The function $type_{Pa} : Pa \rightarrow PaType$ is defined as the function that assigns a participant type to each participant reference which is of this type.

Definition 10 (set of scopes).

The set *Scope* is defined as the set containing the scopes and <forEach> activities of all PBDs having a *wsu:id* attribute assigned and that are referenced by one or more single participant references. If different <scope> or <forEach> activities bear the same *wsu:id* and are declared in the same name space, they will be represented by the same element of the set *Scope*.

The following function will be used later to assign a partner link to a scope in which it needs to be declared.

Definition 11 (the function indicating to which scope a participant reference is limited).

The function $\mathit{scope}_{Pa} : Pa \rightarrow \mathit{Scope} \cup \{\perp\}$ is defined as the function that assigns a scope to each participant reference which is limited to this scope. Let $pa \in Pa$ be a participant reference and let $sc \in \mathit{Scope} \cup \{\perp\}$ be a referenced scope or the element $\perp$. Then, $\mathit{scope}_{Pa}(pa)$ is set to sc if and only if one of the following three conditions holds:

1. pa is a single participant reference, and it is associated with sc by a *scope* attribute in its <participant> declaration.
2. pa is a single participant reference, and it is associated with sc by a *forEach* attribute in its <participant> declaration.
3. Neither condition one nor condition two holds, and we heve $sc = \perp$.

A single participant reference assigned to a *forEach* attribute is limited to the scope nested in the appropriate <forEach> activity. So, we need to handle a referenced <forEach> activity as like a referenced <scope> activity. This means that we need to store it in the set *Scope* and modify the function scope_{Pa} of the single participant reference to the <forEach> activity accordingly. Thus, the stored <forEach> activity can be interpreted as the <scope> activity nested in it.

If a participant reference $pa \in Pa$ is not limited to an inner scope of any process, other participant references will be able to communicate with pa within any scope of their processes. In such a case, we set $\mathit{scope}_{Pa}(pa) = \perp$, which is expressed by the third condition of definition 11.

The sets and functions defined in definitions 8 to 11 can be derived from the <participants> declaration of the participant topology. The identifier of an element of Pa is the NCName of the *name* attribute of the appropriate <participant> or <participantSet> declaration. The NCName of its *type* attribute identifies the element of PaType which is assigned to the participant reference by the function type_{Pa}.

In [KML08] $\mathcal{A}_{scope}$ is the set of <scope> activities of one process and $\mathit{process} \in \mathcal{A}_{scope}$ is the <process> activity. Let A^{p}_{scope} be the set $\mathcal{A}_{scope}$ of process $p \in \mathit{Process}$. Then, we have $\mathit{Scope} = (\cup_{p \in \mathit{Process}}(A^{p}_{scope} \setminus (\{\mathit{process}\} \cup \{sc \in A^{p}_{scope} \mid \text{no single participant reference is associated with } sc\}))) = (\cup_{p \in \mathit{Process}}\{sc \in A^{p}_{scope} \mid \text{at least one single participant reference is associated with } sc\})$, while a single participant reference can be associated with a scope $sc \in \mathit{Scope}$ either by a *scope* or a *forEach* attribute.

The identifier of an element $sc \in \mathit{Scope}$ is the QName of the appropriate *scope* or *forEach* attributes which are associated with one or more single participant references. A QName consists of two colon-separated NCNames. The first NCName is the name of the name space prefix which refers to the name space which is the target name space of the process in which the <scope> or <forEach> activity is included, and the second NCName is the value of the *wsu:id* attribute of the corresponding <scope> or <forEach> activity. In that way, each element of *Scope* is named uniquely. For each scope and for each <forEach> activity referenced in the <participants> declaration by a single participant reference there has to be exactly one element of *Scope*. The elements of *Scope* can be derived by traversing the <participants> declaration and adding each scope and each <forEach> activity that is referenced by a single participant reference to *Scope*. We do not need to check whether a newly added element of *Scope* has been added previously since there cannot be any duplicates of elements in a mathematical set.

The function $scope_{Pa}$ can be derived as follows. Let $pa \in Pa$ be the participant reference of the current <participant> or <participantSet> declaration while traversing the <participants> declaration. If *pa* is a single participant reference, and if it is associated with a scope $sc \in Scope$ by a *scope* or *forEach* attribute, $scope_{Pa}(pa)$ will be set to *sc*. Otherwise, it will be set to $\perp$.

Example 3.3 (participant references and scopes). The following code snippet of a participant topology defines two single participant references and one set of participant references and limits one of the single references to a scope. The sets and functions of definitions 1 to 7 are the same as in examples 3.1 to 3.2.

```
<participants>
  <participant name="buyerref" type="Buyer" selects="sellerref" />
  <participant name="sellerref" type="Seller"
    scope="seller:innerscope" />
  <participantSet name="sellers" type="Seller" />
</participants>
```

We get the set of participant references $Pa = \{buyerref, sellerref, sellers\}$ and the function $type_{Pa}$ with $type_{Pa}(buyerref) = Buyer$, $type_{Pa}(sellerref) = Seller$ and $type_{Pa}(sellers) = Seller$.

The participant reference $sellerref$ is associated with a scope named *innerscope*. The value of the corresponding *scope* attribute is the QName *seller:innerscope*. Thus, we get the set $Scope = \{seller : innerscope\}$. For the participant reference $sellerref$ we get $scope_{Pa}(sellerref) = seller : innerscope$. The participant references $buyerref$ and $sellers$ are not associated with a scope in their <participant> or <participantSet> declarations. So, we get $scope_{Pa}(buyerref) = scope_{Pa}(sellers) = \perp$.

Definition 12 (set of message constructs).

The set MC is defined as the set containing the message constructs (BPEL constructs used for communication) of all PBDs. If different message constructs bear the same *wsu:id* and are declared in the same name space, they will be represented by the same element of the set MC.

In [KML08] $\mathcal{CO}_{message}$ is the set of the message constructs of one process. Let $CO^{p}_{message}$ be the set $\mathcal{CO}_{message}$ of process $p \in Process$. Then, we have $MC = \cup_{p \in Process} CO^{p}_{message}$.

Definition 13 (set of message links).

The set ML is defined as the set containing all message links of the BPEL4Chor choreography.

Definition 14 (the function assigning a send and a receive activity to each message link).

The function $constructs_{ML} : ML \rightarrow MC \times MC$ is defined as the function that assigns two message constructs to each message link. Let $ml \in ML$ be a message link and $constructs_{ML}(ml) = (mc_1, mc_2)$. Then, mc_1 is the send activity and mc_2 the receive activity of the message link ml.

The function $constructs_{ML}$ is used to determine the message constructs of a message link while analyzing the corresponding <messageLink> declaration in the participant groundings. These message constructs then are associated with partner links, port types and operations (cf. Section 2.2). This makes it possible to add *partnerLink*, *portType* and *operation* attributes to the appropriate message constructs in the PBDs (cf. Section 2.3).

Definition 15 (the function assigning the sender(s) and the receiver to each message link).

The function $parefs_{ML} : ML \rightarrow 2^{Pa} \times Pa$ is defined as the function that assigns a subset of participant references (a subset of Pa) and a single participant reference to each message link. Let $ml \in ML$ be a message link and $parefs_{ML}(ml) = (A, p)$. Then, A contains all participant references which are specified as a potential sender of ml, and p is the participant reference which is specified as its receiver.

Definition 16 (the function assigning the actual sender to each message link).

The function $bindSenderTo_{ML} : ML \rightarrow Pa \cup \{\perp\}$ is defined as the function that assigns a participant reference to each message link of which the actual sender should be bound to this reference indicated by a *bindSenderTo* attribute. Let $ml \in ML$ be a message link. If ml has no *bindSenderTo* attribute assigned, $bindSenderTo_{ML}(ml)$ will be set to $\perp$. Otherwise, if ml has a *bindSenderTo* attribute assigned, and if the value of the attribute is the identifier of the participant reference $p \in Pa$, $bindSenderTo_{ML}(ml)$ will be set to p. Thus, the actual sender of ml is bound to the participant reference p.

The functions parefs_{ML} and $\mathit{bindSenderTo}_{ML}$ are used to determine the combination of communicating participant references of a message link while analyzing the corresponding <messageLink> declaration in the participant groundings. Thus, they help us to create partner link types and partner link declarations (cf. Section 2.2).

The sets and functions defined in definitions 12 to 16 can be derived from the <messageLinks> declaration of the participant topology by traversing all message links one by one. The identifier of an element of ML is the NCName of the *name* attribute of the corresponding <messageLink> declaration. There has to be exactly one element of ML for each <messageLink> declaration.

Each message link has exactly two message constructs referenced by its *sendActivity* and *receiveActivity* attributes. The values of these attributes are NCNames. Since every NCName is unique only within a name space, we cannot use them as identifier of the elements of MC. There might be several message constructs having the same NCName, but which are declared in different name spaces. Instead, we use a QName. Its first NCName is the name space prefix which refers to the target name space of the process which uses the message construct. The second one is the NCName of the *sendActivity* or *receiveActivity* attribute.

For each message link we add both the send and the receive activity to the set MC. The only message construct occurring in two message links instead of one is a synchronous <invoke> activity. It has to be the send activity in one of them and the receive activity in the other message link. But since there cannot be any duplicates of elements in a mathematical set, it will do no harm if we just add it twice to the set MC. In that way, it is not necessary to check whether a message construct is a synchronous <invoke> activity or not, which cannot be seen from the declarations of message links directly. The derivation of the sets and functions defined in definitions 12 to 16 is as follows.

Let $\mathit{ml} \in \mathit{ML}$ be the current message link, *receiver* the participant reference identified by the NCName of the *receiver* attribute, *receiveActivity* the NCName of the *receiveActivity* attribute and *sendActivity* the NCName of the *sendActivity* attribute of *ml*. Furthermore let sender_1, sender_2, ..., sender_n be the participant reference(s) of the sender(s). If only one sender is specified in *ml*, we will have $n = 1$. Then, sender_1 is the single participant reference identified by the NCName of the *sender* attribute. If a set of potentially senders is specified in *ml*, we will have $n \geq 1$. Then, sender_1 to sender_n are the participant references identified by the NCNames of the *senders* attribute, and n is the amount of these NCNames. If we have $n = 1$ additionally, sender_1 will be a set of participant references.

First of all, we build the identifiers for the message constructs of *ml* and add them to the set MC. In sender_{ns} and $\mathit{receiver}_{ns}$ we store the name space prefixes which refer to the target name spaces of the processes of the sender(s) and the receiver. Thus, we get $\mathit{receiver}_{ns} = \mathit{nsprefix}_{\mathit{Process}}(\mathit{process}_{\mathit{PaType}}(\mathit{type}_{\mathit{Pa}}(\mathit{receiver})))$. As each sender has to be of the same participant type and thus is referenced to the same element of *Process*, we can store the value $nsprefix_{Process}(process_{PaType}(type_{Pa}(sender_i)))$ for any $1 \le i \le n$ in $sender_{ns}$. No other participant reference needs to be called in for $sender_{ns}$. Since $sender_1$ always exists, we chose $i = 1$. The identifiers of both message constructs of *ml* can be build by building the QNames $sender_{ns} : sendActivity$ and $receiver_{ns} : receiveActivity$.

After having added the message constructs to MC, we set $constructs_{ML}(ml) = (mc_1, mc_2)$ with $mc_1 = sender_{ns} : sendActivity$ and $mc_2 = receiver_{ns} : receiveActivity$. Furthermore, we set $parefs_{ML}(ml) = (\{sender_1, sender_2, \dots, sender_n\}, receiver)$. The value of $bindSenderTo_{ML}(ml)$ will be set to $\perp$ if *ml* has no *bindSenderTo* attribute assigned. Otherwise, it will be set to the NCName of the *bindSenderTo* attribute, which is an identifier of a single participant reference.

Algorithm 1 states a procedure in pseudo code which executes the derivation of the sets and functions defined in definitions 12 to 16 for one message link. This procedure has to be called for each <messageLink> declaration. The procedures and functions of the algorithms of this chapter use data types and functions which are similar to the data types and functions of JDOM[1]. JDOM stores an xml-file as tree of Java objects. For algorithm 1 we assume the following:

- We assume `QName` to be a data type for QNames. `QName` inherits of the data type `String`.
- We assume `NCName` to be a data type for NCNames. `NCName` inherits of the data type `String`. There is the following function on elements of the type `NCName`
  - `buildQName( name :NCName ) returns QName`: this function builds a QName by concatenating the NCName with a colon and the NCName `name`.
- We assume `DT(S)` to be a data type for the elements of the mathematical set S. S may be any set of the definitions of this chapter. `DT(S)` inherits of the type of the identifiers of the elements of S (`NCName or QName`).
- We assume `Element` to be a data type for a node of a tree which represents an xml-file. So, an element of the type `Element` represents a tag within an xml-file. There are the following functions on elements of the type `Element`:
  - `getAttributeValue( name :String ) returns String`: this function returns the value of the attribute having the name `name`. It will return $\perp$ if the element has no attribute assigned having the name `name`.

[1]`http://www.jdom.org/`

- `getAttributeValueAsList( name :String ) returns List(String)`: this function returns the value of the attribute having the name `name` as list of NCNames or QNames. Each NCName or QName of the value of the attribute is one element of the list. The function will return an empty list if the element has no attribute assigned having the name `name`.
- `hasAttribute( name :String ) returns Boolean`: this function will return `true` if the element has an attribute assigned having the name `name`. Otherwise, it will return `false`.

These assumptions hold for each algorithm of this chapter, and they will be extended successively. A summary of all assumptions made for the algorithms of this chapter is presented in Appendix B.

Algorithm 1 Analysis of one <messageLink> declaration of the participant topology

```
procedure ANALYZEMESSAGELINK(messageLink :Element)
                // the input messageLink points on the current <messagLink> tag

    ml :DT($ML$);                                   // DT($ML$) inherits of NCName
    receiver, sender1 :DT($Pa$);                    // DT($Pa$) inherits of NCName
    senders :List(DT($Pa$));
    receiveActivity, sendActivity :NCName;
    receiverns, senderns :NCName;   // name space prefixes $receiver_{ns}$ and $sender_{ns}$
    mc1, mc2 :DT($MC$);                             // DT($MC$) inherits of QName
     // mc1 will be the send activity and mc2 the receive activity of the message link
begin
                                                    // add message link to $ML$
    ml ←messageLink.getAttributeValue("name");
    $ML$ ←$ML$ ∪ ml;
                          // get participant references of the sender and the receiver
    receiver ←messageLink.getAttributeValue("receiver");
    if messageLink.hasAttribute("sender") then      // only one sender is specified
        sender1 ←messageLink.getAttributeValue("sender");
    else                                            // more than one sender specified
                                        // get participant references of the senders
        senders ←messageLink.getAttributeValueAsList("senders");
        sender1 ←senders[1];                        // the first element of senders
    end if
        // build identifiers for the message constructs and add them to the set $MC$
    receiveActivity ←messageLink.getAttributeValue("receiveActivity");
    sendActivity ←messageLink.getAttributeValue("sendActivity");
    receiverns ←$nsprefix_{Process}(process_{PaType}(type_P($receiver$)))$;
    senderns ←$nsprefix_{Process}(process_{PaType}(type_P($sender1$)))$;
    mc2 ←receiverns.buildQName(receiveActivity);
    mc1 ←senderns.buildQName(sendActivity);
    $MC$ ←$MC$ ∪ mc2;
    $MC$ ←$MC$ ∪ mc1;
                                        // derive functions of definitions 14 to 16
    $constructs_{ML}$(ml) ←(mc1, mc2);
    $parefs_{ML}$(ml) ←(senders, receiver);   // senders is a set (list) of sender references
    $bindSenderTo_{ML}$(ml) ←messageLink.getAttributeValue("bindSenderTo");
            // getAttributeValue returns ⊥ if $BindSenderTo$ attribute does not exist
end procedure
```

Example 3.4 (message links and message constructs). The following code snippet of a participant topology defines three message links. The sets and functions of definitions 1 to 11 are the same as in examples 3.1 to 3.3.

```
<messageLinks>
  <messageLink name="ProductInformation"
    senders="sellers"
    sendActivity="SendPI"
    receiver="buyerref"
    receiveActivity="ReceivePI"
    bindSenderTo="sellerref"
    messageName="ProductInformation" />
  <messageLink name="PurchaseOrder"
    sender="buyerref"
    sendActivity="SendPO"
    receiver="sellerref"
    receiveActivity="ReceivePO"
    messageName="PurchaseOrder" />
  <messageLink name="POConfirmation"
    sender="sellerref"
    sendActivity="SendConf"
    receiver="buyerref"
    receiveActivity="ReceiveConf"
    messageName="POConfirmation" />
</messageLinks>
```

To derive the sets and functions of definition 12 to 16 we need to use the procedure `analyzeMessageLink` on each of the three <messageLink> declarations. In that way, we get the following:

- $ML = \{ProductInformation, PurchaseOrder, POConfirmation, \}$
- $MC = \{seller : SendPI, buyer : ReceivePI, buyer : SendPO, seller : ReceivePO, seller : SendConf, buyer : ReceiveConf\}$
- $constructs_{ML}(ProductInformation) = (seller : SendPI, buyer : ReceivePI)$
- $constructs_{ML}(PurchaseOrder) = (buyer : SendPO, seller : ReveivePO)$
- $constructs_{ML}(POConfirmation) = (seller : SendConf, buyer : ReveiveConf)$
- $parefs_{ML}(ProductInformation) = (\{sellers\}, buyerref)$
- $parefs_{ML}(PurchaseOrder) = (\{buyerref\}, sellerref)$

- $parefs_{ML}(POConfirmation) = (\{sellerref\}, buyerref)$
- $bindSenderTo_{ML}(ProductInformation) = sellerref$
- $bindSenderTo_{ML}(PurchaseOrder) = \perp$
- $bindSenderTo_{ML}(POConfirmation) = \perp$

2.2. Analyzing the Participant Groundings

Until now, we have seen how the data which is stored in step 1 of our procedure can be derived from the participant topology. Now, we describe the data of step 2. First of all, when analyzing the participant groundings, we extend the sets *NS* and *NSPrefix* and the function $prefix_{NS}$ by the name space references in the <grounding> tag.

Example 3.5 (name spaces and name space prefixes of the participant groundings). In our participant groundings we have the following name space references. The sets and functions of definitions 1 to 16 are the same as in examples 3.1 to 3.4

```
xmlns:buyerPTs="example:buyer:portTypes"
xmlns:sellerPTs="example:seller:portTypes"
```

We get the sets $NS = NS \cup \{example : buyer : portTypes, example : seller : portTypes\}$ and $NSPrefix = NSPrefix \cup \{buyerPTs, sellerPTs\}$, and the function $prefix_{NS}$ is extended by the assignments $prefix_{NS}(buyerPTs) = example : buyer : portTypes$ and $prefix_{NS}(sellerP$
$: seller : portTypes$.

The following definitions 17 to 31 represent the data which is stored in step 2 of our procedure while analyzing the participant groundings of the BPEL4Chor choreography.

Definition 17 (set of WSDL port types).

The set PT is defined as the set containing all WSDL port types which are referenced in the participant groundings of the BPEL4Chor choreography.

Definition 18 (the function assigning a name space prefix to each port type).

The function $nsprefix_{PT} : PT \to NSPrefix$ is defined as the function that assigns a name space prefix to each port type which is declared in the name space which is associated with this name space prefix.

Definition 19 (set of WSDL operations).

The set O is defined as the set containing all WSDL operations which are referenced in the participant groundings of the BPEL4Chor choreography.

Definition 20 (the function assigning a port type to each message construct).

The function $\mathit{portType}_{MC} : MC \to PT$ is defined as the function that assigns a WSDL port type to each message construct.

Definition 21 (the function assigning an operation to each message construct).

The function $\mathit{operation}_{MC} : MC \to O$ is defined as the function that assigns a WSDL operation to each message construct.

The functions of definitions 20 and 21 will be used later to add *portType* and *operation* attributes to the message constructs.

The sets and functions defined in definitions 17 to 21 can be derived from the <messageLinks> declaration of the participant groundings by traversing each <messageLink> declaration one by one. At each message link we add the port type and operation referenced by the *portType* and *operation* attributes to the sets PT and O. The identifier of an element of PT is the QName of the *portType* attribute. The first NCName of this QName is the name space prefix that refers to the name space of the WSDL definitions in which the port type is declared, and the second one is the local name of the port type. The function $\mathit{nsprefix}_{PT}$ is set to the first NCName of the identifier of the corresponding element of the set PT.

As identifier of an element of O we cannot use the NCName of the *operation* attribute since it is unique only within the name space of the corresponding WSDL definitions. Instead, we use a QName. Like in the case of an element of PT, the first NCName of this QName is the name space prefix that refers to the name space of the appropriate WSDL definitions. It can be derived from the function $\mathit{nsprefix}_{PT}$ and the port type referenced by the *portType* attribute. The second NCName is the one being the value of the *operation* attribute.

To derive the functions $\mathit{portType}_{MC}$ and $\mathit{operation}_{MC}$, we first need to retrieve the message constructs of the current message link. Let $ml \in ML$ be the current message link, which can be found by using the *name* attribute of the <messageLink> tag. Then, we get the message constructs by using the function $\mathit{constructs}_{ML}$. Let $(mc1, mc2)$ be $\mathit{constructs}_{ML}(ml)$, and let $pt \in PT$ and $o \in O$ be the port type and operation referenced in the current <messageLink> declaration. Then, we set $\mathit{portType}_{MC}(mc1) = \mathit{portType}_{MC}(mc2) = pt$ and $\mathit{operation}_{MC}(mc1) = \mathit{operation}_{MC}(mc2) = o$.

Example 3.6 (port types and operations). The following code snippet of a participant groundings presents three message links associated with WSDL port types and operations. The sets and functions of definitions 1 to 16 are the same is in examples 3.1 to 3.5.

```
<messageLinks>
  <messageLink name="ProductInformation"
    portType="buyerPTs:buyerPT"
    operation="getPI" />
  <messageLink name="PurchaseOrder"
    portType="sellerPTs:sellerPT"
    operation="getPO"’ />
  <messageLink name="POConfirmation"
    portType="buyerPTs:buyerPT"
    operation="getConf" />
</messageLinks>
```

We get the set $PT = \{sellerPTs : sellerPT, \; buyerPTs : buyerPT\}$, the function $\mathit{nsprefix}_{PT}$ with $\mathit{nsprefix}_{PT}(sellerPTs : sellerPT) = sellerPTs$ and $\mathit{nsprefix}_{PT}(buyerPTs : buyerPT) = buye$
Furthermore, we get the set $O = \{buyerPTs : getPI, \; sellerPTs : getPO, \; buyerPTs : getConf\}$. The message constructs of each message link can be derived from the function $\mathit{constructs}_{ML}$ (cf. example 3.4) and we get the functions $\mathit{portType}_{MC}$ and $\mathit{operation}_{MC}$ as follows:

- $\mathit{portType}_{MC}(SendPI) = \mathit{portType}_{MC}(ReceivePI) = buyerPTs : buyerPT$
- $\mathit{operation}_{MC}(SendPI) = \mathit{operation}_{MC}(ReceivePI) = buyerPTs : getPI$
- $\mathit{portType}_{MC}(SendPO) = \mathit{portType}_{MC}(ReceivePO) = sellerPTs : sellerPT$
- $\mathit{operation}_{MC}(SendPO) = \mathit{operation}_{MC}(ReceivePO) = sellerPTs : getPO$
- $\mathit{portType}_{MC}(SendConf) = \mathit{portType}_{MC}(ReceiveConf) = buyerPTs : buyerPT$
- $\mathit{operation}_{MC}(SendConf) = \mathit{operation}_{MC}(ReceiveConf) = buyerPTs : getConf$

Definition 22 (set of partner link declarations).

The set PL is defined as the set containing all partner link declarations which are created during our procedure.

The following function will be used later to add a *partnerLink* attribute to the message constructs

Definition 23 (the function assigning a partner link declaration to each message construct).

The function $partnerLink_{MC} : MC \rightarrow PL$ is defined as the function that assigns a partner link declaration to each message construct which uses the corresponding partner link to communicate with another message construct.

Definition 24 (the function limiting a set of partner link declarations to each scope or process).

The function $partnerLinks_{scope} : (Scope \cup Process) \rightarrow 2^{PL}$ is defined as the function that assigns a set of partner link declarations to each scope or process in which these partner link declarations will be enclosed. If the empty set is assigned to a scope or process, no partner links will be declared within it.

Let $pl \in PL$ be a partner link, $sc \in (Scope \cup Process)$ be a process or an inner scope of a process, and let pl be an element of $partnerLinks_{scope}(sc)$. If $sc \in Scope$, pl will be declared in the scope sc. Otherwise, if $sc \in Process$, pl will be declared in the outermost scope of the process sc, which is the process itself.

The function of definition 24 is used to limit a partner link declaration to a scope. If a partner link declaration needs to be limited to a scope or process $sc \in (Scope \cup Process)$, we will add it to the set $partnerLinks_{scope}(sc)$. In step 3 of our procedure the partner link declaration is declared within the scope or process sc (cf. Section 2.3). Initially, each scope or process needs to be assigned to the empty set by the function $partnerLinks_{scope}$. This has to be done in step 1 of our procedure while deriving the elements of the sets *Scope* and *Process*.

Definition 25 (set of partner link types).

The set *PLType* is defined as the set containing all partner link types which are created during our procedure.

Definition 26 (the function assigning a partner link type to each partner link declaration).

The function $type_{PL} : PL \rightarrow PLType$ is defined as the function that assigns a partner link type to each partner link declaration which is of this type.

The function of definition 26 will be used later to add *partnerLinkType* attributes to the <partnerLink> declarations.

Definition 27 (the relation *Comm*).

The relation $Comm \subseteq (2^{Pa} \times (PT \cup \{\perp\})) \times (Pa \times PT)$ is defined as the relation that assigns a subset of participant references and another participant reference to a pair of port types which they use to communicate. Let $((A, c), (b, d))$ be an element of the relation *Comm*. Then, the participant references contained in A communicate with the participant reference b while all participant references contained in A are realized by the port type c, and the participant reference b is realized by the port type d. If $c = \perp$, the communication will be one-way. That means the participant references contained in A send something to b, but not vice versa. If $c \neq \perp$, the communication will be request/response. That means the participant references contained in A send something to b and vice versa.

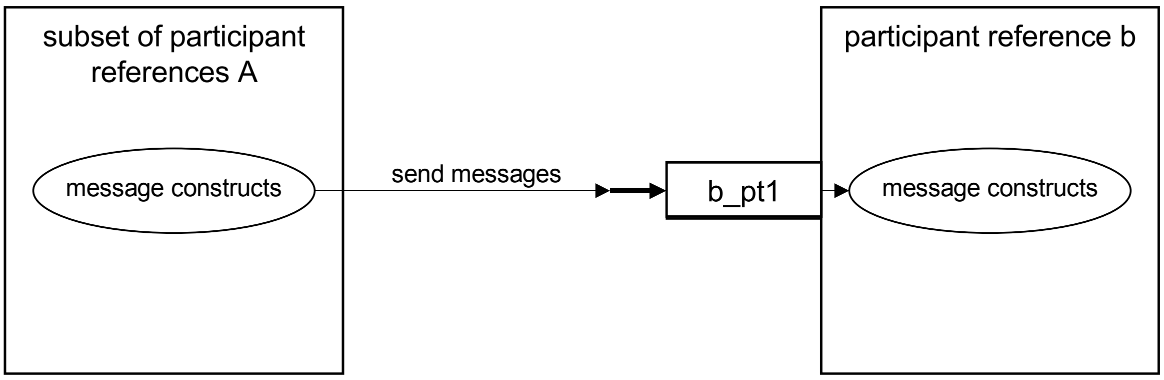

Figure 3.: One-way communication and one port type

In Figure 3 a subset of participant references $A \subseteq Pa$ sends one or more messages to a single participant reference b (maybe using several message links), and b is realized by the port type b_pt1 to receive these messages. Since b does not send any message to a participant reference contained in A, the latter does not need to be realized by any port type. In that case, we store the element $((A, \perp), (b, b\_pt1))$ in the relation *Comm*. Furthermore, we need to create one partner link type and two partner link declarations, one for the sender(s) contained in A and one for the receiver b. Each of the partner link declarations and the partner link type gets one role specified which is associated with the port type b_pt1. The partner link declaration of the participant references contained in A gets a *partnerRole* and the partner link declaration of b a *myRole* assigned.

Figure 4 looks similar to Figure 3, but b is realized by two port types b_pt1 and b_pt2 with $b\_pt1 \neq b\_pt2$ to receive the messages of the participant references contained in A. In that case, we store the two elements $((A, \perp), (b, b\_pt1))$ and $((A, \perp), (b, b\_pt2))$ in the relation *Comm*. We need to create one partner link type and two partner link declarations for each of the two elements since b is realized by two different port types. Like in the case of Figure 3, each of them gets one role specified.

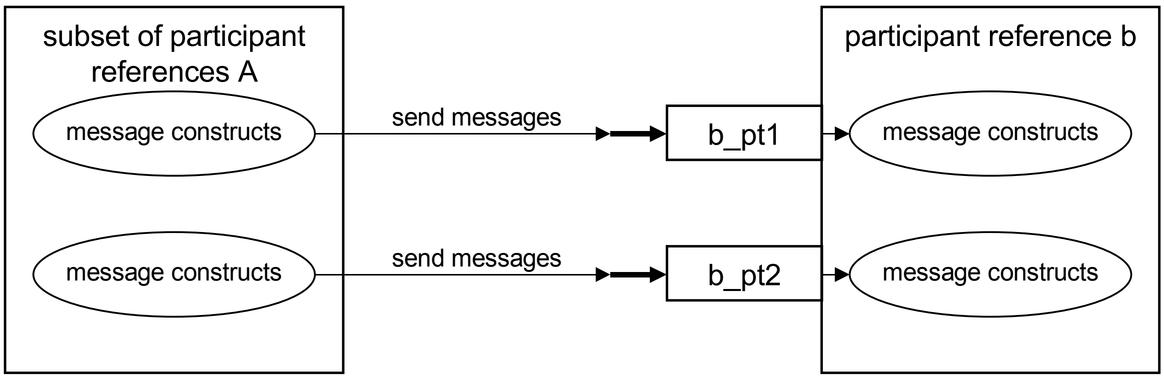

Figure 4.: One-way communication and two port types

If we continue the last two examples and add one or more port types by which the participant reference b is realized, we will get one element of *Comm*, one partner link type and two partner link declarations for each newly added port type.

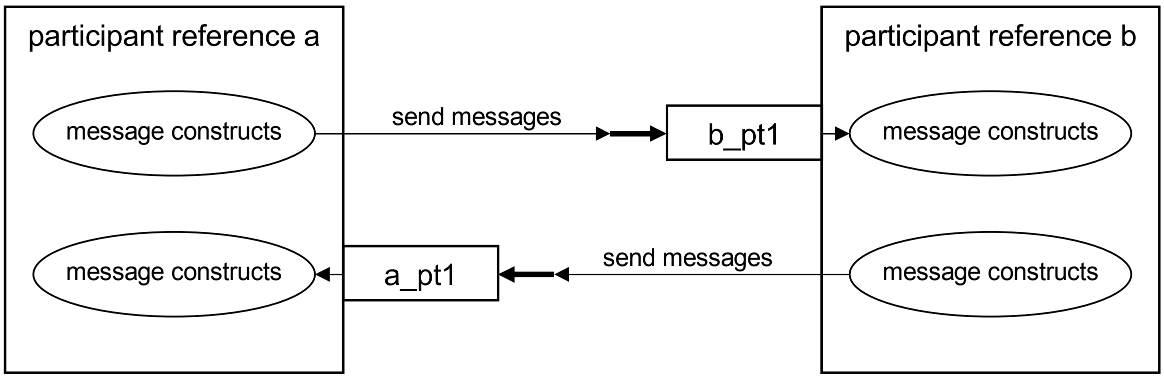

Figure 5.: Request/Response communication and two port types

In Figure 5 we see a request/response communication between the participant references a and b. The participant references a is realized by the port type a_pt1 and the participant reference b by the port type b_pt1 to receive the messages sent to each other. In that case, we store one element $((\{a\}, a\_pt1), (b, b\_pt1))$ in the relation *Comm*. Since a partner link type and its partner links can be associated with two port types, one for each partner, we can use the same partner link type and partner link declarations for each direction of communication. Thus, we need to create, like in the case of Figure 3, one partner link type and two partner link declarations. But this time each of them gets both roles specified. One role is associated with the port type a_pt1 and the other with the port type b_pt1.

Since the receiver of a message link in the participant topology of a BPEL4Chor choreography has to be a single participant reference, we do not have multiple participant references on any side of a request/response communication. If a message link specifies more than one sender, this can be realized by a *bindSenderTo* attribute associated with one single participant reference. The receiver of a message sent over the message link may reply to it by sending a message to this single participant reference. This is presented in listing 2.2.

```
<messageLink name"MessageLinkName"
  senders="a1 a2 ... an"
  sendActivity="send"
  receiver="b"
  receiveActivity="receive"
  bindSenderTo="a"?
  messageName="MessageLinkName" />
```

Listing 2.2: Message link having a *bindSenderTo* attribute assigned

In this message link we have a subset $A = \{a1, a2, ..., an\} \subseteq Pa$ of potentially senders with $n \geq 1$. The receiver is the participant reference b, and the actual sender is bound to the participant reference a. If b wants to reply to a message sent over this message link, it needs to send the reply to a. In such a case, we create partner link declarations between the participant references a and b. Each participant reference contained in A may use the partner link declaration of the participant reference a.

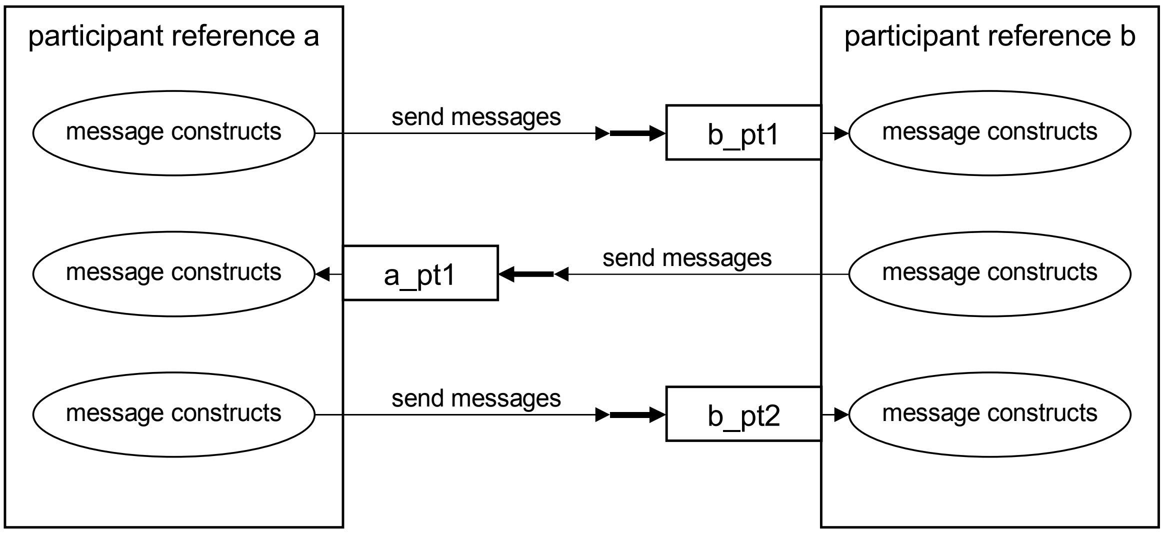

Figure 6.: Request/Response communication and three port types

Figure 6 presents the same communication like Figure 5, but the participant reference b is realized by two port types instead of one. The second one is b_pt2. In addition to the element $((\{a\}, a\_pt1), (b, b\_pt1))$, which we had in Figure 5, we store another element $((\{a\}, \perp), (b, b\_pt2))$ in the relation *Comm*. As this is a combination of the communication seen in Figures 5 and 3, we also need to create a combination of partner link types and partner link declarations of the communication of these figures. So, we get two partner link types, one for each newly added element of *Comm*, and four partner link declarations, two for each of the participant references involved. The partner link type and the partner link declarations for the element $((\{a\}, a\_pt1), (b, b\_pt1))$ get both roles specified, but the partner link type and partner link declaration for $((\{a\}, \perp), (b, b\_pt2))$ get only one role specified like in the case of Figure 3.

Figures 5 and 3 have presented two elementary kinds of communication between combinations of participant references and Figures 4 and 6 a combination of these kinds. We have seen that we get exactly one element of the relation *Comm* for each of the two elementary kinds of communication and that a combination of the kinds results in a combination of the elements of *Comm*. This can be carried on for any kind of communication between a combination of participant references. If we have another combination of communicating participant references, we will just get new elements of *Comm* for it. Furthermore, we have seen that for each element of *Comm* we need to create exactly one partner link type and two partner link declarations. The definitions 28 and 29 below are used to associate an element of *Comm* with its partner link type and partner link declarations.

To associate a partner link with the scope or process in which it needs to be declared, we can use the values of the function $scope_{Pa}$ of definition 11 of the participant references involved in the corresponding communication. Let $((A, A\_pt), (b, b\_pt))$ be an element of the relation *Comm*, let $pl_A \in PL$ be the partner link declaration for the subset of participant references A, and let $pl_b \in PL$ be the partner link declaration for the participant reference b. If the participant reference b is limited to an inner scope of the process of the sender(s), that is, we have $scope_{Pa}(b) \neq \perp$, the partner link declaration pl_A will be limited to this scope, too. In that way, we guarantee that every time the scope is entered, a new instance of the partner link is created. Then, the endpoint reference of the concrete participant bound to b can be copied to the partner role of this partner link instance without having any conflict with another endpoint reference which is copied to a partner role of any partner link. Otherwise if $scope_{Pa}(b) = \perp$, the participant reference b is not limited to an inner scope. In that case, pl_A is limited to the process of the participant type of the participant references contained in A.

If a message link has a *senders* attribute assigned and if the partner link declaration at receiver side needs to be declared within a scope, we will need to assign a *bindSenderTo* attribute to the appropriate message link declaration. The *bindSenderTo* attribute needs to be associated with a single participant reference which is limited to the corresponding scope (cf. beginning of Section 2). So, the set A is of the form $A = \{a\}$ with $a \in Pa$. If the message link has a *senders* attribute assigned, a will be the single participant reference associated with the *bindSenderTo* attribute. Otherwise, a will be the single participant reference associated with the *sender* attribute. Again, if we have $\mathit{scope}_{Pa}(a) \neq \bot$, pl_b will be limited to this scope. Otherwise, it will be limited to the process of the participant type of b.

Definition 28 (the function assigning a pair of partner link declarations to each element of *Comm*).

The function $\mathit{partnerLinks}_{\mathit{Comm}} : \mathit{Comm} \to \mathit{PL} \times \mathit{PL}$ is defined as the function that assigns a pair of partner link declarations to each element of the relation *Comm*. Let $\mathit{comm} = ((A,\, c),\, (b,\, d))$ be an element of the relation *Comm*, and let $(pl_1,\, pl_2) \in \mathit{PL} \times \mathit{PL}$ be $\mathit{partnerLinks}_{\mathit{Comm}}(\mathit{comm})$. Then, pl_1 is the partner link declaration used by the participant references contained in A and pl_2 the one used by the participant reference b.

Definition 29 (the function assigning a partner link type to each element of *Comm*).

The function $\mathit{plt}_{\mathit{Comm}} : \mathit{Comm} \to \mathit{PLType}$ is defined as the function that assigns a partner link type to each element of the relation *Comm*.

The function of definition 29 is bijective since there is exactly one element of *PLType* for each element of *Comm* and vice versa. We will use this function later to add roles and port types to the WSDL definitions of partner link types (cf. Section 5).

Definition 30 (the function assigning a *myRole* to each partner link declaration).

The function $\mathit{myRole}_{\mathit{PL}} : \mathit{PL} \to \mathit{Pa} \cup \{\bot\}$ is defined as the function that assigns a participant reference to a partner link declaration which may be interpreted as its *myRole*. If the value of this function is $\bot$ the corresponding partner link declaration will have no *myRole* specified.

Definition 31 (the function assigning a *partnerRole* to each partner link declaration).

The function $\mathit{partnerRole}_{\mathit{PL}} : \mathit{PL} \to \mathit{Pa} \cup \{\bot\}$ is defined as the function that assigns a participant reference to a partner link declaration which may be interpreted as its *partnerRole*. If the value of this function is $\bot$ the corresponding partner link declaration will have no *partnerRole* specified.

Let $comm = ((A,\ c),\ (b,\ d))$ be an element of the relation *Comm*, and let $(pl_1,\ pl_2) \in PL \times PL$ be $partnerLinks_{Comm}(comm)$. If $c = \bot$, the element *comm* will specify a one-way communication. In that case, we set $partnerRole_{PL}(pl1) = b$ and $myRole_{PL}(pl2) = b$. The values $myRole_{PL}(pl1)$ and $partnerRole_{PL}(pl2)$ are set to $\bot$ since the participant references contained in A are not realized by any port type. If $c \neq \bot$, the element *comm* will specify a request/response communication. In that case, we have $|A| = 1$. With $a \in Pa \wedge A = \{a\}$ we set $myRole_{PL}(pl1) = a$, $partnerRole_{PL}(pl1) = b$, $myRole_{PL}(pl2) = b$ and $partnerRole_{PL}(pl2) = a$.

The functions of definitions 30 and 31 will be used later to add *myRole* and *partnerRole* attributes to the partner link declarations. As name of the corresponding role we use the identifier of the participant reference associated with it.

The identifiers of the elements of *PL* and *PLType* are built as follows. Let $comm = ((A,\ c),\ (b, d))$with $A = \{a_1,\ a_2,\ ...,\ a_n\} \subseteq Pa$, $b \in Pa$, $c \in PT \cup \{\bot\}$ and $d \in PT$ be an element of the relation *Comm*. Furthermore, let $(pl_1,\ pl_2) \in PL \times PL$ be $partnerLinks_{Comm}(comm)$, and let $plt \in PLType$ be $plt_{Comm}(comm)$. As identifier of the partner link declaration pl_1 we use the character string "$a_1\_a_2\_...\_a_n - b\_isRealizedBy\_d'$". The last part d' of this character string is the identifier of the port type d, but the colon in it is replaced by an underline. The identifier of an element of *PL* is used as value of the *name* attribute assigned to the corresponding partner link declaration (cf. Section 2.3). This value has to be an NCName, but the identifier of a port type includes a colon. Thus, we need to eliminate this colon. The same goes for the identifier of an element of *PLType*. It is used as the second NCName of a QName. This QName is the value of the *partnerLinkType* attributes assigned to the corresponding partner link declarations. Furthermore, it is used as the value of the *name* attribute assigned to the corresponding <partnerLink-Type> declaration in the WSDL definitions of partner link types (cf. Section 5). The identifier of the partner link declaration pl_2 is "$b\_isRealizedBy\_d' - a_1\_a_2\_...\_a_n$". Furthermore, we chose "$a_1\_a_2\_...\_a_n - b\_isRealizedBy\_d' - plt$" as identifier of *plt*.

Since the port type c of the element *comm* might be changed while traversing the <messageLinks> declarations of the participant groundings (see below), we do not include its identifier in the identifiers of the elements of *PL* and *PLType*. Otherwise, we would also need to change those identifiers. This is not allowed since an element of a mathematical set has to be immutable. If it is not, it and the elements which are associated with it by a function may not to be found after having changed its identifier.

The sets, functions and the relation defined in definitions 22 to 31 can be derived while traversing the <messageLink> declarations of the participant groundings. Let $ml \in ML$ be the current message link, and let $(mc_1,\ mc_2) \in MC \times MC$ be $constructs_{ML}(ml)$. Thus, mc_1 is the send activity and mc_2 the receive activity of *ml*. Furthermore, let $pt \in PT$ be the port type referenced in the *portType* attribute of the current <messageLink> declaration, and let $(A,\ b) \in 2^{Pa} \times Pa$ be $parefs_{ML}(ml)$. This means that a participant reference contained in A sends a message to the participant reference b, and that b is realized by the port type *pt* to receive this message. If $bindSenderTo_{ML}(ml) \neq \bot$, we will set $A = \{\mathit{bindSenderTo}_{ML}(ml)\}$. Now, we need to traverse the relation *Comm* and search for elements of it which include the subset of participant references A and the participant reference b. In the process we distinguish between six different cases:

(1): $\exists\, comm \in Comm : (comm = ((A,\, c),\, (b, pt)) \wedge c \in PT \cup \{\bot\})$**:**

This means that we have already had a message link specifying the same direction of communication of ml, and in which the participant reference b has been realized by the same port type. So, we do not need to create new partner link declarations and a new partner link type. Instead, we can use the ones which have been created for the former message link. We just need to set the values of the function $\mathit{partnerLink}_{MC}$ of the message constructs mc_1 and mc_2 to the corresponding partner link declarations, which can be derived from the function $\mathit{partnerLinks}_{Comm}$. Let $(pl_1,\, pl_2) \in PL \times PL$ be $\mathit{partnerLinks}_{Comm}(comm)$. Then, we need to set $\mathit{partnerLink}_{MC}(mc_1) = pl_1$ and $\mathit{partnerLink}_{MC}(mc_2) = pl_2$ since by definition 28 pl_1 is the partner link declaration used by the participant references contained in A, and pl_2 is the one used by b.

(2): $\neg$**(1)** $\wedge\, A = \{a\} \wedge a \in Pa \wedge \exists\, comm \in Comm : (comm = ((\{b\},\, pt),\, (a, d)) \wedge d \in PT)$**:**

This means that we have already had at least two message links specifying a request/response communication between the participant references a and b, and in which b has been realized by the same port type as it is in the message link ml. As like in case **(1)**, we do not need to create new partner link declarations and a new partner link type. We just need to assign the message constructs mc_1 and mc_2 to the partner link declarations they will use. Let $(pl_1,\, pl_2) \in PL \times PL$ be $\mathit{partnerLinks}_{Comm}(comm)$. By definition 28 pl_1 is the partner link declaration used by the participant reference b, and pl_2 is the one used by the participant reference a. Since mc_1 is the message construct used by a and mc_2 the one used by b, we need to set $\mathit{partnerLink}_{MC}(mc_1) = pl_2$ and $\mathit{partnerLink}_{MC}(mc_2) = pl_1$.

(3): $\neg$(**(1)** $\vee$ **(2)**) $\wedge\, A = \{a\} \wedge a \in Pa \wedge \exists\, comm \in Comm : (comm = ((\{b\},\, \bot),\, (a, d)) \wedge d \in PT)$**:**

This means that we have already had a message link specifying a one-way communication in the opposite direction of ml, but no message link specifying the same direction of communication, and in which b has been realized by the same port type. In that case, we can use the same partner link declarations and partner link type which have been created for the former message link, and we need to change the one-way communication into a request/response communication. Therefore, we set the port type c of the element *comm* to pt. The values $\mathit{partnerLink}_{MC}(mc_1)$ and $\mathit{partnerLink}_{MC}(mc_2)$ can be set in the same way as in case **(2)**. So, we will get $\mathit{partnerLink}_{MC}(mc_1) = pl_2$ and $\mathit{partnerLink}_{MC}(mc_2) = pl_1$ if $(pl_1,\, pl_2) \in PL \times PL$ is $\mathit{partnerLinks}_{Comm}(comm)$.

Since the participant reference b has not been realized by a port type formerly, the values $myRole_{PL}(pl_1)$ and $partnerRole_{PL}(pl_2)$ have been set to $\perp$. Now, b is realized by the port type pt. So, we need to specify the two roles. Therefore, we set $myRole_{PL}(pl_1) = partnerRole_{PL}(pl_2) = b$.

(4): $\neg(\mathbf{(1)} \vee \mathbf{(2)} \vee \mathbf{(3)}) \wedge \exists\, comm \in Comm : (comm = ((A, c), (b, d)) \wedge c \in PT \cup \{\perp\} \wedge d \in PT \setminus \{pt\})$**:**

This means that we have already had a message link specifying the same direction of communication of ml, but the participant reference b has been realized by another port type. In addition, there has been no message link which would lead to the creation of an element of the relation $Comm$ for which one of the cases **(1)**, **(2)** or **(3)** holds. So, we need to create a new element of the relation $Comm$, new partner link declarations and a new partner link type and store them in the corresponding sets. The new element of the relation $Comm$ is $comm_{new} = ((A, \perp), (b, pt))$. Note that this is a one-way communication. Let pl_1 be the newly created partner link declaration used by the participant references contained in A, let pl_2 be the one used by the participant reference b, and let plt be the newly created partner link type. With $A = \{a_1, a_2, ..., a_n\}$ we have $pl_1 =$ "$a_1\_a_2\_...\_a_n - b\_isRealizedBy\_pt'$", $pl_2 =$ "$b\_isRealizedBy\_pt' - a_1\_a_2\_...\_a_n$" and $plt =$ "$a_1\_a_2\_...\_a_n - b\_isRealizedBy\_pt' - plt$". As mentioned above, pt' is the identifier of the port type pt, but the colon in it is replaced by an underline.

The values $partnerLink_{MC}(mc_1)$ and $partnerLink_{MC}(mc_2)$ can be set in the same way as in case **(1)**. So, we get $partnerLink_{MC}(mc_1) = pl_1$ and $partnerLink_{MC}(mc_2) = pl_2$ if $(pl_1, pl_2) \in PL \times PL$ is $partnerLinks_{Comm}(comm)$..

If $scope_{Pa}(b) = \perp$, it will be added to $partnerLinks_{scope}(process_{PaType}(type_{Pa}(a_1)))$. Note that $process_{PaType}(type_{Pa}(a_1))$ is the process of the participant type of the participant references contained in A. We can use the participant reference a_1 since all other participant references contained in A have to be of the same participant type. If $scope_{Pa}(b) \neq \perp$, the partner link declaration pl_1 will be added to $partnerLinks_{scope}(scope_{Pa}(b))$. This means that pl_1 is declared within the scope $scope_{Pa}(b)$.

If $bindSenderTo_{ML}(ml) = \perp \vee scope_{Pa}(bindSenderTo_{ML}(ml)) = \perp$, the sender will not be bound to a single participant reference, or the participant reference to which the sender will be bound is not limited to an inner scope of any process. In that case, we add the partner link declaration pl_2 to the process of the participant type of b. This is $partnerLinks_{scope}(process_{PaType}(type_{Pa}(b)))$. Otherwise, we will add it to the scope to which the single participant reference associated with the *bindSenderTo* attribute is limited, which is $partnerLinks_{scope}(scope_{Pa}(bindSenderTo_{ML}(ml)))$.

Furthermore, we set the following:

- $type_{PL}(pl_1) = plt$,

- $type_{PL}(pl_2) = plt$,
- $partnerLinks_{Comm}(comm_{new}) = (pl_1, pl_2)$,
- $plt_{Comm}(comm_{new}) = plt$,
- $myRole_{PL}(pl_1) = \bot$,
- $myRole_{PL}(pl_2) = b$,
- $partnerRole_{PL}(pl_1) = b$ and
- $partnerRole_{PL}(pl_2) = \bot$.

The values $myRole_{PL}(pl_1)$ and $partnerRole_{PL}(pl_2)$ are set to $\bot$ since the participant references contained in A are not realized by a port type in the communication represented by the element $comm_{new}$.

(5): $\neg(\mathbf{(1)} \vee \mathbf{(2)} \vee \mathbf{(3)} \vee \mathbf{(4)}) \wedge A = \{a\} \wedge a \in Pa \wedge \exists\, comm \in Comm : (comm = ((\{b\}, c), (a, d)) \wedge d \in PT \wedge c \in PT \setminus \{pt\})$**:**

This means that we have already had at least two message links specifying a request/response communication between the participant references a and b, and in which b has been realized by another port type than it is in the message link ml. Note that the condition $c \in PT \setminus \{pt\}$ implies $c \neq \bot$. In addition, there has been no message link specifying another kind of communication between the participant references a and b. As like in case **(4)**, we need to create a new element of the relation $Comm$, new partner link declarations and a new partner link type and store them in the corresponding sets. The new element of the relation $Comm$ is again $comm_{new} = ((A, \bot), (b, pt))$. Let pl_1 be the newly created partner link declaration used by the participant reference a, let pl_2 be the one used by the participant reference b, and let plt be the newly created partner link type. Then, we have $pl_1 =$ "$a - b\_isRealizedBy\_pt'$", $pl_2 =$ "$b\_isRealizedBy\_pt' - a$" and $plt =$ "$a - b\_isRealizedBy\_pt' - plt$". The character string pt' is the same as in case **(4)**. The remaining steps are also the same as like in case **(4)**.

(6): $\neg(\mathbf{(1)} \vee \mathbf{(2)} \vee \mathbf{(3)} \vee \mathbf{(4)} \vee \mathbf{(5)})$**:**

This means that there has been no message link specifying any kind of communication between the participant references contained in A and the participant reference b. In that case, we need to do exactly the same as like in cases **(4)** and **(5)**.

Note that a partner link declaration is associated with a scope or process by the function $partnerLinks_{scope}$ only while it is created. So, each partner link declaration will be limited to exactly one scope or process.

Algorithm 2 states a procedure in pseudo code which executes the derivation of the sets, functions and the relation defined in definitions 17 to 31 for one message link. This procedure has to be called for each <messageLink> declaration of the participant groundings. It uses the procedure `traverseComm` of algorithm 3, which, for it part, uses the procedure `createPartnerLinkDeclarations` of algorithm 4. In addition to the assumptions made for algorithm 1, we assume the following:

- There is the following function modifying elements of the data type `String`, which represents a data type of character strings:
  - `replaceColons( string :String ) returns String`: this function replaces each colon in the character string `string` by an underline.
- We assume `DT(`$Comm$`)` to be a data type for the elements of the mathematical relation *Comm*. There is the following function on elements of the type `DT(`$Comm$`)`:
  - `changeFirstPortType( pt : DT(`PT`))`: let $comm = ((A,\, c),\, (b,\, d))$ be an element of the relation *Comm*. If we use this function on the element *comm*, the port type c of it will change to pt.
- There is the following function on elements of the type `Element`:
  - `getAttributeNamespacePrefix( name :String ) returns NCName`: this function returns the name space prefix of the attribute having the name `name`, which will be the first NCName of its value if this is a QName. It will return $\perp$ if there is no attribute having the name `name` or if the attribute value does not include a name space prefix.

The function `repplaceColons` is used on the identifiers of port types when building the identifiers of partner link declarations and partner link types.

Example 3.7 (partner link declarations and partner link types). This example illustrates the derivation of the sets, functions and the relation defined in definitions 22 to 31. The message link declarations of the participant groundings relating to this example are the same as in example 3.6. The sets and functions of definitions 1 to 21 are the same as in examples 3.1 to 3.6.

To derive the sets, functions and the relation defined in definitions 22 to 31, we need to use the procedure `analyzeMessageLinkGrounding` on each of the three <messageLink> declarations. In the first message link $ProductInformation$ the participant reference *sellers* sends a message to the participant reference $buyerref$. Since $bindSenderTo_{ML}(ProductInformation) = sellerref$, we set $A = \{sellerref\}$. The send activity is $seller : SendPI$, and the receive activity is $buyer : ReceivePI$. When reaching this message link, we have not created an element of the relation *Comm* before. So, case **(6)** holds, and we need to create a new element of *Comm*, new partner link declarations and a new partner link type. Consequently, we get the following:

- $Comm = \{((\{sellerref\},\, \perp),\, (buyerref,\, buyerPTs : buyerPT))\}$

- $PL = \{sellerref{-}buyerref\_isRealizedBy\_buyerPTs : buyerPT,\ buyerref\_isRealizedB$ $\_buyerPTs : buyerPT - sellerref\}$
- $PLT = \{sellerref - buyerref\_isRealizedBy\_buyerPTs : buyerPT - plt\}$
- $partnerLink_{MC}(seller : SendPI) = sellerref{-}buyerref\_isRealizedBy\_buyerPTs : buyerPT$
- $partnerLink_{MC}(buyer : ReceivePI) = buyerref\_isRealizedBy\_buyerPTs : buyerPT{-}sellerref$

Since the participant reference $sellerref$ is limited to the scope $seller : innerscope$ and the participant reference $buyerref$ to the process $buyer : buyer$ (cf. examples 3.2 and 3.3), we get the following:

- $partnerLinks_{scope}(buyer : buyer) = \{sellerref{-}buyerref\_isRealizedBy\_buyerPTs : buyerPT\}$
- $partnerLinks_{scope}(seller : innerscope) = \{buyerref\_isRealizedBy\_buyerPTs : buyerPT{-}$ $sellerref\}$

If we set $pl1$ to the partner link declaration $sellerref - buyerref\_isRealizedBy\_buyerPTs : buyerPT$, $pl2$ to $buyerref\_isRealizedBy\_\ buyerPTs : buyerPT - sellerref$ and plt to the partner link type $sellerref - buyerref\_isRealizedBy\_buyerPTs : buyerPT - plt$, and if $comm_{new}$ is the newly created element of $Comm$, we will get the following for the message link $ProductInformation$:

- $type_{PL}(pl1) = plt$
- $type_{PL}(pl2) = plt$
- $partnerLinks_{Comm}(comm_{new}) = (pl1,\ pl2)$
- $plt_{Comm}(comm_{new}) = plt$
- $myRole_{PL}(pl1) = \perp$
- $myRole_{PL}(pl2) = buyerref$
- $partnerRole_{PL}(pl1) = buyerref$
- $partnerRole_{PL}(pl2) = \perp$

In the second message link $PurchaseOrder$ the participant reference $buyerref$ sends a message to the participant reference $sellerref$ and $sellerref$ is realized by the port type $sellerPTs : sellerPT$ to receive this message. The send activity is $buyer : SendPO$, and the receive activity is $seller : ReveivePO$. When reaching this message link, we have already created an element of the relation $Comm$ specifying a one-way communication in the opposite direction (see message link $ProductInformation$). So, case **(3)** holds, and we can use the same partner link declarations and partner link type again. After having changed the one-way communication into a request/response communication we get the following with $pl1 = sellerref - buyerref\_isRealizedBy\_buyerPTs : buyerPT$ and $pl2 = buyerref\_isRealizedBy\_\ buyerPTs : buyerPT - sellerref$:

- $Comm = \{((\{sellerref\}, sellerPTs : sellerPT), (buyerref, buyerPTs : buyerPT))\}$
- $myRole_{PL}(pl1) = sellerref$
- $partnerRole_{PL}(pl2) = sellerref$

Furthermore, the message constructs $buyer : SendPO$ and $seller : ReveivePO$ are set to their partner link declarations as follows:

- $partnerLink_{MC}(buyer : SendPO) = buyerref\_isRealizedBy\_buyerPTs : buyerPT - sellerref$
- $partnerLink_{MC}(seller : ReveivePO) = sellerref - buyerref\_isRealizedBy\_buyerPTs : buyerPT$

In the last message link $POConfirmation$ the participant reference $sellerref$ sends a message to the participant reference $buyerref$, and $buyerref$ is realized by the port type $buyerPTs : buyerPT$ to receive this message. This is the same kind of communication as like in the first message link. So, case **(1)** holds, and we just need to assign the message constructs $seller : SendConf$ and $buyer : ReveiveConf$ of this message link to their partner link declarations as follows:

- $partnerLink_{MC}(seller : SendConf) = sellerref - buyerref\_isRealizedBy\_buyerPTs : buyerPT$
- $partnerLink_{MC}(buyer : ReveiveConf) = buyerref\_isRealizedBy\_buyerPTs : buyerPT -$
  $sellerref$

Algorithm 2 Analysis of one <messageLink> declaration of the participant groundings

```
procedure ANALYZEMESSAGELINKGROUNDING(messageLink :Element)
                // the input messageLink points on the current <messagLink> tag

    ml :DT($ML$);                                   // DT($ML$) inherits of NCName
    mc1, mc2 :DT($MC$);                             // DT($MC$) inherits of QName
      // mc1 will be the send activity and mc2 the receive activity of the message link
    pt :DT($PT$);
    o :DT($O$);                            // DT($PT$) and DT($O$) inherit of QName
    pt_nsprefix :DT($NSPrefix$);          // name space prefix of the port type pt
                                             // DT($NSPrefix$) inherits of NCName
    b :DT($Pa$);
    A :List(DT($Pa$));                      // a set (list) of participant references
begin
                                                            // get message link $ml$
    ml ←messageLink.getAttributeValue("name");
                                       // get port type of $ml$ and store it in the set $PT$
    pt ←messageLink.getAttributeValue("portType");
    $PT \leftarrow PT \cup$ pt;
    pt_nsprefix ←messageLink.getAttributeNamespacePrefix("portType");
    $nsprefix_{PT}$(pt) ←pt_nsprefix;      // name space prefix of pt is set to pt_nsprefix
                                       // get operation of $ml$ and store it in the set $O$
    o ←pt_nsprefix.buildQName(messageLink.getAttributeValue("operation"));
    $O \leftarrow O \cup$ o;
                   // assign message constructs of $ml$ to port type pt and operation o
    (mc1, mc2) ←$constructs_{ML}$(ml);
    $portType_{MC}$(mc1) ←pt;
    $portType_{MC}$(mc2) ←pt;
    $operation_{MC}$(mc1) ←o;
    $operation_{MC}$(mc2) ←o;
                                                    // derive definitions 22 to 31
    (A, b) ←$parefs_{ML}$(ml);                  // sender/receiver combination of $ml$
    if $bindSenderTo_{ML}$(ml) $\neq \perp$ then         // $bindSenderTo$ attribute is specified
        A ←{$bindSenderTo_{ML}$(ml)};
    end if
                                                  // traverse the relation $Comm$
    TRAVERSECOMM;                                // call procedure traverseComm
end procedure
```

Algorithm 3 Procedure traverseComm

```
procedure TRAVERSECOMM
    comm, comm_new :DT(Comm);
    pl1, pl2 :DT(PL);                                    // DT(PL) inherits of NCName
    plt :DT(PLType);                                     // DT(PLType) inherits of NCName
begin
    if condition (1) holds then
        comm ←first element of Comm for which condition (1) holds;
                // assign message constructs of ml to their partner link declarations
        (pl1, pl2) ←$partnerLinks_{Comm}$(comm);
        $partnerLink_{MC}$(mc1) ←pl1;
        $partnerLink_{MC}$(mc2) ←pl2;
    else
        if condition (2) holds then
            comm ←first element of Comm for which condition (2) holds;
                // assign message constructs of ml to their partner link declarations
            (pl1, pl2) ←$partnerLinks_{Comm}$(comm);
            $partnerLink_{MC}$(mc1) ←pl2;
            $partnerLink_{MC}$(mc2) ←pl1;
        else
            if condition (3) holds then
                comm ←first element of Comm for which condition (3) holds;
                // assign message constructs of ml to their partner link declarations
                (pl1, pl2) ←$partnerLinks_{Comm}$(comm);
                $partnerLink_{MC}$(mc1) ←pl2;
                $partnerLink_{MC}$(mc2) ←pl1;
                    // change communication into request/response communication
                comm.changeFirstPortType(pt);
                $myRole_{PL}$(pl1) ←b;
                $parnterRole_{PL}$(pl2) ←b;
            else
                                                // one of condition (4) to (6) holds
      // new partner link declarations, a new partner link type and a new element of
  Comm need to be created
                comm_new ←((A, ⊥), (b, pt));
                Comm ←Comm ∪ comm_new;
                CREATEPARTNERLINKDECLARATIONS(comm_new);
                                // call procedure createPartnerLinkDeclarations
            end if
        end if
    end if
end procedure
```

Algorithm 4 Procedure createPartnerLinkDeclarations

```
procedure CREATEPARTNERLINKDECLARATIONS(comm_new :DT($Comm$))
    pl1, pl2 :DT($PL$);                                // DT($PL$) inherits of NCName
    plt :DT($PLType$);                                 // DT($PLType$) inherits of NCName
    senders_ids :String ←A[1];        // initially the identifier of the first participant
reference of A
    a :DT($Pa$);                                       // DT($Pa$) inherits of NCName
    sc :QName;                  // sc will be used for elements of ($Scope \cup Process$)
begin
    for all a ∈ A \ A[1] do
        senders_ids ←senders_ids + "_" + a;
      // adds an underline and the identifier of the participant reference a at the end
of the string senders_ids, senders_ids results in the character string "$a_1\_a_2\_...\_a_n$"
    end for
                                                    // create partner link declarations
    pl1 ←senders_ids + "-" + b + "_isRealizedBy_" + replaceColons(pt);
    pl2 ←b + "_isRealizedBy_" + replaceColons(pt) + "-" + senders_ids;
    $PL$ ←$PL \cup$ pl1;
    $PL$ ←$PL \cup$ pl2;
                                                           // create partner link type
    plt ←senders_ids + "-" + b + "_isRealizedBy_" + replaceColons(pt) + "-plt";
                              // results in $a_1\_a_2\_...\_a_n - b\_isRealizedBy\_pt' - plt$
    $PLType$ ←$PLType \cup$ plt;
             // assign the message constructs of $ml$ to their partner link declarations
    $partnerLink_{MC}$(mc1) ←pl1;
    $partnerLink_{MC}$(mc2) ←pl2;
                                   // assign partner link declarations to their scopes
    if $scope_{Pa}$(b) = ⊥ then
        sc ←$process_{PaType}(type_{Pa}$(A[1]));
    else                                                           // $scope_{Pa}$(b) ≠ ⊥
        sc ←$scope_{Pa}$(b);
    end if
    $partnerLinks_{scope}$(sc) ←$partnerLinks_{scope}$(sc) ∪ pl1;
    if $bindSenderTo_{ML}$(ml) = ⊥ ∨ $scope_{Pa}(bindSenderTo_{ML}$(ml)) = ⊥ then
        sc ←$process_{PaType}(type_{Pa}$(b));
    else           // $bindSenderTo_{ML}$(ml) ≠ ⊥ ∨ $scope_{Pa}(bindSenderTo_{ML}$(ml)) ≠ ⊥
        sc ←$scope_{Pa}(bindSenderTo_{ML}$(ml));
    end if
    $partnerLinks_{scope}$(sc) ←$partnerLinks_{scope}$(sc) ∪ pl2;
                                                       // modify the remaining functions
    $type_{PL}$(pl1) ←plt;
    $type_{PL}$(pl2) ←plt;
    $partnerLinks_{Comm}$(comm_new) ←(pl1, pl2);
    $plt_{Comm}$(comm_new) ←plt;
    $myRole_{PL}$(pl1) ←⊥;                                        // no $myRole$ specified
    $myRole_{PL}$(pl2) ←b;
    $partnerRole_{PL}$(pl1) ←b;
    $partnerRole_{PL}$(pl2) ←⊥;                              // no $partnerRole$ specified
end procedure
```

2.3. Converting the PBDs

No.	Definition	Explanation
(1)	NS	The set of name spaces.
(2)	$NSPrefix$	The set of name space prefixes.
(3)	$prefix_{NS} : NSPrefix \rightarrow NS$	The bijective function that assigns a name space to each name space prefix.
(4)	$PaType$	The set of participant types.
(5)	$Process$	The set of BPEL processes (PBDs).
(6)	$process_{PaType} : PaType \rightarrow Process$	The bijective function that assigns a PBD to each participant type.
(7)	$nsprefix_{Process} : Process \rightarrow NSPrefix$	The function that assigns a name space prefix to each PBD of which the target name space is associated with this name space prefix.
(8)	Pa	The set of participant references.
(9)	$type_{Pa} : Pa \rightarrow PaType$	The function that assigns a participant type to each participant reference.
(10)	$Scope$	The set of the scopes and <forEach> activities of all PBDs that have an wsu:id and that are referenced by one or more (single) participant references.
(11)	$scope_{Pa} : Pa \rightarrow Scope \cup \{\perp\}$	The function that assigns a scope to each participant reference which is limited to this scope.
(12)	MC	The set of the message constructs of all PBDs.
(13)	ML	The set of message links.
(14)	$constructs_{ML} : ML \rightarrow MC \times MC$	The function that assigns a send and a receive activity to each message link.
(15)	$parefs_{ML} : ML \rightarrow 2^{Pa} \times Pa$	The function that assigns a set of senders and a receiver to each message link.
(16)	$bindSenderTo_{ML} : ML \rightarrow Pa \cup \{\perp\}$	The function that assigns a participant reference to each message link of which the actual sender should be bound to this reference.
(17)	PT	The set of WSDL port types.

(18)	$\mathit{nsprefix}_{PT} : PT \to \mathit{NSPrefix}$	The function that assigns a name space prefix to each port type which is declared in the name space which is associated with this name space prefix.
(19)	O	The set of WSDL operations.
(20)	$\mathit{portType}_{MC} : MC \to PT$	The function that assigns a WSDL port type to each message construct.
(21)	$\mathit{operation}_{MC} : MC \to O$	The function that assigns a WSDL operation to each message construct.
(22)	PL	The set of partner link declarations.
(23)	$\mathit{partnerLink}_{MC} : MC \to PL$	The function that assigns a partner link declaration to each message construct.
(24)	$\mathit{partnerLinks}_{scope} : (\mathit{Scope} \cup \mathit{Process}) \to 2^{PL}$	The function that assigns a set of partner link declarations to each scope or process in which these partner link declaration will be enclosed.
(25)	PLType	The set of partner link types.
(26)	$\mathit{type}_{PL} : PL \to \mathit{PLType}$	The function that assigns a partner link type to each partner link declaration.
(27)	$\mathit{Comm} \subseteq (2^{Pa} \times (PT \cup \{\perp\})) \times (Pa \times PT)$	This relation associates a combination of participant references with a (pair of) port type(s) which they use to communicate.
(28)	$\mathit{partnerLinks}_{\mathit{Comm}} : \mathit{Comm} \to PL \times PL$	The function that assigns a pair of partner link declarations to each element of the relation *Comm*.
(29)	$\mathit{plt}_{\mathit{Comm}} : \mathit{Comm} \to \mathit{PLType}$	The bijective function that assigns partner link type to each element of the relation *Comm*.
(30)	$\mathit{myRole}_{PL} : PL \to Pa \cup \{\perp\}$	The function that assigns a *myRole* to a partner link declaration.
(31)	$\mathit{partnerRole}_{PL} : PL \to Pa \cup \{\perp\}$	The function that assigns a *partnerRole* to a partner link declaration.

Table 1.: Summary of definitions 1 to 31

Table 1 presents an overview of the definitions of this chapter. Now, we need to describe how the data represented by these definitions can be used to convert the single PBDs into BPEL processes including partner link declarations and having *parnterLink*, *portType* and *operation* attributes assigned to the message constructs. Algorithm 5 states a procedure in pseudo code which executes the conversion of one PBD. This procedure has to be called for each PBD of the BPEL4Chor choreography. The procedures of algorithms 6 to 11 are also used during the conversion. In addition to the assumptions made for algorithms 1 to 4, we assume the following:

- There is the following function on elements of the type `QName`:
  - `removeNSPrefix() returns NCName`: this function returns the second NCName of the QName. This means that it removes the name space prefix from the QName.
- There are the following functions on elements of the type `Element`:
  - `new Element( name :String ) returns Element`: this function is a constructor that creates a new element having the local name `name`.
  - `getName() returns String`: this function returns the local name of the element, e. g. a <scope> activity will return the character string "scope".
  - `getChild( name :String ) returns Element`: this function returns the first child element having the local name `name`. It will return $\perp$ if there is no child element having the name `name`.
  - `getChildren() returns List(Element)`: this function returns a list of all child elements of the element. It will return an empty list if there is no child element.
  - `addChild( element :Element )`: this function adds a new child element after the existing child elements.
  - `addPartnerLinks()`: this function adds a <partnerLinks> declaration at the right place of the element taking the BPEL specification ([Org07]) into account. If the element already includes a <partnerLinks> declaration, it will not be changed.
  - `addAttribute( name :String, value :String )`: this function adds an attribute having the name `name` and the value `value`. If an attribute having the name `name` already exists, its value will be overwritten by `value`.
  - `removeAttribute( name :String )`: this function removes the attribute having the local name `name`.

The function `addPartnerLinks` is used on <process> or <scope> activities to create a <partnerLinks> declaration and to put it to the right place within the BPEL code. The single partner link declarations can be declared within such a <partnerLinks> declaration. This guarantees that the resulting BPEL process does not violate the BPEL specification.

Starting the conversion of one PBD at the corresponding <process> activity (cf. algorithm 5), we first store the name space prefix referencing to the target name space of the current process in the participant topology. It can be derived by using the inverse function of the bijective function $prefix_{NS}$ on the value of the *targetNamespace* attribute assigned to the <process> activity. We will use it later to get the elements of the sets *Process*, *Scope* and *MC* by adding a colon and the local names of the corresponding activities or constructs to it.

Afterward, we add the declaration of the name space of the WSDL definitions of partner link types to the <process> activity. As name of this name space we use the target name space of the participant topology concatenated by the character string "/partnerLinkTypes" (cf. Section 5). The target name space of the participant topology can be derived by using the function $prefix_{NS}$ on the designated element $topologyNS \in NSPrefix$. As name space prefix of the newly created name space we chose "plt".

We assume that there is no other name space prefix in the PBD named by "plt" and referencing to another name space than our newly created one (cf. beginning of Chapter 1). Otherwise, the existing name space declaration would be overwritten.

Algorithm 5 Conversion of one PBD

```
procedure CONVERTPBD(process :Element)
            // the input process points on the <process> activity of the current PBD
    process_nsprefix :DT($NSPrefix$);
                // the name space prefix of the target name space of the current PBD
    nsprefixList :List(DT($NSPrefix$)) ←$\emptyset$;
          // a list of name space prefixes referencing to the name spaces of the WSDL
    definitions of port types used in this process
begin
    process_nsprefix ←$prefix_{NS}^{-1}$(process.getAttributeValue("targetNamespace"));
                // $prefix_{NS}^{-1}$ is the inverse function of the bijective function $prefix_{NS}$
          // add the declaration of the name space of the partner link type definitions
    process.addAttribute("xmlns:plt", $prefix_{NS}(topologyNS)$ + "/partnerLinkTypes")
        // $prefix_{NS}(topologyNS)$ is the target name space of the participant topology
                                                        // start depth-first search
    EXECUTEDEPTH-FIRSTSEARCH(process);                  // cf. algorithm 3.7
                                          // add partner link declarations to the process
    DECLAREPARTNERLINKS(process, process.getAttributeValue("name"));      // cf.
algorithm 3.6
                    // process.getAttributeValue("name") is the local name of the PBD
                // add the declarations of the name spaces of the port type definitions
    DECLARENAMESPACES(process, nsprefixList);           // cf. algorithm 3.8
end procedure
```

To add <partnerLink> declarations to the <process> activity, we call the procedure `declarePartnerLinks` of algorithm 6 with the <process> activity and the value of its *name* attribute as input. This value is the local name of the <process> activity. The procedure will later be used on <scope> activities, too. First, it builds the global name of the current <scope> or <process> activity by concatenating the previously stored name space prefix of the target name space of the PBD with a colon and the local name of the activity. If the resulting QName is an element of $(\mathit{Scope} \cup \mathit{Process})$, it is a <process> activity, or it is a <scope> activity and a single participant reference is limited to it. In that case, the function $\mathit{partnerLinks}_{\mathit{scope}}$ can be used on it to determine the partner link declarations that need to be declared within it.

Let $\mathit{partnerLinkList} \subseteq PL$ be the set of partner link declarations associated with the current scope or process by the function $\mathit{partnerLinks}_{\mathit{scope}}$. If it is the empty list, no partner links need to be declared. Otherwise, we will add a <partnerLinks> declaration to the <scope> or <process> activity. Furthermore, we add a <partnerLink> declaration within the newly created <partnerLinks> declaration for each element of $\mathit{partnerLinkList}$, and we add *name*, *partnerLinkType*, *myRole* and *partnerRole* attributes to it, if necessary.

Let $pl \in \mathit{partnerLinkList}$ be the current partner link declaration. Then, the value of the *name* attribute is the identifier of pl. The partner link type of the current partner link declaration can be determined by $\mathit{type}_{PL}(pl)$. The resulting NCName is concatenated to the character string "plt:". This results in the global name of the partner link type and the QName associated with the *partnerLinkType* attribute of the newly created <partnerLink> declaration.

To add a *myRole* attribute to the <partnerLink> declaration of pl, the value of $\mathit{myRole}_{PL}(pl)$ can be used. If $\mathit{myRole}_{PL}(pl) = \perp$, no *myRole* will be specified for pl. In that case, no *myRole* attribute needs to be added to the partner link declaration. Otherwise, let $a \in Pa$ be $\mathit{myRole}_{PL}(pl)$. Then, the partner link declaration gets a *myRole* attribute assigned having the identifier of the participant reference a as value. The *partnerRole* attribute is added in the same way as the *myRole* attribute only using the function $\mathit{partnerRole}_{PL}$ instead of myRole_{PL}.

To modify the constructs nested in the <process> activity, a depth-first search with backtracking is started traversing the tree of BPEL constructs having the <process> activity as root (cf. algorithm 7). During this depth-first search we are searching for message constructs, for <scope> activities having a *wsu:id* attribute assigned and for <forEach> activities (cf. algorithm 9). Note that the depth-first search is started before the partner links (and variables) are declared. These declarations are added to the <process> activity when it is reached for the second time during the backtracking. The same goes for <scope> activities (cf. algorithm 9). This guarantees that the newly added declarations are not visited during the search, which makes it more efficient.

Algorithm 6 Procedure declarePartnerLinks

```
procedure DECLAREPARTNERLINKS(scope :Element, id :NCName)
      // the input scope points on the scope or process in which the partner links need
  to be declared, and the input id is the name or wsu:id of that scope or process or of
  the corresponding <forEach> activity
    sc :QName;                                   // an element of ($Scope \cup Process$)
    partnerLinkList :List(DT($PL$));          // a set (list) of partner link declarations
    pl :DT($PL$);                                   // a single partner link declaration
    partnerLinks, partnerLink :Element;                       // single BPEL constructs
    role :DT($Pa$);                                          // a participant reference
    s :NCName;
begin
    sc ←process_nsprefix.buildQName(id);
      // the global name of the current scope or process and the corresponding element
  of ($Scope \cup Process$)
    if sc ∈ ($Scope \cup Process$) then
          // sc is a process or a single participant reference is limited to the scope sc
                          // thus the function $partnerLinks_{scope}$ can be used on sc
           // determine the set of partner link declarations that need to be declared
        partnerLinkList ←$partnerLinks_{scope}$(sc);
        if partnerLinkList $\neq \emptyset$ then            // there are partner links to be declared
            scope.addPartnerLinks();           // adding a <partnerLinks> declaration
            partnerLinks ←scope.getChild("partnerLinks");
                           // partnerLinks becomes the <partnerLinks> declaration
            for all pl ∈ partnerLinkList do
                                          // create a new partner link declaration for pl
                partnerLink ←new Element("partnerLink");
               // add name, partnerLinkType, myRole and partnerRole attributes to it
                partnerLink.addAttribute("name", p));
                s ←$type_{PL}$(pl);
                partnerLink.addAttribute("partnerLinkType", "plt:" + s);
          // "plt" is the name space prefix of the name space of the partner link type
  declarations
                role ←$myRole_{PL}$(pl);                                  // the myRole of pl
                if role $\neq \perp$ then                                // a myRole is specified
                    partnerLink.addAttribute("myRole", role);
                end if
                role ←$partnerRole_{PL}$(pl);                        // the partnerRole of pl
                if role $\neq \perp$ then                            // a partnerRole is specified
                    partnerLink.addAttribute("partnerRole", role);
                end if
          // add the new partner link declaration to the <partnerLinks> declaration
                partnerLinks.addChild(partnerLink);
            end for
        end if
    end if
end procedure
```

Algorithm 7 Procedure executeDepth-firstSearch

```
procedure EXECUTEDEPTH-FIRSTSEARCH(currentConstruct :Element)
        // the input currentConstruct points on the tag of the current BPEL construct
    constructList :List(Element);                         // a list of BPEL constructs
    construct :Element;                                     // a single BPEL construct
begin
    constructList ←currentConstruct.getChildren();
    for all construct ∈ constructList do
        MODIFYCONSTRUCT(construct);                                 // cf. algorithm 9
    end for
end procedure
```

If the current BPEL construct is a message construct, we will add *partnerLink*, *portType* and *operation* attributes to it, and we will change the *wsu:id* attribute to a *name* attribute if this is possible (cf. algorithm 10). The latter may only be done if the message construct is an <invoke>, <receive> or <reply> activity since <onMessage> or <onEvent> constructs are not allowed to have a *name* attribute assigned (cf. [Org07]).

Let *mc* be the element of *MC* of the current message construct. This element can be derived by concatenating the previously stored name space prefix, which references to the target name space of the current process in the participant topology, with a colon and the value of the *wsu:id* attribute assigned to the message construct. The partner link, port type and operation associated with *mc* can be derived from the functions $partnerLink_{MC}$, $portType_{MC}$ and $operation_{MC}$. The value of the *partnerLink* attribute is the identifier of $partnerLink_{MC}(mc) \in PL$. The value of the *portType* attribute is the identifier of $portType_{MC}(mc) \in PT$. The value of the *operation* attribute has to be an NCName (cf. [Org07]), which is the local name of the corresponding WSDL operation. To get this local name, we need to use the function `removeNSPrefix` on the identifier of $operation_{MC}(mc) \in O$.

Since the value of the *portType* attribute is a QName, which includes a name space prefix, we need to add the name space declaration of this name space prefix and the corresponding name space to the <process> activity. To do so, we add the name space prefix of the port type associated with the current message construct to a global list of name space prefixes if it has not been added before. The name space prefix of that port type can be derived by using the function $nsprefix_{PT}$ on $portType_{MC}(mc)$. Having finished the depth-first search completely, a name space declaration is added to the <process> activity for each element of the global list of name space prefixes (cf. algorithm 8). The name of a name space can be derived by using the function $prefix_{NS}$ on the current element of the global list.

Algorithm 8 Procedure declareNameSpaces

```
procedure declareNameSpaces(construct :Element, nsprefixList :List(DT($NSPrefix$)))
        // the input construct points on the tag of the current BPEL construct, and nsprefixList includes the name space prefixes referencing to the name spaces which need to be declared within construct
    nsprefix :DT($NSPrefix$);                               // a single name space prefix
begin
                                        // add each name space declaration to the construct
    for all nsprefix ∈ nsprefixList do
        construct.addAttribute("xmlns:" + nsprefix, $prefix_{NS}$(nsprefix));
          // $prefix_{NS}$(nsprefix)) is the name space referenced by the name space prefix nsprefix
    end for
end procedure
```

Like in the case of the partner link types, we assume that there is no other name space prefix having the same name as like any name space prefix stored in our global list (cf. beginning of Chapter 1). Furthermore, there must not be any name space prefix in the global list having the name "plt" since this name is reserved for the name space prefix referencing to the target name space of the WSDL definitions of partner link types.

If the current message construct is modified completely, the depth-first search will be continued on it if it is an <invoke> activity or an <onMessage> or <onEvent> construct (cf. algorithm 9). This has to be done since an <onMessage> or <onEvent> construct may contain other BPEL activities, and an <invoke> activity may contain <catch> or <catchAll> constructs and a compensation handler, which, for it part, may contain other BPEL activities (cf. [Org07]). The subtree of a <reply> or <receive> activity must not contain any other BPEL activities or message constructs. So, the depth-first search can be stopped at them. Note that the depth-first search may be continued on <invoke> activities or <onMessage> or <onEvent> constructs after having modified them. This does not affect the efficiency of our procedure since no BPEL constructs are added to the message constructs during the modification.

When reaching a <scope> activity having a *wsu:id* attribute assigned, we need to declare partner links within it, if necessary. To declare partner links, the procedure `declarePartnerLinks` is called with the <scope> activity and the value of its *wsu:id* attribute as input. The latter is the local name of the <scope> activity.

When reaching a <forEach> activity having a *wsu:id* attribute assigned, we need to declare partner links within the <scope> activity nested in the <forEach> activity, if necessary. To do so, the procedure `declarePartnerLinks` is called with the <scope>

activity and the value of the *wsu:id* attribute assigned to the <forEach> activity as input.

Since no subtree of another BPEL construct nested in the <forEach> activity may contain BPEL activities or message constructs (cf. [Org07]), we only need to continue the depth-first search on the <scope> activity. This may also be done at <forEach> activities having no *wsu:id* attribute assigned. A <scope> activity nested in a <forEach> activity may have a *wsu:id* attribute assigned, too. In that case, additional partner link declarations may be limited to this scope. Therefore, we need to call the procedure `modifyConstruct` and not the procedure `executeDepth-firstSearch` on such a <scope> activity. Otherwise, only its child elements would be analyzed. So, the additional partner link declarations would not be added to the <scope> activity.

The execution time of the depth-first search rises at an exponential rate with the depth of the tree of BPEL constructs. Our depth-first search could possibly be more efficient if we stop it at more BPEL constructs of which the subtree must not contain any BPEL activities or message constructs. Such a construct is e. g. the <variables> construct. But we argue that this would even be more costly since an additional comparison at algorithm 7 would be needed whether the subtree of a BPEL construct may contain any BPEL activities or message constructs. Furthermore, the subtrees of such constructs only have a depth of at most one (cf. [Org07]). So, it is even more efficient just to traverse the whole subtree until each node of it has been reached.

Algorithm 9 Procedure modifyConstruct

```
procedure modifyConstruct(construct :Element)
            // the input construct points on the tag of the current BPEL construct
    constructName :String ←construct.getName(); // the name of the current BPEL construct
    fEScope :Element;                              // a single BPEL construct
begin
    if (constructName = "invoke" ∨ constructName = "onMessage" ∨ constructName = "onEvent") then
            // construct is an <invoke> activity or an <onMessage> or <onEvent> construct
        modifyMessageConstruct(construct, constructName); // cf. algorithm 10
                                                   // continue depth-first search
        executeDepth-firstSearch(construct);             // cf. algorithm 7
    else
        if (constructName = "receive" ∨ constructName = "reply") then
                                // construct is a <receive> or <reply> activity
            modifyMessageConstruct(construct, constructName); // cf. algorithm 10
        else
            if (constructName = "scope" ∧ construct.hasAttribute("wsu:id")) then
                // construct is a <scope> activity having a wsu:id attribute assigned
                                                   // continue depth-first search
                executeDepth-firstSearch(construct);         // cf. algorithm 7
                                // add partner link declarations to the scope
                declarePartnerLinks(construct, construct.getAttributeValue("wsu:id"));
                                                   // cf. algorithm 6
            else
                if constructName = "forEach" then
                                        // construct is a <forEach> activity
                    fEScope ←construct.getChild("scope");
        // fEScope points on the <scope> activity nested in the <forEach> activity
                            // continue depth-first search on the scope
                    modifyConstruct(fEScope);                // recursion
                    if construct.hasAttribute("wsu:id") then
                                // add partner link declarations to the scope
                        declarePartnerLinks(fEScope, construct.getAttributeValue("wsu:id"));
                                                   // cf. algorithm 6
                    end if
                else
                    // construct may be any BPEL construct, e.g. a structured activity
                                                   // continue depth-first search
                    executeDepth-firstSearch(construct);     // cf. algorithm 7
                end if
            end if
        end if
    end if
end procedure
```

Algorithm 10 Procedure modifyMessageConstruct

```
procedure modifyMessageConstruct(construct :Element, constructName :String)
        // the input construct points on the tag of the current message construct, and constructName is the local name of it
    mc :DT($MC$);                                    // the current element of $MC$
    constructID :NCName ←construct.getAttributeValue("wsu:id");
    pl :DT($PL$);                                    // a single partner link declaration
    pt :DT($PT$);                                    // a single port type
    op :DT($O$);                                     // a single operation
    pt_nsprefix :DT($NSPrefix$) ←$nsprefix_{PT}$(pt); // the name space prefix of the port type pt
begin
                        // change the wsu:id attribute to a name attribute if possible
    if (constructName $\neq$ "onMessage" $\wedge$” constructName $\neq$ "onEvent") then
        // <onMessage> and <onEvent> constructs are not allowed to have a name attribute assigned
        construct.removeAttribute("wsu:id");
        construct.addAttribute("name", constructID);
    end if
    mc ←process_nsprefix.buildQName(constructID);
            // mc becomes the global name of the current message construct and the corresponding element of $MC$;
                        // add a partnerLink attribute to the message construct
    pl ←$partnerLink_{MC}$(mc);
    // pl becomes the partner link declaration used by the current message construct
    construct.addAttribute("partnerLink", pl);
                        // add a portType attribute to the message construct
    pt ←$portType_{MC}$(mc);
    construct.addAttribute("portType", pt);
        // add the name space prefix of pt to the global list nsprefixList if it has not been added before
    if pt_nsprefix $\notin$ nsprefixList then
        nsprefixList ←nsprefixList $\cup$ pt_nsprefix;
    end if
                        // add an operation attribute to the message construct
    op ←$operation_{MC}$(mc);
    construct.addAttribute("operation", op.removeNSPrefix());
        // the operation attribute needs to be associated with the local name of the operation
end procedure
```

2.4. Why not Using Pairs of Communicating Participant Types

In Section 2 we have mentioned that there is an alternative way to create partner link types and partner link declarations. In contrast to our approach, this way creates them for pairs of communicating participant types instead of combinations of communicating participant references. Again, if each participant type of the pair is not realized by multiple port types, exactly one partner link type and two partner link declarations will be created for this pair. Otherwise, let n be the number of port types by which the first participant type is realized, and let m be the number of port types for the second participant type. Then, we need exactly $max(n, m)$ partner link types and $2 * max(n, m)$ partner link declarations for the pair of participant types. Figure 7 states a counter example that shows why this alternative way is not suited to the transformation of a BPEL4Chor choreography into BPEL Abstract Processes.

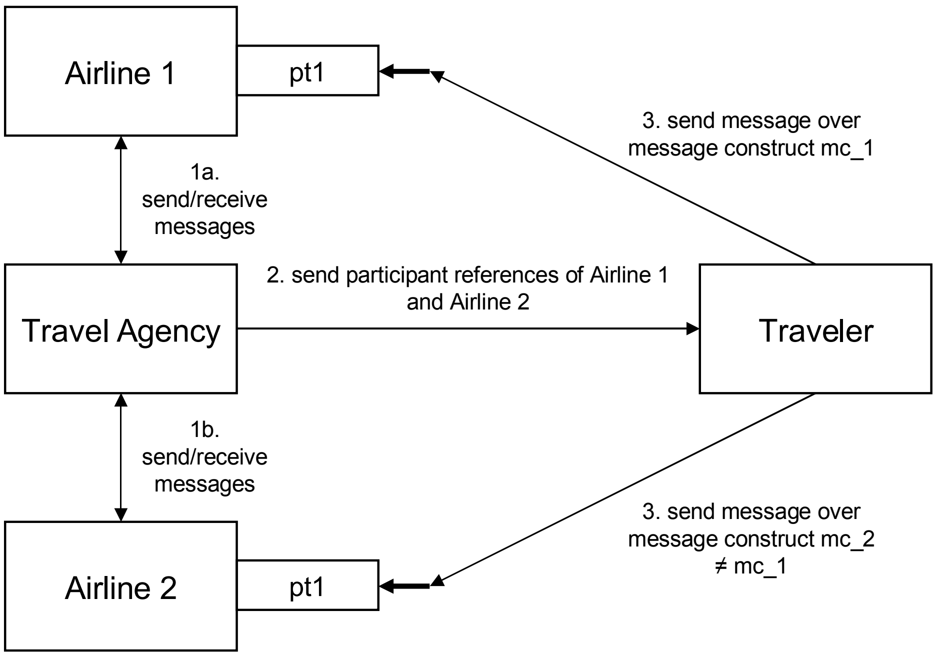
Airline 1
pt1
3. send message over message construct mc_1
1a. send/receive messages
2. send participant references of Airline 1 and Airline 2
Travel Agency
Traveler
1b. send/receive messages
3. send message over message construct mc_2 ≠ mc_1
Airline 2
pt1

Figure 7.: A counter example

In this example there are four communicating participants: a travel agency, two airlines and a traveler. We assume that each of them is bound to a different participant reference and that both airlines are of the same participant type. Initially, the travel agency knows both the airlines and the traveler. The airlines and the traveler only know the travel agency. First, the travel agency communicates with both airlines either in parallel or in sequence. Then, it forwards their participant references to the traveler. Afterward, the traveler sends a message to each airline using different message constructs. These message constructs are executed in parallel, e. g. they are nested in a <flow> activity. Both airlines are realized by the same port type to receive the corresponding message.

Using the approach presented in Section 2, we get two partner link declarations on the side of the traveler, one for each airline, since they are bound to different participant references. In that case, each of the two transmitted endpoint references is copied to the *partnerRole* of a different partner link. So, the communication can take place in parallel.

Using the alternative way, we get one partner link declaration on the side of the traveler. This partner link declaration is used to communicate with both airlines since the latter are of the same participant type and are realized by the same port type. In that case, both transmitted endpoint references will be copied to the *partnerRole* of the same partner link if its declaration is not nested in a separate <scope> activity. Since the communication takes place in parallel, this leads to an overwriting of endpoint references on a partner link. Then, both messages are sent only to one of the airlines. So, the traveler is not able to communicate with both airlines in parallel.

This contradiction has shown that the alternative way to create partner link types and partner link declarations is not suited to the transformation of a BPEL4Chor choreography into BPEL Abstract Processes.

3. Correlation Properties

The following definitions will be used to replace the NCNames of the correlation properties in the PBDs by the QNames of the corresponding references to WSDL properties. Since the binding of these NCNames to the appropriate QNames is done in the participant groundings, the data which is represented by these definitions is stored in step 2 of our procedure.

Definition 32 (set of NCNames of correlation properties).

The set *CorrPropName* is defined as the set containing all NCNames that are used as names of correlation properties and that are referenced to WSDL properties in the participant groundings.

Definition 33 (set of WSDL properties).

The set *Property* is defined as the set containing all WSDL properties which are referenced in the participant groundings.

Note that the set *Property* contains both WSDL properties associated with correlation properties and WSDL properties associated with transmitted participant references (cf. Section 5.1).

Definition 34 (the function assigning a property to each property name).

The function $\mathit{property}_{\mathit{CorrPropName}} : \mathit{CorrPropName} \rightarrow \mathit{Property}$ is defined as the function that assigns a WSDL property to each name of a correlation property which is referenced to the property in the participant groundings.

The function $\mathit{property}_{\mathit{CorrPropName}}$ will be used later to replace the names of correlation properties in the PBDs by their references to WSDL properties. We can assume that each name of a correlation property used in the PBDs is included in the set *CorrPropName* and thus is referenced to a WSDL property since the BPEL4Chor choreography is assumed to be completely grounded. This includes that each name of a correlation property used in the PBDs is referenced to a WSDL property in the participant groundings.

Definition 35 (the function assigning a name space prefix to each WSDL property).

The function $\mathit{nsprefix}_{\mathit{Property}} : \mathit{Property} \rightarrow \mathit{NSPrefix}$ is defined as the function that assigns a name space prefix to each WSDL property which is declared in the name space referenced by the name space prefix.

The function $\mathit{nsprefix}_{\mathit{Property}}$ will be used later to add name space declarations to a PBD. These name space declarations refer to the name spaces in which the WSDL properties are declared which are referenced by the QNames of the correlation properties used within the converted PBD. The WSDL files including the declaration of these properties need to be included in the input of our procedure (cf. beginning of Chapter 1). The corresponding elements of *NSPrefix* have been created while extending the sets *NS* and *NSPrefix* and the function $\mathit{prefix}_{\mathit{NS}}$ by the name space references in the <grounding> tag of the participant groundings (cf. Section 2.2).

The sets and functions defined in definitions 32 to 35 can be derived from the <properties> declaration of the participant groundings by traversing each <property> declaration one by one. At each <property> declaration we add the property name referenced by the value of the *name* attribute to the set *CorrPropName*. Furthermore, we add the WSDL property referenced by the value of the *WSDLproperty* attribute to the set *Property*. The identifier of an element of *CorrPropName* is the NCName of the respective *name* attribute. The identifier of an element of *Property* is the QName of the respective *WSDLproperty* attribute. The first NCName of this QName is the name space prefix that refers to the name space of the WSDL definitions in which the corresponding property is declared, and the second one is the local name of that property.

The functions $\mathit{property}_{\mathit{CorrPropName}}$ and $\mathit{nsprefix}_{\mathit{Property}}$ of definitions 34 and 35 can be derived as follows. Let $propName \in \mathit{CorrPropName}$ and $\mathit{property} \in \mathit{Property}$ be the property name and WSDL property referenced in the current <property> declaration. Then, $\mathit{property}_{\mathit{CorrPropName}}(propName)$ is set to $\mathit{property}$, and $\mathit{nsprefix}_{\mathit{Property}}(\mathit{property})$ is set to the first NCName of the QName $\mathit{property}$.

Algorithm 11 states a procedure in pseudo code which executes the derivation of the sets and functions defined in definitions 32 to 35 for one <property> declaration. This procedure has to be called for each <property> declaration within the <properties> declaration of the participant groundings. For this algorithm we assume the same data types and functions as we did for algorithms 1 to 10.

Example 3.8 (property names and WSDL properties). The following code snippet of a participant groundings presents a <properties> declaration including two <property> declarations. We assume that the name space prefix `prop` is included within a name space declaration in the <grounding> tag and thus is included in the set *NSPrefix*.

```
<properties>
```

Algorithm 11 Analysis of one <property> declaration of the participant groundings

```
procedure analyzePropertyGrounding(construct :Element)
                    // the input construct points on the current <property> tag

    propName :DT($CorrPropName$);      // DT($CorrPropName$) inherits of NCName
    property :DT($Property$);                  // DT($Property$) inherits of QName
begin
                                   // get property name and WSDL property
    propName ←construct.getAttributeValue("name");
    property ←construct.getAttributeValue("WSDLproperty");
                          // add them to the sets $CorrPropName$ and $Property$
    $CorrPropName \leftarrow CorrPropName \cup$ propName;
    $Property \leftarrow Property \cup$ property;
                              // assign property name to its WSDL property
    $property_{CorrPropName}$(propName) ←property;
                            // assign WSDL property to its name space prefix
    $nsprefix_{Property}$(property) ←construct.getAttributeNamespacePrefix("WSDLproperty");
end procedure
```

```
  <property name="ProcessID" WSDLproperty="prop:ProcessID" />
  <property name="InstanceID" WSDLproperty="prop:InstanceID" />
</properties>
```

To derive the sets and functions defined in definitions 32 to 35, we need to use the procedure `analyzePropertyGrounding` on each of the two <property> declarations. This results in the sets $CorrPropName = \{ProcessID, InstanceID\}$ and $Property = \{prop : ProcessID, prop : InstanceID\}$. Furthermore, we get the functions $property_{CorrPropName}$ and $nsprefix_{Property}$ as follows:

- $property_{CorrPropName}(ProcessID) = prop : ProcessID$
- $property_{CorrPropName}(InstanceID) = prop : InstanceID$
- $nsprefix_{Property}(prop : ProcessID) = prop$
- $nsprefix_{Property}(prop : InstanceID) = prop$

To replace the NCNames of the correlation properties in the PBDs by the QNames of the corresponding references to WSDL properties, the depth-first search in step 3 of our procedure (cf. Section 2.3) needs to be refined. In addition to message constructs, <scope> activities having a *wsu:id* attribute assigned and <forEach> activities, we also search for <correlationSets> constructs. Algorithm 12 on page 59 presents the resulting refined procedure `modifyConstruct` of algorithm 9 on page 50. The beginning and the end of the code section which is added to the procedure is indicated by extra comments. For the sake of simplicity, some code elements of the original procedure have been left out, which is indicated by three dots. The procedure of algorithm 12 uses the procedure `modifyCorrelationSet` of algorithm 13. In addition to the assumptions made for algorithms 1 to 11, we assume the following:

- There is the following function on elements of the type `Element`:
  - `addAttribute( name :String, value :List(String) )`: this function adds an attribute having the name `name`. The input variable `value` needs to be either a list of NCNames or a list of QNames. As value the attribute gets the corresponding list, while each element of it is separated by a space. If an attribute having the name `name` already exists, its value will be overwritten by the list `value` accordingly.

Note that this new function `addAttribute` differs from the one assumed before (cf. Section 2.3) by the type of the input variable `value`.

When reaching a <correlationSets> construct during our depth-first search, we use the function `modifyCorrelationSet` of algorithm 13 on each <correlationSet> construct nested in the current <correlationSets> construct. Since a <correlationSets> construct must not include other BPEL constructs than <correlationSet> constructs, the current list of <correlationSet> constructs is just the list of all child elements of the current <correlationSets> construct.

To modify a <correlationSet> construct, we first get the list of property names associated with the *properties* attribute of the construct. Then, we use the function $property_{CorrPropName}$ of definition 34 on each property name included in the list to get the corresponding list of WSDL properties. Note that the latter is a list of QNames. Later, the value of the *properties* attribute will be overwritten by it. In that way, we have replaced each property name by its reference to a WSDL property.

Furthermore, we need to add some additional name space declarations to the <process> activity of the current PBD. These name space declarations refer to the name spaces in which the WSDL properties are declared which are referenced by the currently regarded correlation properties. To do so, we will add the name space prefixes associated with these WSDL properties by the function $nsprefix_{Property}$ of definition 35 to the global list of name space prefixes `nsprefixList` if they have not been added before. The corresponding name space declarations are added to the <process> activity after having finished the depth-first search (cf. algorithms 5 and 8).

We assume that the PBD has not included any name space declarations before in which a name space prefix is used which has been added to the global list `nsprefixList` currently

Algorithm 12 Refined procedure modifyConstruct

```
procedure MODIFYCONSTRUCT(construct :Element)
                    // the input construct points on the tag of the current BPEL construct
    constructName :String ←construct.getName();        // the name of the current BPEL construct
    ...
begin
    if (constructName = "invoke" ∨ constructName = "onMessage" ∨ constructName = "onEvent") then
        // construct is an <invoke> activity or an <onMessage> or <onEvent> construct
        ...
    else
        if (constructName = "receive" ∨ constructName = "reply") then
                                        // construct is a <receive> or <reply> activity
            ...
        else
            if (constructName = "scope" ∧ construct.hasAttribute("wsu:id")) then
                    // construct is a <scope> activity having a wsu:id attribute assigned
                ...
            else
                if constructName = "forEach" then
                                                // construct is a <forEach> activity
                    ...

                                                // here the refinement begins
                else
                    if constructName = "correlationSets"  then
                                        // construct is a <correlationSets> construct
                    // it includes one or more <correlationSet> constructs (and nothing else)
                        corrSet :Element;                       // a single BPEL construct
                        corrList :List(Element) ←construct.getChildren();
            // corrList becomes the list of <correlationSet> constructs nested in the current <correlationSets> construct
                                    // modify each of these <correlationSet> constructs
                        for all corrSet ∈ corrList do
                            MODIFYCORRELATIONSET(corrSet);              // cf. algorithm 13
                        end for
                                                // here the refinement ends

                    else
                        // construct may be any BPEL construct, e.g. a structured activity
                        ..
                    end if
                end if
            end if
        end if
    end if
end procedure
```

(cf. beginning of Chapter 1). Furthermore, there must not be any currently added name space prefix having the name “plt” since this name is reserved for the name space prefix referencing to the target name space of the WSDL definitions of partner link types.

Example 3.9 (modifying correlation sets). The following code snippet of a PBD presents a declaration of one correlation set. The sets, relations and functions of definitions 1 to 35 are the same as like in examples 3.1 to 3.8.

```
<correlationSets>
  <correlationSet name="correlationSetName"
    properties="ProcessID InstanceID" />
</correlationSets>
```

After having converted the whole PBD, the property names `ProcessID` and `IncstanceID` have been changed to the references to WSDL properties `prop:ProcessID` and `prop:IncstanceID`. Thus, the resulting declaration of the correlation set looks like as follows:

```
<correlationSets>
  <correlationSet name="correlationSetName"
    properties="prop:ProcessID prop:InstanceID" />
</correlationSets>
```

Algorithm 13 Procedure modifyCorrelationSet

```
procedure MODIFYCORRELATIONSET(corrSet :Element)
        // the input corrSet points on the tag of the current <correlationSet> construct
    propNameList :List(DT(CorrPropName));                  // a list of property names
    propName :DT(CorrPropName);                              // a single property name
    propertyList :List(DT(Property)) ←∅;                  // a list of WSDL properties
                                                              // initially the empty list
    property :DT(Property);                                // a single WSDL property
    nsprefix :DT(NSPrefix);                               // a single name space prefix
begin
                      // get the list of property names of the current correlation set
    propNameList ←corrSet.getAttributeValueAsList("properties");

      // use the function $property_{CorrPropName}$ to get the corresponding list of WSDL
properties
    for all propName ∈ propNameList do
        property ←$property_{CorrPropName}$(propName);
        propertyList ←propertyList ∪ property;

        // add the name space prefix of the current WSDL property to the global list
nsprefixList of name space prefixes if it has not been added before
     // the corresponding name space declaration is added to the <process> activity
after having finished the depth-first search (cf. algorithms 5 and 8)

        nsprefix ←$nsprefix_{Property}$(property);
        if nsprefix ∉ nsprefixList then
            nsprefixList ←nsprefixList ∪ nsprefix;
        end if
    end for
            // change the property names to the corresponding references to WSDL
properties
    corrSet.addAttribute("properties", propertyList);
                                                 // the properties attribute is overwritten
end procedure
```

4. Set-Based ForEach

A set-based <forEach> activity is a <forEach> activity iterating over a set of participant references. The set and the iterator are specified in the participant topology. In a BPEL process a set-based <forEach> activity is not allowed. It has to be converted into a <forEach> activity iterating over a variable, and the amount of iterations needs to be defined by a combination of a <startCounterValue> and a <finalCounterValue>. If the owner of the <forEach> activity sends one or more messages to the iterator, the endpoint reference which refers to the iterator needs to be copied to the corresponding partner link at each branch of the <forEach> activity. In Section 4.1 the matter of converting a set-based <forEach> activity is exemplified. Afterward, Section 4.2 describes how our procedure has to be refined to accomplish this conversion.

4.1. Exemplifying the Conversion of a Set-Based ForEach

Assume the code snippet of a participant topology of listing 4.1 and the set-based <forEach> activity of listing 4.2. This <forEach> activity is used by the participant reference `p`. It iterates over the set of participant references `set`, and the iterator is the single participant reference `i`.

```
<participants>
  <participant name="p" type="p"/>
  <participantSet name="set" type="q" forEach="s:forEach1">
    <participant name="i" type="q" forEach="s:forEach1" />
  </participantSet>
</participants>
```

Listing 4.1: Topology for a set-based <forEach> activity

```
<forEach wsu:id="forEach1">
  <scope>
  ... nested activity ...
  </scope>
</forEach>
```

Listing 4.2: Set-based <forEach> activity used by participant reference p

We assume that the participant reference `p` sends one or more messages to the iterator `i` at each branch of the <forEach> activity. Then, the <forEach> activity is rewritten to the <forEach> activity presented in listing 4.3. The partner link declaration in this listing has been created formerly (cf. Section 2). For the sake of clearness, it has been enclosed to the listing, too. It creates a partner link between the participant references `p` and `i`. We assume that `i` is realized by the port type `PTs:pt` to receive the messages sent by `p`. So, the partner link declaration gets the name `p-i_isRealizedBy_PTs_pt` and the type `p-i_isRealizedBy_PTs_pt-plt`. Note that the colon in the name of the port type is replaced by an underline since the name of a partner link declaration or a partner link type has to be an NCName (cf. Section 2). The *partnerRole* of the partner link declaration is `i`. Its *myRole* may be `p` if `i` responds to the messages sent by `p`. Otherwise, the *myRole* will not be specified. Note that the partner link declaration is enclosed to the <scope> activity nested in the <forEach> activity. So, a new partner link is created at each branch of the <forEach> activity.

First of all, we need to add a *counterName* attribute to the <forEach> activity. This generates a counter variable for the loop which is declared implicitly within the <scope> activity (cf. [Org07]). The value of the attribute and the name of the counter variable is the character string "i_" concatenated by the value of the *wsu:id* attribute of the <forEach> activity ("i_forEach1" in listing 4.3). We assume that there is no other variable declared within the <scope> activity and having the same name as like the counter variable (cf. beginning of Chapter 1).

```
<forEach wsu_id="forEach1" counterName="i_forEach1">
  <startCounterValue>0</startCounterValue>
  <finalCounterValue>count($set/)-1</finalCounterValue>
  <scope>
    <partnerLinks>
      <partnerLink name="p-i_isRealizedBy__PTs_pt"
          type="plt:p-i_isRealizedBy__PTs_pt-plt"
                  partnerRole="i" myRole="p"? />
    </partnerLinks>
    <sequence>
     <assign>
       <copy>
         <from variable="set">
           <query>[$i_forEach1]</query>
         </from>
         <to partnerLink="p-i_isRealizedBy__PTs_pt" />
       </copy>
     </assign>
    ... nested activity ...
    </sequence>
  </scope>
</forEach>
```

Listing 4.3: Rewritten <forEach> activity

The variable `set` in listing 4.3 is a variable containing an array of endpoint references. It needs to include exactly those endpoint references that refer to the endpoints of the participants bound to the participant references contained in the set `set` of listing 4.1. Furthermore, it is declared globally in the <process> activity. Its name is the name of the corresponding set of participant references specified in the participant topology. We assume that there is no other variable declared in the hierarchical path between the process and the scope of the <forEach> activity and having the same name (cf. beginning of Chapter 1).

The variable `set` can be filled in two ways. The first way is a UDDI query nested in a <while> activity. But this is part of the executable completion of the BPEL processes (step 3 of Figure 1 on page 5). Thus, it is out of scope for this work. The second way is to receive it from another participant transmitting a set of participant references. In that case, the endpoint references are copied to the variable `set` after the relevant receive activity (cf. Section 5.1).

The syntax of the XML Schema definition describing the data type of the newly created variable `set` can be seen in listing 4.4.

```
<srefs:service-refs>
  <sref:service-ref ... />+
</srefs:service-refs>
```

Listing 4.4: Syntax for service-refs

`sref:service-ref` is the data type used to store a single endpoint reference, while the name space prefix `sref` refers to the name space `http://docs.oasis-open.org/wsbpel/2.0/serviceref.service-ref` (cf. [Org07]). The data type `srefs:service-refs` is a container for a sequence of single endpoint references, while the name space prefix `srefs` refers to the name space `http://www.bpel4chor.org/service-references`. Since a service reference needs to be added to the set, an XSL transformation needs to be specified enabling this insertion. The provisional style sheet can be seen in listing 4.5. Its completion and the creation of the new XML Schema definition and the corresponding name space is out of scope for this work.

```
<xsl:transform version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform" ...
  targetNamespace="http://www.bpel4chor.org/service−references"
  xsl:name="add.xsl" >
  <xsl:param name="NewSRef">
    <sref:service-reference />
  </xsl:param>
  <xsl:template match="sref:service−reference">
    <xsl:copy-of select="." />
    <xsl:if test="position()=last()">
      <xsl:copy-of select="$NewSRef" />
    </xsl:if>
  </xsl:template>
</xsl:transform>
```

Listing 4.5: XSL style sheet to append a service reference to a sequence of service references

To set the amount of iterations of the <forEach> activity, we need to add both a <startCounterValue> and a <finalCounterValue> construct to it. The <startCounterValue> is set to zero and the <finalCounterValue> to the amount of elements in the variable `set` minus one. The latter can be derived by using the XPath expression `count($set/)-1` (cf. [CD99]). In the first iteration the counter variable will be set to zero. Each subsequent iteration will increment the previously initialized counter variable value by one until the final iteration has been reached where it will be set to the <finalCounterValue>. In that way, the <forEach> activity will be executed exactly `count($set/)` times (cf. [Org07]).

To copy the proper endpoint reference to the partner link `p-i_isRealizedBy_PTs_pt`, an <assign> activity is added before the nested activity of the <scope> activity. This <assign> activity copies the endpoint reference from the array `set` having the value of the counter variable as index to the partner link between `p` and `i`. The index can be derived by using the XPath expression `[$i_{forEach1}]`. As the counter variable is assigned to each integer value between zero and the amount of elements of the array `set` minus one, each endpoint reference is copied to a partner link exactly once. In that way, the participant bound to the participant reference `p` communicates with each participant bound to the set of participant references `set` of listing 4.1 exactly once, too. Note that the <assign> activity and the nested activity are rewritten to a <sequence> activity. This guarantees that the endpoint reference is copied to the partner link before the participant reference `p` can send a message to the iterator `i`.

If the participant bound to `p` does not send a message to the iterator `i` in any branch of the <forEach> activity, there will be no partner link to which the respective endpoint reference may be copied. Thus, the <assign> activity must not be enclosed to the <scope> activity in that case. We just need to declare the variable `set` and add the *counterName* attribute and the <startCounterValue> and <finalCounterValue> constructs to the <forEach> activity. This results in the <forEach> activity presented in listing 4.6.

```
<forEach wsu_id="forEach1" counterName="i_forEach1">
  <startCounterValue>0</startCounterValue>
  <finalCounterValue>count($set/)-1</finalCounterValue> %$
  <scope>
    ... nested activity ...
  </scope>
</forEach>
```

Listing 4.6: Rewritten <forEach> activity without an <assign> activity

Note that the participant bound to `p` will send one or more messages to the iterator `i` if and only if a partner link declaration between them exists (cf. Section 2). This fact is used in Section 4.2 to determine whether the <assign> activity has to be added or not.

The owner of a <forEach> activity might send several messages to its iterator over several message links. If the iterator is realized by multiple port types to receive these messages, there will be more than one partner link declaration between them. In that case, the owner of the <forEach> activity is possibly not able to send all messages to the iterator since an endpoint reference is associated with only one port type. Hence, we restrict that the iterator is realized by one port type to receive messages from the owner of the <forEach> activity (cf. Section 5.1). This implies that there is at most one partner link declaration between the owner and the iterator. Consequently the <assign> activity will only be added before the nested activity if and only if there is exactly one partner link declaration between the owner and the iterator.

A <forEach> activity has a *parallel* attribute assigned, which is set either to the value *yes* or to the value *no*. If it is set to *yes*, the branches of the <forEach> activity will be executed in parallel. Otherwise, they will be executed in sequence (cf. [Org07]). Since the partner link and the counter variable are declared within the <scope> activity, the <forEach> activity can be converted similarly in both cases. So, the *parallel* attribute needs not to be removed or changed.

The optional <completionCondition> evaluates to an integer value `B`. If at least `B` branches of the <forEach> activity have been completed, the <completionCondition> will be triggered. Then, no other branches will be started, and, in the case of a parallel <forEach> activity, all still running branches need to be terminated (cf. [Org07]). We forbid that the unsigned-integer expression of a <completionCondition> includes something which is specified in the participant topology or the participant groundings since both will not be reachable to the converted PBDs. Then, the usage of a <completionCondition> does no harm to our conversion.

4.2. Refining our Procedure

The following definition will be used to get the name of the variable of endpoint references used within a converted set-based <forEach> activity.

Definition 36 (the function assigning a set of participant references to each <forEach> activity).

The function $set_{forEach} : Scope \to Pa \cup \{\bot\}$ is defined as the function that assigns a set of participant references to each set-based <forEach> activity which iterates over this set. Let $sc \in Scope$ be a <scope> or <forEach> activity, and let $set \in Pa \cup \{\bot\}$ be a set of participant references or the element $\bot$. Then, $set_{forEach}(sc)$ is set to set if and only if one of the following two conditions holds:

1. sc is a <forEach> activity, set is a set of participant references, and set is associated with sc by a *forEach* attribute in its <participantSet> declaration.
2. sc is a <scope> activity, and we have $set = \bot$.

Note that a set of participant references can be associated with more than one <forEach> activitiy by a *forEach* attribute, but a <forEach> activity can be associated with at most one set of participant references. Otherwise, the activity would iterate over more than one set, which does not make sense. If a set of participant references is associated with more than one <forEach> activity, each of them will iterate over the same set of participant references.

The function defined in definition 36 can be derived while analyzing the <participants> declaration of the participant topology. Let $sc \in Scope$ be a <scope> or <forEach> activity. If sc is associated with a participant reference by a *scope* attribute, $set_{forEach}(sc)$ will be set to $\bot$. If sc is associated with a set of participant references by a *forEach* attribute, $set_{forEach}(sc)$ will be set to the name of the set.

In Section 2.1 we have specified that a <forEach> activity will only be added to the set *Scope* if it is associated with a single participant reference by a *forEach* attribute. Now, we specify that it will only be added to the set *Scope* if it is associated with a set of participant references. Otherwise, we would set the function $set_{forEach}$ for <forEach> activities that have not been added to *Scope* yet since the <particpantSet> declarations are above the <particpant> declarations of the iterators. Actually, the same set *Scope* is generated by both variants since a <participantSet> declaration having a *forEach* attribute assigned must contain exactly one <participant> declaration for every <forEach> activity listed with a matching *forEach* attribute.

Algorithm 14 presents the refined procedure `modifyConstruct` of algorithms 9 and 12. Again, the beginning and the end of the code section which is refined is indicated by extra comments, and some code elements of the original procedure are left out indicated by three dots. The procedure uses the procedure `modifyForEach` of algorithm 15 to modify a set-based <forEach> activity in the way described in Section 4.1. Algorithm 17 on page 73 presents the refined procedure `convertPBD` of algorithm 5. In addition to the original procedure, the list `PaSetList` is declared as list of sets of participant references over which the set-based <forEach> activities in the current process iterate. They are used as names of variables containing an array of endpoint references. To declare these variables the procedure `declareSrefVariables` of algorithm 18 is called at the end of the procedure `convertPBD`. In addition to the assumptions made for algorithms 1 to 13, we assume the following:

- There are the following functions on elements of the type `Element`:
  - `addContent( text :String )`: adds the character string stored in the variable `text` to the content of the element.
  - `addVariables()`: this function adds a <variables> declaration at the right place of the element taking the BPEL specification ([Org07]) into account. If the element already includes a <variables> declaration, it will not be changed.
  - `removeChild( element :Element )`: this function removes the child element `element`. If `element` is not a child element of the current element, the latter will not be changed.
  - `removeChildren()`: this function removes all child elements of the element.

If the current <forEach> activity has a *wsu:id* attribute assigned, the procedure `modifyForEach` will be started with the <forEach> activity, the <scope> activity and the value of the *wsu:id* attribute of the <forEach> activity as input. Note that the value of the *wsu:id* attribute is now used twice. Therefore, it is stored in a variable. In that way, we only need one access to the attribute instead of two. The procedure `modifyForEach` first builds the global name of the current <forEach> activity by concatenating the previously stored name space prefix of the target name space of the PBD with a colon and the local name of the activity. If the resulting QName is an element of *Scope*, a set of participant references will be associated with the <forEach> activity by a *forEach* attribute. Thus, it is a set-based <forEach> activity. Otherwise, it will not be a set-based <forEach> activity and does not need to be changed.

Afterward, the *counterName* attribute is added to the <forEach> activity. Then, the <startCounterValue> and <finalCounterValue> constructs are created. The name of the set of participant references over which the <forEach> activity iterates can be determined by using the function $set_{forEach}$ on the element of *Scope* representing the activity. This name will be added to the global list `PaSetList` if it has not been added before. The corresponding variable containing an array of endpoint references will be declared later (cf. algorithms 16 and 17). The <startCounterValue> and <finalCounterValue> constructs need to be enclosed before the optional <completionCondition> construct and the <scope> activity of the <forEach> activity (cf. [Org07]). To do so, the <completionCondition> construct and the <scope> activity are removed from the activity. After having added the <startCounterValue> and <finalCounterValue> constructs, they are added to the end of the <forEach> activity again.

In Section 4.1 we have mentioned that the <assign> activity needs to be enclosed before the nested activity if and only if there is exactly one partner link declaration between the owner and the iterator of the <forEach> activity. Let $fe \in \mathit{Scope}$ be a <forEach> activity. If fe is a set-based <forEach> activity, there will be exactly one set of participant references over which fe iterates. Furthermore, there is exactly one single participant reference being the iterator of fe. Since the iterator is the only single participant reference which may be associated with the <forEach> activity by a *forEach* attribute, the set $\mathit{partnerLinks}_{\mathit{Scope}}(fe)$ only includes partner link declarations between the owner and the iterator of the <forEach> activity. If the amount of elements in this set is not one, the <assign> activity must not be enclosed before the nested activity (cf. Section 4.1). Otherwise, the procedure `addAssignsToForEach` will be started having the <scope> activity, the set of participant references over which the <forEach> activity iterates, the first and only element of the set of partner link declarations limited to the <forEach> activity and the value of its *wsu:id* attribute as input.

First, the nested activity is stored in a variable and removed from the <scope> activity. Then, the <copy> construct is specified and added to the new <assign> activity as described in Section 4.1. Afterward, the <assign> activity and the nested activity are added to the <sequence> activity in that order. Finally, the <sequence> activity is added to the <scope> activity.

If variables containing an array of endpoint references need to be declared, a name space declaration will be added to the <process> activity. This name space declaration associates the name space prefix `sref` with the name space `http://www.bpel4chor.org/service-references`. We assume that there is no other name space declaration associating the name space prefix `sref` with another name space (cf. beginning of Chapter 1). After the declaration of the name space, a <variables> construct will be added to the <process> activity if it does not already include one. For each element of the global list `PaSetList` a <variable> construct is added to the <variables> construct. The name of the variable is the name of the corresponding set of participant references stored in the list `PaSetList`. The type of each variable is `sref:service-refs` (cf. Section 4.1).

Algorithm 14 Refined procedure modifyConstruct

```
procedure modifyConstruct(construct :Element)
            // the input construct points on the tag of the current BPEL construct
    constructName :String ←construct.getName(); // the name of the current BPEL construct
    fEScope :Element;                                   // a single BPEL construct
begin
    ...
    if constructName = "forEach" then
                                             // construct is a <forEach> activity
        fEScope ←construct.getChild("scope");
        // fEScope points on the <scope> activity nested in the <forEach> activity
                                          // continue depth-first search on the scope
        modifyConstruct(fEScope);                                      // recursion
        if construct.hasAttribute("wsu:id") then

                                                     // here the refinement begins
            id :String ←construct.getAttributeValue("wsu:id");
            // the value of the wsu:id attribute is stored in id since it is used twice

                                          // modify the set-based <forEach> activity
            modifyForEach(construct, fEScope, id);              // cf. algorithm 15

                                        // add partner link declarations to the scope
            declarePartnerLinks(fEScope, id);                    // cf. algorithm 6
                                                       // here the refinement ends

        end if
    else
        ...
    end if
end procedure
```

Algorithm 15 Procedure modifyForEach

```
procedure modifyForEach(forEach, fEScope :Element, id :NCName)
        // the input forEach points on the tag of the current <forEach> activity, the input fEScope on the tag of the <scope> activity nested in the <forEach> activity, and the input id is its wsu:id
    fe :DT($Scope$);                                  // DT($Scope$) inherits of QName
    set :DT($Pa$);                                    // DT($Pa$) inherits of NCName
    startCounter :Element ←new Element("startCounterValue");     // the <startCounterValue>
    finalCounter :Element ←new Element("finalCounterValue"); // the <finalCounterValue>
    completionCondition :Element ←forEach.getChild("completionCondition");
      // the optional <completionCondition> construct of the current <forEach> activity
begin
    fe ←process_nsprefix.buildQName(id);
      // the global name of the current <forEach> activity and the corresponding element of $Scope$
    if fe ∈ $Scope$ then
              // a set of participant references is associated with fe by a forEach attribute
                                       // thus it is a set-based <forEach> activity
                                             // add the counterName attribute
        forEach.addAttribute("counterName", "i_" + id);
             // add the contents to the <startCounterValue> and the <finalCounterValue>
        startCounter.addContent("0");
        set ←$set_{forEach}$(fe);
        finalCounter.addContent("count($" + set + "/)-1");
      // $set_{forEach}$(fe) is the name of the set of participant references over which the current <forEach> activity iterates
              // add this name to the global list PaSetList if it has not been added before
          // The corresponding variable containing an array of endpoint references will be declared later (cf. algorithms 16 and 17)
        if set ∉ PaSetList then
            PaSetList ←PaSetList ∪ set;
        end if
      // enclose the <startCounterValue> and the <finalCounterValue> constructs before the <completionCondition> construct and the <scope> activity of the <forEach> activity
        forEach.removeChild(completionCondition);
        forEach.removeChild(fEScope);
        forEach.addChild(startCounter);
        forEach.addChild(finalCounter);
        forEach.addChild(completionCondition);
        forEach.addChild(fEScope);
        // add the <assign> activities to copy the endpoint references on the partner links
        plList :List(DT($PL$)) ←$partnerLinks_{Scope}$(fe);
        if |plList| = 1 then
            addAssignsToForEach(fEScope, set, plList[1], id);       // cf. algorithm 16
        end if
    end if
end procedure
```

Algorithm 16 Procedure addAssignsToForEach

```
procedure addAssignsToForEach(fEScope :Element, set :DT($Pa$), pl :DT($PL$),
id :NCName)
        // the input fEScope points on the tag of the <scope> activity nested in the
current <forEach> activity, the input set is the set participant references over which
the <forEach> activity iterates, the input pl is the partner link declaration between
the owner and the iterator of the <forEach> activity, and the input id is its wsu:id

    nestedActivity :Element ←fEScope.getChildren().last();
            // the last child of the <scope> activity → this is the nested activity
    sequence :Element ←new Element("sequence");  // the new <sequence> activity
    assign :Element ←new Element("assign");            // the new <assign> activity
    copy :Element ←new Element("copy");                // the new <copy> construct
    from :Element ←new Element("from");                // the new <from> construct
    query :Element ←new Element("query");             // the new <query> construct
    to :Element←new Element("to");                          // the new <to> construct
begin
                                                            // remove the nested activity
    fEScope.removeChild(nestetActivity);

                                                          // specify the <copy> construct
    from.addAttribute("variable", set);
    query.addContent("[$i_" + id + "]");
    to.addAttribute("partnerLink", pl);

            // arrange the <copy> construct and add it to the <assign> activity
    from.addChild(query);
    copy.addChild(from);
    copy.addChild(to);
    assign.addChild(copy);

                        // add the <assign> activity to the new <sequence> activity
    sequence.addChild(assign);
                         // add the nested activtiy to the new <sequence> activity
    sequence.addChild(nestedActivity);
                          // add the <sequence> activity to the <scope> activity
    fEScope.addChild(sequence);
end procedure
```

Algorithm 17 Refined procedure convertPBD

```
procedure CONVERTPBD(process :Element)
                // the input process points on the <process> activity of the current PBD
    ...
    PaSetList :List(DT($Pa$)) ←∅;
            // a list of sets of participant references over which the set-based <forEach>
  activities in this process iterate
        // they are used as names of variables containing an array of endpoint references
begin
    ...
                                                          // here the refinement begins
                        // declare the variables containing an array of endpoint references
    DECLARESREFVARIABLES(process, PaSetList);                    // cf. algorithm 18
end procedure
```

Algorithm 18 Procedure declareSrefVariables

```
procedure declareSrefVariables(construct :Element, PaSetList :List(DT($Pa$)))
        // the input construct points on the tag of the current BPEL construct, and
  PaSetList is the list of sets of participant references for which a variable containing
  the corresponding array of endpoint references needs to be declared
    variables, variable :Element;                          // single BPEL constructs
    PaSet : DT($Pa$);                                       // a participant reference
begin
    if PaSetList $\neq \emptyset$ then                      // there are variables to be declared
                   // add the name space declaration for the data type service-refs
        srefsNamespace :String ←"http://www.bpel4chor.org/service-references";
        construct.addAttribute("xmlns:srefs", srefsNamespace);
                                                       // add a <variables> declaration
        construct.addVariables();
        variables ←construct.getChild("variables");
                                      // variables becomes the <variables> declaration
                                                       // declare all necessary variables
        for all PaSet $\in$ PaSetList do
            variable ←new Element("variable");
                                      // variable becomes a new <variable> declaration
                   // add a name and a type attribute to the <variable> declaration
            variable.addAttribute("name", PaSet);
            variable.addAttribute("type", "srefs:service-refs");
                   // add the <variable> declaration to the <variables> declaration
            variables.addChild(variable);
        end for
    end if
end procedure
```

5. Generating WSDL Definitions of Partner Link Types

The WSDL definitions of partner link types are written to a new WSDL file. The root element of this file is a <wsdl:definitions> tag. The name space prefix `wsdl` refers to the name space `http://schemas.xmlsoap.org/wsdl/` (cf. [Org07]).

Algorithm 19 presents a procedure in pseudo code which adds one partner link type declaration for each element of the set *PLType* to the new WSDL file. It is used on the newly created <wsdl:defnitions> tag. To add roles to the partner link type declarations, the procedure `declareNewRole` of algorithm 20 is used. For algorithms 19 and 20 we assume the same data types and functions as we did for algorithms 1 to 18.

First of all, the <wsdl:definitions> tag gets a *name* and a *targetNamespace* attribute assigned. The value of the *name* attribute is "partnerLinkTypes". As target name space of the declarations of partner link types we chose the target name space of the participant topology concatenated by the value "/partnerLinkTypes" (cf. algorithm 5 in Section 2.3). We assume that there are no conflicts with existing name spaces and the newly created one (cf. beginning of Chapter 1).

Afterward, the name space declarations for the name space prefixes `wsdl` and `plnk` are added to the <wsdl:definitions> tag. The name space prefix `wsdl` needs to refer to the name space `http://schemas.xmlsoap.org/wsdl/` and the name space prefix `plnk` to the name space `http://docs.oasis-open.org/wsbpel/2.0/plnktype` (cf. [Org07]).

For each element of the set *PLType* a new <plnk:partnerLinkType> declaration needs to be added to the <wsdl:definitions> tag. Let $plt \in PLType$ be the current partner link type. Then, the name of the new <plnk:partnerLinkType> declaration is the identifier of plt. Since each element of the set *PLType* is named uniquely, there are no naming conflicts between the individual partner link type declarations.

The roles of the partner link type declaration can be derived from the element of the relation *Comm* which is associated with *plt* by the function plt_{Comm} of definition 29. Let $((A, c), (b, d)) \in Comm$ be ${plt_{Comm}}^{-1}(plt)$. Since plt_{Comm} is a bijective function, it inverse function can be used. Each role is declared in a <plnk:role> construct added to the current <plnk:partnerLinkType> construct. The name of the first role is the identifier of the participant reference b. The value of its *portType* attribute is the identifier of the port type d. Note that b is an NCName, and d is a QName.

If we have $c = \perp$, the participant references contained in A will not receive any messages from the participant reference b. In that case, the second role must not be declared.

If we have $c \neq \perp$, A will contain only one participant reference since the receiver of a message link has to be a single participant reference (cf. Section 2). In that case, the name of the second <plnk:role> construct is the identifier of the first and only element of the subset of participant references A. The value of its *portType* attribute is the identifier of the port type c.

After having declared each partner link type with its roles, we need to add name space declarations to the <wsdl:defintions> tag. These name space declarations refer to the name spaces of the WSDL definitions of port types associated with the roles. To do so, we will add each name space prefix of a currently associated port type to the global list of name space prefixes `nsprefixList` if it has not been added before. The name space prefix of a port type can be derived using the function $nsprefix_{PT}$ of definition 3.18 on it. Later, the procedure `declareNameSpaces` of algorithm 8 is used on the <wsdl:definitions> tag and the list `nsprefixList` to add the relevant name space declarations.

Algorithm 19 Procedure declarePartnerLinkTypes

```
procedure DECLAREPARTNERLINKTYPES(definitions :Element)
        // the input definitions points on the <wsdl:definitions> tag of the newly created WSDL file
    plt :DT($PLType$);                                        // DT($PLType$) inherits of NCName
    partnerLinkType, role :Element;                           // single BPEL constructs
    A :List(DT($Pa$))                                         // a list of participant references
    b :DT($Pa$);                                              // DT($Pa$) inherits of NCName
    c, d :DT($PT$);                                           // DT($PT$) inherits of QName
    nsprefixList :List(DT($NSPrefix$)) $\leftarrow\emptyset$;
        // a list of name space prefixes referencing to the name spaces of the WSDL definitions of port
    types used by the new partner link types
begin
                                            // add a name attribute to the <definitions> tag
    definitions.addAttribute("name", "partnerLinkTypes");
                                            // declare the target name space for the WSDL definitions
    targetNS :String $\leftarrow prefix_{NS}(topologyNS)$ + "/partnerLinkTypes";
                        // $prefix_{NS}(topologyNS)$ is the target name space of the participant topology
    definitions.addAttribute("targetNamespace", targetNS);    // cf. algorithm 5
        // add the name space declarations for the name space prefixes wsdl and plnk (cf. [Org07])
    definitions.addAttribute("xmlns:wsdl", "http://schemas.xmlsoap.org/wsdl/");
    plnkNS :String $\leftarrow$"http://docs.oasis-open.org/wsbpel/2.0/plnktype";
    definitions.addAttribute("xmlns:plnk", plnkNS);

                            // add a partner link type declaration for each element of $PLType$
    for all plt $\in$ $PLType$ do
                                            // create a new <plnk:partnerLinkType> construct
        partnerLinkType $\leftarrow$new Element("plnk:partnerLinkType");
                            // add a name attribute to the new partner link type declaration
        partnerLinkType.addAttribute("name", plt);

                    // get the element of the relation Comm of the current partner link type
        ((A,c),(b,d)) $\leftarrow {plt_{Comm}}^{-1}$(plt);              // $plt_{Comm}$ is a bijective function
                    // create the first role element and add it to the partner link type declaration
        role $\leftarrow$DECLARENEWROLE(b, d);                    // cf. algorithm 20
        partnerLinkType.addChild(role);
                                    // the second role must not be declared if we have c = $\perp$
        if c $\neq \perp$ then
            role $\leftarrow$DECLARENEWROLE(A[1], c);             // cf. algorithm 20
                    // in this case, A includes only one participant reference (cf. Section 2)
            partnerLinkType.addChild(role);
        end if
                    // add the partner link type declaration to the <definitions> construct
        definitions.addChild(partnerLinkType);
    end for
                    // add the declarations of the name spaces of the port type definitions
    DECLARENAMESPACES(definitions, nsprefixList);             // cf. algorithm 8
end procedure
```

Algorithm 20 Function declareNewRole

```
function DECLARENEWROLE(pa :DT($Pa$), pt :DT($PT$)) returns Element
        // the input pa is the participant reference which may be interpreted as the new
    role, the input pt is the port type of this role
                                  // this function returns the new <plnk:role> construct
    role :Element ←new Element("plnk:role");          // a new <plnk:role> construct
    pt_nsprefix :DT($NSPrefix$) ←$nsprefix_{PT}$(pt); // the name space prefix of the port
    type pt
begin
                                                    // add the name attribute to the role
    role.addAttribute("name", pa);                                // pa is an NCName
                                                // add the portType attribute to the role
    role.addAttribute("portType", pt);                              // pt is a QName
        // add the name space prefix of pt to the global list nsprefixList if it has not
    been added before
    if pt_nsprefix $\notin$ nsprefixList then
        nsprefixList ←nsprefixList $\cup$ pt_nsprefix;
    end if
                                               // return the new <plnk:role> construct
    return role;
end function
```

5.1. Endpoint References

In a completely grounded BPEL4Chor choreography each transmitted participant reference is associated with a WSDL property in the participant groundings. In the case of a single participant reference, the type of such a property is `sref:service-ref`. The name space prefix `sref` refers to the name space `http://docs.oasis-open.org/wsbpel/2.0/serviceref.service-ref` (cf. [Org07]). This universal container for storing an endpoint reference allows the usage of different versions of service referencing or endpoint addressing schemes within BPEL (cf. [Org07]). If the transmitted reference is a set of participant references, the type of the property will be `srefs:servicerefs`. The name space prefix `srefs` refers to the name space `http://www.bpel4chor.org/servicereferences` (cf. Section 4.1). In a BPEL process we need to pass on endpoint references instead of participant references. If the receiver of a participant reference sends a message to the transmitted reference, the corresponding endpoint reference needs to be copied to the *partnerRole* of the partner link between the receiver and the reference. This ensures that the message is sent to the right participant.

In the following, we distinguish explicitly between two different cases regarding the behavior of the receiver of a transmitted participant reference. The first case is presented in Figure 8, and the second one in Figure 9. Figure 10 presents a combination of both cases.

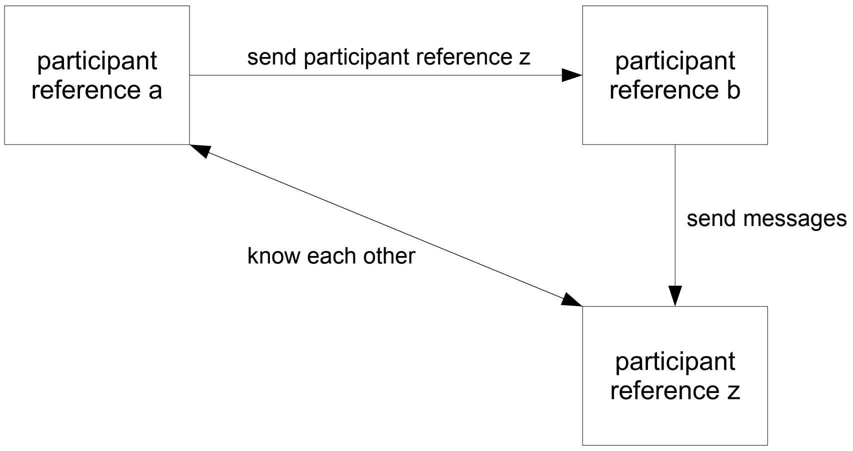

Figure 8.: Sending and using a participant reference

In the scenario of Figure 8 a participant reference `a` knows another participant reference `z` and sends it to the participant reference `b`. The latter uses the transmitted participant reference to send one or more messages to it.

The scenario of Figure 9 differs from the one of Figure 8 in that the participant reference `b` does not use the transmitted participant reference `z` to communicate with it. Instead,

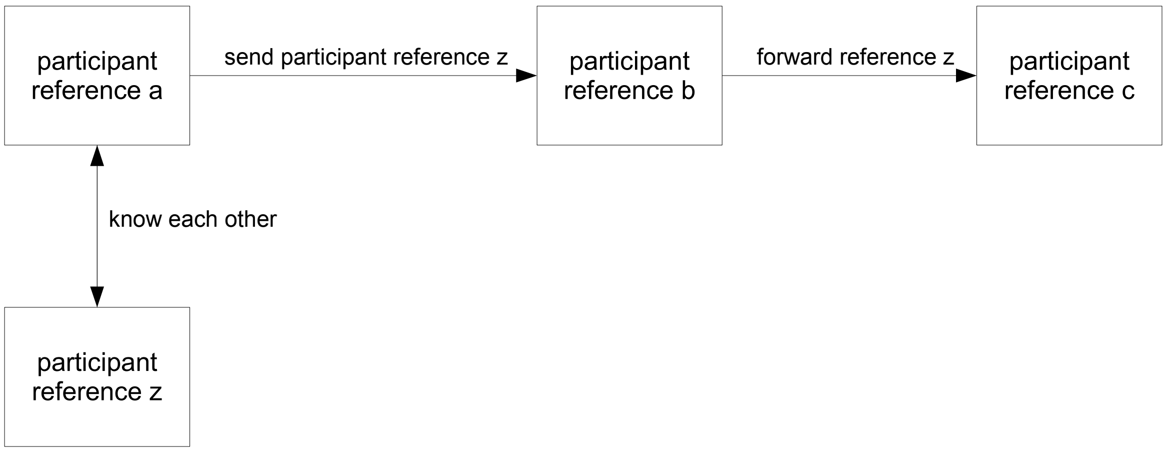

Figure 9.: Sending and forwarding a participant reference

it forwards `z` to another participant reference `c`.

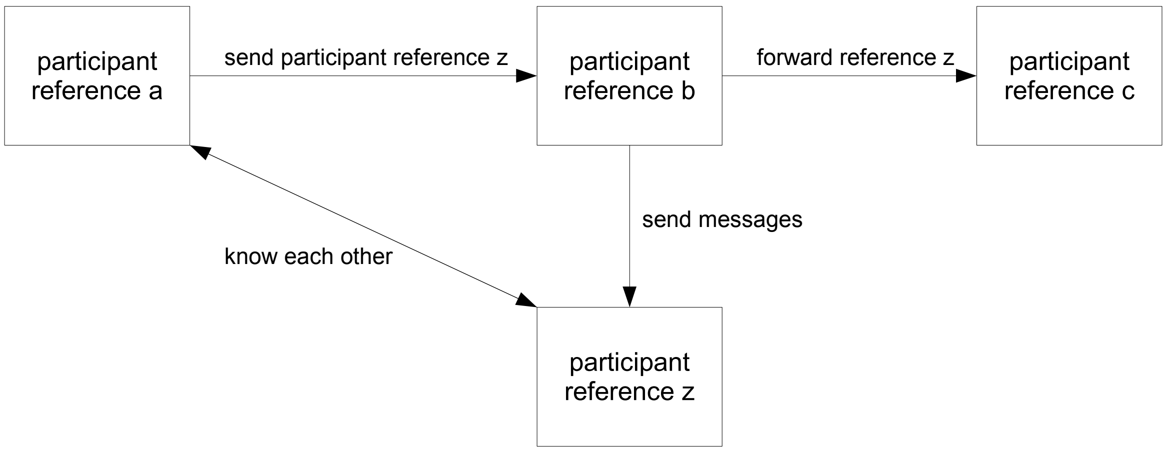

Figure 10.: Sending, using and forwarding a participant reference

Finally, Figure 10 presents a scenario in which the participant reference `b` both sends one or more messages to the transmitted participant reference `z` and forwards it to another participant reference `c`.

A WSDL property can be found in the messages in which it is transmitted. The property is connected to the messages by property aliases also defined in WSDL. For each WSDL property there has to be one property alias definition for each WSDL message in which the property should be found (cf. [Org07]). If a send activity sending or forwarding a participant reference is reached, it will be assumed that the corresponding endpoint reference is stored in the message. Therefore, nothing additional can be done before or in the send activity.

If a participant reference uses the transmitted reference to send one or more messages to it, the corresponding endpoint reference needs to be copied to the partner link between them. Therefore, an <assign> activity has to be added after the receive activity. Note that the receiver of a participant reference will send one or more messages to it if and only if a partner link declaration between them exists (cf. Section 2). This fact can be used to determine whether the <assign> activity has to be added or not.

A single transmitted participant reference is associated with exactly one single endpoint reference. With this restriction the receiver of a reference actually receives only one endpoint reference. Thus, if the participant bound to the transmitted participant reference is realized by multiple port types, the receiver of the reference will possibly not be able to send him all messages. Hence, we furthermore restrict that the transmitted participant reference is realized by one port type when the reference is used to send messages to it. This implies that there is at most one partner link declaration between the receiver of a participant reference and the reference itself.

Furthermore, an endpoint reference is associated with a concrete port to which a message may be sent. The *partnerRole* of the partner link declaration to which the reference is copied needs to be associated with the port type of this port. Each transmitted participant reference may be realized by multiple port types. Thus, it is usually not known at design time with which concrete port type a transmitted endpoint reference is associated. If there are multiple partner link declarations between the receiver of a participant reference and the reference itself, we need to determine the partner link declaration which is associated with the correct port type. This is not always possible before the runtime of the process. Future work will investigate how multiple endpoint references can be transmitted using one participant reference.

Assume the participant references defined in listing 5.1 and the message links of listing 5.2. There are three single participant references. The pariticipant reference `a` sends the participant reference `z` to the participant reference `b`. Afterward, `b` sends a message to `z`.

```
<participants>
  <participant name="a" type="A" />
  <participant name="b" type="B" />
  <participant name="z" type="Z" />
</participants>
```

Listing 5.1: Participant references in our example

```
<messageLinks>
  <messageLink name="a−b"
    sender="a" sendActivity="sa"
    receiver="b" receiveActivity="rb"
    messageName="a−b"
    participantRefs="z" />
  <messageLink name="b−z"
    sender="b" sendActivity="sb"
    receiver="z" receiveActivity="rz"
    messageName="b−z" />
</messageLinks>
```

Listing 5.2: Message links including a transmitted participant reference

The groundings of these message links is presented in listing 5.3. The transmitted participant reference `z` is associated with the WSDL property `z:prop`.

```
<messageLinks>
  <messageLink name="a−b"
    portType="b:pt" operation="op" />
  <messageLink name="b−z"
    portType="z:pt" operation="op" />
</messageLinks>
<participantRefs>
  <participantRef name="z"
    WSDLproperty="z:prop" />
</participantRefs>
```

Listing 5.3: Code snippet of the participant groundings of our example

Listing 5.4 presents the new <assign> activity which needs to be added to the receive activity `rb`. In this example `rb` is a <receive> activity. Note that it might also be an <onMessage> or <onEvent> construct or a synchronous <invoke> activity. In each of the four cases, the <assign> activity has to be added after the message construct in the same way. Assume the receive activity stores the received message in the variable `rb_variable`. Then, the <assign> activity reads the endpoint reference out of that variable and copies it to the appropriate partner link. Since the participant reference *z* is realized by the port type `z:pt` to receive the messages from `b`, this partner link has the name `b-z_isRealizedBy_z_pt` (cf. Section 2). To read the endpoint reference out of the variable, a property alias of the property `z:prop` is used.

The <receive> and the <assign> activity in listing 5.4 are rewritten to a <sequence> activity. The optional links of the original <receive> activity are moved to the new <sequence> activity (indicated by the three dots). This ensures that the received endpoint reference is copied to the partner link after it has been received. In the PBD the semantics of the message links implied that behavior. In an abstract BPEL process, this behavior must be made explicit. The only way is to tie the <assign> activity to the <receive> activity in order to execute them as a group.

```
<sequence ...>
...
  <receive name="rb" variable="rb_variable" ... />
  <assign>
```

```
    <copy>
      <from variable="rb_variable" property="z:prop" />
      <to partnerLink="b−z_isRealizedBy_z_pt" />
    </copy>
  </assign>
</sequence>
```

Listing 5.4: Copying the endpoint reference to a partner link

If the transmitted participant reference is a set of participant references, the receiver cannot send a message to the whole set since the receiver of a message link has to be a single participant reference. Instead, the set can be forwarded to another participant reference (cf. Figure 9), or it can be used as set for a set-based <forEach> activity (cf. Section 4). In such a case, the endpoint references associated with the set need to be copied to a variable of the type `srefs:servicerefs`. To do so, the <to> construct of the <copy> construct needs to be adjusted. The target is not a partner link, but the corresponding variable. Assume that a message construct called `receive` receives the set of participant references `set`. Furthermore assume that `set` is associated with the WSDL property `s:prop`. Then, the <assign> activity copying the endpoint references to a variable is presented in listing 5.5.

```
<sequence ...>
...
  <receive name="receive" variable="receive_variable" ... />
  <assign>
    <copy>
      <from variable="receive_variable" property="s:prop" />
      <to variable="set" />
    </copy>
  </assign>
</sequence>
```

Listing 5.5: Copying the endpoint references to a variable

The name of the variable is the name of the corresponding set of participant references. It is declared globally in the <process> activity. We assume that there are no naming conflicts with existing variables (cf. beginning of Chapter 1).

The transmitted participant reference may be stored in another participant reference after its reception. This is indicated by a *copyParticipantRefsTo* attribute assigned to the message link declaration. Assume the modified message link declarations of listing 5.6. At the side of the receiver `b`, the transmitted participant reference `z` is stored in the participant reference `x`. Afterward, `b` sends a message to `x` using the message link `b-x`. In that case, the <to> construct of the <copy> construct in listing 5.4 uses the partner link between `b` and `x` as target. Note that now there has to be exactly one partner link declaration between `b` and `x`.

```
<messageLinks>
  <messageLink name="a−b"
    sender="a" sendActivity="sa"
    receiver="b" receiveActivity="rb"
    messageName="a−b"
    participantRefs="z"
    copyParticipantRefsTo="x" />
  <messageLink name="b−x"
    sender="b" sendActivity="sb"
    receiver="x" receiveActivity="rx"
    messageName="b−x" />
</messageLinks>
```

Listing 5.6: Message links including a *copyParticipantRefsTo* attribute

Again, if `b` only forwards the stored participant reference `x` and if `x` is a single participant reference, nothing additional can be changed at the side of `b`. If the transmitted participant reference is a set of participant references, the name of the variable which is the target of the <to> construct will change to `x`.

If more than one participant reference is transmitted over a message link, one <copy> construct for each reference needs to be added to the new <assign> activity. Each <copy> construct copies the corresponding endpoint reference to the relevant partner link or variable as described above.

Until now, we have described how the endpoint references are copied from the message variables to the partner links or variables containing an array of endpoint references. But some message constructs in the PBDs may have no message variables specified. In that case, a new variable needs to be created for both the send and the receive activity of a message link if endpoint references need to be copied either to partner links or to variables after the receive activity. The name of a newly created variable is the value of the *wsu:id* attribute of the message construct concatenated by the character string "_variable". We assume that there are no naming conflicts with existing variables (cf. beginning of Chapter 1).

A synchronous <invoke> activity occurs both as send and receive activity of two different message links. Its *inputVariable* is associated with the outgoing message and its *outputVariable* with the incoming message (cf. [Org07]). If both variables need to be created newly in the way described above, they will get the same name. Thus, we employ a different naming for these variables. We concatenate the character string "_inputVariable" to the *wsu:id* of the <invoke> activity to get the name of the *inputVariable*. The name of the *outputVariable* is the *wsu:id* concatenated by the character string "_outputVariable". For the sake of simplicity, this may be done for each <invoke> activity either being synchronous or asynchronous.

Different message constructs of one process may have the same *wsu:id* assigned. Since such message constructs are associated with the same WSDL operation, their message variables need to get the same message type assigned. Furthermore, they cannot be executed in parallel since at most one message may be sent over a message link at the same point in time. Thus, these message constructs may get the same message variables assigned without creating any conflict regarding different message types or the overwriting of endpoint references on variables.

In a BPEL process following the *Abstract Process Profile for Observable Behavior* the declarations of message variables may be omitted. Nevertheless, we present how these declarations can be created automatically. If a message variable is declared, its message type must match the type of the corresponding WSDL message. The latter can be derived from the WSDL definition of the operation assigned to the relevant message link. Note that both the outgoing variable of the send activity and the incoming variable of the receive activity of a message link need to get the same message type assigned (cf. [Org07]). If the receive activity is a synchronous <invoke> activity (and the send activity is a <reply> activity), the output message of the relevant WSDL operation needs to be used. Otherwise, we need to use the input message of this operation.

Note that the message variables of the PBDs may have already been declared by the modeler. In that case, it needs to be checked whether he has chosen the right message types. The declarations can be created automatically for newly created message variables. The same goes for message variables which have been assigned to message constructs, but which have not been declared before the automated transformation.

In which scopes the message variables need to be declared, depends on which scopes the transmitted participant references are limited to. Again, if the modeler has already declared a message variable, it needs to be checked whether it is declared in the right scope. Otherwise, this can be done automatically. Declaring the variables in their respective scopes prevents an overwriting of endpoint references in the messages if they are sent in parallel.

Assume that the participant reference `a` sends the participant reference `z` to the participant reference `b` as presented in listing 5.2. For the message variables, we do not need to distinguish between a single participant reference and a set of participant references explicitly. Thus, `z` can either be a single participant reference or a set of participant references. Furthermore assume that `b` does not store `z` into another participant reference using a *copyParticipantRefsTo* attribute. Then, we need to look at the following limitations regarding `z`:

1. `z` is not limited to any scope nested in the processes of `a` or `b`.
2. `z` is limited to a scope nested in the process of `a`, but not to any scope nested in the process of `b`.

3. `z` is limited to a scope nested in the process of `b`, but not to any scope nested in the process of `a`.
4. `z` is limited to both a scope nested in the process of `a` and a scope nested in the process of `b`.

In case 1 both the outgoing variable of the send activity and the incoming variable of the receive activity may be declared globally in the respective <process> activities. In case 2 the outgoing variable of the send activity needs to be declared in the corresponding scope. The incoming variable of the receive activity may be declared globally in the <process> activity of `b`. In case 3 the incoming variable of the receive activity needs to be declared in the corresponding scope. The outgoing variable of the send activity may be declared globally in the <process> activity of `a`. In case 4 both the outgoing variable of the send activity and the incoming variable of the receive activity need to be declared in their respective scopes.

Now we assume that `z` is again a single participant reference, but that `b` does store it into another participant reference $\texttt{x} \neq \texttt{z}$ using a *copyParticipantRefsTo* attribute. Copying `z` to `x` is done only at the side of `b`. Thus, any limitation of `x` to a scope nested in the process of `a` does not force us to declare the outgoing variable of the send activity in this scope. At the side of `a` only the limitations regarding `z` have to be taken into account.

If `x` is not limited to any scope nested in the process of `b`, the incoming variable of the receive activity may be declared globally in the <process> activity of `b`. Otherwise, it needs to be declared in the corresponding scope. Note that only the limitations regarding `x` have to be taken into account. The participant reference `z` may be limited to any scope at the side of `b`. But since the participant bound to `z` at the side of `a` is copied to `x`, such limitations do not force us to declare the variable in any scope. Actually, another participant than the transmitted one may be bound to the participant reference `z` at the side of `b`.

A participant reference may be limited to multiple scopes nested in the same process. In that case, the relevant message variables need to be declared in each of these scopes in which the send or respectively the receive activity of the message link is nested directly. The same goes if multiple participant references are transmitted over one message link and each of them is limited to a different set of scopes. Note that both the send and the receive activity can be found in their process multiple times having the same *wsu:id* assigned.

The <toPart> and <fromPart> constructs are an alternative way to variable attributes in message constructs. A <fromPart> construct is used to copy the content of a part of the incoming message to a BPEL variable. It can be used in <reveive> or synchronous <invoke> activities or in <onMessage> or <onEevent> constructs. It creates an anonymous temporary WSDL variable being of the type specified by the relevant WSDL operation's output or input message (cf. [Org07]). A <fromPart> construct must not be used to copy an endpoint reference to a partner link. Thus, if a receive activity receives a participant reference and the corresponding endpoint reference needs to be copied to a partner link, it will be mandatory that the activity has an incoming BPEL variable assigned by a *variable* or *outputVariable* attribute. In that case, the <fromPart> constructs need to be converted into an equivalent <assign> activity including one <copy> construct for each <fromPart> construct (cf. [Org07]).

Assume the <receive> activity presented in listing 5.7. It uses two <fromPart> constructs. The first construct copies the part `messagePart1` of the incoming message to the BPEL variable `variable1`. The second one does the same for the message part `messagePart2` and the BPEL variable `variable2`. The three dots within the <receive> activity indicate its optional links.

```
<receive name="r" ...>
...
       <fromParts>
              <fromPart part="messagePart1" toVariable="variable1" />
              <fromPart part="messagePart2" toVariable="variable2" />
       </fromParts>
</receive>
```

Listing 5.7: <receive> activity including <fromPart> constructs

Listing 5.8 presents the modified <receive> activity. Its <fromPart> constructs are removed. Instead, we get a new message variable (called `r_variable`) and a new <assign> activity including two <copy> statements. The first <copy> statement copies the message part `messagePart1` from the new message variable to the BPEL variable `variable1`. The second one does the same for the message part `messagePart2` and the BPEL variable `variable2`. Thus, the new <assign> activity is equivalent to the original <fromParts> construct.

Again, the <receive> and the <assign> activity are rewritten to a <sequence> activity. The optional links of the original <receive> activity are moved to the new <sequence> activity. This ensures that the message parts are copied to the relevant BPEL variables after the message has been received. The <copy> statements copying endpoint references to partner links or variables can be added to the new <assign> activity as described above.

```
<sequence ...>
...
  <receive name="r" variable="r_variable" ... />
  <assign>
    <copy>
      <from variable="r_variable" part="messagePart1" />
      <to variable="variable1" />
    </copy>
    <copy>
      <from variable="r_variable" part="messagePart2" />
      <to variable="variable2" />
    </copy>
    <!-- here the <copy> statements copying endpoint references to partner links or
        variables can be added -->
  </assign>
</sequence>
```

Listing 5.8: Converting the <fromPart> constructs into an equivalent <assign> activitiy

If a set of participant references `set` is transmitted over the message link, it is not necessary to convert the <fromPart> constructs into a new <assign> activity. Listing 5.9 presents how a new <fromPart> construct can be used to copy the endpoint references to the corresponding variable of the type `srefs:servicerefs`. Assume that `EPRs` is the part of the message in which the endpoint references are transmitted. Then, the third <fromPart> construct copies this message part to the variable `set` containing the array of endpoint references.

```
<receive name="r" ...>
...
        <fromParts>
                <fromPart part="messagePart1" toVariable="variable1" />
                <fromPart part="messagePart2" toVariable="variable2" />
                <fromPart part="EPRs" toVariable="set" />
        </fromParts>
</receive>
```

Listing 5.9: <receive> activity with a new <fromPart> construct

This alternative way in the case of a set of participant references requests knowledge about the definitions of WSDL messages and their message parts. Optionally, the <fromPart> constructs can be converted into an <assign> activity as described in listing 5.8, too. In that case, the locating mechanism of WSDL properties and their property aliases can be used to find the endpoint references in the message. Since <fromPart> constructs, as a group, act as a single virtual <assign> activity (cf. [Org07]), the same semantics is achieved by both variants. Which variant should be used, depends on the degree of knowledge about WSDL definitions of messages and message parts and on the attitude of the converter. The properties can be derived directly from the participant groundings. To get the message parts, we also need to analyze the WSDL definitions of messages. Furthermore, there might be more than one message part for an array of endpoint references in the same message. Then, the definitions of properties and property aliases need to be analyzed in order to determine which set of participant references is associated with witch message part. Therefore and for the sake of simplicity, we recommend the first alternative way converting the <fromPart> constructs into an <assign> activity.

A <toPart> construct is used to copy the content of a BPEL variable to a part of the outgoing message. It can be used in <invoke> or <reply> activities. It creates an anonymous temporary WSDL variable being of the type specified by the relevant WSDL operation's input or output message (cf. [Org07]). If an endpoint reference needs to be copied from a variable to the message, an additional <toPart> construct can be used for this. Similar to a <fromPart> construct, a <toPart> construct must not be used to copy an endpoint reference from a partner link to the message. Again, if a send activity sends a participant reference and the corresponding endpoint reference needs to be copied from a partner link to the message, it will be mandatory that the activity has an outgoing BPEL variable assigned by a *variable* or *inputVariable* attribute. In that case, the <toPart> constructs need to be converted into an equivalent <assign> activity including one <copy> construct for each <toPart> construct (cf. [Org07]). As mentioned above, it is assumed that this has already been done beforehand. It can be done in a similar way as in the case of <fromPart> constructs, but the <assign> activity has to be added before the send activity. This ensures that the message parts are copied to the message before it is sent.

6. Summary and Future Work

This work has described how the PBDs of a BPEL4Chor choreography can be transformed into BPEL processes following the *Abstract Process Profile for Observable Behavior*. The next two sections state a summary of this work and an outlook to future work.

Summary

The main task of this work was to find an automated procedure that fulfills the transformation. In Section 2 we have described how parter link declarations and the *partnerLink*, *portType*, and *operation* attributes at the message constructs are generated. Section 3 has dealt with replacing the NCNames of correlation properties by the corresponding QNames of WSDL properties and Section 4 with converting a set-based <forEach> activity into a <forEach> activity iterating over a BPEL variable. Section 5 has presented how the WSDL definitions of partner link types can be generated automatically. All other WSDL definitions (WSDL messages, port types, operations and properties and property aliases for endpoint references and correlation sets) have to be generated before starting the automated transformation. Finally, transmitted participant references have to be converted into the corresponding endpoint references. A transmitted endpoint reference needs to be copied to the *partnerRole* of the partner link between the receiver of the reference and the transmitted participant reference if they want to communicate. This has been described conceptionally in Section 5.1. The integration of this part of the transformation is out of scope for this work.

The underlying procedure of the transformation is subdivided into four steps. In the first step the participant topology is analyzed, and all relevant data needed to execute the transformation are stored. Afterward, the participant groundings are analyzed, and the data are extended. The third step represents the transformation of the PBDs into BPEL Abstract Processes following the *Abstract Process Profile for Observable Behavior*. The missing information for this step is derived from the data stored in the first two steps. In the last step the WSDL definitions of partner link types are created and written to a new WSDL file.

Future Work

The transformation introduced in Chapter 1 represents step 2 of Figure 1. Future work will include detailed investigations whether steps 1 and 3 can be realized by an automatic or semi-automatic transformation.

As example for Figure 1, we are going to model the RosettaNet PIPs in BPEL4Chor and show their transformation into BPEL Abstract Processes. Rania Khalaf of the IBM TJ Watson Research Center in Hawthorne, New York has developed another procedure transforming RosettaNet PIPs to BPEL Executable Processes (cf. [Kha07]). In the context of modeling the PIPs in BPEL4Chor, this procedure will be compared to our procedure.

Some points regarding the transformation from BPEL4Chor to BPEL are out of scope for this work. They can serve as basis for future work. These points are the following:

- The conversion of participant references to endpoint references (cf. Section 5.1) needs to be completed and integrated into the automated transformation. The points which need to be included in the transformation are amongst others that <assign> activities need to be added to the send activities sending or forwarding participant references. These <assign> activities copy the endpoint references from partner links or variables to the respective messages.
- A transmitted participant reference is associated with exactly one endpoint reference. This will create a conflict if the transmitted participant reference is realized by multiple port types, and if the reference is used to send messages to it (cf. Section 5.1). Future work will investigate how multiple endpoint references can be transmitted using one participant reference, and how this can be used in the transformation.
- In the case of a *senders* attribute assigned to a message link, we will committ the modeler to assign a participant reference to the message link by a *bindSenderTo* attribute if the partner link declaration at receiver side needs to be declared within a scope (cf. Section 2). The participant reference associated with the *bindSenderTo* attribute needs to be limited to the relevant scope. As subject to future work, one could investigate whether there is an alternative way to declare the partner link declaration of the receiver within a scope. If the sending participant references are limited to different scopes, the partner link declaration may be associated with a set of scopes. Then, it has to be declared in the right of these scopes.
- Over and above that, a participant reference may be limited to multiple scopes. In the automated transformation this has been disregarded (cf. Section 2). Thus, it also needs to be integrated.

In top-down scenarios, we usually start with an abstract business process description which will be converted into executable processes (cf. Figure 1). In such scenarios, the WSDL definitions of messages, port types, operations and properties and property aliases for endpoint references and correlation sets do not exist. Thus, it is desirable to generate them automatically. As we do not have any information about the data types of the correlation properties, their WSDL definitions cannot be generated automatically from a BPEL4Chor choreography. This has to be done beforehand. The automatic generation of the other WSDL definitions may be integrated as switch for the procedure.

In the context of the transformation, we have assumed that the BPEL4Chor choreography is correct regarding its semantics. An example for missing correctness is that a participant receives a participant reference and does not use it for anything (cf. Figure 11). To verify a BPEL4Chor choreography, an additional tool may be developed.

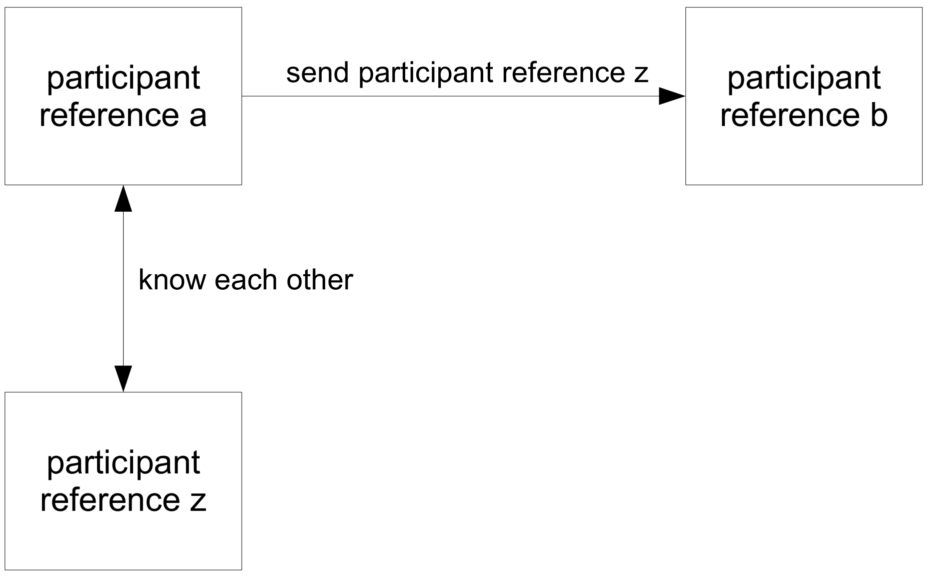

Figure 11.: Sending and not using or forwarding a participant reference

Acknowledgments

This work is a modified version of the student thesis by Peter Reimann [Rei07]. Oliver Kopp is funded by the German Federal Ministry of Education and Research (project Tools4BPEL, number 01ISE08B).

A. Summary of all Definitions

No.	Definition	Explanation
(1)	NS	The set of name spaces.
(2)	$NSPrefix$	The set of name space prefixes.
(3)	$prefix_{NS} : NSPrefix \rightarrow NS$	The bijective function that assigns a name space to each name space prefix.
(4)	$PaType$	The set of participant types.
(5)	$Process$	The set of BPEL processes (PBDs).
(6)	$process_{PaType} : PaType \rightarrow Process$	The bijective function that assigns a PBD to each participant type.
(7)	$nsprefix_{Process} : Process \rightarrow NSPrefix$	The function that assigns a name space prefix to each PBD of which the target name space is associated with this name space prefix.
(8)	Pa	The set of participant references.
(9)	$type_{Pa} : Pa \rightarrow PaType$	The function that assigns a participant type to each participant reference.
(10)	$Scope$	The set of the scopes and <forEach> activities of all PBDs that have an wsu:id and that are referenced by one or more (single) participant references.
(11)	$scope_{Pa} : Pa \rightarrow Scope \cup \{\perp\}$	The function that assigns a scope to each participant reference which is limited to this scope.
(12)	MC	The set of the message constructs of all PBDs.
(13)	ML	The set of message links.
(14)	$constructs_{ML} : ML \rightarrow MC \times MC$	The function that assigns a send and a receive activity to each message link.
(15)	$parefs_{ML} : ML \rightarrow 2^{Pa} \times Pa$	The function that assigns a set of senders and a receiver to each message link.
(16)	$bindSenderTo_{ML} : ML \rightarrow Pa \cup \{\perp\}$	The function that assigns a participant reference to each message link of which the actual sender should be bound to this reference.
(17)	PT	The set of WSDL port types.

(18)	$nsprefix_{PT} : PT \rightarrow NSPrefix$	The function that assigns a name space prefix to each port type which is declared in the name space which is associated with this name space prefix.
(19)	O	The set of WSDL operations.
(20)	$portType_{MC} : MC \rightarrow PT$	The function that assigns a WSDL port type to each message construct.
(21)	$operation_{MC} : MC \rightarrow O$	The function that assigns a WSDL operation to each message construct.
(22)	PL	The set of partner link declarations.
(23)	$partnerLink_{MC} : MC \rightarrow PL$	The function that assigns a partner link declaration to each message construct.
(24)	$partnerLinks_{scope} : (Scope \cup Process) \rightarrow 2^{PL}$	The function that assigns a set of partner link declarations to each scope or process in which these partner link declaration will be enclosed.
(25)	$PLType$	The set of partner link types.
(26)	$type_{PL} : PL \rightarrow PLType$	The function that assigns a partner link type to each partner link declaration.
(27)	$Comm \subseteq (2^{Pa} \times (PT \cup \{\perp\})) \times (Pa \times PT)$	This relation associates a combination of participant references with a (pair of) port type(s) which they use to communicate.
(28)	$partnerLinks_{Comm} : Comm \rightarrow PL \times PL$	The function that assigns a pair of partner link declarations to each element of the relation *Comm*.
(29)	$plt_{Comm} : Comm \rightarrow PLType$	The bijective function that assigns partner link type to each element of the relation *Comm*.
(30)	$myRole_{PL} : PL \rightarrow Pa \cup \{\perp\}$	The function that assigns a *myRole* to each partner link declaration.
(31)	$partnerRole_{PL} : PL \rightarrow Pa \cup \{\perp\}$	The function that assigns a *partnerRole* to each partner link declaration.
(32)	$CorrPropName$	The set of NCNames that are used as names of correlation properties and that are referenced to WSDL properties in the participant groundings.
(33)	$Property$	The set of WSDL properties which are referenced in the participant groundings.
(34)	$property_{CorrPropName} : CorrPropName \rightarrow Property$	The function that assigns a property to each property name.

(35)	$\mathit{nsprefix}_{\mathit{Property}} : \mathit{Property} \rightarrow \mathit{NSPrefix}$	The function that assigns a name space prefix to each WSDL property.
(36)	$\mathit{set}_{\mathit{forEach}} : \mathit{Scope} \rightarrow \mathit{Pa} \cup \{\perp\}$	The function that assigns a set of participant references to each <forEach> activity which iterates over this set.

Table 2.: Summary of all Definitions

B. Summary of all Data Types and their Functions

- There is the following function modifying elements of the data type `String`, which represents a data type of character strings:
  - `replaceColons( string :String ) returns String`: this function replaces each colon in the character string `string` by an underline.
- We assume `QName` to be a data type for QNames. `QName` inherits of the data type `String`. There is the following function on elements of the type `QName`:
  - `removeNSPrefix() returns NCName`: this function returns the second NC-Name of the QName. This means that it removes the name space prefix from the QName.
- We assume `NCName` to be a data type for NCNames. `NCName` inherits of the data type `String`. There is the following function on elements of the type `NCName`
  - `buildQName( name :NCName ) returns QName`: this function builds a QName by concatenating the NCName with a colon and the NCName `name`.
- We assume `DT(S)` to be a data type for the elements of the mathematical set S. S may be any set of the definitions of this chapter. `DT(S)` inherits of the type of the identifiers of the elements of S (`NCName or QName`).
- We assume `DT(`*Comm*`)` to be a data type for the elements of the mathematical relation *Comm*. There is the following function on elements of the type `DT(`*Comm*`)`:
  - `changeFirstPortType( pt : DT(`PT` ))`: let $comm = ((A, c), (b, d))$ be an element of the relation *Comm*. If we use this function on the element *comm*, the port type c of it will change to pt.
- We assume `Element` to be a data type for a node of a tree which represents an xml-file. So, an element of the type `Element` represents a tag within an xml-file. There are the following functions on elements of the type `Element`:
  - `new Element( name :String ) returns Element`: this function is a constructor that creates a new element having the local name `name`.
  - `getName() returns String`: this function returns the local name of the element, e. g. a <scope> activity will return the character string "scope".

- `getAttributeValue( name :String ) returns String`: this function returns the value of the attribute having the name `name`. It will return ⊥ if the element has no attribute assigned having the name `name`.
- `getAttributeValueAsList( name :String ) returns List(String)`: this function returns the value of the attribute having the name `name` as list of NC-Names or QNames. Each NCName or QName of the value of the attribute is one element of the list. The function will return an empty list if the element has no attribute assigned having the name `name`.
- `getAttributeNamespacePrefix( name :String ) returns NCName`: this function returns the name space prefix of the attribute having the name `name`, which will be the first NCName of its value if this is a QName. It will return ⊥ if there is no attribute having the name `name` or if the attribute value does not include a name space prefix.
- `hasAttribute( name :String ) returns Boolean`: this function will return `true` if the element has an attribute assigned having the name `name`. Otherwise, it will return `false`.
- `addAttribute( name :String, value :String )`: this function adds an attribute having the name `name` and the value `value`. If an attribute having the name `name` already exists, its value will be overwritten by `value`.
- `addAttribute( name :String, value :List(String) )`: this function adds an attribute having the name `name`. The input variable `value` needs to be either a list of NCNames or a list of QNames. As value the attribute gets the corresponding list, while each element of it is separated by a space. If an attribute having the name `name` already exists, its value will be overwritten by the list `value` accordingly.
- `removeAttribute( name :String )`: this function removes the attribute having the local name `name`.
- `getChild( name :String ) returns Element`: this function returns the first child element having the local name `name`. It will return ⊥ if there is no child element having the name `name`.
- `getChildren() returns List(Element)`: this function returns a list of all child elements of the element. It will return an empty list if there is no child element.
- `addChild( element :Element )`: this function adds a new child element after the existing child elements.

- `addPartnerLinks()`: this function adds a <partnerLinks> declaration at the right place of the element taking the BPEL specification ([Org07]) into account. If the element already includes a <partnerLinks> declaration, it will not be changed.
- `addVariables()`: this function adds a <variables> declaration at the right place of the element taking the BPEL specification ([Org07]) into account. If the element already includes a <variables> declaration, it will not be changed.
- `removeChild( element :Element )`: this function removes the child element `element`. If `element` is not a child element of the current element, the latter will not be changed.
- `removeChildren()`: this function removes all child elements of the element;
- `addContent( text :String )`: adds the character string stored in the variable `text` to the content of the element.

Bibliography

[CD99] J. Clark, S. DeRose. *XML Path Language (XPath) Version 1.0*. World Wide Web Consortium Recommendation, November 16th, 1999.

[DKLW07] G. Decker, O. Kopp, F. Leymann, M. Weske. BPEL4Chor: Extending BPEL for Modeling Choreographies. In *ICWS*, pp. 296–303. IEEE Computer Society, 2007.

[DKLW09] G. Decker, O. Kopp, F. Leymann, M. Weske. Interacting services: From specification to execution. *Data & Knowledge Engineering*, 68(10):946–972, 2009. doi:10.1016/j.datak.2009.04.003.

[Kha07] R. Khalaf. From RosettaNet PIPs to BPEL processes: A three level approach for business protocols. *Data & Knowledge Engineering*, 61(1):23–38, 2007.

[KML08] O. Kopp, R. Mietzner, F. Leymann. Abstract Syntax of WS-BPEL 2.0. Technical Report 2008/06, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2008.

[Org07] Organization for the Advancement of Structured Information Standards (OASIS). *Web Services Business Process Execution Language Version 2.0, OASIS Standard*, 2007.

[Rei07] P. Reimann. Generating BPEL Processes from a BPEL4Chor Description. Student Thesis: University of Stuttgart, Institute of Architecture of Application Systems, 2007.