BPEL4Chor: Extending BPEL for Modeling Choreographies

Gero Decker[1], Oliver Kopp[2], Frank Leymann[2], Mathias Weske[1]
[1]Hasso-Plattner-Institute, University of Potsdam, Germany
{gero.decker,mathias.weske}@hpi.uni-potsdam.de
[2]Institute of Architecture of Application Systems, University of Stuttgart, Germany
{oliver.kopp,frank.leymann}@iaas.uni-stuttgart.de

Abstract

The Business Process Execution Language (BPEL) is a language to orchestrate web services into a single business process. In a choreography view, several processes are interconnected and their interaction behavior is described from a global perspective. This paper shows how BPEL can be extended for defining choreographies. The proposed extensions (BPEL4Chor) distinguish between three aspects: (i) participant behavior descriptions, i.e. control flow dependencies in each participant, (ii) the participant topology, i.e. the existing participants and their interconnection using message links and (iii) participant groundings, i.e. concrete configurations for data formats and port types. As BPEL itself is used unchanged, the extensions facilitate a seamless integration between service choreographies and orchestrations. The suitability of the extensions is validated by assessing their support for the Service Interaction Patterns.

1 Introduction

The service oriented architecture (SOA) is an architectural style for building software systems based on services. Services are loosely coupled components described in a uniform way that can be discovered and composed. One realization of a SOA is the web services platform architecture where services are offered as web services ([7]). The Business Process Execution Language (BPEL, [1]) is an established standard to describe long-running business processes in such an environment.

BPEL is well suited for describing the communication behavior of an individual service. In scenarios, where several such conversational services are to be interconnected, we need a global view on the overall interaction behavior. Therefore, choreographies were introduced as a new view on interacting services (cf. [9]). They describe message exchanges between services from the perspective of an observer who is able to see all interactions and their flow dependencies. Existing choreography languages, as the Web Service Choreography Description Language (WS-CDL, [11]), support a top-down approach for choreography design and implementation. Here, choreographies serve as starting point for generating participant behavior descriptions for each service which are then used for implementing new services or for adapting existing services. Vice versa, bottom-up approaches, where existing BPEL processes are interconnected, are helpful for analyzing the overall interaction behavior between services and optimizing it.

Since BPEL is an accepted standard and has a defined execution semantics, we use it as foundation for describing choreographies. An additional layer, namely BPEL4Chor, is added to shift BPEL from an orchestration language to a complete choreography language. It decouples non-technical specifications from web-service-specific configurations. That way, reuse of choreographies for different technical groundings is facilitated.

The remainder of this paper is structured as follows: After introducing a choreography example in section 2, related work is discussed in section 3. The main contribution can be found in section 4, where BPEL4Chor is introduced. A validation for BPEL4Chor is given in section 5, where its support for the Service Interaction Patterns ([5]) is investigated. Finally, section 6 draws a conclusion and gives an outlook on future work.

2 Example

Figure 1 shows a sample choreography modeled using BPMN ([13]). A traveler wants to book a flight. She therefore submits a trip order to a travel agency of her choice. The travel agency requests the price for the selected route and date from a set of airlines. Each airline responds and the travel agency selects the airline offering the best price. The agency orders a ticket from

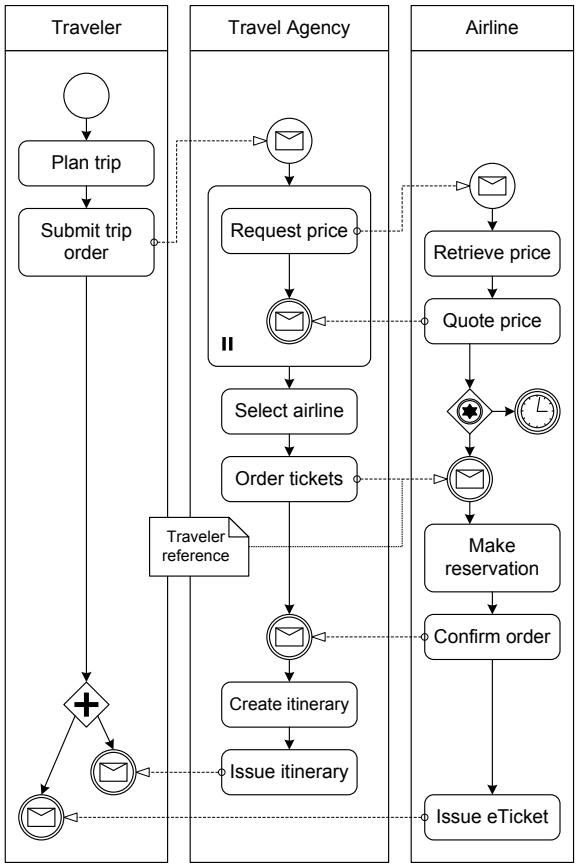

Figure 1. Choreography example

this airline and passes the traveler's email address on to the airline. As soon as the airline has confirmed the booking, the travel agency sends the itinerary to the traveler and the airline issues the eTicket. The airlines not selected for booking, stop waiting for the order as soon as a timeout occurs.

3 Related Work

Different viewpoints for service-oriented design have been proposed in [9]; the differences between choreographies, interface behaviors, provider behaviors, and orchestrations are explained. "Observable behavior" ([1]) and "local model" ([16]) are used as synonyms for provider behavior. For describing a choreography, we are using provider behavior descriptions given as "participant behavior descriptions" and their interconnections given in the "participant topology". The orchestration of each participant is out of scope of a BPEL4Chor choreography description.

The BPEL standard includes the notion of *abstract processes*. Abstract processes can be seen as participant behavior descriptions for services. However, they can be tightly linked to WSDL, since port types, operations, and XML schema types for data elements may be defined. This hampers reuse of choreography models. Suppose the activity for receiving the eTicket is coupled to the port type `travellerPT` and to the operation `receiveETicket`. After the complete choreography design, the role of the traveler should be taken by an interface for travelers realized on the port type `travelInterfacePT`. Now, the processes of the travel agency and the airline have to be modified to communicate with the new port type, even though the behavior has not changed. The decision which port types to use should be made as late as possible.

BPSS ([6]) is not tied to any particular technology. However, only bi-lateral collaboration scenarios can be specified. WSCI ([2]) and WSCL ([3]) are also languages for describing behavioral dependencies between web service interactions. WSCI does not provide a global view and WSCL is limited to bi-lateral interaction scenarios. WS-CDL ([11]) was introduced as successor for WSCI and WSCL. WS-CDL is also tightly linked to WSDL. It has been criticized that WS-CDL does not easily integrate with BPEL, as WS-CDL comes with its own set of control flow constructs that can hardly be mapped to those of BPEL (cf. [4]). In [8], the suitability of WS-CDL is assessed by investigating which of the Workflow Patterns and Service Interaction Patterns are supported. It turns out that WS-CDL does not directly support scenarios where the number of participants involved in a choreography is only known at runtime.

WSFL ([12]) is a predecessor of BPEL, where the global view ("Global Model") is distinguished from the local view ("Service Providers"). In contrast to BPEL, all sending and all receiving operations are modeled in port types for each service. Each sending operation is wired to a receiving operation in the global model. In the Service Component Architecture (SCA, [10]), instead of each operation, the interfaces as a whole are wired together to form a complete application. Both a wiring on the operation level and on the interface level are too coarse-grained. We claim that a wiring on the activity level provides a more detailed view on the dependencies between processes. That means, an activity sending data to a receiving activity is explicitly modeled and not hidden behind an interface. This allows to see where exactly in the receiving process the message is consumed, what happened before the consumption and what is going to happen afterwards.

Let's Dance ([15]) is a visual choreography language targeted at business analysts. It does not allow any technology-specific configurations. However, interface behavior descriptions out of the global interaction model can be generated ([16]).

The Business Process Modeling Notation (BPMN, [13]) is a graphical modeling language for intra- or inter-organizational business processes. It allows to interconnect processes using message flows and therefore to express choreographies. BPMN lacks formal semantics and is not executable, however mappings from BPMN to BPEL are available (e.g. [14]).

4 BPEL4Chor

In the choreography space there are two different modeling approaches: interaction models and interconnected interface behavior models. In case of *interaction models* (e.g. defined using WS-CDL and Let's Dance), elementary interactions, i.e. request and request-response message exchanges, are the basic building blocks. Behavioral dependencies are specified between these interactions and combinations of interactions are grouped into complex interactions. Due to the fact that these models capture the dependencies from a truly global perspective, the modeler is able to define dependencies that cannot be enforced. E.g., she might specify that a shipper can only send the delivery details to a buyer after the supplier has notified the insurance about the delivery. In this case it is left unexplained how the shipper can learn about whether the notification has been sent. Additional synchronization messages would be necessary to turn such a *locally unenforceable* interaction model into an enforceable one [16]. In the case of *interconnected interface behavior models* (e.g. expressed in BPMN) such unenforceability issues cannot arise since control flow is defined per participant. However, on the other hand, interface behavior models might be *incompatible*, i.e. the different participant cannot interact successfully with each other. Deadlocks are typical outcomes of such incompatibility. For instance, imagine a participant expecting a notification of another participant before being able to proceed and the other participant never sends such a notification.

It has not been investigated yet which approach is more suitable for the human modeler. We adopt interconnected interface behavior descriptions for specifying choreographies since this approach is closer to the history of BPEL. More precise, we use the *Abstract Process Profile for Observable Behavior* of BPEL ([1]) and add an interconnection layer on top of that leading to interconnected interface behavior descriptions.

In addition, unlike WS-CDL, BPEL4Chor decouples the "heart" of choreographies, i.e. the communication activities, their behavioral dependencies and their interconnection, from technical configuration, e.g. the definition of WSDL port types. That way, higher reusability of the choreography models is achieved.

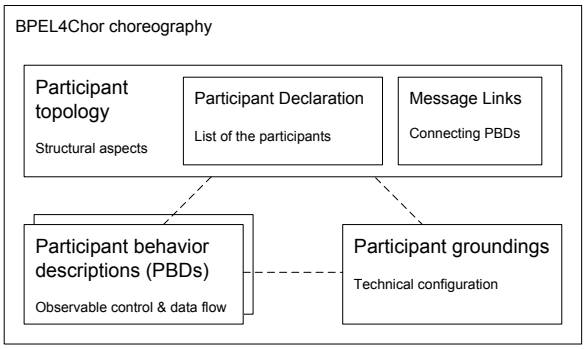

Figure 2. BPEL4Chor artifacts

BPEL4Chor is a collection of three different artifact types (cf. Figure 2): (i) *Participant behavior descriptions* define the control flow dependencies between activities, in particular between communication activities, at a given participant. (ii) A *Participant topology* defines the structural aspects of a choreography by specifying participant types, participant references, and message links. Participants of the same type have to provide the same set of communication activities. The communication activities of different participants are connected through message links. (iii) *Participant groundings* define the actual technical configuration of the choreography. Here, the choreography becomes web-service-specific and the link to WSDL definitions and XSD types is established.

The following subsections are going to introduce these artifact types. Corresponding code snippets will be given for the example from section 2.

4.1 Participant Behavior Descriptions

Communication activities, i.e. message send and receive activities, together with their control and data flow dependencies are at the center of attention in choreographies. BPEL comes with a rich set of constructs for control flow and data manipulation, which are used unchanged in BPEL4Chor. Therefore, existing BPEL tools can also be reused for choreographies.

In abstract processes, some language constructs needed to specify executable BPEL processes may be omitted. Such constructs are for example the `partnerLink` and the `operation` attribute of a message activity. A profile can force or forbid the usage of certain attributes. We will introduce the *Abstract Process Profile for Participant Behavior Descriptions* stating the requirements for defining the behavior of one participant. This profile inherits all constraints of the *Abstract Process Profile for Observable Behavior* specified by BPEL. We have to uniquely reference activities

from abstract process models to define the participant topology and therefore need an identifier for each activity in each process. Since `onMessage` branches do not offer a `name` attribute, we introduce `wsu:id` having the type `xsd:id` as new attribute for communication activities and `onMessage` branches. Besides the unique naming, the profile we introduce forbids the usage of `partnerLink`, `portType`, and `operation` attributes at the communication activities. In that way, the strong dependency between BPEL and WSDL interfaces is removed. All structural configuration will be defined in the participant topology and the participant groundings. Note that it is still allowed, but not required, to specify variables or use untyped variables at the BPEL constructs used for communication, as it is allowed in the *Abstract Process Profile for Observable Behavior*. Omitting variables and especially variable types allows branching conditions to be formulated as plain text. If variables or variables types are not used, they have to be filled in at the *executable completion* after the grounding of the abstract process. Since the relation between a `receive` and a `reply` activity cannot be established using `portType` and `operation`, we force the attribute `messageExchange` to be present at each pair of `receive` and `reply` activities. If a `receive` models an asynchronous operation, the attribute `messageExchange` must not be specified.

Listing 1 Participant behavior description

```
<process name="agency"
  targetNamespace="urn:booking:agency"
  abstractProcessProfile=
    "urn:HPI_IAAS:choreography:profile:2006/12">
  <sequence>
    <receive wsu:id="ReceiveTripOrder"
     createInstance="yes" />
    <forEach wsu:id="RequestPriceFE" parallel="yes">
      <scope><sequence>
        <invoke wsu:id="RequestPrice" />
        <receive wsu:id="ReceivePrice" />
      </sequence></scope>
    </forEach>
    <opaqueActivity name="SelectAirline" />
    <invoke wsu:id="OrderTickets" />
    <receive wsu:id="ReceiveOrderConfirmation" />
    <opaqueActivity name="CreateItinerary" />
    <invoke wsu:id="IssueItinerary" />
  </sequence>
</process>
```

Listing 1 shows the participant behavior description (PBD) for the travel agency in the example given in section 2. The BPEL profile for participant behavior descriptions is referenced. This allows us to add opaque activities into a PBD, which is useful for documentation purposes. Each communication activity and `onMessage` branch carries an identifier. These identifiers will be used later on for interconnecting corresponding send and receive activities of different participants.

In the example given above, the number of airlines is not known at design time. If no `counterName` attribute is specified in a `forEach` used in a PBD, the semantics of the `forEach` changes: The `forEach` is then iterating over a set of participants, which is specified in the participant topology. After transformation to BPEL the set will be represented as a `xsd:sequence` of `sref:service-ref` elements, where the current element is accessed via an XPath statement.

Executable BPEL demands that process instantiation is defined in every process. That means the receipt of a certain message leads to the creation of a process instance. In our scenario our travel agency is triggered by the receipt of a trip order from a traveler. However, it can be left open what triggers the traveler. Therefore, it is not required in BPEL4Chor that a process begins with an incoming message activity with the `createInstance` attribute set to `yes`.

BPEL comes with a built-in handling of message correlation. Since BPEL4Chor choreographies depend on BPEL and should not introduce any implementation dependencies, the correlation mechanism of BPEL is used unchanged: Correlation may be specified in the participant behavior description. We allow the usage of `correlationSet`s, but use the QNames of the properties specified for a correlation set as names. Thus, the names of properties change to NCNames and therefore have no connection to property aliases. Hence, properties get untyped and not bound to WSDL. On the other hand, properties always need to be typed in BPEL. Since we see typing as web service specific configuration detail, we leave properties untyped in the participant behavior descriptions and do the actual typing in the groundings.

4.2 Participant Topology

The participant topology describes the structural aspects of a choreography and serves as "glue" between the participant behavior descriptions. It introduces the notions of *participant type* and *participant reference* as well as *message link*. Every participant behavior description represents one participant type. Therefore, the same participant behavior description applies to all participants of the same type. Participant references point to participants.

Concerning the relationship between participants and participant types we can distinguish between three cases: (i) There is only one participant of a certain type in one conversation (choreography instance). E.g., there is one traveler and one travel agency involved

in a booking conversation. (ii) Several participants of a certain type appear in one conversation and the number of participants is known at design-time. (iii) An unbounded number of participants are involved and the exact number might only be determined at runtime. E.g., there are many airlines involved in our sample scenario. In order to provide support for all cases, *participant sets* are introduced.

Listing 2 Participant topology

```
<topology name="bookingtopology"
  targetNamespace="urn:booking"
  xmlns:agency="urn:booking:agency">
 <participantTypes>
   <participantType name="Agency"
     participantBehaviorDescription="agency:agency" />
   <participantType name="Traveler" ... />
   <participantType name="Airline" ... />
 </participantTypes>
 <participants>
   <participant name="traveler" type="Traveler"
     selects="agency" />
   <participant name="agency" type="Agency"
     selects="airlines" />
   <participantSet name="airlines" type="Airline"
    forEach="agency:RequestPriceFE">
     <participant name="currentAirline"
      forEach="agency:RequestPriceFE" />
     <participant name="selectedAirline" />
   </participantSet>
 </participants>
 <messageLinks>
   <messageLink name="tripOrderLink"
    sender="traveler" sendActivity="SubmitTripOrder"
    receiver="travelagency"
    receiveActivity="ReceiveTripOrder"
    messageName="tripOrder" />
   <!-- ... -->
   <messageLink name="ticketOrderLink"
    sender="travelagency" sendActivity="OrderTickets"
    receiver="selectedAirline"
    receiveActivity="ReceiveOrder"
    messageName="ticketOrder"
    participantRefs="traveler" />
   <messageLink name="eTicketLink"
    sender="selectedAirline" sendActivity="IssueETicket"
    receiver="traveler" receiveActivity="ReceiveETicket"
    messageName="eTicket" />
 </messageLinks>
</topology>
```

Listing 2 presents the participant topology for our example from section 2. In addition to the references for the traveler and the travel agency we find a participant set for representing the airlines.

The attribute `selects` defines which participant selects which other participants. In our scenario the traveler starts a conversation and selects a travel agency of her choice. The travel agency is in turn responsible for determining which airlines will be asked for a quote. The knowledge about participants during a conversation is local to individual participants. Only the travel agency knows all airlines involved. The traveler will only get into contact with one airline and an airline might not know which other airlines are involved.

Participant sets typically appears in combination with the notion of *containment*: a participant reference can be contained in a participant set. One usage scenario for this is the case where one participant is selected from the set. E.g., the travel agency will order a ticket only from the airline with the best offer. In another usage scenario a participant reference is needed in `forEach` constructs. The reference then represents the one participant selected in each of the parallel branches.

Message links state which participant can potentially communicate with which other participants. The identifiers given in `receiveActivity` and `sendActivity` refer to the activities in the participant behavior descriptions and therefore specify an interconnection of the participant behavior descriptions. The ordering constraints between these interactions are not given in the topology as they have already been defined in the participant behavior descriptions. Every time, a sending activity is active, a message is sent over the message link. If there are multiple senders with the same target specified, only one sender is allowed to send a message. If a receiving activity is executed multiple times, several interactions can take place over one message link. In the case of more than one potential sender, instead of `sender` containing the concrete sender, `senders` containing the set of potential senders is specified. Each sender has to be of the same participant type.

A message link has to obey the following restrictions: (i) A `receive` activity, a `onMessage` branch, or a `invoke` are valid as `receiveActivity` in a message link. If the output variable is specified, an `invoke` activity must appear as `receiveActivity` in a message link. Assume a message link l with an invoke i as `sendActivity` and a `receive` r as `receiveActivity`. If r is not associated with a reply y activity through a `messageExchange` attribute, the invoke i must not appear as `receiveActivity` in another message link. (ii) `reply` and `invoke` activities are valid as `sendActivity` in message links. Take i, r, and y as defined above. The message link where y is the `sendActivity` is required to have i as `receiveActivity`. That means, a `reply` may only answer a synchronous request by an `invoke` and must not send the answer to another `invoke`, `receive`, or `onMessage` branch. (iii) For every `invoke` and `reply` activity, there is exactly one message link in which this activity is the `sendActivity`. This implies: If there are several `receive`s for the same `invoke` or `reply` (e.g. through branching on the receiver's side) then all these receives must have the same identifier. Otherwise we would need several message links

for one invoke and this would violate the constraint iii. (iv) For every `receive` activity and `onMessage` branch, there is exactly one message link in which the activity respectively the branch is the `receiveActivity`. This implies: In the case of several sends for one `receive` (or `onMessage` branch), a list of senders is given in the message link. (v) If `senders` is specified in a message link, each listed sender has to be of the same type. (vi) If `senders` is specified in a message link l and the receiving activity r is connected to a reply activity p through a `messageExchange` attribute, `bindSenderTo` has to be specified in the link l. (vii) If corresponding variables are typed at both sender's and receiver's side, the types have to match.

The selection of participants might happen at runtime or already at design-time. The topology in listing 2 does not exclude the case that every traveler has exactly one travel agency she always goes to or that the list of airlines is fixed. As knowledge about participants is local, it might be required that participant references are passed on. E.g., the travel agency needs to pass on the traveler's reference to the airline so that the airline knows who to send the eTicket to. This phenomenon is called *link passing mobility*, realized through the attribute `participantRefs`.

Selection and reference passing lead to the *binding* of concrete participants to participant references. Binding might also happen in the case of a message receipt: If no participant is bound to a participant reference that is used in a receive activity, a message of any participant might be received and the sender of the message is bound to the reference. This second case is especially interesting in multi-lateral scenarios: E.g., consider a bidding scenario where bids from arbitrary bidders are accepted and stored. If the scope of the respective participant reference is limited, it can be reused. If such a participant is contained in a participant set, every binding of a participant that is not yet part of the set leads to adding this reference to the set.

We do not use the notion of *partner link* in the participant topology since in BPEL each partner link is related to one port type only. We leave it open in the participant topology whether a participant relates to exactly one port type or a collection of port types.

So far, we have only considered the case where participant behavior descriptions exist for every participant type. However, we might also envisage top-down approaches, where the participant topology is the first artifact to be created and then refined into a full choreography in a step-wise manner. Alternatively, the topology could simply be used as participant landscape if the behavioral constraints between message exchanges are not of interest.

4.3 Participant Grounding

While the participant topology and the participant behavior descriptions should be free of technical configuration details, the participant groundings introduce the mapping to web-service-specific configurations. So far, port types, operations, and XML schema types for messages have been avoided.

Listing 3 Participant grounding

```
<grounding topology="top:bookingtopology"
  xmlns:top="urn:booking" xmlns:...>
  <messageLinks>
    <messageLink name="tripOrderLink"
     portType="ag1:travelAgency_pt"
     operation="getTripRequest" />
    <messageLink name="ticketOrderLink"
     portType="lhx:web_pt"
     operation="getOrder" />
    <!-- ... -->
  </messageLinks>
  <participantRefs>
    <participantRef name="traveler"
     WSDLproperty="msgs:travelerProp" />
  </participantRefs>
</grounding>
```

Listing 3 shows the participant grounding for the example from section 2. For each link a port type / operation combination is given. This allows for realizing one participant through different port types. A grounding is only valid, if all message links are grounded. If variables were specified at the sending or receiving activity, the message type of the specified operation must match the given variable types.

The attribute `participantRefs` enables link passing mobility in BPEL4Chor choreographies. In the case of executable BPEL, end point references are passed in messages. In analogy to properties that are used for message correlation, we do not specify where exactly the references can be found in the message. The concrete location in the messages is specified using existing property aliases.

The grounding of properties is included, since correlation in the choreography is only specified on a name basis. If it comes to an execution, the concrete WSDL property has to be known.

After a choreography is completely grounded, every participant behavior description can be transformed to an executable BPEL process following the *Abstract Process Profile for Observable Behavior*. That profile ensures that the interactions between the participants will not be changed during the executable completion of each BPEL process. Future work will give a detailed elaboration of the mapping from BPEL4Chor over abstract BPEL to executable BPEL.

5 Validation

The Service Interaction Patterns [5] have been put forward as a benchmark for evaluating choreography languages. They capture common interaction scenarios between two or more participants. An assessment of WS-CDL can be found in [8].

The three simple patterns ***Send***, ***Receive***, and ***Send/receive*** are directly supported in BPEL4Chor through `invoke`, `reply`, and `receive` activities when being interconnected using message links in the participant topology. BPEL4Chor allows that a receiver of a message is bound at design-time or at runtime.

In the case of the ***Racing incoming messages*** pattern a party expects to receive one among a set of messages. Solution: This is expressed using a `pick`. If the participant reference used for the respective receive activity is not bound when this activity is reached, messages from arbitrary senders can be received.

One-to-many send: A party sends messages to several parties. Solution: In BPEL4Chor this pattern is directly supported through the notion of participant sets in combination with a `forEach` construct. The number of recipients does not need to be known at design-time. The sender is responsible for selecting the recipients, indicated by the `selects` attribute.

The ***One-from-many receive*** pattern describes that a party receives a number of logically related messages that arise from autonomous events occurring at different parties. Solution: This can be expressed using a `while` construct in BPEL4Chor with a corresponding participant topology: A participant set `senders` represents the set of all possible senders. The second set `mySenders` contains all participants the messages of which are actually received. Within the `while` structure we find a scope the participant reference `s` is limited to. This means that every time the scope is entered, there is no participant bound to `s` and a message of any sender can be received. The containment relationship between `s` and `mySenders` has the semantics that if a sender that is not contained in `mySenders` is bound to `s`, then this new sender is added to the set.

The ***One-to-many send/receive*** pattern is similar to *One-to-many send*: A party sends a request to several other parties. Responses are expected within a given timeframe. The interaction may complete successfully or not depending on the set of responses gathered. Solution: The timeframe aspect is supported in BPEL4Chor through scopes with an `onAlarm` event handler attached to it. Successful vs. unsuccessful completion is directly supported through exception mechanisms.

In the case of ***Multi-responses*** a party X sends a request to another party Y. Subsequently, X receives any number of responses from Y until no further responses are required. Solution: This pattern is directly supported through `while` structures.

Contingent requests: A party X makes a request to another party Y. If X does not receive a response within a certain timeframe, X alternatively sends a request to another party Z, and so on. Responses from previous requests might be still considered or discarded. Solution: The limited timeframe can be specified through an `onAlarm` structure. Should responses for previous requests also be considered we need to introduce two different participant references for the responders. However, we cannot ensure that the responder has actually received a request before. If responses from previous requests should be discarded, we only need to employ one participant reference for the responders. That way we ensure that the sender of the response is the same participant like the recipient of the latest request.

In the case of ***Atomic multicast notification*** a party sends notifications to several parties and a certain number of parties are required to accept the notification. For example, all parties or just one party are required to accept the notification. BPEL4Chor does not directly support this pattern. As a workaround an *One-to-many send/receive*-like implementation can be used.

Request with referral: Party A sends a request to party B indicating that any follow-up response should be sent to a number of other parties depending on the evaluation of certain conditions. Solution: BPEL4Chor directly supports link passing mobility through the `participantRefs` attribute of the `messageLink` element. Since also participant sets can be used as value for that attribute, the number of references passed does not need to be known at design-time in BPEL4Chor.

Relayed request: Party A makes a request to party B which delegates the request to other parties (P1, ..., Pn). Parties P1, ..., Pn then continue interactions with party A while party B observes a "view" of the interactions including faults. Solution: Using a `flow` structure responses are sent to A and to B.

6 Conclusion

This paper has shown how BPEL can be reused for describing choreographies. Although only few new constructs were added on top of BPEL, BPEL4Chor provides direct support for all Service Interaction Patterns, except Atomic Multicast Notification. BPEL4Chor was introduced as an alternative to WS-CDL. It compares with WS-CDL as follows:

WS-CDL follows the approach of interaction modeling, while BPEL4Chor expresses interconnected participant behavior descriptions. Therefore, unenforceable

models are possible with WS-CDL and BPEL4Chor allows to define incompatible behavior descriptions.

While a WS-CDL choreography is tightly coupled to WSDL files, web-service-specific details only appear in BPEL4Chor's participant groundings. Therefore, the same choreography can be reused with different port type definitions by changing the groundings.

Unknown numbers of participants are natively supported in BPEL4Chor through the notion of participant set. WS-CDL does not directly support parallel conversations with an unknown number of participants although these multi-lateral scenarios are very common as stated in the Service Interaction Patterns.

As BPEL4Chor is based on BPEL, a seamless integration between choreographies and orchestrations is possible. While WS-CDL comes with a different set of control flow constructs, the same constructs are found in BPEL and BPEL4Chor. All constructs introduced in topologies and groundings can be mapped to BPEL.

Future work will include detailed investigations on transformations from BPEL4Chor to BPEL and vice versa. Another aspect will be the investigation of modeling methods for choreographies using BPEL4Chor.

We demanded the variable types of a `sendActivity` and the `receiveActivity` to match. A next research step is to investigate how *message mediation* can help here. The participant groundings link activities directly to WSDL port types and operations. We suppose this is not the only way to do grounding and will investigate other possibilities like semantical groundings.

Acknowledgments. This work was partially supported by the German Federal Ministry of Education and Research (project number 01ISE08B).

References

[1] Web Services Business Process Execution Language Version 2.0 – Committe Specification. Technical report, OASIS, Jan 2007.

[2] A. Arkin et al. Web Service Choreography Interface (WSCI) 1.0. Technical report, Aug 2002.

[3] A. Banerji, C. Bartolini, D. Beringer, V. Chopella, K. Govindarajan, A. Karp, H. Kuno, M. Lemon, G. Pogossiants, S. Sharma, and S. Williams. Web Services Conversation Language (WSCL) 1.0, W3C Note. Technical report, March 2002.

[4] A. Barros, M. Dumas, and P. Oaks. A Critical Overview of WS-CDL. *BPTrends*, 3(3), 2005.

[5] A. Barros, M. Dumas, and A. ter Hofstede. Service Interaction Patterns. In *BPM 2005*, LNCS, pages 302–318, Nancy, France, 2005. Springer Verlag.

[6] J. Clark, C. Casanave, K. Kanaskie, B. Harvey, N. Smith, J. Yunker, and K. Riemer. ebXML Business Process Specification Schema Version 1.01. Technical report, UN/CEFACT and OASIS, May 2001. `http://www.ebxml.org/specs/ebBPSS.pdf`.

[7] F. Curbera, F. Leymann, T. Storey, D. Ferguson, and S. Weerawarana. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR, 2005.

[8] G. Decker, H. Overdick, and J. M. Zaha. On the Suitability of WS-CDL for Choreography Modeling. In *EMISA 2006*, Hamburg, Germany, Oct 2006.

[9] R. Dijkman and M. Dumas. Service-oriented Design: A Multi-viewpoint Approach. *International Journal of Cooperative Information Systems*, 13(4):337–368, 2004.

[10] D. F. Ferguson and M. Stockton. Enterprise Business Process Management – Architecture, Technology and Standards. In J. Eder and S. Dustdar, editors, *BPM 2006*, volume 4103 of *LNCS*, pages 1–15, Nancy, France, 2006. Springer Verlag.

[11] N. Kavantzas, D. Burdett, G. Ritzinger, and Y. Lafon. Web Services Choreography Description Language Version 1.0, W3C Candidate Recommendation. Technical report, November 2005. `http://www.w3.org/TR/ws-cdl-10`.

[12] F. Leymann. Web Services Flow Language (WSFL 1.0), May 2001.

[13] Business Process Modeling Notation (BPMN) Specification, Final Adopted Specification. Technical report, OMG, Feb 2006. `http://www.bpmn.org/`.

[14] C. Ouyang, M. Dumas, S. Breutel, and A. H. ter Hofstede. Translating Standard Process Models to BPEL. In *CAiSE 2006*, Luxembourg, June 2006.

[15] J. M. Zaha, A. Barros, M. Dumas, and A. ter Hofstede. A Language for Service Behavior Modeling. In *CoopIS 2006*, Montpellier, France, Nov 2006.

[16] J. M. Zaha, M. Dumas, A. ter Hofstede, A. Barros, and G. Decker. Service Interaction Modeling: Bridging Global and Local Views. In *EDOC 2006*, Hong Kong, Oct 2006.