Interacting services: From specification to execution

Gero Decker[1], Oliver Kopp[2], Frank Leymann[2], Mathias Weske[1]

[1]Hasso-Plattner-Institute, University of Potsdam, Germany
{gero.decker,weske}@hpi.uni-potsdam.de

[2]Institute of Architecture of Application Systems, University of Stuttgart, Germany
lastname@iaas.uni-stuttgart.de

BibTeX:

```
@article{DKLW2009,
  author    = {Gero Decker and Oliver Kopp and
               Frank Leymann and Mathias Weske},
  title     = {Interacting services:
               From specification to executio},
  jounal    = {Data \& Knowledge Engineering},
  year      = {2009},
  volume    = 68,
  number    = 10,
  pages     = {946--972},
  doi       = {10.1016/j.datak.2009.04.003},
  publisher = {Elsevier Science Publishers}
}
```


Paper available at http://dx.doi.org/10.1016/j.datak.2009.04.003
See also Elsevier homepage: http://www.sciencedirect.com

Interacting Services: From Specification to Execution

Gero Decker [a], Oliver Kopp [b], Frank Leymann [b], Mathias Weske [a]

[a] *Hasso-Plattner-Institute, University of Potsdam, Germany*

[b] *Institute of Architecture of Application Systems, University of Stuttgart, Germany*

Abstract

Interacting services play a key role to realize business process integration among different business partners by means of electronic message exchange. In order to provide seamless integration of these services, the messages exchanged as well as their dependencies must be well-defined. Service choreographies are a means to describe the allowed conversations. This article presents a requirements framework for service choreography languages, along which existing choreography languages are assessed. The requirements framework provides the basis for introducing the language BPEL4Chor, which extends the industry standard WS-BPEL with choreography-specific concepts. A validation is provided and integration with executable service orchestrations is discussed.

Key words: Choreography, Service Interaction Modeling

1 Introduction

In a networked business world, companies need to interact with their customers, suppliers and other stakeholders as part of their day-to-day business. While such interactions used to be realized mostly by phone calls, letters and faxes, more and more companies increase efficiency by exchanging electronic

Email addresses: gero.decker@hpi.uni-potsdam.de (Gero Decker), oliver.kopp@iaas.uni-stuttgart.de (Oliver Kopp), leymann@iaas.uni-stuttgart.de (Frank Leymann), weske@hpi.uni-potsdam.de (Mathias Weske).

URLs: http://www.iaas.uni-stuttgart.de (Frank Leymann), http://bpt.hpi.uni-potsdam.de/ (Mathias Weske).

messages. This can lower costs, increase the overall process performance, and sometimes even lead to new business opportunities. On the other hand, electronic communication also creates a need to specify the desired message formats and interaction sequences. Since a fast setup of new collaborations increases competitiveness of companies, powerful means to specify interaction scenarios between business partners are required. To this end, service choreographies serve as contract that the companies have to conform to.

Interactions by electronic messages also involve technical aspects. In this context, service-oriented architectures (SOA) have received a lot of attention lately. SOA is an architectural style prescribing systems to be made up of loosely coupled services with clearly defined interfaces [9]. These services are not just software components but have business meaning. They are meant to be reusable in different business processes. The possibility of easily rearranging and exchanging services in business processes promises quick adaption of software systems to changing business requirements [33].

In a first generation of services, only simple variants of request/response message exchanges were considered. This view is sufficient when considering simple interactions, for instance, a stock information service, where the current or a past value of a share can be requested. However, more complex interactions must be considered in many real-world business scenarios. For instance, in a typical purchasing scenario, goods can be ordered, orders can be modified or canceled, orders must be acknowledged and delivery can be rerouted, or alternative products or quantities are offered in out-of-stock situations. Also multi-lateral scenarios involving, for instance, external payment, shipment and insurance services need to be considered. These scenarios involve multiple interactions and the complex dependencies between them must be addressed.

As typically multiple parties are involved, each offering their own services, successful service integration can be roughly divided into two phases. (i) The different service providers need to agree on a certain interaction behavior. This includes ordering constraints, alternatives and time constraints. Therefore, the result of the specification phase is an interaction contract, also called service choreography (cf. [23]). (ii) At runtime all service providers need to adhere to what was initially agreed upon in the choreography. Violations could lead to exceptions in operations and might have legal consequences. Examples for exceptions in operations are ignored incoming messages or the suspension of process instances. The latter happens only if a proper infrastructure is in place: the infrastructure has to be capable to detect violations during the runtime of a choreography and be able to suspend the sender of a message causing the violation.

A large number of industry initiatives such as RosettaNet (`http://www.rosettanet.org/`) aim at facilitating integration between different compa-

nies of a particular domain. However, due to a lack of suitable choreography modeling languages available, these initiatives have mostly resorted to textual descriptions of the overall choreographies, while providing detailed message format descriptions. Therefore, a clear need for choreography languages that allow for technical configurations can be observed. The initiatives' artifacts, in turn, allow the derivation of requirements for choreography languages.

The Web Service Choreography Description Language (WS-CDL, [28]) is a prominent language in the field of service choreographies. However, as this paper will point out, it has a number of drawbacks, including limited support for scenarios with an unknown number of services, integration with service orchestration languages, and different technical realizations of the used services.

The Web Services Business Process Execution Language (WS-BPEL [51], BPEL for short) is the de facto language for service composition in web services environments. In addition, it is very valuable in the context of interaction services: BPEL can be used to implement web services that show complex interaction behavior with their environment. Both external services (i.e. web services offered by other service providers) and internal web services can be composed into a single executable business processes. BPEL processes themselves are offered as web services to the environment once they are deployed in a BPEL engine. Therefore, BPEL is well suited as implementation language for the runtime of web services. Although BPEL can also be used to describe the communication behavior of a web service, it is not suited as choreography language: BPEL focuses on describing the behavior of a single partner. The interconnection of multiple partners cannot be represented appropriately [36].

In this article we present a requirements framework for service choreography languages, which is largely based on the service interaction patterns [7]. We are going to assess WS-CDL, BPEL, and a number of other languages along this framework. As this assessment will show, none of the existing languages meets the identified requirements. Therefore, extensions to existing languages have to be introduced or a new language needs to be designed.

As main contribution, this article introduces BPEL4Chor as additional layer on top of BPEL. Core concepts of this language have already been presented in [17]. We show that BPEL4Chor is indeed suited to express choreographies. Furthermore, we investigate the integration of BPEL4Chor choreographies and executable BPEL processes. More specifically, we show how BPEL processes can be generated out of BPEL4Chor choreographies.

The remainder of this article is structured as follows: the next section presents the main concepts of service choreographies and introduces a choreography example. Section 3 presents the requirements framework, points to related work

and assesses WS-CDL and other languages along the framework. BPEL4Chor and its artifact types are discussed in Section 4. The suitability of BPEL4Chor for choreography modeling is assessed in Section 5. Section 6 discusses the integration of BPEL4Chor with executable BPEL. Section 7 concludes the paper and points to directions of future work.

2 Service Choreographies

A motivating example illustrates the main concepts in service choreographies. Figure 1 presents a sample choreography describing an auctioning scenario in the financial sector. Three types of participants are involved in this scenario: a seller, several bidders and a broker. The seller wants to sell stock options for the highest possible price. Brokers are able to carry out auctions more efficiently than the seller. That is why the seller outsources the operation of the auction to a broker. Different bidders can join in if they are interested in the options and place their bids accordingly. The Business Process Modeling Notation (BPMN [48]) is used in this example. All interactions are realized as exchange of electronic documents. *Broker services* provide all functionality needed. On the seller and bidder side, electronic services are also available to carry out most of the work. The *seller service* automatically handles the exchange of options for payment. The *bidder service* automatically issues bids, based on pre-defined rules (which are not in the scope of this work).

An auction works as follows. First, the seller service sends an auction creation request to the broker service, who acknowledges it with a confirmation message. As soon as the auction begins, bidder services can place bids that are in turn confirmed by the broker service. Each bidder service is allowed to place several bids during an auction. The auction ends at a given point in time. When an auction has ended, the broker service notifies the seller service about which bidder won the auction, i.e., which bidder service has placed the bid with the highest amount. The bidder service, which has won the auction also gets a notification. The unsuccessful bidder services are informed, too. Finally, the exchange of the options payment can happen in parallel. The seller service sends payment details to the successful bidder service, and the bidder issues the payment. On the option side, the seller service grants the stock options to the bidder service and the bidder service acknowledges this.

This scenario shows how interacting processes are realized as interacting services. The focus is placed on the actual message exchanges between the services. Human involvement is not documented in this example, although it might be present. The different seller and bidder services might differ in the degree of automation, e.g. a bidder service might need to be triggered to enter an auction or it constantly observes all offered stock options and decides to enter

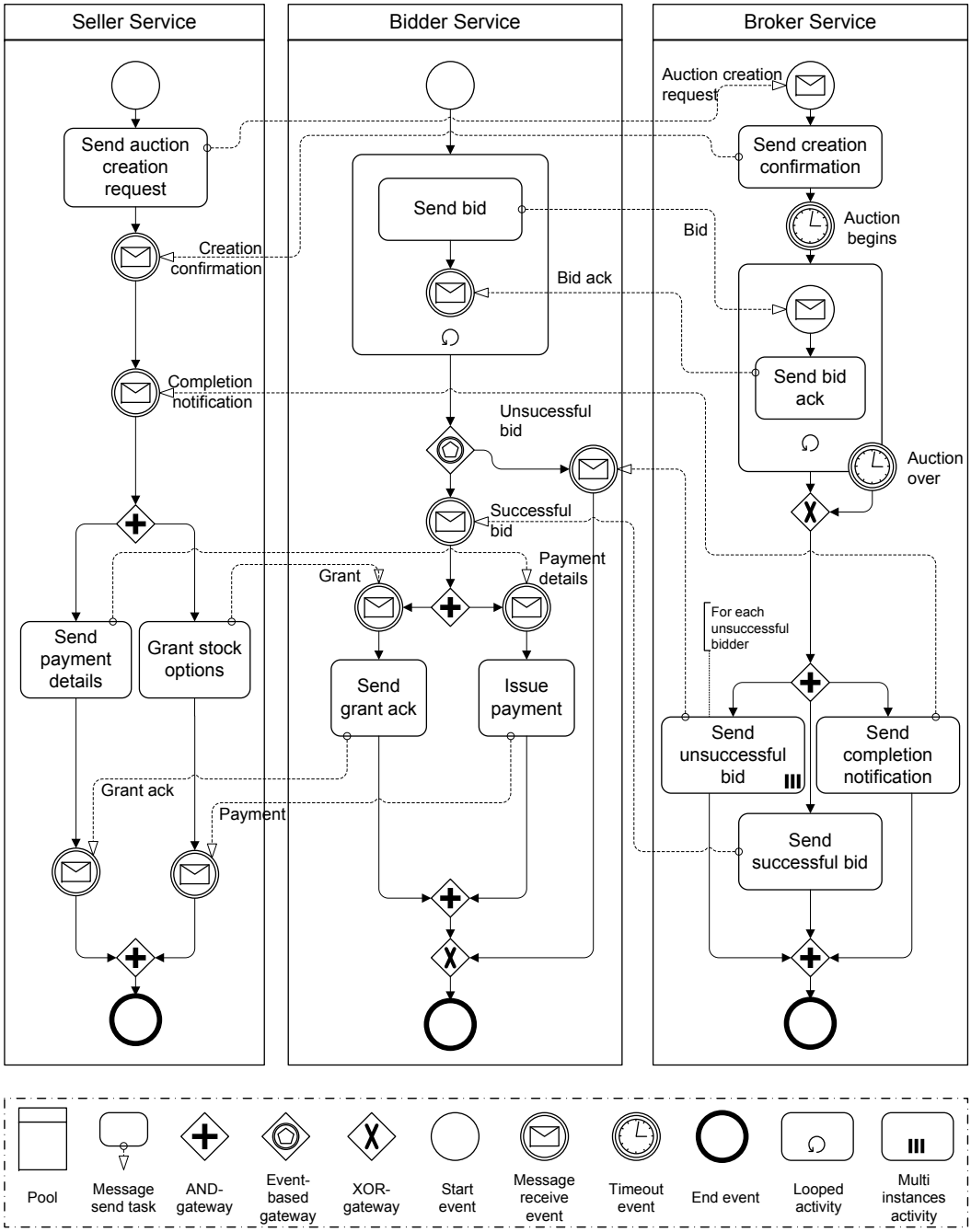

Fig. 1. Auctioning Scenario

an auction based on sophisticated algorithms and extensive financial data sets.

This example also shows the correspondence between the terms *participants* and *services*, as well as *participant types* and *service types*: the individual sellers, brokers and bidders are choreography participants, realized as seller, bidder and broker services, respectively. All sellers are of participant type "seller", while all seller services are of service type "seller service".

While choreographies in general show how participant of different types interact with each other, we focus on service choreographies where all interactions are realized as electronic message exchanges.

The Business Process Modeling Notation (BPMN [48]), used to express the example, is a popular language to model choreographies. The service types are represented by pools (big rectangles in the background). Message flows (dashed arrows) indicate which messages might be sent from a service of one type to a service of another type. Message events (circles containing a letter icon) represent message receipt activities while the rounded rectangles with outgoing message flow represent message send activities. Additionally you find timer events (circles containing a clock icon) and a number of control flow constructs defining the behavioral dependencies between the different message exchange activities.

While the BPMN diagram provides an appropriate overview by representing participant types and interactions, the diagram alone would not be sufficient as interaction contract. It is too imprecise in many aspects. Time information about the start of the auction and its completion are only given in an informal way, i.e., in plain text. The case that no bids are placed is not covered and the distinction between different bidders, i.e., between different participants of a given type, is not specified. Finally, the message formats are not defined. While handling exceptions such as "no bids" could be represented in BPMN, other aspects are only poorly supported and require further refinement in a service choreography.

3 Motivation of the Approach

The example in the previous section showed that choreographies easily go beyond simple sequences of message exchanges between two services. What happens to the bidder service that has not placed the highest bid? What happens if an answer does not arrive on time? Does the payment need to be carried out prior to delivery? These are sample questions a choreography must answer.

In this section, requirements are developed that can be used to identify shortcomings of choreography languages.

The service interaction patterns [7] represent a catalog of typical scenarios for interacting services. They can be used to benchmark modeling languages or systems regarding their support for typical use cases. While the workflow patterns [2] concentrate on control flow structures within service implementations, they do not capture scenarios where two or more services engage in complex

interactions. The service interaction patterns fill this gap. However, there is some overlap between the pattern sets. Often, a service interaction pattern is based on a certain workflow pattern, while adding certain aspects that only apply to choreographies.

The service interaction patterns are divided into four categories. The distinguishing criteria are: (i) the number of services involved (bi-lateral vs. multilateral interactions), (ii) the number of messages exchanged (single-transmission vs. multi-transmission interactions) and (iii) in the case of two-way interactions, if the receiver of a response is necessarily the same as the sender of the request (round-trip vs. routed interactions). The resulting categories therefore are: (1) single-transmission bilateral interaction patterns, (2) single-transmission multilateral interaction patterns, (3) multi-transmission interaction patterns and (4) routing patterns.

Through the analysis of these patterns, we extracted key requirements of service choreographies. At least following requirements must be fulfilled by a choreography language:

- **R1. Multi-lateral interactions**. The example in the previous section shows that more than two services can be involved in a choreography. The higher the number of services, the more important the service choreography becomes: while it is possible to make small runtime adjustments in the case of two interacting services, these adjustments are more difficult if multiple services are involved. Therefore, a choreography language must support multi-lateral interactions.
- **R2. Service topology**. The choreography must specify what types of services are involved and how services of these types are interlinked. The service topology provides an essential structural view on the choreography from a global perspective.
- **R3. Service sets**. It can be observed that often several services of the same type are involved in a choreography. As example our auctioning scenario includes a set of bidder services. A choreography language must support a (potentially unknown) number of services of the same type.
- **R4. Selection of services and reference passing**. In some cases, the selection of services is done at design-time or deployment-time of services. In other cases, selection is only done at runtime. In both cases it must be ensured that other services are made aware of the selection if they are to interact with these services as well. In our example the broker service has to pass on the reference of the successful bidder service to the seller service. Otherwise the seller service would not know where to send the payment details to.

In addition to the requirements derived from the service interaction patterns, the following technical requirements are very important for the suitability of a

service choreography language:

- **R5. Message formats**. Perfect match regarding the order of messages does not ensure that the different services can process the messages they receive. Often the message formats simply do not match. Initiatives such as RosettaNet show that it is possible and fundamental to agree on the message formats exchanged. Therefore, a choreography language has to natively support the definition of message formats.
- **R6. Interchangeability of technical configurations**. A drawback of tightly integrating message format specifications into choreographies is the limited reusability of the choreography.

  Currently, the Web Service Description Language (WSDL, [12]) is a widely used standard to describe the structural interface ("port type" as a collection of "operations") of a web service. A WSDL definition includes the definition of message formats, usually using XML Schema, as well as the information needed to interact with a physically deployed service—on the wire manifestation of messages and the transport protocols ("binding") as well as the physical endpoints ("ports") [14]. Standards such as RosettaNet show that it is possible to standardize the exchanged messages, but that it is not possible to standardize the interfaces to be used. Therefore, WSDL's ability to standardize messages definitions is important in choreography design, whereas the description of operations in an interface as well as port and binding information are not needed, because their use would force the choreography adopters to follow these technical realizations.

  Changes in port type or operation names should not require changing the choreography. Also other technical details, e.g. whether a service is realized using one port type or two port types or whether two messages are exchanged in a synchronous request/response cycle or asynchronously, should not require major changes in the choreography. Sometimes it is not possible to agree on one message format for a particular interaction. In these cases, message mediation is necessary. There should at least be an extension point to plug in corresponding configurations. A choreography language must support interchangeability of technical configurations.
- **R7. Time constraints**. Time is an important aspect of choreographies. It must be agreed upon how long a certain message must be waited for and for how long certain messages are allowed. In our auctioning example, bids are not accepted once the auction is over. Therefore, a choreography language must support the definition of time constraints.
- **R8. Exception handling**. There is typically an ideal path in a choreography. However, there might be purely technical reasons or other reasons why messages are not sent on time or contain the wrong content. For these cases, control flow handling faults and exceptions must be defined. Reactions in the form of cancellation messages or a simple termination of a conversation might be necessary. Therefore, a choreography language

must allow the definition of exception handling.

- **R9. Correlation**. It cannot be expected that each service is involved in at most one choreography instance at a time. It must be possible to distinguish different conversations and to relate an incoming message to the messages previously sent or received. Such correlation is typically realized by placing correlation information into messages. In the example scenario, a correlation information can be an auction id or a bid id. In general, the service providers must agree upon where in the message such information can be found and who is responsible to generate which identifier. By checking the equality of an id included in a message and an expected id, correlation is carried out. A choreography language must support the definition of such correlation configurations.
- **R10. Integration with service orchestration languages**. BPEL is the de-facto standard to implement business processes based on web services. Therefore, choreography languages must allow an integration with BPEL, including easy generation of BPEL processes out of choreographies and extracting choreographies out of existing interacting BPEL processes.

In the following, we first give an overview of related work in the field of choreography languages and then evaluate the most prominent choreography languages and workflow languages using the ten requirements listed above. A summary is given in Section 3.12.

3.1 Related Work

Different viewpoints for service-oriented design have been proposed in [23]; the differences between choreographies, interface behaviors, provider behaviors, and orchestrations are explained. "Observable behavior" [51] and "local model" [60] are used as synonyms for provider behavior.

In addition to the Service Interaction Patterns framework [7] that is used in this paper, there is a family of interaction patterns reported by Mulyar et al. [42]. Here, different variation points for multi-party multi-message conversations are identified, also allowing for a detailed analysis of choreography languages.

In general, we can distinguish between two different modeling approaches in the choreography space: interaction models and interconnected interface behavior models [16]. The choreography model in Fig. 1 is modeled using interconnected participant behavior descriptions. In the case of *interaction models*, 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. For example, the modeler 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 open 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 [60]. In the case of *interconnected participant behavior descriptions* such unenforceability issues cannot arise since control flow is defined per participant. However, participant behavior descriptions might be *incompatible*, i.e. the different participants cannot interact successfully with each other. Deadlocks are typical outcomes of such incompatibility. Imagine a participant expecting a notification of another participant before being able to proceed and the other participant never sends such a notification (cf. [38]). It has not yet been investigated which approach is more suitable for the human modeler.

Most academic work in the area of choreographies mainly focuses on formal verification of properties such as compatibility of interacting services [10, 38, 40, 55] or the conformance of a service implementation to a behavioral specification [1, 8, 31, 39]. In this context, *compatibility checking* addresses the question whether two or more services can interact successfully with each other given the different behaviors plugged together. *Conformance* addresses the question whether a service will show a communication behavior as specified [21].

A choreography definition can be used to enable proper message routing in an enterprise service bus. In general, a service bus is a middleware platform for realizing service oriented computing based on web service standards [11,14,35]. If a service-bus is aware of the choreography, it is able to monitor the compliance to the choreography of the participants and to react on violations of choreographies [27, 32]. In the auctioning scenario, a violation of the choreography is that a broker does not send an unsuccessful bid notification to all unsuccessful bidders. While there is work on conformance, a process can be verified to conformance only if the implementation is available. Usually, service providers claim to follow a choreography, but do not verify their business process. Then, the only possibility to check conformance is conformance checking at runtime. Most important, a message violating the choreography is not delivered to the recipient and does not cause any more wrong actions at the recipient. Further actions to be taken have to be specified within the enterprise service bus and are currently not standardized or further investigated [32].

We presented an overview over BPEL4Chor in [17]. [19] shows a mapping of selected concepts of BPEL4Chor to Pi-Calculus and [37] shows how BPEL4Chor choreographies can be verified using Petri nets. This verification checks the absence of deadlocks in choreographies.

BPEL centers around message send and receive activities, defining if and when messages are sent or received by the BPEL engine. These communication activities (invoke, receive, reply, pick, ...) together with other basic activities such as data manipulation can be arranged using control flow constructs into processes. BPEL comes with a wide range of such constructs covering sequences, parallelism, alternatives, repetitions and even multiple-instances where the number of parallel instances does not need to be known at design-time. Furthermore, BPEL covers exception handling and compensation, correlation and data flow.

BPEL comes in two flavors: abstract BPEL and executable BPEL. In the latter case, the process definitions (together with the service interface and binding descriptions as well as additional discovery information) represent artifacts that can be directly executed. Abstract BPEL processes define business processes which are not intended to be directly executable. Each abstract BPEL process is annotated with a BPEL process profile, which states the intended usage type of the BPEL process. The BPEL specification defines profiles for process templates and observable behavior. Process templates serve as templates for executable BPEL processes, whereas observable behavior processes serve as description of the interaction behavior of the business process. Modelers are free to define new profiles. This enables modelers to describe service behavior in BPEL without the need to implement the described services using BPEL.

BPEL defines the notion of partner links. Partner links are connectors between the WSDL port type offered by the process and the WSDL port type required from the partner [14]. Through partner links, a service implemented as BPEL processes can interact with multiple other services. Therefore, BPEL supports Requirement *R1*. However, as BPEL only focuses on behavior of individual participants and sees the services of other participants it interacts with as black boxes, BPEL does not provide a big picture on how services are involved in a choreography. In many cases, it is impossible to derive such a picture, since discovery information does not form a part of BPEL. BPEL refers to port types, not to ports. Therefore, a given set of BPEL processes with matching port types does not guarantee that these BPEL processes are actually interconnected. The lack of a topology and the lack of interconnection between different communication activities make BPEL miss Requirement *R2*.

Services endpoints can be described by endpoint references, which in turn can be stored in BPEL variables. This storage allows to receive endpoint references from other services or sending endpoint references to other services. Since an *assign* activity can be used to assign an endpoint reference to a partner link, it is possible to interact with a service described by an endpoint reference. In

addition, endpoint reference sets can be handled through corresponding XML schema types. In combination with the parallel `forEach` construct in BPEL, the "One-to-many send/receive" pattern [7] is supported by BPEL. Relying on external type systems, sets or lists are not first-class citizens in BPEL and therefore working with endpoint reference sets is cumbersome. Hence, we conclude that BPEL only partially supports Requirement *R3*.

Although reference passing is fully supported in BPEL, the notion of service selection is out its scope. Service selection is done through deployment configurations, either realizing service selection at deployment-time or at runtime. Again, we conclude that there is only partial support for Requirement *R4*.

BPEL is tightly integrated with WSDL, where message formats are specified and thus make BPEL supporting Requirement *R5*. The variables used at send and receive activities have to match the used operation in the expected data structure. Partner link types are configured with WSDL port types and BPEL's communication activities depend on WSDL operations. This makes BPEL rigid in terms of interchangeable technical configurations (*R6*): changes on the WSDL side often require changes in the BPEL process.

BPEL has built-in timing capabilities (*R7*). Timers can be attached to the so called scopes and can be put into regular control flow, defining the maximum time that should be spent waiting. Furthermore, delays can be realized through wait activities. Also a variety of possibilities to react to erroneous situations are at hand (*R8*). Exceptions can be handled and completed activities can be compensated if necessary.

In BPEL, correlation sets are used to define which process instance an incoming message should be routed to. These correlation sets can be initialized by the BPEL engine and correlation information is included into the messages exchanged. The so called properties define where exactly in the message correlation information can be found. BPEL supports Requirement *R9*.

The integration between executable BPEL and abstract BPEL is already described in the specification. Although issues like conformance checking lie outside the scope of the BPEL specification, it is defined what elements must be used and what attributes must be set in either of the two BPEL flavors. As the elements and attributes available do not significantly differ from abstract BPEL to executable BPEL, we conclude that Requirement *R10* is supported.

3.3 Assessment of BPEL^{light}

BPEL$^{\text{light}}$ [44] has been introduced as extension of BPEL to decouple BPEL and its technical details. BPEL$^{\text{light}}$ introduces the new activity type

`interactionActivity`, which replaces BPEL's communication activities. Instead of using WSDL artifacts, each interaction activity is assigned to a `conversation`. To enable the usage of BPEL^{light} processes in a web service environment, the concept of "assignment" is introduced. There, each `conversation` is assigned to a "partner link" and each activity is assigned to a WSDL operation. This makes BPEL^{light} fulfill requirement *R6*.

In [45] the concept of a BPEL^{light} `partner` element is extended to cover multiple partners of the same type. BPEL^{light} still describes the viewpoint of one participant and thus lacks a topology. Finally, it does not offer an explicit mechanism to describe the interconnection of activities, which makes it still missing Requirement *R2*.

BPEL^{light} supports a set of services. The declaration of a `partner` element declares an unbounded set of partners. An `interactionActivity` is always linked to a partner. The respective partner of a received message is looked up in the referenced partner set. If the partner is not contained in the set, the partner is added to the set. All in all, BPEL^{light} does not distinguish between a single participant and a set of participants and thus BPEL^{light} partially supports Requirement *R3*.

As in BPEL, reference passing is fully supported in BPEL^{light}, but the notion of service selection is out of its scope. Service selection is realized through the concrete assignment to a communication infrastructure and thus delegated to the infrastructure. As with BPEL, we conclude that there is partial support for Requirement *R4*.

Since BPEL^{light} not change anything else in BPEL, the assessment for the other requirements *R1,R5,R7-R10* remains unchanged in comparison to the assessment of BPEL.

3.4 Assessment of WSFL

WSFL [34] is a predecessor of BPEL, where control flow is modeled as graph. WSFL distinguishes between a "Flow Model" and a "Global Model". In the Flow Model, control flow of one process is defined, whereas a service topology is provided in the Global Model (*R2*). Each Flow Model offers and requires operations. These operations are connected together in a "Global Model" using so-called plug links. This enables different names for the same operations. Each time, a service provider is replaced, the Global Model has to be adopted and the Flow Models remains unchanged (*R6*). Sets of services are not supported by the Global Model (*R3*). Message formats are specified using XML Schema (*R5*).

The semantics of WSFL control flow is similar to that of BPEL's `flow` activity with `links`. However, there is no transformation available from WSFL models to BPEL is available, even if it should be possible due to the control flow similarities (*R10*). By using different operations for different participants, multi-lateral interactions can be realized using WSFL (*R1*). While services can be selected by using the respective operation, WSFL does not support reference passing (*R4*). Time constraints are not supported (*R7*), but WSFL supports exception handling by a special attributes of control links (*R8*). The WSFL specification mentions correlation, but not specify how correlation works (*R9*).

3.5 Assessment of WS-CDL

The Web Services Choreography Interface (WSCI, [3]) and the Web Service Conversation Language (WSCL, [4]) are the predecessors of the Web Services Choreography Description Language (WS-CDL [28]). WS-CDL is an interaction modeling languages for web service choreographies. Being a candidate recommendation by the W3C since 2005, WS-CDL is mostly criticized for not easily integrating with BPEL. WS-CDL comes with its own set of control flow constructs that can hardly be mapped to those of BPEL (cf. [6]). In [20], the suitability of WS-CDL is assessed by investigating which of the workflow patterns and service interaction patterns are supported. This assessment reveals that WS-CDL does not directly support scenarios where the number of participants involved in a choreography is only known at runtime. However, WS-CDL is the most prominent example of a language following the interaction modeling style.

The basic building blocks of WS-CDL are interactions. They are bi-lateral (between two web services) and involve either one message (request-only or response-only) or two messages (request-response). Each interaction takes place via a channel instance identifying the responding service. An important feature of WS-CDL is the possibility to pass channel instances from one service to another in an interaction. The structure of other information that can be passed is specified using information types. Role types define what behaviors (WSDL interfaces in the default case) have to be implemented by a corresponding service. Relationship types are pairs of role types, services of which can directly interact with each other. Interactions can be composed to activities using a range of control flow constructs. These include sequence, parallel, choice and workunit. Listing 1 shows how the auction creation request and the response by the broker service can be described in WS-CDL.

A more detailed overview of WS-CDL can be found in [6] and an assessment of WS-CDL regarding its support for the workflow patterns and the service interaction patterns in [20].

Listing 1 Sample interaction in WS-CDL

```
<interaction name="auctionCreation"
    channelVariable="tns:broker-channel"
    operation="requestAuctionCreation">
  <participate relationshipType="tns:SellerBrokerRel"
      fromRole="tns:Seller" toRole="tns:Broker"/>
  <exchange name="request" informationType="tns:creationReq"
     action="request">
     <send/>
     <receive variable="..." />
  </exchange>
  <exchange name="response"...>...</exchange>
  <timeout time-to-complete="..."/>
</interaction>
<exceptionBlock name="handleTimeoutException">
  <noAction/>
</exceptionBlock>
```

WS-CDL directly supports scenarios where multiple services are involved in one choreography (Requirement *R1*). This is realized through an unlimited number of role types that can be defined in a choreography. As concrete services are identified by channel instances, there can be potentially many services involved in one choreography of a particular role type. As all role types are enumerated in the choreography, a topology is present in the case of WS-CDL. However, the number of services per role is unspecified. Although in most cases there is at most one service per role type, this cannot always be assumed. In some cases it can be derived from the choreography whether there might be several services of one role type, but not in all. Therefore, we opt for partial support for Requirement *R2*.

Two important features are missing to support service sets (*R3*): (a) sets are not first-class citizens in WS-CDL and (b) there is no control flow construct allowing multiple branches to be executed in parallel where the number of branches might only be known at runtime. This results in the fact that the workflow pattern "Multiple instances with a priori runtime knowledge" is not supported by WS-CDL. The service interaction pattern "One-to-many send/receive" comes in two flavors: the exact number of services might be known at design- or only at runtime. While the first case is supported by WS-CDL, the second one is not. Therefore, we conclude that this service interaction pattern is only partially supported.

The notion of reference passing (*R4*) is directly integrated into WS-CDL. However, it is not possible to explicitly model which service selects which other service. This can only be derived from data flow dependencies at best. Therefore, we conclude that there is only partial support for Requirement *R4*.

While WS-CDL focuses on the behavioral aspects of a choreography, it relies on WSDL for the specification of message formats. Therefore, it supports Requirement *R5*. However, as a drawback of this integration with WSDL, the WSDL-configurations heavily influence the way the choreography looks like. Changes in WSDL files often require changes in the WS-CDL choreography. E.g. splitting a port type into several port types requires major refactoring of the choreography. Therefore, WS-CDL does not support Requirement *R6*.

It is possible to specify timeouts for interactions (Requirement *R7*). Listing 1 includes such a timeout. It is realized through an exception block with a dedicated exception type. Furthermore, there is a variety of other exception types available to cover message transmission exceptions, security failures and application failures (Support of *R8*).

Correlation of interactions is addressed using identity tokens that are to be included in messages (Support of *R9*). The so called token locators are a mechanism to retrieve tokens from messages: an XPath expression defines where in the message the correlation information is placed. This is an example of how tightly WS-CDL is linked to WSDL. Whenever the message format changes, the token locators have to be adapted to the new format. This direct influence of message format changes to the choreography definition is not desired.

There have been proposals to generate abstract BPEL processes out of WS-CDL choreographies [41]. However, it remains unclear how constructs like blocking work units can be realized in BPEL. WS-CDL also allows the specification of mixed choices on a global level, i.e. choices between send and receive activities. Mixed choices on a global are a major challenge when trying to properly translate them to interacting BPEL processes. Here, sophisticated synchronization mechanisms might need to be applied. Furthermore, round-tripping between WS-CDL and BPEL seems unrealistic given e.g. the fact that BPEL includes a `forEach` construct which does not have a correspondence in WS-CDL. Therefore, we conclude that Requirement *R10* is not supported.

3.6 Assessment of Let's Dance

Let's Dance [59] is a visual choreography language targeted at business analysts. Its origins can be found in the service interaction patterns initiative, where Let's Dance was positioned as first language to support most patterns. It does not allow any technology-specific configurations. Like WS-CDL, Let's Dance is also an interaction modeling language. Let's Dance distinguishes the concepts *roles* and *actor references*. Roles correspond to what we call service types and actor references to service references.

Interactions between more than two services can be expressed in Let's Dance (*R1*). The core version of Let's Dance only comes with a behavioral view of choreographies, which allows to express the behavioral constraints between interactions (*R2*). Extensions have been proposed for also representing service topologies in [5]. Service sets are an essential feature in Let's Dance, where a number of services of the same role can be addressed in "for each" interactions. The concrete number does not need to be known at design-time of the choreography. This leads to full support for requirement *R3*. While the notion of reference passing is present in Let's Dance, service selection relationships cannot be modeled (*R4*).

Let's Dance is a language on the conceptual level, meaning that there is no technology-specific details in the language. No integration with WSDL or any other language for describing message formats has been done (*R5*). Therefore, interchangeability of technical configurations is not given, as the notion of technical configurations is completely absent in the language (*R6*).

Timeouts are realized using timers, a special form of interaction in Let's Dance. However, time constraints are only given as free text, therefore only leading to partial support for *R7*. Exception handling is not present (*R8*) and correlation is not addressed, either (*R9*).

The generation of abstract BPEL processes was shown in [60]. While most control flow constructs introduced in Let's Dance are mapped to constructs in BPEL, the mapping does not cover a number of aspects present in Let's Dance. As examples, timer interactions are not mapped, and roles and actor references are ignored. Therefore, we conclude that Requirement *R10* is at most partially supported.

3.7 Assessment of BPMN

The Business Process Modeling Notation (BPMN, [48]) 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. Thus, BPMN allows to model interconnected participant behavior descriptions. However, BPMN lacks formal semantics and thus there is no standardized execution semantics. There are approaches to interpret the informal execution semantics of BPMN and use this interpretation to map BPMN to BPEL (e.g. [52]). We show in [54] how BPMN can be extended to enable the generation of fully defined BPEL4Chor choreographies. [54] also specifies how the transformation is carried out. Our work of [18] shows how BPMN can be used in conjunction with BPEL4Chor to model service choreographies.

Involvement of two or more services can be represented by a corresponding

number of so called pools (*R1*). By collapsing pools and only showing message flow between them, service topologies can be modeled. However, as pools represent service types and it cannot be defined how many services of one type will be involved, there is only partial support for *R2*. The notions of service sets (*R3*), selection of services and reference passing cannot be found in BPMN (*R4*). While BPMN is primarily targeted at conceptual modeling, there are attributes to define concrete message formats (*R5*). However, interchangeability of such technical configurations is not given (*R6*). Time constraints can be modeled using intermediate timer events (*R7*) and exception handling via error events (*R8*). Correlation configurations cannot be set in BPMN (*R9*).

An integration with existing service orchestration languages is given through the extensive work on mapping BPMN to BPEL. While mapping in both directions is possible for a large number of constructs, different coverage of concepts and semantic differences between corresponding constructs do not allow for complete round-tripping (cf. [58]). Therefore, we conclude that there is only partial support for Requirement *R10*.

3.8 Assessment of iBPMN

iBPMN [15] is an extension to BPMN allowing for interaction modeling. While most of BPMN's control flow constructs are reused, certain restrictions are imposed, e.g. there are no tasks and only events, complex interactions and gateways can be arranged in the control flow. Decision points and non-message events, such as timeouts or exceptions, must be assigned to pools, indicating who own the choice or who can observe the events.

In terms of support for the different requirements, there is no difference between BPMN and iBPMN regarding *R1* and *R3* through *R9*. iBPMN comes with the concept of shadowed pools indicating that a number of services of the same type will be involved in a choreography. Therefore, *R2* is now fully supported. However, referencing particular sets of services is still not possible.

R10 is not supported in iBPMN as the interaction model cannot directly be mapped to orchestrations and a transformation of iBPMN to interface behavior descriptions (e.g. given in BPMN) is largely missing.

3.9 Assessment of BPSS/UMM

The ebXML initiative (`http://ebxml.org/`) proposed the Business Process Schema Specification (BPSS, [13]) to describe choreographies.

BPSS is closely related to the UN/CEFACT Modeling Methodology (UMM, [25]) which mainly describes the different steps to specify choreographies in a technology-independent manner. UMM also provides a meta-model for choreographies, including the business transactional view, the business service view and business collaboration view. Business transactions serve as basic building blocks, each involving a request-response interaction plus additional business signals to synchronize the state of the two business partners involved. These bi-lateral transactions are described using UML 1.4 Activity Diagrams [47]. Transactions are composed into a business collaboration protocol, the choreography. UMM's business service view finally specifies the services and operations (or messages) participants must support in order to implement a role.

On the behavioral level, BPSS is limited to bi-lateral scenarios, having no support for the first three requirements (*R1*, *R2*, *R3*). Service selection and reference passing are not covered, either (*R4*).

While BPSS is technology-independent, WSDL files can be referenced in the Collaboration Profile Agreement (*R5*). Timeouts and exception handling can also be specified (*R7*, *R8*). However, interchangeability and correlation configurations are not given (*R6*, *R9*).

There has been a mapping of business transactions to executable BPEL code [26]. However, as business collaborations are not covered, we conclude that there is at most partial support for an integration with orchestration languages (*R10*).

3.10 Assessment of SCA

The Service Component Architecture (SCA) provides a model for building applications based on SOA [24, 49]. Implementations of services are wrapped in components. Each component provides a set of services and requires a set of interfaces. The required services have to be wired with provided services form a valid SCA composite. SCA itself does not provide a specific language for the implementation of components and thus evaluations targeting the implementation are not applicable (n/a). SCA supports BPEL as a possible implementation language [50]. Thus SCA integrates with service orchestration languages (*R10*). If BPEL is taken as implementation language, SCA also supports all of the requirements BPEL supports.

In the SCA assembly model, the used components, the wires and the multiplicity of these wires are specified (*R2*). By specifying 1..n as multiplicity at a wire, one can denote service sets (*R3*). Each interface is specified by a Java interface, WSDL 1.1 port types or WSDL 2.0 interfaces. Thus, the specification of

message formats is possible (*R5*). Due to the dependency to a concrete interface, technical configurations cannot be interchanged (*R6*).

3.11 Other Choreography Approaches

Several industry initiatives have worked towards domain-specific choreography models in order to enable an easier integration between different companies of that domain. Examples for such initiatives are RosettaNet (`http://www.rosettanet.org/`) for the supply chain domain, SWIFTNet (Society for Worldwide Interbank Financial Transfer, `http://www.swift.com/`) for financial services or HL7 (Health Level Seven, `http://www.hl7.org/`) for health care services. Due to a lack of choreography modeling languages available, these initiatives have mostly resorted to textual descriptions of the choreographies. Rather adhoc-notations where used for illustration. However, both the textual descriptions and the illustrations have many ambiguities, allowing for different interpretations. When it comes to execution, the current way is to map the proprietary notations to BPEL by making heavy assumptions about the concrete semantics. Such a mapping is currently available for RosettaNet only and presented in [29].

WSDL 2.0 Message Exchange Patterns [46] offer means to specify the order of messages an operation expects and sends back. This ordering is specified using text and not a designated process description language. It is possible to specify node types, which enables describing multi-lateral interactions from the view of one participant. However, the specification is limited to one operation and thus does not cover a whole choreography which may span over multiple operations.

Most languages mentioned above are procedural languages, where the approach is to explicitly enumerate all interactions possible in a certain situation. As an alternative to these procedural languages there are also approaches for a declarative style of modeling. Declarative in this context means that all constraints are enumerated that apply for a set of interactions. That way, an "empty" choreography would mean that all interaction sequences are allowed, while adding constraints limits the number of allowed sequences. In procedural languages, an "empty" choreography would mean that no interaction is allowed and adding constructs enlarges the set of possible interaction sequences. The goal of declarative languages is to avoid overspecification, a common phenomenon that can be observed whenever procedural languages are used. An overview of declarative workflow models is given in [53]. Let's Dance can be seen as a hybrid of procedural and declarative languages. It has control flow constructs for loops and enabling like procedural languages but also contains an "Inhibits" relationship, where an interaction is no longer allowed once another

interaction has happened.

3.12 Summary of the Assessments

Table 1
Assessment of WS-CDL and BPEL

Requirements	BPEL	BPEL^{light}	WSFL	WS-CDL	Let's Dance	BPMN	iBPMN	BPSS/UMM	SCA
R1. Multi-lateral interactions	+	+	+	+	+	+	+	–	n/a
R2. Service topology	–	–	+	+/–	–	+/–	+	–	+
R3. Service sets	+/–	+/–	–	–	+	–	–	–	+
R4. Selection of services and reference passing	+/–	+/–	+/–	+/–	+/–	–	–	–	n/a
R5. Message formats	+	+	+	+	–	+	+	+	+
R6. Interchangeability of technical configurations	–	+	+	–	–	–	–	–	–
R7. Time constraints	+	+	–	+	+/–	+	+	+	n/a
R8. Exception handling	+	+	+	+	–	+	+	+	n/a
R9. Correlation	+	+	–	+	–	–	–	–	n/a
R10. Integration with service orchestration languages	+	+	+/–	–	+/–	+/–	–	+/–	n/a

Table 1 gives an overview of the assessment of the different languages. As drawbacks of WS-CDL we see that service sets and service selection are not fully supported. Especially the fact that an unknown number of services of the same type is not supported in WS-CDL decreases the suitability of WS-CDL to represent complex choreographies. Furthermore, WS-CDL is too inflexible in terms of changing technical configurations. However, the main drawback of WS-CDL is that it is not well aligned with BPEL, which leaves service choreographies and service implementation disconnected.

BPEL provides better support for service sets than WS-CDL, but still service sets need to be a first-class citizen in choreographies. Implementations requiring complex XPath expressions can only be seen as workaround. Also the notion of service selection is conceptually too important in choreographies to leave it to engine-specific deployment configurations. By "service selection" we do not mean concrete endpoints. It should not be specified, how and when a concrete service is bound, but which service selects which other services. In

the example scenario, it should be possible to specify that the seller selects the broker and not vice versa. BPEL is equally inflexible regarding changing technical configurations as WS-CDL is. BPEL^{light} already resolves this problem. However, the main reason why BPEL and BPEL^{light} simply cannot be used as choreography languages is that they do not provide a global view on interacting services.

The other languages have far less support for the requirements. WSFL does not support advanced control-flow constructs such as exception handling and timing. The handling of service sets as well as the support of link passing are also open issues. SCA is an architectural principle and not a language to specify control flow. The problem of Let's Dance is that technical configurations cannot be done. Let's Dance operates on a purely conceptual level. BPMN and iBPMN lack support for distinguishing different services of the same type and they are also weak on the technical side. BPSS/UMM are early approaches where fundamental concepts such as multi-lateral choreographies are not supported.

The comparison of these languages shows that BPEL, BPEL^{light} and WS-CDL already provide the broadest support for the requirements. However, they are not satisfying with respect to our framework. Introducing a completely new language to overcome the limitations is not desirable, as this would hamper reuse of existing tools and techniques. Therefore, there are two possible solutions:

(i) Enhance WS-CDL with service set capabilities, including a parallel `forEach`-like control flow construct and loosen the coupling with WSDL.
(ii) Introduce choreography extensions for BPEL/BPEL^{light}, providing a topology, a loose link between behavioral definitions and technical configurations, service sets as first-class citizens and the notion of service selection.

Regarding the first solution, Enhancing WS-CDL would still leave the gap to orchestration languages. We opted for the second solution, because of the broad acceptance of BPEL as orchestration language, the huge set of supporting tools and execution platforms, as well as the well-established processes to define and standardize extensions to BPEL. To gain interchangeability of technical configurations, BPEL^{light} replaced BPEL's communication activities. Since the way of BPEL^{light}is not the only way to make BPEL WSDL independent, we introduce BPEL4Chor as new choreography language on top of BPEL.

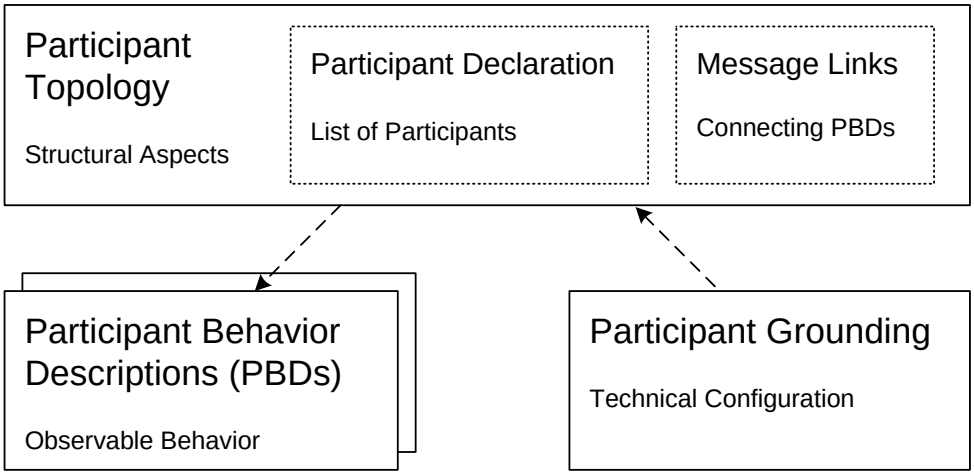

Fig. 2. BPEL4Chor artifacts

4 BPEL4Chor

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. Thus, a higher degree of reusability of the choreography models is achieved.

Modeling a choreography using BPEL4Chor mainly follows the modeling approach presented in [5].

(1) Specify participant types and the participants. In our example this would be the sellers, bidder and the broker service.
(2) Specify the message links between the participants. This specifies the business documents that are exchanged between the participants. E.g. the bid that is sent from a bidder to the broker service and the payment details sent from the seller to a bidder.
(3) Specify the behavioral dependencies and data flow between message exchanges by defining each participant's behavior. Here, the allowed message orders are defined. E.g. bids are only received after the auction has been set up by the seller, while the payment details and the stock options grant can be sent in any order.
(4) (Optional) Ground each message link to concrete services. E.g. it is specified what concrete serialization formats for the messages are used. It is defined how the interaction will look "on the wire".

BPEL4Chor is a collection of three different artifact types. Each artifact type is represented as a rectangle in Fig. 2, while the dashed arrows between the rectangles symbolize that there exist references from artifacts of one type to artifacts of the other type.

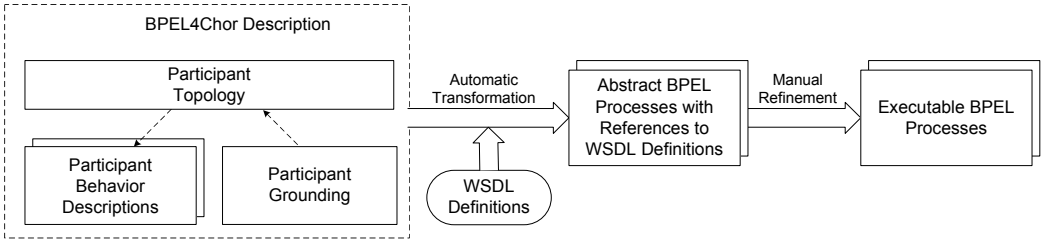

Fig. 3. Getting from a BPEL4Chor choreography to executable BPEL processes

(1) A *participant topology* defines the structural aspects of a choreography by specifying participant types, participant references, and message links.
(2) *Participant behavior descriptions* define the control flow and data flow dependencies between activities, in particular between communication activities, at a given participant. By assigning a message link to a communication activity, participant behavior descriptions plug into the topology.
(3) A *Participant grounding* defines 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. We use the term "grounding" similar to the Semantic Web terminology, where "grounding" denotes that the semantically described service is bound to a concrete technical configuration.

It is possible to get an executable process from each participant behavior description as shown in Figure 3. First of all, all technical configurations of each participant have to be specified in the participant grounding. This serves as input for a transformation can be run. This transformation transforms each participant behavior description into a BPEL process, where the technical details given in the participant grounding are included. However, process-internal activities, such as data manipulation or calling other services to fulfill the requirement are missing. These have to be manually added afterwards. The details of the transformation itself are described in Section 6.

The following sections are going to introduce the artifact types of BPEL4Chor. Corresponding code snippets are given for the example given in Section 2.

4.1 Participant Topology

The participant topology describes the structural aspects of a choreography. As most the important aspect, the participant topology enumerates different *participant types*. BPEL4Chor supports the following three cases: (i) there is only one participant of a certain type in one conversation (choreography instance). As an example, an individual seller and a single broker service involved in a specific auction can be imagined. (ii) Several participants of a certain type appear in one conversation and the number of participants is

known at design-time. As an example imagine a scenario where two shippers are involved in one conversation. (iii) An unbounded number of participants is involved and the exact number might only be determined at runtime. Imagine a large number of bidders involved in our sample scenario.

Participant types are not sufficient to support cases (ii) and (iii), as we need to distinguish between different participants of the same type. E.g. we need to distinguish between a bidder who has won the auction from a bidder who has not. Therefore, the notions of *participant reference* and *participant set* are introduced. A participant reference describes an instance of a participant type, e.g. one particular bidder, while participant sets describe a set of participant references, e.g. the set of all unsuccessful bidders.

Cardinality of participant types is implicitly given through the participant reference and participant set declarations. (i) If there is only a participant reference for a given participant type, we can conclude that there will only be at most one participant of that type involved in a conversation. (ii) If there are several participant references but no participant set for a given type, then the number of participants in one conversation is limited by the number of references. (iii) If there is a participant set declared for a given type, then the number of participants is not defined at design-time.

Listing 2 shows the participants in the participant topology for the auctioning scenario. Three participant types, namely seller, broker service and bidder, are listed along with a participant reference for a seller and an broker service, respectively. A participant set for the bidders is also given. The participant reference `currentBidder` contained in the set will be used as iterator on the set later on. `successfulBidder` is one particular participant from the set of bidders. In general, the semantics of a reference p contained in a set s is that if a sender p is not contained in s, then this new sender is added to the set.

Using different participant references does not guarantee that the referenced participants are different at runtime. The same applies to sets, which may overlap in terms of referenced participants.

In Listing 2, the declaration of the seller declares a `selects` attribute that refers to the broker service. This indicates that the seller chooses which broker service she actually wants to use. This in turn implies that there are potentially many broker services available. 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 seller has exactly one broker service she always goes to.

The attribute `forEach` on the set of unsuccessful bidders denotes that the `forEach` activity having the name `notifyUnsuccesfulBidders` at the auctioning service should iterate over that set. The attribute `forEach` at the nested participant `currentBidder` denotes that this participant reference

Listing 2 Participants in the participant topology

```
<topology name="topology"
  targetNamespace="urn:auction"
  xmlns:sns="urn:auction:seller" ...>
 <participantTypes>
   <participantType name="Seller"
     participantBehaviorDescription="sns:seller" />
   <participantType name="BrokerService" ... />
   <participantType name="Bidder" ... />
 </participantTypes>
 <participants>
   <participant name="seller" type="Seller"
     selects="brokerService" />
   <participant name="brokerService" type="BrokerService" />
   <participantSet name="bidders" type="Bidder">
     <participant name="bidder" selects="brokerService" />
     <participant name="successfulBidder" />
   </participantSet>
   <participantSet name="unsuccessfulBidders" type="Bidder"
    forEach="as:notifyUnsuccesfulBidders">
     <participant name="currentBidder"
      forEach="as:notifyUnsuccesfulBidders" />
   </participantSet>
 </participants>
 ...
</topology>
```

should store the current value of the iterator. A `forEach` may only iterate on one set. Thus, we require that for each `forEach` there is at most one participant set and at most one participant reference pointing to it.

Listing 3 shows the message links in the participant topology for the auctioning scenario. *Message links* state which participant can potentially communicate with which other participants. However, the topology does not include any constraint about ordering sequences or the cardinality of message exchanges. BPEL4Chor is based on a *closed world assumption*. In particular, message links only connect participants listed in the topology. I.e. the sender and receiver attributes must always be set.

Some of the message links in Listing 3 also contain the attribute `participantRefs`. This attribute realizes link passing mobility in BPEL4Chor: as part of the exchanged business documents, participant references are passed from one participant to another. E.g. consider the message link `completion-NotificationLink`. Here, the reference to the successful bidder is passed from the auctioning service to the seller. This enables the seller to directly communicate with this bidder later on.

Listing 3 Message links in the participant topology

```
<topology name="topology"
  targetNamespace="urn:auction" ...>
 ...
 <messageLinks>
   <messageLink name="auctionRequestLink"
    sender="seller" sendActivity="sendAuctionCreationRequest"
    bindSenderTo="seller"
    receiver="auctioningService"
    receiveActivity="receiveAuctionCreationRequest"
    messageName="auctionCreationRequest" />
   ...
   <messageLink name="bidLink"
    senders="bidders" sendActivity="sendBid"
    bindSenderTo="bidder"
    receiver="auctioningService" receiveActivity="receiveBid"
    messageName="bid" />
   <messageLink name="bidAckLink"
    sender="auctioningService" sendActivity="sendBidAck"
    receiver="bidder" receiveActivity="receiveBidAck" />
   ...
   <messageLink name="completionNotificationLink"
    sender="auctioningService"
        sendActivity="sendCompletionNotification"
    receiver="seller" receiveActivity="completionNotification"
    messageName="notification"
    participantRefs="successfulBidder" />
   <messageLink name="unsuccessfulBidLink">
    sender="auctioningService" sendActivity="sendUnsuccessfulBid"
    receiver="currentBidder" receiveActivity="receiveUnsuccessfulBid"
    messageName="notification" />
   </messageLink>
   ...
 </messageLinks>
</topology>
```

Reference passing must also happen whenever the attribute `bindSenderTo` is set for a message link. In contrast to `participantRefs`, where a reference to a third participant is passed, `bindSenderTo` implies that the sender of the message must include a reference to herself in the message. As knowledge about participants is local, the usage of `bindSenderTo` is required even in those scenarios where only one participant of a type is involved. Take `auctionRequestLink`. Here, the seller sends a reference pointing to herself, enabling the auctioning service to reply later on.

Selection and reference passing lead to the *binding* of concrete participants to participant references. In the case of reference passing, rebinding occurs if

a participant was already bound to a reference. The reference is simply overwritten. If a referenced participant should be bound to a different participant reference, the attribute `copyParticipantRefsTo` is used.

Special semantics apply if binding occurs for a participant reference contained in a participant set. Here, the reference must be added to the set upon binding, provided that a reference to that participant is not yet contained in the set.

The listing also shows that either the attribute `sender` or `senders` is used in a message link. `sender` is used if one participant reference applies, `senders` is used if any participant out of a set can be the sender.

Topologies by themselves only describe the basic structural aspects of a choreography. However, the typical usage for topologies is to glue together participant behavior descriptions. Therefore, one already finds connections to constructs from participant behavior descriptions in the topology. These attributes are explained in the next section.

4.2 Participant Behavior Descriptions

Participant behavior descriptions (PBDs) cover the behavioral aspects of a choreography. Control flow dependencies between communication activities are defined per participant type. These dependencies determine the ordering sequences of message exchanges the different participants have to adhere to. Furthermore, data flow aspects are covered in the participant behavior descriptions. It is important to specify what data can be expected by the receiver of a message and how this data relates to data previously exchanged.

Abstract BPEL is used as basis for participant behavior descriptions. It already provides most constructs that are needed. In contrast to executable BPEL, some language constructs do not need to occur and some attributes do not need to be set. BPEL profiles force or forbid the usage of certain attributes in abstract BPEL process. Therefore, we will introduce the *Abstract Process Profile for Participant Behavior Descriptions* stating the requirements for the definition the behavior of one participant. This profile inherits all constraints of the *Abstract Process Profile for Observable Behavior* from the BPEL specification. This allows us to add e.g. opaque activities into a PBD, which is useful for documentation purposes.

Communication activities. Message send and receive activities are at the center of attention in choreographies. BPEL introduces `invoke` and `reply` as send activities and `receive` and `onMessage` as receive activities. These activities are reused in BPEL4Chor.

Listing 4 Constraints of the "Abstract Process Profile for Participant Behavior Descriptions" on the invoke activity.

```
<invoke
  partnerLink="NCName"
  portType="QName"?          must not be used
  operation="NCName"
  inputVariable="BPELVariableName"?
  outputVariable="BPELVariableName"?
  standard-attributes
  wsu:id="NCName"            must be used
>
...
</invoke>
```

The usage of `partnerLink`, `portType` and `operation` attributes at the communication activities of BPEL link the BPEL process tightly to WSDL operations. The communication activities are linked together using message links in the topology as described in the last section. Thus, a linkage to WSDL artifacts is obsolete. Therefore, we forbid the usage of the attributes `partnerLink`, `portType` and `operation` at communication activities.

In the topology, a message link references two communication activities. To enable proper referencing, we need an identifier for each communication activity in a participant behavior description. Since `onMessage` branches do not offer an attribute `name`, we introduce the attribute `wsu:id` having the type `xsd:id` as new attribute for communication activities and `onMessage` branches. To simplify reading, the `wsu:id` attribute defaults to the attribute `name` of the BPEL activity. Listing 4 presents the modifications in the case of the `invoke` activity. Typically, one message link references exactly one send and one receive activity. However, there are scenarios, where one message link references several send and/or receive activities. E.g. several send activities can have the same target activity or several receive activities refer to the same message type.

BPEL directly adopts some of the WSDL interaction styles. BPEL distinguishes between one-way interaction and request/response interactions. We argue that the choice of the interaction style is a configuration issue and should normally happen in the phase of the technical configuration. Nevertheless, we allow that both interaction styles can be used in the participant behavior descriptions. This allows higher similarity between participant behavior descriptions and orchestrations. Especially in a bottom-up approach where existing BPEL files are the starting point, we do not want to force the modeler having to split e.g. a request/response activity into two activities in the BPEL code. In this context, we force the attribute `messageExchange` to be present in order to relate pairs of `receive` and `reply` activities. This constraint is necessary, because corresponding pairs can no longer be determined by matching `portType` and `operation` values.

Control flow. BPEL comes with a rich set of control flow constructs, which are used unchanged in BPEL4Chor. This enables the reuse of existing BPEL tools to model choreographies. As examples `if`, `pick`, `flow` and `forEach` are available to model branching structures and concurrency.

Timing constraints play an important role in choreographies. Offers might only be valid for a certain timespan. In our example, an auction begins at a certain point in time and only lasts for a certain timespan. BPEL's facilities to model time constraints, namely `wait` and event handlers with `onAlarm`, are also reused unchanged.

Support for participant sets. Scenarios where a number of participants of the same type are involved in one conversation are recurrent in the choreography world. BPEL supports parallel instances the number of which might only be known at runtime through the `forEach` construct. Therefore, there is sufficient support for participant sets from the control flow side. In contrast to this, data sets are not natively supported in BPEL. Special XSD types have to be used for this purpose.

As a realization for the iteration on participant sets, the attribute `forEach` can be set for participant references and participant sets in the participant topology. The semantics is that the corresponding set determines the number of branches spawned and the participant reference is used as iterator.

Data flow. It should be specified in a choreography what contents can be expected by the receiver of a message. As an example the quantity of ordered products in an order message and the quantity in an order acknowledgment message must not differ. Furthermore, data values can determine branching behavior in choreographies. E.g. a seller might need to be ready to provide further details when the goods offered have a certain value.

BPEL offers a rich set of constructs to model data access and data manipulation. Variables are used as output of receive activities and input of send activities. Data "flows" from one activity to another by using one variable as output variable for the first activity and input variable of the other, provided that no other activity overwrites the variable in between.

In the general case, it should be possible to leave variables untyped to offer a greater flexibility to the business user. This is in line with the *Abstract Process Profile for Observable Behavior*, which does not require the specification of a type for each variable. However, typed variables are necessary to define data-based decisions in detail. Otherwise, branching conditions can only be formulated as plain text.

For documentation purposes it is sometimes useful to add hints about dependencies between data values. E.g. it could be specified that the decision about

who is the successful bidder in an auction should be based on the height of the bid. Assign activities with opaque `from` or `to` parts can be used in this context.

Listing 5 Participant behavior description for the auctioning service

```
<process name="auctioningService"
  targetNamespace="urn:auction:auctioningService"
  abstractProcessProfile=
    "urn:HPI_IAAS:choreography:profile:2006/12">
<sequence>
  <receive name="receiveAuctionCreationRequest"
   createInstance="yes" />
  ...
  <scope>
    <eventHandlers><onAlarm .../></eventHandlers>
    <sequence>
      <receive name="receiveBid" />
      <invoke name="sendBidAck" />
    </sequence>
  </scope>
  <flow>
    <invoke name="sendCompletionNotification" />
    <forEach name="notifyUnsuccesfulBidders">
      <scope><invoke name="sendUnsuccessfulBid" /></scope>
    </forEach>
    <invoke name="sendSuccessfulBid" />
  </flow>
</sequence>
</process>
```

Message correlation. 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 the attribute `correlationSet`, 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, in contrast to BPEL, properties can be left untyped and are not bound to WSDL. Actual typing will happen in the participant grounding.

Listing 5 shows the participant behavior description (PBD) for the auctioning service in the example given in Section 2. The abstract BPEL profile for participant behavior descriptions is referenced.

4.3 Participant Grounding

While the participant topology and the participant behavior descriptions are free of technical configuration details, the participant grounding introduces the mapping to web-service-specific configurations. So far, port types and operations are left out and XML schema types for messages are not mandatory. In the participant grounding, these aspects are brought in. The participant grounding is specific to the target platform. While we use BPEL as target platform, it is possible to replace the participant grounding by a participant grounding specific to other target platforms, such as BPEL4SWS [43], to enable a semantic-based execution of each participant.

After a process is grounded, the participant behavior descriptions can be transformed to abstract BPEL processes, where partner links, port types and operations are defined (cf. Section 6). Since BPEL depends on WSDL 1.1 [12], message links, participant references and properties have to be assigned to WSDL artifacts in a participant grounding.

Message links are targeted at one or more receive activities. A message link models the sending and consumption of one message and does not model multicast. Therefore, one message link is grounded to one WSDL operation. This allows for realizing one participant through different port types. The attributes `participantRefs` and `bindSenderTo` enable link passing mobility in BPEL4Chor choreographies. In the case of executable BPEL, end service references are passed in messages. The mapping of participant references to concrete service references is done by grounding a participant reference to a WSDL property. A WSDL property states where a certain element is located in different message types. Using that property, a BPEL process can extract the concrete service reference out of an incoming message regardless of the type of the incoming message. In that way, the service reference of a passed participant reference can be located in different messages. For correlation, we allowed untyped correlation sets. With the participant grounding, these sets get typed.

Listing 6 presents the participant grounding for the example given in Section 2. Each message link is grounded to a WSDL operation and each participant reference is grounded to a WSDL property.

4.4 Consistency between BPEL4Chor Artifacts

This section will introduce a number of constraints. These constraints are used to ensure consistency between BPEL4Chor artifacts.

Listing 6 Participant grounding

```
<grounding topology="auc:topology"
  xmlns:top="urn:auction" ...>
  <messageLinks>
    <messageLink name="auctionRequestLink"
     portType="auc:auction_pt"
     operation="auctionRequest" />
    <messageLink name="bidLink"
     portType="auc:auction_pt"
     operation="bid" />
    <messageLink name="bidAckLink"
     portType="bid:bidder_pt"
     operation="bidAck" />
    ...
  </messageLinks>
  <participantRefs>
    <participantRef name="seller"
     WSDLproperty="msgs:sellerProp" />
    <participantRef name="bidder"
     WSDLproperty="msgs:bidderProp" />
  </participantRefs>
</grounding>
```

Participant references and message links. If there are several senders in a message link and the attribute `bindSenderTo` is set, all senders must be of the same type as the participant reference defined in the attribute. If the attribute `copyParticipantRefsTo` is set, the list must match the `participantRefs` list in terms of length, participant types and cardinality.

A participant set having an attribute `forEach` must contain exactly one participant with a matching attribute `forEach` for every `forEach` listed.

Message links and communication activities. For every `invoke` and `reply` (`receive` and `onMessage`) activity there must be at least one message link in which this activity is a send (receive) activity. In the other direction, the send (receive) activities given in a message link must be `invoke` or `reply` (`receive`, `onMessage` or `invoke`) activities.

If `senders` is specified in a message link l and the receiving activity is connected to a reply activity through an attribute `messageExchange`, `bindSenderTo` has to be specified in the link l.

Synchronism issues. If the output variable is specified for an `invoke` activity, it must appear as `receiveActivity` in a message link. The synchronous call must be matched on the receiving side by `receive` activities with corresponding `reply` activities. On the other hand, each pair of `receive` / `reply` activities must be matched by a corresponding `invoke` activity.

Completeness of participant grounding. All passed participant references, used message properties and message links must be grounded. If variable types are defined for a send or receive activity, the variable type must match the type of the expected port type. Furthermore, references passed via a message link must be grounded in a WSDL property. This applies to the two attributes `participantRefs` and `bindSenderTo`.

Conflicting groundings of message links can occur especially in choreographies with synchronous interactions. The corresponding message links must be grounded in one WSDL request-response operation.

Further consistency issues. The consistency constraints mentioned above can be detected easily in a BPEL4Chor choreography on a syntactical level. However, there are certain anomalies that are not covered yet and which require behavioral analysis. The most obvious anomaly is a deadlock, where a participant waits for messages that will never be sent by other participants. Such deadlocks are typically be detected in compatibility checking techniques [21]. An approach for checking the absence of deadlocks and proper termination was presented by Lohmann et al. [37], where the participant behavior descriptions are translated to Petri nets [57] and connected via communication places as it can be derived from the topology. However, this approach abstracts from data flow and data-based decisions are treated as non-deterministic choices.

Another challenge is the verification of proper reference passing. In scenarios where dynamic binding is used, it must be ensured that the selection of participants is properly propagated to the corresponding participants. We have already presented a technique to check "instance isolation" in the context of concurrent conversations [22]. It checks whether correlation information was properly set. However, a mapping from BPEL4Chor to ν^*-nets, an extension for Petri nets used in instance isolation analysis, is not available yet.

When multiple participants of the same type are involved, it may frequently occur that some of the combinations of communication activities and participant references are not used in message links. For example, the receive activity `receiveUnsuccessfulBid` is contained in the participant behavior description for the participant type `Bidder`. Concrete participants for this type are `bidder`, `successfulBidder` and `currentBidder`. The message link `unsuccessfulBidLink` is only one message link targeting `receiveUnsuccessfulBid`. This message link specifies `currentBidder` as receiver. Thus, the activity `receiveUnsuccessfulBid` of the participants `successfulBidder` and `currentBidder` will never be used. In the example process, this does not cause a problem, since this activity is only reached for unsuccessful bidders. In order to decide reachability for particular participants, the overall behavior has to be considered. However, these analyses are beyond the scope of this article.

These constraints have to be ensured directly by a BPEL4Chor modeling tool or by a dedicated tool to check BPEL4Chor choreographies. Currently, BPEL2oWFN [37] implements an analysis of a subset of the presented constraints.

5 Assessment of BPEL4Chor

This section presents an assessment of BPEL4Chor using the service interaction patterns [7]. It illustrates how suitable BPEL4Chor is with regard to complex interaction scenarios. The examples provided further illustrate the capabilities of BPEL4Chor beyond what was already presented in the listings of Section 4. Most importantly, the assessment serves as input to present which of the requirements presented in Section 3 are partially or fully supported by BPEL4Chor. The result is presented in Section 5.5.

5.1 Single-transmission bilateral interaction patterns

The two patterns ***Send*** and ***Receive*** are directly supported in BPEL4Chor through the send activities `<invoke>` and `<reply>` and `<receive>` activities when being interconnected using message links in the participant topology. The requesting part of ***Send/receive*** can be realized in two ways: either by using the synchronous `<invoke>` activity or using a combination of one-way `<invoke>` and `<receive>`. The responding party can be realized using a combination of `<receive>` and `<reply>` or `<receive>` and `<invoke>`. In all these cases, `<onMessage>` is an alternative to use `<receive>`. BPEL4Chor allows that a receiver of a message is bound at design-time or at runtime. Runtime re-binding is allowed through the `copyParticipantRefsTo` attribute in a `<messageLink>` definition. Listing 7 lists the topology and Listing 8 lists the PBD for the requestor. Since there is one single message link leaving `sendRequest`, the `<invoke>` is a one-way invoke, otherwise there would have been a message link targeting `sendRequest`.

5.2 Single-transmission multilateral interaction patterns

In the case of the ***Racing incoming messages*** pattern, a party expects to receive one message among a set of messages. This pattern is directly supported by the `<pick>` activity. It is both possible that messages have different types and that messages can originate from different senders. If the participant reference that is used for the respective receive activity is not bound when this

Listing 7 Participant topology for the Send/receive pattern

```
<participants>
  <participant name="a" type="Requestor" />
  <participant name="b" type="Responder" />
</participants>
<messageLinks>
  <messageLink sender="a" sendActivity="sendRequest" receiver="b"
      receiveActivity="receiveRequest" messageName="request" />
  <messageLink sender="b" sendActivity="sendResponse" receiver="a"
      receiveActivity="receiveResponse" messageName="response" />
</messageLinks>
```

Listing 8 Participant behavior description for participant type Requestor in the Send/receive pattern

```
<sequence>
  <invoke name="sendRequest" />
  <receive name="receiveResponse" />
</sequence>
```

activity is reached, messages from arbitrary senders can be received. Listing 9 lists the topology and Listing 10 lists the PBD for the receiver.

Listing 9 Participant topology for the Racing incoming messages pattern

```
<participants>
  <participant name="a" type="Receiver" />
  <participantSet name="senders" type="Sender">
    <participant name="b" />
  </participantSet>
</participants>
<messageLinks>
  <messageLink senders="senders" sendActivities="sendX"
      bindSenderTo="b" receiver="a" receiveActivity="receiveX"
      messageName="documentX" />
  <messageLink senders="senders" sendActivities="sendY"
      bindSenderTo="b" receiver="a" receiveActivity="receiveY"
      messageName="documentY" />
</messageLinks>
```

Listing 10 Participant behavior description for participant type Receiver in the Racing incoming messages pattern

```
<pick>
  <onMessage wsu:id="receiveX" /> activity </onMessage>
  <onMessage wsu:id="receiveY" /> activity </onMessage>
</pick>
```

In the case of the ***One-to-many send*** pattern a party sends messages to several parties. Solution: in BPEL4Chor this pattern is directly supported through the notion of participant sets in combination with the `<forEach>`

construct. The number of recipients does not need to be known at design-time. The sender is responsible to select the recipients. This obligation is specified using the `selects` attribute at the declaration of the sender `s`. Listing 11 lists the topology and Listing 12 lists the PBD for the sender.

Listing 11 Participant topology for the One-to-many send pattern

```
<participants>
  <participant name="s" type="Sender" selects="receivers" />
  <participantSet name="receivers" type="Receiver" forEach="s:fe1">
    <participant name="r" forEach="s:fe1" />
  </participantSet>
</participants>
<messageLinks>
  <messageLink sender="s" sendActivity="sendDocument" receiver="r"
     receiveActivity="receiveDocument" messageName="document" />
</messageLinks>
```

Listing 12 Participant behavior description for participant type Sender in the One-to-many send pattern

```
<forEach name="fe1" parallel="yes"><scope>
  <invoke name="sendDocument" />
</scope></forEach>
```

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. The arrival of messages needs to be timely so that they can be correlated as a single logical request. 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 the participants whose messages are actually received. Within the `<while>` structure we find a `scope` to which the participant reference `s` is limited. This means that every time the scope is entered, no participant is bound to `s` and a message from any sender can be received. The containment relationship between `s` and `mySenders` has the semantics that if a sender not contained in `mySenders` is bound to `s`, then this new sender is added to the set. Listing 13 lists the topology and Listing 14 lists the PBD for the receiver.

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 attached `<onAlarm>` event handler. Successful vs. unsuccessful completion is directly supported through exception mechanisms. Otherwise, the implementation in BPEL4Chor looks similar to the one for *One-to-many-send*.

Listing 13 Participant topology for the One-from-many receive pattern

```
<participants>
  <participant name="r" type="Receiver" />
  <participantSet name="senders" type="Sender" />
  <participantSet name="mySenders" type="Sender">
    <participant name="s" scope="rcvScope" />
  </participantSet>
</participants>
<messageLinks>
  <messageLink senders="senders" sendActivity="sendDoc"
      bindSenderTo="s" receiver="r" receiveActivity="receiveDoc"
      messageName="document" />
</messageLinks>
```

Listing 14 Participant behavior description for participant type Receiver in the One-from-many receive pattern

```
<while><condition />
  <scope name="rcvScope"><receive name="receiveDoc" /></scope>
</while>
```

5.3 Multi-transmission interaction patterns

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.

The ***Contingent requests*** pattern describes that 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 than the recipient of the last request. Listing 15 lists the topology and Listing 16 lists the PBD for the requestor.

In the case of ***Atomic multicast notification*** a party sends notifications to several parties implying that a certain number of parties are required to accept the notification within a certain timeframe. For example, two or more parties have to accept the notification. In general, the constraint for successful notification applies over a range between a minimum and a maximum number.

Listing 15 Participant topology for the Contingent requests pattern

```
<participants>
  <participant name="a" type="Requestor" />
  <participantSet name="responders" type="Responder"
      forEach="rs:fe1" >
    <participant name="currR" forEach="rs:fe1" />
  </participantSet>
  <participant name="r" type="Responder" />
</participants>
<messageLinks>
  <messageLink sender="a" sendActivity="sendReq"
      receiver="currR" receiveActivity="receiveReq"
      messageName="request" />
  <messageLink sender="r" sendActivity="sendResp" receiver="a"
      receiveActivity="receiveResp" messageName="response" />
</messageLinks>
```

Listing 16 Participant behavior description for participant type Requestor in the Contingent requests pattern

```
<forEach name="fe1"><scope>
  <sequence>
    <invoke name="sendRequest" />
    <pick>
      <onMessage wsu:id="receiveResp" /><empty /></onMessage>
      <onAlarm><for>3m</for><empty /></onAlarm>
    </pick>
  </sequence>
</scope></forEach>
```

There is no direct support for this pattern in BPEL4Chor. As a workaround we could use a similar implementation like the one used for *One-to-many send/receive*.

5.4 Routing patterns

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 (P1, P2, ..., Pn) depending on the evaluation of certain conditions. Solution: BPEL4Chor directly supports link passing mobility through the `participantRefs` attribute of the `<messageLink>` element. Since participant sets can also be used as values for that attribute, the number of passed references does not need to be known at design-time in BPEL4Chor. Listing 17 lists the topology and Listing 18 lists the PBD for the party B.

Relayed request: Party A makes a request to party B which delegates

Listing 17 Participant topology for the Request with referral pattern

```
<participants>
  <participant name="a" type="A" selects="b p"/>
  <participant name="b" type="B" />
  <participantSet name="p" type="P" forEach="b:fe1">
    <participant name="pi" forEach="b:fe1" />
  </participantSet>
</participants>
<messageLinks>
  <messageLink sender="a" sendActivity="sendMsg1" receiver="b"
      receiveActivity="receiveMsg1" messageName="msg1"
      participantRefs="p" />
  <messageLink sender="b" sendActivity="sendMsg2" receiver="pi"
      receiveActivity="receiveMsg2" messageName="msg2" />
</messageLinks>
```

Listing 18 Participant behavior description for participant type B in the Request with referral pattern

```
<sequence>
  <receive name="receiveMsg1" />
  <forEach name="fe1" parallel="yes"><scope>
    <invoke name="sendMsg2" />
  </scope></forEach>
</sequence>
```

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: by using a `<flow>` structure, the responses will be sent to A and to B.

5.5 Summary

BPEL4Chor covers multi-lateral choreographies and introduces a topology. Service selection and reference passing are also present in BPEL4Chor. Therefore, BPEL4Chor fully supports Requirements *R1*, *R2* and *R4*. It was shown in Section 3 that WS-CDL does not support the notion of service sets. On the contrary, BPEL4Chor introduces sets as first class citizen and therefore supports Requirement *R3*.

Concrete message formats can be defined in the participant behavior descriptions and in the participant grounding (Requirement *R5*). As participant grounding files can be replaced without affecting the topology and the participant behavior descriptions, Requirement *R6* is also fulfilled. Regarding Requirements *R7*, *R8* and *R9* BPEL4Chor directly inherits the capabilities

Table 2
Pattern support in BPEL4Chor and WS-CDL

Service Interaction Patterns	BPEL4Chor	BPEL / BPEL^{light}	WS-CDL
S1. Send	+	+	+
S2. Receive	+	+	+
S3. Send/receive	+	+	+
S4. Racing incoming messages	+	+	+
S5. One-to-many send	+	+/–	+/–
S6. One-from-many receive	+	+	+
S7. One-to-many send/receive	+	+/–	+/–
S8. Multi-responses	+	+	+
S9. Contingent requests	+	+	+/–
S10. Atomic multicast notification	–	–	–
S11. Request with referral	+	+	+
S12. Relayed request	+	+	+

of BPEL. The integration of BPEL4Chor and executable BPEL (*R10*) is discussed in the following section.

Table 2 summarizes which service interaction patterns are supported in BPEL4Chor. In analogy to the assessments of WS-CDL in [20] we assign a "+" for direct support of a pattern, "+/–" for partial support and "–" for lack of support.

When comparing BPEL4Chor with WS-CDL the following main differences can be identified:

(1) Unknown numbers of participants are natively supported in BPEL4Chor through the notion of participant sets (cf. S5 and S7 in Table 2). WS-CDL does not directly support parallel conversations with an unknown number of participants.
(2) While a WS-CDL choreography is tightly coupled to WSDL files, web-service-specific details only appear in BPEL4Chor's participant grounding. Therefore, the same choreography can be reused with different port type definitions by changing the participant grounding.
(3) 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 participant grounding can be mapped to BPEL as shown in Section 6.

BPEL4Chor compares to BPEL/BPEL^{light} as follows:

(1) BPEL4Chor is a true choreography language in that it specifies the communication behavior for all services involved in a choreography. The introduction of a service topology provides a structural overview of choreographies, which is not in place in the case of BPEL/BPEL^{light} (*R3*).
(2) The native support for service sets leads to direct support of requirement *R3* and patterns S5 and S7, which were not fully supported in BPEL/BPEL^{light}.
(3) Service selection is natively supported in BPEL4Chor, which was not the case for BPEL/BPEL^{light}.

6 From BPEL4Chor to Executable BPEL

While choreographies serve as interaction contract between business partners, they are not meant to be executed by themselves. However, they can serve as blueprint for actual business process implementation for the different partners. As the usage of executable BPEL for process implementation is a typical case, we discuss the transformation of BPEL4Chor to executable BPEL in this section. An overview of the transformation has been given in Section 4. Recall BPEL offers both, the definition of abstract BPEL processes and executable BPEL processes. Executable BPEL are processes which can be deployed an run at a BPEL engine. The intend usage of abstract processes is defined by a so-called "profile". Recall from Section 3.2 that the BPEL specification defines a profile for process templates and a profile for observable behavior. The profile for observable behavior ensures that the interactions between the participants will not be changed during an *executable completion* of an abstract BPEL process. "Executable completion" is defined in the BPEL specification [51] and describes constraints on the manual actions taken to advance from an abstract BPEL process to an executable BPEL process.

When advancing from BPEL4Chor to executable BPEL two typical scenarios can be distinguished. (i) A BPEL4Chor choreography was set up and agreed upon by a set of business partners. This choreography not only includes the topology and the participant behavior descriptions but also a *complete participant grounding*. A complete participant grounding includes that port types and operations are defined for all message links. Imagine there is one auctioning service dictating the participant grounding. Now each partner uses his generated abstract BPEL process as starting point for internal refinement. (ii) A BPEL4Chor choreography is set up. This time, the choreography is used with different participant groundings. For example, there still is only one auctioning service but some of the very important sellers require *different participant groundings*. The internal process of the auctioning service stays

unchanged for different sellers. In this case, the auctioning service refines the BPEL4Chor choreography by adding internal activities and variables. Then an executable BPEL process is generated for each participant grounding and directly fed into an execution engine.

In both scenarios there is a transformation step, where—combined with the topology and participant grounding—a participant behavior description is mapped to a BPEL process following the *Abstract Process Profile for Observable Behavior* (observable BPEL for short). The transformation from a participant behavior description to observable BPEL processes is an automatic step and described in detail in [56]. While most aspects are copied unchanged from the participant behavior description to the observable BPEL process, four challenges are to be tackled:

- **Generation of partner link types and partner links**. Partner links are excluded in BPEL4Chor, but required in the observable BPEL to specify the partner to interact with. Therefore, partner link types and partner links have to be generated.
- **Realization of BPEL4Chor binding semantics**. Participants are bound if their reference is passed over a message link. The pendant for participant references are service references in observable BPEL. Thus, participant references have to be mapped to service references.
- **Realization of participant sets**. A participant set contains multiple participant references. Since the BPEL specification is not aware of a set of service references, we have to introduce them.
- **Inclusion of participant grounding details**. A participant behavior description leaves out port types and operations. If it comes to the BPEL processes following the "Abstract Process Profile for Observable Behavior", these technical details have to be put into communication constructs, such as `invoke` or `receive`.

6.1 Generation of Partner Link Types and Partner Links

BPEL uses the notion of partner link types to connect two services. A partner link is an instance of a partner link type and is declared in a BPEL process. A partner link denotes which port type is used to send a message and which port type is offered to receive a message. The semantics of partner links is that a partner link holds service references at runtime and denotes the port type to be used. The BPEL specification does not introduce any other runtime semantics at the atomic service level. The concept of partner links is specified in the BPEL specification and added as an extension to WSDL. At the participant grounding, we use WSDL without BPEL specific extensions. Therefore, we generate partner links out of the port types given in the participant grounding.

There may be more than one partner link for a participant if the participant is realized by multiple port types: for example, the participant grounding of the auctioning scenario (Listing 6) shows two different port types for two message links targeting at the bidder.

In general, for each participant reference and message link, a partner link type and a partner link are generated. A partner link is reused, if it is visible at the current activity and if the message link belongs to the same participant reference and the message link is grounded to the same port type. The grounded port type is set to `myRole` if the message link is inbound and the port type is put as `partnerRole` if the message link is outbound.

6.2 Realization of BPEL4Chor Binding Semantics

In the context of BPEL, partner links and correlation sets are the artifacts used to realize binding semantics: if an endpoint reference is copied to a partner link, the participant belonging to the port type is bound. As soon as a reference is passed over a message link, the reference is copied to the partner link. In the case of correlation sets, the correlation set has to be defined and the content of the message has to match it. As a next step, the message is passed to the receiving activity. The port to which the message is sent to may be bound at deployment time or at runtime. At runtime, the port may be bound at process instantiation or at the first usage of the port. If correlation sets are used to realize BPEL4Chor binding semantics, the user has to ensure that the correlation sets are properly used. The last moment when this may happen is when an executable completion of the abstract BPEL process is made.

In the presented auctioning scenario, references are passed over the message link, since arbitrary sellers can register at the auctioning service and the bidders are unknown at modeling time. The seller has to send his service reference to the auctioning service, which in turn uses this service reference to communicate with the seller. The message link `auctionRequest` specifies `bindSenderTo`. Therefore, the auctioning service can use the grounded WSDL property to fetch the service reference out of the received message. That service reference is then copied to the partner link used to send messages to the seller. If there are multiple partner links used for the seller, the service reference has to be a virtual service reference. A virtual service reference can be used by an Enterprise Service Bus (ESB) to determine the endpoint of the service [11]. An endpoint is the concrete point, where a service can be reached.

The participant reference of the successful bidder is passed to the seller in the message link `completionNotificationLink` using the attribute `participantRefs`. Similar to a reference being passed using `bindSenderTo`,

each passed reference is copied to the respective partner link. If a participant set is passed, the set is copied to the variable representing the set (cf. Section 6.3).

6.3 Realization of Participant Sets

A participant set is a set of participant references. The BPEL specification does not specify the XML type of a set of service references. Therefore, we define `sref:service-references` to be a `sequence` of `sref:service-reference` elements.

The `forEach` of BPEL may only iterate over numbers. Therefore, `forEach` activities iterating over participant sets are mapped to `forEach` activities iterating over a number and a nested `assign` activity copying the current service reference to a partner link.

6.4 Inclusion of Participant Grounding Details

In the participant behavior description, communication activities have no partner link and no operation assigned. With the participant grounding and the generation of partner links (cf. Section 6.1), partner links and operations are generated for each communication activity. Thus, the attributes `partnerLink` and `operation` get written at the mapping of each activity.

7 Conclusion

This article has positioned service choreographies as an important artifact to realize successful interaction between business partners. Service choreographies describe the electronic messages exchanged and the dependencies between these exchanges. We presented a list of requirements for service choreography languages. These requirements were used to highlight the shortcomings of using BPEL, WS-CDL and languages used in the context of choreography modeling.

As main contribution, this paper introduced a choreography layer on top of abstract BPEL, namely BPEL4Chor. The different choreography-specific extensions were described and a detailed assessment using the service interaction patterns was provided. The paper presented the modeling approach and presented how to get from a choreography model to executable BPEL processes, where all details necessary for execution are contained. By using abstract BPEL during the choreography modeling, the modeler is free to leave out process internal details and focus on the interaction behavior.

Regarding the participant groundings, 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 in this context. The participant grounding links activities directly to WSDL port types and operations. We suppose this is not the only way to do grounding and will investigate other possibilities such as semantical grounding.

As mentioned earlier, BPEL4Chor assumes a closed world. It is not possible to model message exchanges with an environment that is not further defined. Explicitly introducing a participant "environment" without participant behavior description can serve as a workaround at the moment. Furthermore, there is currently no possibility to compose different BPEL4Chor choreographies into bigger choreographies. This is left open for future work. Proposals such as BPEL for subprocesses [30] are a promising basis for solving this issue. However, BPEL for subprocesses has not been released yet.

An infrastructure to monitor a BPEL4Chor choreography does not yet exist. Currently, violations of a choreography can only be detected by additional logic at the participants or by (manually) analyzing audit log information. We foresee an infrastructure, where the execution of choreographies is actively monitored and violations are recognized as soon as they happen. This enables a proper reaction to violations, such as not delivering wrong messages and suspending senders of wrong messages.

Acknowledgments. This work was partially supported by the German Federal Ministry of Education and Research (Tools4BPEL, project number 01ISE08B).

References

[1] W. M. P. v. d. Aalst, N. Lohmann, P. Massuthe, C. Stahl, K. Wolf, From Public Views to Private Views – Correctness-by-Design for Services, in: Proceedings 4th International Workshop on Web Services and Formal Methods (WS-FM), vol. 4937 of Lecture Notes in Computer Science, Springer Verlag, 2007.

[2] W. M. P. v. d. Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, A. P. Barros, Workflow Patterns, Distributed and Parallel Databases 14 (1) (2003) 5–51.

[3] A. Arkin, et al., Web Service Choreography Interface (WSCI) 1.0, World Wide Web Consortium, `http://www.w3.org/TR/wsci/` (Aug 2002).

[4] A. Banerji, et al., Web Services Conversation Language (WSCL) 1.0, W3C Note, `http://www.w3.org/TR/wscl10/` (March 2002).

[5] A. Barros, G. Decker, M. Dumas, Multi-staged and Multi-viewpoint Service Choreography Modelling, in: Proceedings Workshop on Software Engineering

Methods for Service Oriented Architecture (SEMSOA), No. 244 in CEUR Workshop Proceedings, 2007.

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

[7] A. Barros, M. Dumas, A. ter Hofstede, Service Interaction Patterns, in: Proceedings 3rd International Conference on Business Process Management (BPM), vol. 3649 of Lecture Notes in Computer Science, Springer Verlag, 2005.

[8] T. Basten, W. M. P. van der Aalst, Inheritance of Behavior, Journal of Logic and Algebraic Programming 47 (2) (2001) 47–145.

[9] S. Burbeck, The Tao of e-business services: The evolution of Web applications into service-oriented components with Web services, `http://www.ibm.com/developerworks/webservices/library/ws-tao/` (October 2000).

[10] C. Canal, E. Pimentel, J. M. Troya, Compatibility and Inheritance in Software Architectures, Science of Computer Programming 41 (2) (2001) 105–138.

[11] D. Chappell, Enterprise Service Bus, O'Reilly Media, Inc., 2004.

[12] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, Web Services Description Language (WSDL) 1.1, `http://www.w3.org/TR/2001/NOTE-wsdl-20010315` (Mar 2001).

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

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

[15] G. Decker, A. Barros, Interaction Modeling using BPMN, in: Proceedings 1st International Workshop on Collaborative Business Processes (CBP), vol. 4928 of Lecture Notes in Computer Science, Springer Verlag, 2007.

[16] G. Decker, O. Kopp, A. Barros, An Introduction to Service Choreographies, Information Technology 50 (2) (2008) 122–127.

[17] G. Decker, O. Kopp, F. Leymann, M. Weske, BPEL4Chor: Extending BPEL for Modeling Choreographies, in: Proceedings IEEE 2007 International Conference on Web Services (ICWS), IEEE Computer Society, 2007.

[18] G. Decker, O. Kopp, F. Leymann, M. Weske, Modeling Service Choreographies using BPMN and BPEL4Chor, in: Proceedings of the 20th International Conference on Advanced Information Systems Engineering (CAiSE), vol. 5074 of Lecture Notes in Computer Science, Springer Verlag, 2008.

[19] G. Decker, O. Kopp, F. Puhlmann, Service Referrals in BPEL-based Choreographies, in: 2nd European Young Researchers Workshop on Service Oriented Computing (YR-SOC), University of Leicester, 2007.

[20] G. Decker, H. Overdick, J. M. Zaha, On the Suitability of WS-CDL for Choreography Modeling, in: Proceedings of Methoden, Konzepte und Technologien für die Entwicklung von dienstebasierten Informationssystemen (EMISA), vol. P-95 of Lecture Notes in Informatics, 2006.

[21] G. Decker, M. Weske, Behavioral Consistency for B2B Process Integration, in: Proceedings 19th International Conference on Advanced Information Systems Engineering (CAiSE), vol. 4495 of Lecture Notes in Computer Science, Springer Verlag, 2007.

[22] G. Decker, M. Weske, Instance Isolation Analysis for Service-Oriented Architectures, in: Proceedings of the IEEE 2008 International Conference on Services Computing (SCC), IEEE Computer Society, 2008.

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

[24] D. F. Ferguson, M. Stockton, Enterprise Business Process Management – Architecture, Technology and Standards, in: Proceedings 4th International Conference on Business Process Management (BPM), vol. 4102 of Lecture Notes in Computer Science, Springer Verlag, 2006.

[25] B. Hofreiter, C. Huemer, P. Liegl, R. Schuster, M. Zapletal, UN/CEFACT'S Modeling Methodology (UMM): A UML Profile for B2B e-Commerce, in: Advances in Conceptual Modeling – Theory and Practice, ER 2006 Workshops BP-UML, CoMoGIS, COSS, ECDM, OIS, QoIS, SemWAT, vol. 4231 of Lecture Notes in Computer Science, Springer Verlag, 2006.

[26] B. Hofreiter, C. Huemer, P. Liegl, R. Schuster, M. Zapletal, Deriving executable BPEL from UMM Business Transactions, in: IEEE International Conference on Services Computing (SCC), IEEE Computer Society, 2007.

[27] Z. Kang, H. Wang, P. C. Hung, WS-CDL+: An Extended WS-CDL Execution Engine for Web Service Collaboration, in: Proceedings IEEE 2007 International Conference on Web Services (ICWS), IEEE Computer Society, 2007.

[28] N. Kavantzas, D. Burdett, G. Ritzinger, Y. Lafon, Web Services Choreography Description Language Version 1.0, W3C Candidate Recommendation, W3C, `http://www.w3.org/TR/ws-cdl-10` (Nov. 2005).

[29] R. Khalaf, From RosettaNet PIPs to BPEL processes: A three level approach for business protocols, Data & Knowledge Engineering 61 (2006) 23–38.

[30] M. Kloppmann, D. Koenig, F. Leymann, G. Pfau, A. Rickayzen, C. von Riegen, P. Schmidt, I. Trickovic, WS-BPEL Extension for Sub-processes – BPEL-SPE, IBM, SAP (2005).

[31] D. König, N. Lohmann, S. Moser, C. Stahl, K. Wolf, Extending the Compatibility Notion for Abstract WS-BPEL Processes, in: Proceedings 17th International Conference on World Wide Web (WWW), ACM, 2008.

[32] O. Kopp, T. van Lessen, J. Nitsche, The Need for a Choreography-aware Service Bus, in: Proceedings 3rd European Young Researchers Workshop on Service Oriented Computing (YR-SOC), 2008.

[33] D. Krafzig, K. Banke, D. Slama, Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series), Prentice Hall PTR, 2004.

[34] F. Leymann, Web Services Flow Language (WSFL 1.0), IBM Software Group (May 2001).

[35] F. Leymann, The (Service) Bus: Services Penetrate Everyday Life., in: Proceedings of the 3rd International Conference on Service Oriented Computing (ICSOC), vol. 3826 of Lecture Notes in Computer Science, Springer Verlag, 2005.

[36] F. Leymann, Workflow-Based Coordination and Cooperation in a Service World, in: Proceedings 14th International Conference on Cooperative Information Systems (CoopIS), vol. 4275 of Lecture Notes in Computer Science, Springer Verlag, 2006.

[37] N. Lohmann, O. Kopp, F. Leymann, W. Reisig, Analyzing BPEL4Chor: Verification and Participant Synthesis, in: Proceedings 4th International Workshop on Web Services and Formal Methods (WS-FM), vol. 4937 of Lecture Notes in Computer Science, Springer Verlag, 2007.

[38] A. Martens, Analyzing Web Service based Business Processes, in: M. Cerioli (ed.), Proceedings of Intl. Conference on Fundamental Approaches to Software Engineering (FASE), Part of the European Joint Conferences on Theory and Practice of Software (ETAPS), vol. 3442 of Lecture Notes in Computer Science, Springer Verlag, 2005.

[39] A. Martens, Consistency between Executable and Abstract Processes, in: Proceedings IEEE International Conference on e-Technology, e-Commerce, and e-Services (EEE), IEEE Computer Society, 2005.

[40] P. Massuthe, W. Reisig, K. Schmidt, An Operating Guideline Approach to the SOA, Annals of Mathematics, Computing & Teleinformatics 1 (3) (2005) 35–43.

[41] J. Mendling, M. Hafner, From Inter-Organizational Workflows to Process Execution: Generating BPEL from WS-CDL, in: Proceedings of OTM 2005 Workshops, vol. 3762 of Lecture Notes in Computer Science, Springer Verlag, 2005.

[42] N. Mulyar, L. Aldred, W. M. P. van der Aalst, The Conceptualization of a Configurable Multi-party Multi-message Request-Reply Conversation, in: Proceedings of the Distributed Objects and Applications (DOA) International Conference, vol. 4803 of Lecture Notes in Computer Science, Springer Verlag, 2007.

[43] J. Nitzsche, T. van Lessen, D. Karastoyanova, F. Leymann, BPEL for Semantic Web Services (BPEL4SWS), in: On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops, vol. 4805 of Lecture Notes in Computer Science, Springer Verlag, 2007.

[44] J. Nitzsche, T. van Lessen, D. Karastoyanova, F. Leymann, BPEL^{light}, in: Proceedings 5th International Conference on Business Process Management (BPM), vol. 4714 of Lecture Notes in Computer Science, Springer Verlag, 2007.

[45] J. Nitzsche, T. van Lessen, F. Leymann, Extending bpel^{light} for expressing multi-partner message exchange patterns, in: 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC), IEEE Computer Society, 2008.

[46] J. Nitzsche, T. van Lessen, F. Leymann, WSDL 2.0 Message Exchange Patterns: Limitations and Opportunities, in: 3rd International Conference on Internet and Web Applications and Services (ICIW), IEEE Computer Society, 2008.

[47] Object Management Group, OMG Unified Modeling Language Specification, v1.4 (2001).

[48] Object Management Group, Business Process Modeling Notation, V1.1 – OMG Available Specification (Jan. 2008).

[49] Organization for the Advancement of Structured Information Standards (OASIS), SCA Assembly Model Specification V1.00 (March 2007).

[50] Organization for the Advancement of Structured Information Standards (OASIS), SCA Client and Implementation Model Specification for WS-BPEL (March 2007).

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

[52] C. Ouyang, M. Dumas, S. Breutel, A. H. ter Hofstede, Translating Standard Process Models to BPEL, in: Proceedings 18th International Conference on Advanced Information Systems Engineering (CAiSE), vol. 4001 of Lecture Notes in Computer Science, Springer Verlag, 2006.

[53] M. Pesic, M. H. Schonenberg, N. Sidorova, W. M. P. van der Aalst, Constraint-Based Workflow Models: Change Made Easy, in: Proceedings 15th International Conference on Coopartive Information Systems (CoopIS), vol. 4803 of Lecture Notes in Computer Science, Springer Verlag, 2007.

[54] K. Pfitzner, G. Decker, O. Kopp, F. Leymann, Web Service Choreography Configurations for BPMN, in: Third International Workshop on Engineering Service-Oriented Applications: Analysis, Design and Compostion (WESOA), 2007.

[55] F. Puhlmann, M. Weske, Interaction Soundness for Service Orchestrations, in: A. Dan, W. Lamersdorf (eds.), Proceedings of the 4th International Conference on Service Oriented Computing (ICSOC), vol. 4294 of Lecture Notes in Computer Science, Springer Verlag, 2006.

[56] P. Reimann, O. Kopp, G. Decker, F. Leymann, Generating WS-BPEL 2.0 Processes from a Grounded BPEL4Chor Choreography, Tech. Rep. 2008/07,

University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany (2008).

[57] W. Reisig, Petri nets, Springer Verlag, 1985.

[58] M. Weidlich, G. Decker, A. Grokopf, M. Weske, BPEL to BPMN: The Myth of a Straight-Forward Mapping, in: Proceedings 16th International Conference on Cooperative Information Systems (CoopIS), Lecture Notes in Computer Science, Springer Verlag, 2008.

[59] J. M. Zaha, A. Barros, M. Dumas, A. ter Hofstede, Let's Dance: A Language for Service Behavior Modeling, in: Proceedings 14th International Conference on Cooperative Information Systems (CoopIS), vol. 4275 of Lecture Notes in Computer Science, Springer Verlag, 2006.

[60] J. M. Zaha, M. Dumas, A. ter Hofstede, A. Barros, G. Decker, Service Interaction Modeling: Bridging Global and Local Views, in: Proceedings 10th IEEE International EDOC Conference (EDOC), IEEE Computer Society, 2006.

All links were last followed on October 10, 2008.