Interaction Choreography Models in BPEL: Choreographies on the Enterprise Service Bus

Oliver Kopp, Lasse Engler, Tammo van Lessen,
Frank Leymann, Jörg Nitzsche

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{wieland, goerlach, schumm, leymann}@iaas.uni-stuttgart.de

Interaction Choreography Models in BPEL: Choreographies on the Enterprise Service Bus

Oliver Kopp, Lasse Engler, Tammo van Lessen,
Frank Leymann, and Jörg Nitzsche*

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

Abstract Interactions between services may be globally captured by choreographies. We introduce BPEL^{gold} supporting modeling interaction choreography models using BPEL. We show the usage of BPEL^{gold} in an enterprise service bus to ensure an executed message exchange complies with a pre-defined choreography.

1 Introduction

Choreographies capture the interaction between services on a global perspective [1]. The behavior of services taking part in the choreography may be expressed by an abstract process, which in turn models the public visible behavior. A choreography model interconnects the communication activities of each abstract process and hence forms an *interconnection model*. The second paradigm offered to model choreographies is the *interaction model* paradigm. Here, a send/receive message exchange is modeled as an atomic activity. The public visible behavior of each participant is not shown any more. Details are presented in Sect. 2.

BPEL [2] and BPEL4Chor [3] are currently the only two choreography approaches tightly integrated in the Web service stack. In this paper, we present BPEL^{gold} as a complement to BPEL4Chor: BPEL4Chor supports the interconnection-based modeling approach and BPEL^{gold} supports the interaction-based modeling approach. "gold" is an abbreviation for **glo**ba**l** **d**efinition emphasizing the global process consisting of interactions. We show that BPEL^{gold} fulfills all requirements put on a choreography language. Subsequently, we show how BPEL^{gold} can be integrated in an enterprise service bus to make it choreography-aware. By aligning BPEL^{gold} with established standards in the field of Web service (e.g., BPEL, WSDL, XML), existing tooling as well knowledge may be reapplied in a BPEL^{gold} setting.

On the one hand, choreographies capture the interaction between services on a global perspective. On the other hand, an enterprise service bus (ESB) is an standard-based integration platform [4]. For instance, all messages sent by a service are routed through the ESB, which selects the appropriate endpoint.

* This work was supported by funds from the European Commission (contract no. 215175 for the FP7-ICT-2007-1 project COMPAS)

Typically, an ESB is not aware of the choreography the service implementers agreed upon before executing the services. This paper introduces the concept of a choreography-aware enterprise service bus. A choreography-aware enterprise service bus can detect violations of the choreography and can react accordingly. Possible reactions include the interruption of the message transfer and informing the participants that an error occurred in the choreography. For that purpose, an extended choreography model can be modeled, which specifies reactions to erroneous situations.

Consequently, this paper is structured as follows: Section 2 presents an overview on the two choreography modeling paradigms. Subsequently, Sect. 3 introduces the choreography language BPEL$^{\text{gold}}$. It is evaluated against requirements on choreography languages in Sect. 4. The concept and implementation of a choreography-aware enterprise service bus is presented in Sect. 5. Section 6 presents an overview on related work in the field of compliance with respect to choreographies. Finally, Sect. 7 concludes and provides and outlook on future work.

2 Choreography Modeling Paradigms

We use a choreography dealing with investment proposals as illustration. A customer talks to a financial consultant. The consultant recommends an investment and hands over an investment proposal including information material. Governmental regulations require a time-window of at least 24 hours for the customer to decide on the investment. After 24 hours have passed, the customer may sign a contract. It is not allowed to receive a signature from the customer beforehand. Figure 1 models the choreography using the interconnection model paradigm. The notation used is the Business Process Model and Notation BPMN 2.0 [5]. The public visible behavior of each participant is modeled using a BPMN pool. The communication elements are connected by message flows. The intermediate event with the arrow label denotes that the choreography continues in the case the customer votes for the investment. The choreography may also be expressed using the interaction model paradigm. Figure 2 presents the choreography using the BPMN 2.0 choreography notation. Here, pairs of sending and receiving elements are collapsed into one choreography task. The gray shaded participant is the participant receiving the message. The event-based gateway denotes that the decision whether to accept or reject the proposal is made available to the consultant by the respective message.

The expressiveness of the two modeling paradigms "interaction-based modeling" and "interconnection-based modeling" is different [6]. On the one hand, interconnection models allow modeling of incompatible processes or even processes for which no partner exists [7]. For instance, service A waits for a message, but service B never sends that message. Decker and Weske [8] identified 8 antipatterns which can be expressed by interconnection models, but not in interaction models. Wolf [7] studies the existence of partners for services, which is called "controllability". On the other hand, interaction models may introduce constraints,

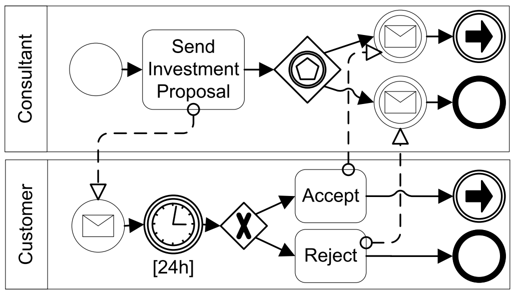

Figure 1. Investment Choreography: Interconnection Model (BPMN 2.0 Collaboration)

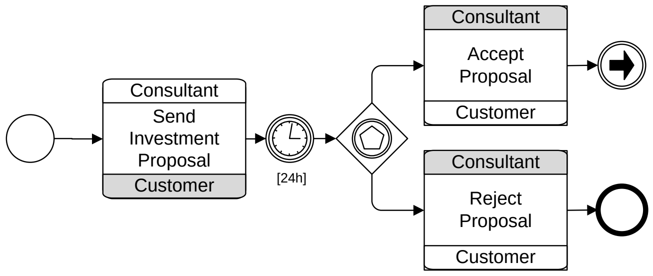

Figure 2. Investment Choreography: Interaction Model (BPMN 2.0 Choreography)

which need to be locally enforced in interconnection models by adding additional communication that is not captured in the original interaction model [9]. This property is called "local enforceability". For instance, if service A sends a message to service B and the interaction model demands that *subsequently* service C sends a message to service D, service C has to know when service B has received the message. "Realizability" is an even stronger requirement, which demands that all conversations produced by the interaction model must also be produced by the corresponding interconnection models [10]. Lohmann and Wolf [11] showed that the property of realizability is equal to the property of controllability.

Although the expressiveness of the two modeling paradigms is not equal, there exist mappings from interaction models to interconnection models and vice versa: Zaha et al. [9] present an approach of mapping interaction models to interconnection models. As shown above, not all interaction models can be mapped to interconnection models. Kopp et al. [12] show that sound and safe BPMN models following the interconnection style and containing control flow without exception handling can be mapped to interaction BPMN models (iBPMN [13]). Having BPEL$^{\text{gold}}$ at hand, the approaches may be adapted to map BPEL4Chor models to BPEL$^{\text{gold}}$ models and thus being able to switch between the two modeling paradigms.

Decker et al. [14] show 10 requirements put on choreography languages. An evaluation using this requirements shows whether a language is suitable for choreography modeling. Decker et al. [14] use these requirements to evaluate current choreography languages including WS-CDL [15] and BPEL4Chor [3]. The result is, that WS-CDL fails in modeling an a priori unknown number of participants and that the integration with orchestration languages such as BPEL [2] is not given. BPEL4Chor on the other hand, fulfills all requirements. These requirements themselves are mainly based on the service interaction patterns [16]. These patterns list possible interactions between business processes. We use the requirements to evaluate BPEL$^{\text{gold}}$ and present them together with the evaluation of BPEL$^{\text{gold}}$ in Sect. 4.

BPMN 2.0 offers both the interconnection model paradigm ("BPMN collaborations") and the interaction model paradigm ("BPMN choreography"). Decker et al. [17] show the integration of BPMN interconnection models and BPEL4Chor. There is currently no study about the integration of BPMN 2.0 interaction models and BPEL4Chor or BPEL$^{\text{gold}}$. This study is out of scope of this paper as we focus on BPEL$^{\text{gold}}$ and its integration in an enterprise service bus. A first evaluation of BPMN 2.0 showed that it still does not support the exchange of participant references between partners. Thus, BPEL4Chor and BPEL$^{\text{gold}}$ are ahead of BPMN 2.0 in this regard. A detailed evaluation of BPMN 2.0 and a comparison to BPEL$^{\text{gold}}$ are our ongoing work.

3 BPEL$^{\text{gold}}$

BPEL4Chor extended BPEL to a choreography language supporting interconnection models. Thereby, a participant topology and a grounding artifact has been added. We re-use these extensions to create a interaction choreography modeling language based on BPEL.

A *participant topology* provides a view on the participants and the connections between them. It consists of a list of participant types, a list of participants, and a list of message links. Each participant type is associated with a BPEL process describing the public visible behavior of this type. Thus, this artifact is called "participant behavior description". Concrete participants are listed subsequently. Each is linked to a participant type and thus specifying its behavior. BPEL4Chor also supports sets of participants, which enables modeling of an a priori unknown number of participants in a choreography. Each participant in a set is merely a reference to a concrete participant. In case a message is received from a participant not being in the set, it is added to the set. Finally, the message links connect communication activities of two participant behavior descriptions. It also lists the name of the message being sent and the participant references contained in the message (which may be empty).

BPEL4Chor is not bound to concrete WSDL service definitions: `partnerLinks`, `portTypes`, and `operations` are not specified in the communication activities. Instead, the message links establish the connection to the partner. When a BPEL4Chor choreography is executed, the choreography description itself is not executed, but services implementing the behavior described by the participant behavior descriptions. The services may be realized using BPEL, but they are not required to do so. Nevertheless, the service needs to know about WSDL information to be able to communicate with the partner service. This information is provided by the *grounding* document. Here, a mapping from a message link to a `portType/operation` pair is provided. This is then used to generate an abstract BPEL process for each participant behavior description, where the required `partnerLinks` are generated [14]. The abstract BPEL processes are not executable by themselves. Internal activities such as assign activities or invokations of internal services have to be added.

3.1 Participant Topology

BPEL^{gold} reuses the concept of the participant topology, but changes the way of the description of the interaction. Instead of specifying the behavior of each participant separately, an abstract BPEL process providing the interaction is used. A reference to the process is provided in the topology by the attribute `gld:interactionDescription`.

Figure 3 presents the topology for the investment choreography. First, the different participant types "Consultant" and "Customer" are enumerated. Subsequently, the concrete participants "consultant" and "customer" of the respective types are declared. In the `participants` element, multiple `participantSet` elements may occur. Here, additional participants can be listed. The semantics is that the participant are contained in the set. Here, a participant is merely a reference to a participant in the set instead of a concrete participant. In case a message is received from one participant out of a possible set of participants, the reference to this participant is added to the set at the receivers' side. The list of participants and participant sets is followed by a list of message links. In the investment choreography, there are three send/receive message exchanges leading to three message links. Each message link takes a `name`, a `sender`, and a `receiver` attribute. The name is unique and required for identification in the interaction description. Additionally, a message link can take a `participantRefs` attribute enabling the transfer of participant references. The attribute `sender` may be replaced by the attribute `senders`, where a set of possible senders may be specified. This enables modeling of an a priori unknown set of participants. The sender may send a reference to himself to enable a reply back to him. In case the reference is included in a set, this reference is added to the set at the receiver's side.

The `participantSet` element may be annotated with a `forEach` attribute. This attribute indicates, that the referenced `forEach` iterates over the respective sets. The current iterator is provided by a `forEach` attribute at a participant reference contained in the set. This enables sending a message to multiple participants. `forEach` is an activity of BPEL, which is used for iterating over a set in a sequential or parallel way. The concrete action has to be modeled as activity nested in the `forEach` activity. In case of BPEL^{gold} this is the `gld:interaction` activity.

```
<topology name="investment" gld:interactionDescription="inv:interactions">
  <participantTypes>
    <participant name="Consultant" />
    <participant name="Customer" />
  </participantTypes>
  <participants>
    <participant name="consultant" type="Consultant" />
    <participant name="customer" type="Customer" />
  </participants>
  <messageLinks>
    <messageLink name="investmentProposal"
       sender="consultant" receiver="customer" />
    <messageLink name="acceptance"
       sender="customer" receiver="consultant" />
    <messageLink name="rejection"
       sender="customer" receiver="consultant" />
  </messageLinks>
</topology>
```

Figure 3. Investment Choreography: BPEL$^{\text{gold}}$ Participant Topology

3.2 Interaction Description

The main building block of the interaction description is the `gld:interaction` activity. This activity is embedded as an `extensionActivity` in the BPEL process. BPEL$^{\text{gold}}$ uses one abstract BPEL process to capture the interactions between the participants. An abstract BPEL process always follows an abstract process profile. BPEL$^{\text{gold}}$ introduces two profiles: (i) The "Abstract Process Profile for Basic Interaction Models" and (ii) the "Abstract Process Profile for Extended Interaction Models". The basic profile enables specification for the interactions between the participants and hence provides a true global view. The extended profile allows additional communication starting from or targeted to the global observer. A global observer observes the interactions made by the participants of the choreography. In our case, this *global observer* is the enterprise service bus.

The "Abstract Process Profile for Basic Interaction Models" forbids the usage of BPEL's standard communication activities and only allows `gld:interaction` as communication activity. Hence, only the interaction between participants can be modeled and the modeled process does not require an active global observer. As the global observer has to track the choreography and uncertainty should not be modeled, the profile requires that the expressions of conditions must be based on data derived from exchanged messages and that all modifications from the message receipt to the condition are modeled. For instance, required variable assignment activities have to be modeled. The requirement ensures that the global observer can evaluate the conditions for itself and thus properly keeps track of the choreography. Opaque activities and attributes are allowed as long as they do not infer with that requirement.

The "Abstract Process Profile for Extended Interaction Models" allows for modeling the interactions between participants. In addition, the global observer may actively participate in the choreography. Reasons include a communication of faults in the choreography execution to participants, which are not notified of the faults otherwise. Besides `gld:interaction` activities, the common BPEL communication activities are allowed. BPEL4Chor's rules forbidding `partnerLink`, `portType`, and `operation` still apply. In addition, the sender or receiver of such a communication activity must be the `GOLDobserver`, which also must be listed as participant in the participant topology. For these communication activities, the message links specified by BPEL4Chor have to be used. The GOLDobserver participant must not be used in message links referred to in `gld:interaction` activities. As a consequence, this profile mixes interaction models and interconnection models together: The interaction between the participants is modeled using the interaction paradigm and the interaction between the global observer and the participants is modeled using the interconnection paradigm. The reason is that the global observer takes a special role in the choreography as it knows the status of the choreography and can also react accordingly if a participant does not comply with the choreography or even disappears. A detailed discussion is provided in Sect. 5.

The extended interaction models defines three standard faults: `gld:interactionInitiationFault`, `gld:interactionCompletionFault`, and `gld:choreographyViolation`. A `gld:interactionInitiationFault` is raised if the interaction cannot be initiated by the sending participant. Reasons include that the participant itself crashed and does not recover, or that message sending activity is on a dead path in the process and will never be executed. A `gld:interactionCompletionFault` is raised if the message cannot be received by the targeted participant. Reasons include that the participant crashed and does not recover, a communication fault to the receiver occurred, or that the message receiving activity is on a dead path in the process and will never be executed. Finally, a `gld:choreographyViolation` fault is raised if the CSB detects a violation of the choreography. The reason is a message which is sent but not allowed in the choreography definition.

A `gld:interaction` activity itself refers to a message link by the `messageLink` attribute. The referred message link in turn states from which participant to with other participant the message is sent. This indirection is inherited from BPEL4Chor, where the connection between participants is also made at the participant topology. The reuse enables a seamless integration with BPEL4Chor described in Sect. 3.3. The child element `correlations` denotes the correlations used in the interaction. Both the sender and the receiver use the same correlation set. Each set may be initiated (`yes`) or an initiation may be forbidden (`no`). In case it is unsure, whether the set has already been initiated, the value `join` may be used. These options are derived from the BPEL specification, where the same values are specified. BPEL$^{\text{gold}}$ introduces the attributes `senderInitiate` and `receiverInitiate` to enable a specification of the sender's and the receiver's behavior according to correlation. The attribute value is transformed to the `initiate` attribute of the respective communication activity in the respective participant behavior description. The attribute may take the additional value `n/a` to indicate that the correlation set is not used at the respective participant. For instance, in our scenario, the customer is not required to use a correlation set as he executes a receive/send message exchange only. Finally, an `gld:interaction`

```
<process>
  <sequence>
    <extensionActivity>
      <gld:interaction messageLink="investmentProposal">
        <correlations>
          <correlation set="cor" gld:senderInitiate="yes"
                                 gld:receiverInitiate="n/a" />
        </correlations>
      </gld:interaction>
    </extensionActivity>
    <wait for="P1D" />
    <pick>
      <gld:onInteraction messageLink="acceptance">
        <correlations>
          <correlation set="cor" gld:senderInitiate="n/a" />
        </correlations>
      </gld:onInteraction>
      <gld:onInteraction messageLink="rejection">
        <correlations>
          <correlation set="cor" gld:senderInitiate="n/a" />
        </correlations>
      </gld:onInteraction>
    </pick>
  </sequence>
</process>
```

Figure 4. Investment Choreography: BPEL$^{\text{gold}}$ Interaction Description

activity may take the attribute `variable` to specify the message format (by the type of the variable) and to enable choreography tracking by the global observer.

Figure 4 presents the interaction description of the investment choreography using the basic profile. The correlation set used for communication is named `cor`, initiated at the sender only. In our scenario, the customer executes a receive/send message exchange only. If one regards the full choreography, the customer has to send initiate a correlation set on his own at the acceptance interaction. This ensures that the consultant reaches him to negotiate the details of the contract. The `pick` activity uses `gld:onInteraction` branches, which replace BPEL's `onMessage` branches.

3.3 From BPEL$^{\text{gold}}$ to Executable BPEL Processes

Going from BPEL$^{\text{gold}}$ to executable BPEL processes is one way to enact the choreography. Other ways include that each participant uses the choreography description and implements the services in his favorite language. Nevertheless, the created endpoints must be made available to the other participants in order to enable them to reach the participant.

When using BPEL as implementation language, the way starting from BPEL$^{\text{gold}}$ is depicted in Fig. 5: A choreography is modeled using a BPEL$^{\text{gold}}$ participant topology, a BPEL$^{\text{gold}}$ interaction description and optionally a grounding. A grounding provides a mapping from the message links to their concrete WSDL implementation in the form of one concrete `portType` and one concrete `operation` for each message link.

The BPEL$^{\text{gold}}$ participant topology is transformed to a BPEL4Chor participant topology, where names for sending and receiving activities have to be

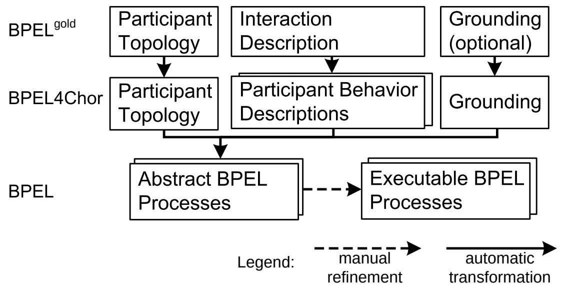

Figure 5. From a BPEL^{gold} Choreography Description to Executable BPEL Processes

generated. This generation is necessary as BPEL^{gold} has no knowledge of the activities used at the participants. The process interaction description is transformed to participant behavior descriptions following the ideas presented in [9]. The main idea is to split an `interactionActivity` into an `invoke` and `receive` activity. A `pick` is translated into an `if` at the sender's side and to a `pick` at the receiver's side. For each `gld:onInteraction` branch, a condition branch in the `if` activity is generated. The first activity in the condition branch is an `invoke` activity. On the receiver's side, each `gld:onInteraction` branch is transformed to an `onMessage` branch. A detailed description of the transformation is out of scope of this paper and will be presented in our future work. In case the grounding description does not exist, it has to be created in order to gain a full BPEL4Chor choreography description. This choreography description can now be used to generate abstract BPEL processes as shown in [14]. These abstract BPEL processes contain `partnerLinks`, `portTypes`, and `operations` at each communication activity. The processes are abstract as they do not contain the internal behavior of each participant. The abstract BPEL processes have to be expanded in order to get executable BPEL processes. This is the only manual refinement step.

4 Evaluation of BPEL^{gold}

In this section, we evaluate BPEL^{gold} using the requirements on choreographies [14], which in turn are based on the service interaction patterns [16]. Other approaches to compare modeling languages are not tailored towards choreography languages, but towards orchestration languages. These approaches include the workflow patterns [18], process instantiation patterns [19], correlation patterns [20], data handling patterns [21], exception handling patterns [22], and a discussion regarding block-structured and graph-based modeling styles [23]. As these patterns are not centered around choreographies, an evaluation using these patterns is out of scope of this paper, but part of our future work.

In the following, we list each requirement of [14] and evaluate BPEL^{gold} according to its fulfillment.

R1. Multi-lateral interactions A choreography language has to support more than two participants in a choreography. This is directly enabled by the BPEL^{gold} topology.

R2. Service topology A choreography language should provide a structural view, where the types and number of participants involved is provided. This is directly supported by the `participant` and `participantSet` elements in BPEL^{gold}'s topology. In case the concrete number of instances are known from a participant, it is listed multiple times as `participant`. A priori unknown numbers are supported by the `participantSet` element.

R3. Service sets A choreography language must support modeling of an a priori unknown arbitrary number of services. This is directly supported by the `participantSet` element.

R4. Reference passing A choreography language must provide support for passing references to participants to enable distribution of knowledge about participants. BPEL^{gold} supports this requirement by the attribute `participantRefs` of a message link.

R5. Message formats It is possible to agree on concrete message formats when agreeing on a choreography. Thus, a choreography language should support the specification of message formats. BPEL^{gold} supports specification of message formats by the `variable` attribute in the `interactionActivity`.

R6. Interchangeability of technical configurations Concrete WSDL `portTypes` and WSDL `operations` typically vary from implementation to implementation even if the implementation itself offers the same functionality. Thus, a choreography language should support changing the concrete identifiers. BPEL^{gold} enables this by the concept of grounding.

R7. Time constraints A choreography has to offer constructs for modeling time constraints. This is offered by BPEL's `wait` activity and the `onAlarm` branch of a `pick` activity and the event handler of a `scope`. Hence, this requirement is supported by BPEL^{gold}.

R8. Exception handling Typically, there is a separation of the "happy path" through a process and the exception path. BPEL^{gold} allows using the `scope` construct of BPEL thus enables explicit modeling of exception handling.

R9. Correlation A process may be instantiated multiple times, each taking part in different choreographies. Thus, a choreography language has to be able to specify the identifiers used for correlation. This is enabled by the `correlationSet` specification at the `interactionActivity`.

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 generation of BPEL processes out of choreographies. Section 3.3 showed that BPEL processes can be generated out of a BPEL^{gold} choreography description.

5 Choreography-aware Enterprise Service Bus

A traditional ESB does not know whether the message it currently routes is part of a conversation. Thus, it cannot react on messages out of band. In contrast, a choreography-aware enterprise service bus (CSB for short) is aware of the choreography and can react accordingly. A CSB should act transparent to the participants. There are use-cases, however, where participants with knowledge about the choreography-awareness of the bus allow enhanced solutions. We refer to such participants as choreography-aware participants (CAP for short). A CAP offers an interface to the CSB, where the CSB may request information or inform the CAP of events not captured in the choreography definition. For instance, a CSB may ask the CAP whether it still runs. In case a CAP is not running any more, the whole choreography cannot be enacted further. Thus, the CSB informs the other CAPs that the choreography faulted. The CAPs in turn can take appropriate fault handing actions including a termination of the process taking part in the choreography. A CAP can also push information on interactions to the CSB. Thus, it can inform the CSB whether an interaction has been skipped. The CSB can then check whether this complies with the choreography definition and take appropriate actions.

We implemented a prototype of an choreography-aware enterprise service bus based on Apache ServiceMix 3.3.1 [24] and call it CASmix—**C**horeography-**A**ware **S**ervice**mix**. We did not change any of ServiceMix code. All extensions were implemented through the extension mechanisms and Java Business Integration components. For tracking the state of the choreography, we extended the Apache ODE engine [25] to ODE$^{\mathrm{gold}}$. The overall architecture and the message flow of one message in CASmix from a participant A to a participant B denoted by the steps 1 to 8 in the diagram.

A binding component provides connectivity to services located outside of the bus. Thus, the message of participant A is received by the binding component (step 1), transforms the message to a normalized message and puts it on the normalized message router (step 2). All communication internal of ServiceMix is based on normalized messages. The *CASmix Message Interceptor* is plugged into the normalized message router and inspects each message being put on the bus (step 3). It checks whether the message belongs to a choreography instance. This is done by checking with the *CASmix Choreography Manager*, which stores and provides information about the currently deployed choreography descriptions. The CASmix Choreography Manager also keeps track of currently running choreographies and is responsible for the fault propagation to choreography-aware participants. In case the message does not belong to a choreography, the CASmix Message Interceptor releases the message unchanged to the normalized message router (step 4). The message then is handed over to the binding component of participant B (step 5’), where it is sent to participant B (step 6’). In case the message belongs to a choreography, the target endpoint is changed to ODE$^{\mathrm{gold}}$ and the original endpoint is stored at a message property. The message is put back to the normalized message router (step 4), where it is forwarded to ODE$^{\mathrm{gold}}$ (step 5). ODE$^{\mathrm{gold}}$ checks whether there is a matching active `receive`

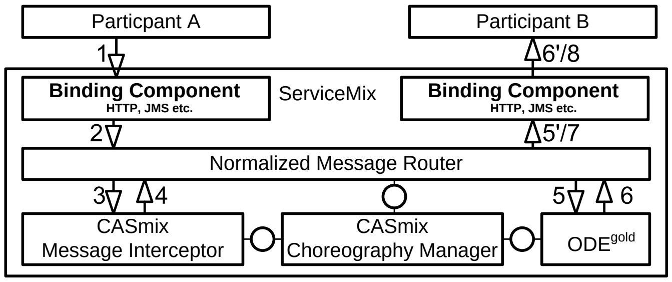

Figure 6. CASmix: Components and Message Routing

activity. In case there is no such activity, the choreography is violated and a `gld:choreographyViolation` fault is thrown in the outermost `scope` of the respective choreography instance. In case an explicit fault handler is modeled, the specified activities are executed. In case no explicit fault handler is modeled, the choreography is terminated and (by using the CASmix Choreography Manager), all choreography-aware participants are notified. In case ODE^{gold} finds a matching choreography instance for the message, the message is routed to the instance and processed there. Otherwise, ODE^{gold} identified a choreography model which can be instantiated with the message and the message can be processed there. After processing, the message is put pack to the normalized message router with the original recipient (step 6). A flag in the message indicates that the CASmix Message Interceptor is not required to redirect the message to ODE^{gold} again. Thus, the message flows via a binding component (step 7) to the participant B.

We have identified three ways to provide support for the `gld:interaction` activity and the `gld:onInteraction` branch in Apache ODE. (i) ODE's mechanism for implementing the behavior of an `extensionActivity`, (ii) implementing the extension directly in the runtime, and (iii) transforming each `gld:interaction` activity and `gld:onInteraction` branches into native BPEL activities. As ODE does not support extensions for the behavior of `pick` activities, the `gld:onInteraction` branch cannot be implemented using an extension mechanism. Thus, approach (i) cannot be taken. Implementing the extensions directly in Apache ODE is a change in ODE's internals and leads to code duplication as the sending and the receipt of messages have to be re-implemented. Therefore, we dropped option (ii). Finally, we opted for option (iii) and transform the BPEL^{gold} model to a standard BPEL model.

Figure 7 shows how a `gld:interaction` activity is transformed to standard BPEL. The `onMessage` element on line 3 is used to receive the specified message. The operation name `opName` and the variable name `varName` is a uniquely name within the choreography definition. The CASmix Message Interceptor changes the destination of a message accordingly. In line 5 the endpoint reference of the original recipient from the message header is assigned to the partner link. In line 5 the message to the recipient is sent. ODE also propagates communication faults into the process [26], thus a fault occurring at the `invoke`

```
1   <pick>
2     <!-- regular interaction -->
3     <onMessage partnerLink="inboundPL" operation="opName" variable="varName">
4       <sequence>
5         <assign><!-- copy original target epr to outboundPL --></assign>
6         <invoke partnerLink="outboundPL" operation="opName" variable="varName">
7           <catchAll>
8             <throw>gld:interactionCompletionFault</throw>
9           </catchAll>
10        <invoke>
11      </sequence>
12    </onMessage>
13    <!-- notification from ChoreographyManager -->
14    <onMessage partnerLink="chorManagerInbound" operation="opNameInteractionFailed" />
15      <throw>gld:interactionInitiationFault</throw>
16    </onMessage>
17  </pick>
```

Figure 7. `gld:interaction` mapped to Standard BPEL Activities (without correlation, simplified)

activity denotes a communication fault. This fault is propagated by throwing an `gld:interactionCompletionFault` (line 8). In case the CASmix Choreography Manager detects that a participant cannot initiate the interaction any more, it sends a message to ODE$^{\text{gold}}$. This message is received by the `onMessage` in line 14 and leads to a propagation in the `gld:interactionInitiationFault`. A similar transformation is done for a `gld:onInteraction` element. Opaque activities are deleted from the process and opaque assignments, too. As the profile definition itself ensures that values for variables are available for conditions, these deletions do not alter the behavior of the choreography. The other elements of the process are transformed one by one.

When a fault occurs and no explicit exception handling is modeled, the CSB has to inform all participants. They can then trigger their individual error handling. Propagating a fault to CAP is achieved using the interface offered by the CAP. For non-choreography aware participants we have to distinguish three possibilities: (i) fault messages may be sent, (ii) erroneous messages may be sent, and (iii) future interactions may be blocked. In case the participant is expecting a message and the respective operation supports fault messages, one of them may be sent by the bus. If no fault message is expected, the bus may send a message of the expected type and fill all parameters with erroneous values (such as empty strings or 0), which in turn leads to a fault in the participants process. In case the participant is currently expecting no messages, we just block any further messages of it and reply with faults or faulty messages, respectively.

6 Related Work

WS-CDL has been introduced to capture choreographies in the field of Web services. It follows the interaction modeling approach. It has been criticized for not supporting all service interaction patterns [27], for the impossibility to specify an a priori unknown number of participants [27], and for not being integrated with

BPEL [28]. Based on WS-CDL, Fredlund [29] present a tool for debugging WS-CDL specifications and to check WS-CDL models against a property formulated as safety automaton. It is not possible to use his tool for runtime checking during the execution of a choreography. Kang et al. [30] extended WS-CDL to support its execution. They mainly added variable initialization to WS-CDL and called it WS-CDL$^+$. As messages are received and sent, WS-CDL$^+$ is a kind of orchestration language without the full capabilities of BPEL. An overview on current available choreography language and an evaluation is presented by Decker et al. [14]. There is no language based on BPEL following the interaction modeling paradigm.

We sketched the concept of a choreography-aware service bus in [31]. This paper extends the work in (i) providing a concrete language for choreography modeling and (ii) presenting a proof-of-concept implementation of a choreography-aware enterprise service bus.

Conformance between a choreography and an orchestration can be checked at design time [32–35]. The approaches assume that at least the public behavior description of each service *used* is available. That might not be possible in case services provided by other companies not offering the code of the deployed implementations. Thus, the approaches cannot prove whether the opaque service implementation adheres to the choreography specification.

Alberti et al. [36] present an approach based on a computational logic for runtime conformance verification. The language does not support an a priori unknown number of participants and the concrete integration to the ESB is not shown. Rozinat and van der Aalst [37] show a conformance approach based on Petri process models. They provide metric definitions of the degree of conformance to the specified model. The support of an a priori unknown number of participants and the integration to the ESB is not shown.

Gheorghe et al. [38] combine enforcement capabilities of a BPEL engine with the ones of an enterprise service bus (ESB). Gheorghe et al. monitor the execution of the process and the message exchange using events, which trigger actions if a certain rule is matched. The actions include altering the process instance and modification of messages. Thus, the possible violations are modeled in a declarative way. This is similar to the approach taken by Montali et al. [39], which offer a declarative way to specify choreographies. In our approach, we opt for the explicit way of specifying choreographies. Once a choreography is explicitly specified, it is not required any more to specify additional declarative properties to ensure compliance: The modeled choreography artifact can directly used to check compliance and take necessary actions.

The work of Gheorghe et al. [38] builds on the work presented by Gheorghe et al. [40]. There, the ESB part of [38] is detailed. The work of Birukou et al. [41] also describes a solution for compliance checking at an ESB. Here, the events produced by a BPEL engine are put to the ESB, where the events are logged and analyzed by a business intelligence engine and by a complex event processing engine. That approach also does not rely on a choreography description to check for compliance with the service interaction to the choreography model.

Daniel et al. [42] survey on business compliance checking, where runtime compliance is a part of. No work uses a choreography description language supporting all choreography requirements by Decker et al. and uses the choreography description itself at the ESB to check for compliance.

A survey on the history on protocol design is provided at by von Bochmann [43]. Although the authors do not explicitly mention the modeling style used, the models presented there follow the interconnection modeling approach.

The S-BPM language PASS (as presented by Fleischmann [44]) builds on the interconnection model paradigm. There currently is no evaluation whether the interaction modeling paradigm is suitable for the S-BPM approach, too.

7 Conclusion and Outlook

This paper presented BPEL^{gold} as alternative to WS-CDL and BPEL4Chor tightly integrated in the Web service stack. Similar to WS-CDL it supports the interaction modeling approach, but supports all common requirements on choreography languages. Similar to BPEL4Chor it builds on the control flow constructs of BPEL, but follows the interaction modeling paradigm instead of the interconnection modeling paradigm.

After presenting BPEL^{gold} and an evaluation of it, we gave an overview on CASmix, a choreography-aware service bus. We have shown how a BPEL^{gold} choreography description can be deployed on CASmix enabling tracking of the choreography and reacting on derivations of the choreography. Neither we measured the efficiency choreography design using BPEL^{gold} nor measured the cost of the additional message flow through the ODE^{gold} component and leave that as future work.

The current limitation of CASmix is the missing enforcement of choreographies. For instance, if messages are sent out of order, it may be the case that CASmix puts the choreography in a faulting state instead of holding the message back until it can be handled by the choreography. Thus, future work is a detailed investigation on possible implementation strategies for choreography-aware enterprise service buses. This includes a transformation of BPEL^{gold} models to state machines, where a choreography-aware enterprise service bus can keep track the changes without the need of a customized BPEL engine.

Acknowledgments This work was supported by funds from the European Commission (contract no. 215175 for the FP7-ICT-2007-1 project COMPAS, `http://www.compas-ict.eu`).

References

1. Peltz, C.: Web Services Orchestration and Choreography. IEEE Computer **36**(10) (2003) 46–52
2. OASIS: Web Services Business Process Execution Language Version 2.0 – OASIS Standard. (2007)

3. Decker, G., Kopp, O., Leymann, F., Weske, M.: BPEL4Chor: Extending BPEL for Modeling Choreographies. In: International Conference on Web Services, IEEE Computer Society (2007)
4. Chappell, D.A.: Enterprise Service Bus. Theory in Practice. 1 edn. O'Reilly Media (July 2004)
5. Object Management Group (OMG): Business Process Model and Notation (BPMN) Version 2.0. (2010) `http://www.omg.org/cgi-bin/doc?dtc/10-06-04`.
6. Decker, G., Kopp, O., Barros, A.: An Introduction to Service Choreographies. Information Technology **50**(2) (February 2008) 122–127
7. Wolf, K.: Does my service have partners? LNCS T. Petri Nets and Other Models of Concurrency **5460**(2) (2009) 152–171
8. Decker, G., Weske, M.: Interaction-centric Modeling of Process Choreographies. Information Systems (2010) in Press.
9. Zaha, J.M., Dumas, M., ter Hofstede, A., Barros, A., Decker, G.: Service Interaction Modeling: Bridging Global and Local Views. In: EDOC, IEEE (2006)
10. Decker, G., Weske, M.: Local Enforceability in Interaction Petri Nets. In: Business Process Management. Volume 4714 of LNCS., Springer (2007) 305–319
11. Lohmann, N., Wolf, K.: Realizability is Controllability. In: Web Services and Formal Methods, 6$^{\text{th}}$ International Workshop (WS-FM 2009, Springer (2009)
12. Kopp, O., Leymann, F., Wu, F.: Mapping interconnection choreography models to interaction choreography models. In: Central-European Workshop on Services and their Composition, ZEUS 2010, CEUR-WS.org (2010)
13. Decker, G., Barros, A.P.: Interaction Modeling Using BPMN. In: 1$^{\text{st}}$ International Workshop on Collaborative Business Processes, Springer (2007)
14. Decker, G., Kopp, O., Leymann, F., Weske, M.: Interacting services: From specification to execution. Data & Knowledge Engineering **68**(10) (April 2009) 946–972
15. Kavantzas, N., Burdett, D., Ritzinger, G., Lafon, Y.: Web Services Choreography Description Language Version 1.0. (November 2005)
16. Barros, A., Dumas, M., ter Hofstede, A.: Service Interaction Patterns. In: 3$^{\text{rd}}$ International Conference on Business Process Management (BPM). Volume 3649 of LNCS., Springer (2005)
17. Decker, G., Kopp, O., Leymann, F., Pfitzner, K., Weske, M.: Modeling Service Choreographies using BPMN and BPEL4Chor. In: International Conference on Advanced Information Systems Engineering (CAiSE '08), Springer (2008)
18. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow Patterns. Distributed and Parallel Databases **14**(1) (2003) 5–51
19. Decker, G., Mendling, J.: Process Instantiation. Data & Knowledge Engineering **68** (2009) 777–792
20. Barros, A.P., Decker, G., Dumas, M., Weber, F.: Correlation Patterns in Service-Oriented Architectures. In: International Conference on Fundamental Approaches to Software Engineering (FASE). LNCS (2007)
21. Russell, N., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Workflow Data Patterns: Identification, Representation and Tool Support. In: 24$^{\text{th}}$ International Conference on Conceptual Modeling (ER 2005), Springer (2005)
22. Russell, N., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Workflow Exception Patterns. In: Advanced Information Systems Engineering (AISE). Volume 4001 of LNCS., Springer (2006)
23. Kopp, O., Martin, D., Wutke, D., Leymann, F.: The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages. Enterprise Modelling and Information Systems **4**(1) (June 2009) 3–13

24. Apache: ServiceMix Website `http://servicemix.apache.org/`.
25. Apache: ODE Website `http://ode.apache.org/`.
26. Kopp, O., Leymann, F., Wutke, D.: Fault Handling in the Web Service Stack. In: ICSOC 2010, Springer (2010)
27. Decker, G., Zaha, J.M.: On the Suitability of WS-CDL for Choreography Modeling. In: EMISA 2006 – Methoden, Konzepte und Technologien für die Entwicklung von dienstbasierten Informationssystemen. Volume 95 of LNI., GI (2006)
28. Barros, A., Dumas, M., Oaks, P.: A Critical Overview of the Web Services Choreography Description Language (WS-CDL) (March 2005) BPTrends.
29. Fredlund, L.R.: Implementing WS-CDL. In: Proceedings of JSWEB 2006 (II Jornadas Cientfico-Tcnicas en Servicios Web). (2006)
30. Kang, Z., Wang, H., Hung, P.C.: WS-CDL+ for web service collaboration. Information Systems Frontiers **9**(4) (2007) 375–389
31. Kopp, O., van Lessen, T., Nitzsche, J.: The Need for a Choreography-aware Service Bus. In: YR-SOC 2008, Online (2008) 28–34
32. Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and Orchestration Conformance for System Design. In: Coordination 2006. LNCS
33. Li, J., Zhu, H., Pu, G.: Conformance Validation between Choreography and Orchestration. In: TASE 2007
34. Hongli, Y., Xiangpeng, Z., Chao, C., Zongyan, Q.: Exploring the Connection of Choreography and Orchestration with Exception Handling and Finalization/Compensation. In: FORTE 2007. LNCS
35. M.Bravetti, Zavattaro, G.: Towards a Unifying Theory for Choreography Conformance and Contract Compliance. In: 6th International Symposium on Software Composition (SC'07). LNCS
36. Alberti, M., et al.: Computational Logic for Run-Time Verification of Web Services Choreographies: Exploiting the SOCS-SI Tool. In: WS-FM 2006. Volume 4184 of LNCS., Springer (2006)
37. Rozinat, A., van der Aalst, W.M.P.: Conformance checking of processes based on monitoring real behavior. Inf. Syst. **33**(1) (2008) 64–95
38. Gheorghe, G., et al.: Combining Enforcement Strategies in Service Oriented Architectures. In: ICSOC 2010, Springer (2010)
39. Montali, M., Pesic, M., Aalst, W.M.P.v.d., Chesani, F., Mello, P., Storari, S.: Declarative Specification and Verification of Service Choreographiess. ACM Trans. Web **4**(1) (2010) 1–62
40. Gheorghe, G., Neuhaus, S., Crispo, B.: xESB: An Enterprise Service Bus for Access and Usage Control Policy Enforcement. In: Trust Management IV. Volume 321 of IFIP Advances in Information and Communication Technology., Springer Boston (2010)
41. Birukou, A., et al.: An integrated solution for runtime compliance governance in SOA. In: ICSOC 2010, Springer (2010)
42. Daniel, F., et al.: Business Compliance Governance in Service-Oriented Architectures. In: Proceedings of the IEEE Twenty-Third International Conference on Advanced Information Networking and Applications (AINA'09), IEEE Press (2009) 113–120
43. von Bochmann, G., Rayner, D., West, C.H.: Some notes on the history of protocol engineering. Computer Networks **54**(18) (2010) 3197–3209
44. Fleischmann, A.: What is S-BPM? In: S-BPM ONE – Setting the State for Subject-Oriented Business Process Management, Springer (2010)