Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 2880

BPEL^{gold}: Choreography on the Service Bus

Lasse Engler

Course of Study: Software Engineering

Examiner: Prof. Dr. Frank Leymann

Supervisor: Dipl.-Inf. Oliver Kopp,
Dipl.-Inf. Tammo van Lessen,
Dipl.-Inf. Jörg Nitzsche

Commenced: January 17, 2009

Completed: October 1, 2009

CR-Classification: H.4.1, C.2.4, D.2.11

Contents

1 Introduction 5

2 Foundations 7
2.1 Choreographies 7
2.1.1 Choreography Languages 8
2.1.2 Service Interaction Patterns 10
2.1.3 Let's Dance 10
2.1.4 Running Example: A Travel Agency 12
2.2 BPEL 12
2.2.1 Extending and Restricting BPEL 13
2.2.2 BPEL4Chor 14
2.3 WS-Addressing and WS-MetadataExchange 15
2.3.1 WS-Addressing 15
2.3.2 WS-MetadataExchange 15
2.4 Enterprise Service Bus 16

3 Related Work 19
3.1 Choreography Languages 19
3.2 Validation of Choreographies 19
3.2.1 Simulating (WS-CDL) Choreographies 19
3.2.2 Run-time Validation 20

4 Concept of a Choreography-aware Service Bus 21
4.1 The Idea 21
4.2 Challenges 22
4.2.1 Identifying Conversation-related Messages 22
4.2.2 Correlation 23
4.2.3 Direct Interactions Between Participants 23
4.3 The Choreography-aware Participant 24
4.3.1 The Need for a Conversation Identifier 24
4.3.2 Interfaces for CAP and CSB 25
4.4 Exception Scenarios 28
4.4.1 Category 1: Missing Interactions 28
4.4.2 Category 2: Unexpected Interactions 29
4.5 Exception Handling Strategies 31
4.5.1 Enforcement 31
4.5.2 Fault Propagation 32

4.6 Summary . . . 32

5 BPEL^{gold} 33
5.1 Requirements . . . 33
5.2 Overview . . . 34
5.3 Participant Topology . . . 35
5.4 Process Interaction Description . . . 36
5.4.1 The Interaction Activity . . . 37
5.4.2 The onInteraction Activity . . . 38
5.4.3 Abstract Process Profiles for BPEL^{gold} . . . 39
5.4.4 Correlation . . . 40
5.4.5 Exception Handling . . . 41
5.4.6 Modeling Internal Behavior . . . 42
5.5 Grounding . . . 42
5.6 Evaluation . . . 43
5.6.1 Mapping Let's Dance to BPEL^{gold} . . . 43
5.6.2 Service Interaction Patterns with BPEL^{gold} . . . 47
5.6.3 Support of Choreography Language Requirements . . . 57
5.7 Summary . . . 57

6 CASmix: A Choreography-aware Service Bus 59
6.1 Architecture . . . 59
6.1.1 ODE Architecture . . . 59
6.1.2 ServiceMix Architecture . . . 61
6.1.3 CASmix' Components . . . 62
6.1.4 Message Routing . . . 63
6.2 Implementation . . . 64
6.2.1 CASmix MessageInterceptor . . . 65
6.2.2 ODE^{gold} . . . 66
6.2.3 CASmix ChoreographyManager . . . 67
6.2.4 Deployment Artifacts . . . 67
6.3 Implementing a CAP using BPEL Events . . . 68
6.4 Summary and Limitations . . . 68

7 Summary and Outlook 71

A Dialect for the Metadata Exchange 73

B BPEL^{gold} Travel Agency Scenario 75
B.1 Participant Topology . . . 75
B.2 Process Interaction Description . . . 76
B.3 Grounding . . . 78

Bibliography 81

Abbreviations 87

1 Introduction

In a *Service Oriented Architecture (SOA)* business functionality is exposed as *(web) services*. These services can be *orchestrated* into a single business process. This allows businesses to rapidly create new processes and adapt existing ones. The *Business Process Execution Language (BPEL)* is the de-facto standard for modeling such orchestrations in the web services world.

Processes usually exchange information with other processes at business partners to achieve their goal. *Choreographies* are used to describe the interactions among them from a global perspective. Today, choreographies are mainly used during design time or to check conformance afterwards using audit logs, but not during execution.

An *Enterprise Service Bus (ESB)* is middleware to connect services—or, in a choreography context, participants. Current ESB implementations do not offer choreography-specific features. For instance, they do not check if the messages they route conform to a choreography description.

The target of the work at hand is to develop an infrastructure for conformance-checking of choreographies based on an ESB. Such a "Choreography-aware" Service Bus is sketched in Figure 1.1. BPEL^{gold} is a choreography-related extension to BPEL introduced in this work.

The remainder of this work is structured as follows: Chapter 2 introduces the foundations that are required to understand the following chapters. This includes a presentation of existing choreography languages, our running example and related web service standards. Also the

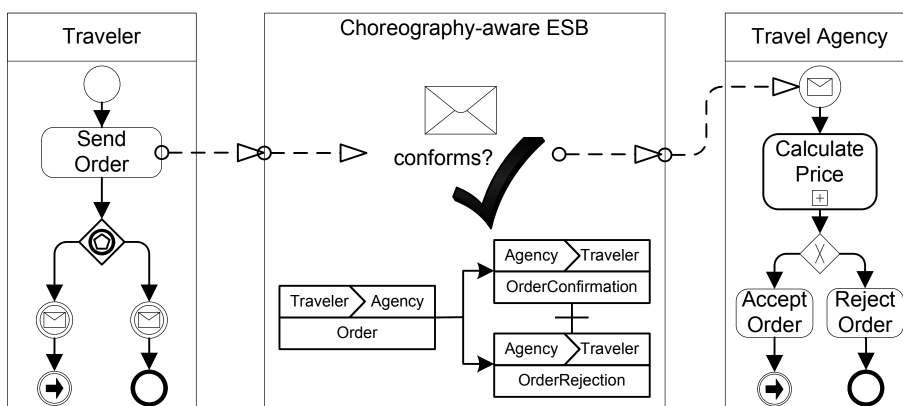


Figure 1.1: A Choreography-aware Service Bus

terminology used in this work and the traditional Enterprise Service Bus (ESB) are introduced. In Chapter 3 we give an overview on related work. The concept of a Choreography-aware Service Bus (CSB) is presented in Chapter 4 along with the related challenges. We show exception scenarios and how a CSB can handle them. BPEL^{gold} is introduced as a choreography language that can be used as the input format for a CSB in Chapter 5. Based on the concept of a CSB and BPEL^{gold} we implemented *CASmix*, a prototype of a CSB that is described in Chapter 6. Finally, we give a summary and an outlook in Chapter 7.

2 Foundations

This chapter presents the terms and concepts that are required to understand the remainder of this work. This involves an introduction to choreographies and their modeling languages (Section 2.1). Furthermore, the running example that is used throughout this work is introduced (Section 2.1.4). Then a short introduction to BPEL (Section 2.2) and other required Web Service standards (Section 2.3) is given. Finally, the concept of an ESB is discussed in Section 2.4.

2.1 Choreographies

In the following we define the terms that are used throughout this work. For the most part we follow the terminology that is defined in [Wes07].

A *business process* "consists of a set of activities that are performed in coordination in an organizational and technical environment. These activities jointly realize a business goal. Each business process is enacted by a single organization, but it may interact with business processes performed by other organizations." [Wes07] When the activities of a business process are realized using (web) services, the model of the business process is also called an *orchestration*. Instances of orchestrations are usually executed and controlled by a *process engine*.

The interactions among a set of business processes can be specified in a *choreography*. Each process plays a specific *role* in the choreography. In contrast to orchestrations, there usually is no central engine that executes a choreography. Choreographies rather act as a specification and "are often the starting point for implementing new orchestrations or for adapting existing ones" [DKB08].

In this work, we use the terms *choreography* or *choreography description* to refer to the model level and the term *conversation* to refer to the instance level. Analogously, a *participant* is an instance—or implementation—of a *role*.

The following life cycle for choreographies is described in [DKB08]: First a choreography is created in the *design and verification phase*. Then, in the *distribution phase* the *observable behavior models* for the individual business processes are derived from the choreography. They also serve as the foundation for orchestrations that are used by the participants. In the *monitoring phase* the choreographies are used to monitor the interaction among the related orchestrations. Finally, the *evaluation phase* is used to analyze completed or running conversations. Information such as the fact that some constraints are frequently not met or some options are never chosen can be used to optimize the choreography.

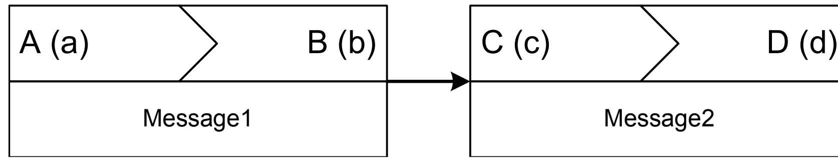


Figure 2.1: Locally unenforceable Choreography

2.1.1 Choreography Languages[1]

Choreography languages can be categorized using two criteria: On the one hand languages are either *implementation-specific* or *implementation-independent*. On the other hand these languages can also be distinguished with respect to their supported modeling style.

Implementation-independent languages are mainly used to describe and communicate about processes from a business perspective. Decisions on how to optimize processes regarding performance, quality or distribution of responsibilities are made based on the models. Defining concrete message formats or communication protocols is in the scope of implementation-specific languages. These target those users that are involved in the actual execution of business processes in the IT departments.

Basically, there exist two modeling approaches for choreography languages: *interaction models* and *interconnected interface behavior models*. In interaction models, the basic building block is an interaction—a message exchange between two participants. The control flow is modeled by specifying dependencies between these interactions. Thus, interaction models provide a truly global view on a choreography. In contrast, for interconnected interface behavior models, the control flow is defined per role. The individual interface behavior models are then stitched together using message links.

In the case of interaction models, there can be dependencies that are not *locally enforceable* [ZDH+06, DBKL08]. This means that it is not possible to create local behavior models that collectively enforce all constraints modeled in the interaction model. An example is given in Figure 2.1. First a message is sent from participant A to participant B. The second interaction between participant C and participant D may only occur after the first message is sent. This constraint can only be enforced globally. A choreography-aware Service Bus as introduced in Chapter 4 can handle such issues. Normally this model must be transformed into another model that is locally enforceable, for example by adding synchronization messages.

Problems that can occur with interconnected behavior models are related to the notion of *compatibility* [DW09, Dec09]. One typical situation is a *deadlock*. Imagine a participant that is waiting for a message from another participant and the message is never sent.

Figure 2.2 shows a categorization of the choreography languages that are presented in the following.

[1] This section is mainly based on [DKB08], other references are explicitly stated.

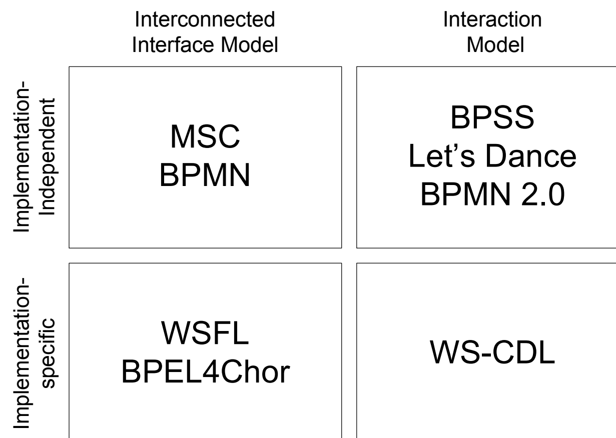


Figure 2.2: Categorization of Choreography Languages [DKB08]

The Business Process Modeling Notation (BPMN) [BPM06] is a graphical modeling language and the de-facto standard for process modeling on the implementation-independent level. BPMN uses the concept of "pools" to assign activities to an organizational unit and explicitly distinguishes between control and message flow. Thus, the interface behavioral models can be realized by using one pool for each participant and connect its activities through the control flow. The message flow is used to connect the pools. As every participant is represented by a pool it is problematic to model choreographies where more than one participant for a role is involved. With iBPMN [DB08] there also exists an extension to BPMN that allows interaction modeling—and also multiple participants per role. The pools in iBPMN are empty and the control flow is modeled for interactions between participants.

BPMN Version 2.0, which is currently in the final stages of becoming an official standard [Sil09], introduces two new diagram types: choreography diagrams and collaboration diagrams. The former allows modeling the control flow between distinct interactions whereas the latter provides an overview on complex scenarios with several participants[All09].

Another possibility to model choreographies on the implementation-independent level are Message Sequence Charts (MSCs) [IT00]. As they do not support conditional branching, parallel branching and iterations, they are more suited for modeling simple interaction sequences than for complex choreographies.

The Business Process Schema Specification (BPSS) [CCK+01] is an XML-based choreography-language that supports modeling implementation-independent interaction models. The main drawback of BPSS is that it only supports choreographies between two participants.

Let's Dance is a graphical modeling language for interaction modeling. We use Let's Dance to illustrate most of the examples in this work and introduce the language in detail in Section 2.1.3.

Most languages on the implementation-specific level are related to web service technologies. One of them is the Web Services Flow Language (WSFL) [Ley01]. A choreography description

in WSFL consists of various local views, so-called flow models, and a global model. Each flow model requires and provides operations. These operations are stitched together in the global model.

BPEL4Chor is another implementation-specific language following the interconnected behavior models approach. It is based on BPEL and its concepts are described in Section 2.2.2.

The Web Services Choreography Description Language (WS-CDL) is the only common implementation-specific interaction modeling language for web service choreographies. Dependencies between interactions are defined through a set of control flow constructs that can hardly be mapped to those of BPEL. This is one of the points that WS-CDL has been criticized for in literature [DOZ06, BDO05].

In Chapter 5 of this work we propose a new interaction modeling language based on BPEL.

Most of the languages presented in this section have been evaluated against their suitability to model choreographies in [DKLW08].

2.1.2 Service Interaction Patterns

The service interaction patterns [BDH05b, BDH05a] are a collection of patterns that cover common scenarios with multilateral, competing, atomic and causally related interactions. They are also a good instrument to benchmark choreography languages. The patterns and an evaluation of BPEL^{gold} against them are presented in Section 5.6.2.

Due to time-constraints, BPEL^{gold} is not evaluated against other patterns such as process instantiation patterns [DM08] or correlation patterns [BDDW07]. An overview on these patterns is also given in [KMWL08].

2.1.3 Let's Dance

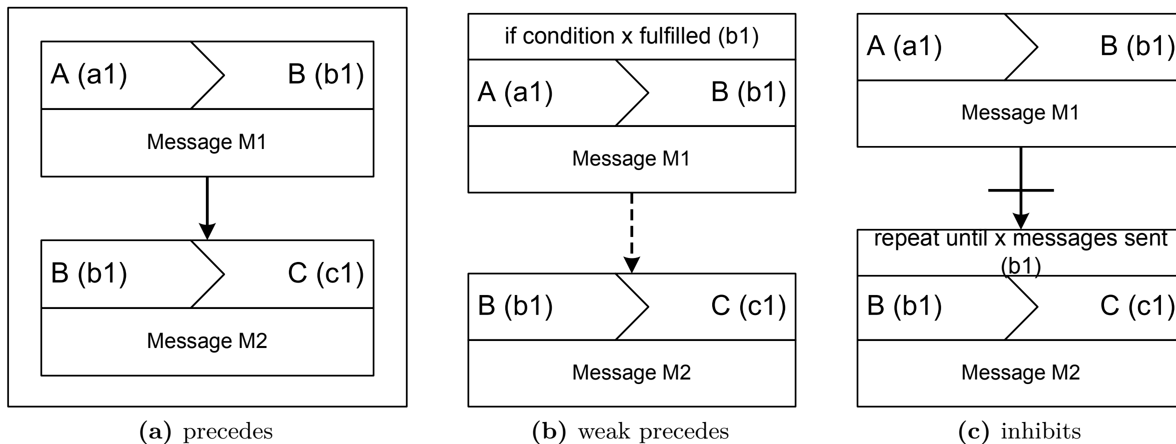
Let's Dance is a visual choreography language that is implementation-independent and supports interaction models. The language has been introduced in [ZBDH06] and a formal definition can be found in [DZD06].

The elementary building block in Let's Dance is an *interaction* or message exchange. An interaction is represented by a rectangle that is formed by two non-regular pentagons. These pentagons stand for the communication actions *send* (exterior apex) and *receive*.

The role of the participant is written in the middle of each communication action. The participant (or *actor*) itself is denoted in brackets.

The message type is written underneath the two communication actions.

Thus, the interaction in the top of Figure 2.3a can be read as follows: Participant “a1”, which plays the role “A”, sends a message of type “message M1” to participant “b1”, which plays role “B”.



(a) precedes **(b)** weak precedes **(c)** inhibits

Figure 2.3: Let's Dance Relationship Types

Interactions can be connected by three different relationship types:

precedes: The target interaction of the directed edge can *only* occur after the source interaction has occurred (Figure 2.3a).

weak-precedes: The target interaction of the dashed directed edge can only occur after the source interaction either has occurred or has been inhibited (Figure 2.3b).

inhibits: The target interaction of the crossed directed edge can *no longer* occur after the source interaction has completed (Figure 2.3c). A special case is the `two-way inhibit`, depicted by an undirected crossed edge. This is an abbreviation for using two `inhibits` relations and means that exactly one of the interactions can occur.

Interactions can be grouped into a *composite interaction* by surrounding them with a rectangle (see Figure 2.3a). The *sub-interactions* within a composite interaction may, but do not have to, be related. If there is no relationship between them, they can occur in any order or in parallel. Composite interactions can be connected to other interactions using any of the relationship types.

Repetition of interactions can be modeled using one of these three constructs: *until*, *while* and *for each*. The latter can be sequential or concurrent and can either iterate over variables or actor references.

The condition under which an interaction may be executed can be defined using a *guard*. An example is shown in the top-right of Figure 2.3. The actor denoted in brackets evaluates the expression.

Finally, Let's dance provides a *timer* construct to model time constraints on interactions. Timers can be connected to interactions using any relationship type and they "occur" when the time elapses.

2.1.4 Running Example: A Travel Agency

The choreography that is used as the running example in this work is presented in Figure 2.4. It is used to illustrate, for instance, our exception scenarios (Section 4.4) and BPEL^{gold} examples (Chapter 5). Furthermore, the test setup for our implementation (Chapter 6) is based on it.

A traveler wants to book a flight. This involves the traveler, the travel agency and an unknown number of airlines. First, the traveler sends his order to the agency. The order contains a destination and a maximum price for the flight. A price request is then sent to each airline by the agency. When the agency has received all offers, they select the airline with the best offer (sa). If the price is within the accepted range, the agency books the flight and sends a confirmation to the customer. The ticket order can no longer be sent after a given time, because the airlines do not wait forever. The traveler finally receives either a confirmation and an e-ticket or a rejection message.

Assume an error occurs at the agency and the process stops after it has received the travel order. Then neither a confirmation nor a rejection is sent to the traveler. This means the traveler's process will wait forever because it is not notified of the error. A solution to this problem is an ESB that is aware of this conversation. Once the ESB has detected that the process at the travel agency has stopped, the ESB can for example send a fault message to the traveler.

Other scenarios that can be handled by a choreography-aware ESB include violations of the behavior that is described by the choreography. The agency could for example send a confirmation to the traveler before they have received the confirmation from the airline. Or they could send both, a rejection to the user and an order to the agency.

More scenarios and how they can be handled by a choreography-aware ESB are discussed in Section 4.4.

2.2 BPEL[2]

In this section we introduce the concepts of the Business Process Execution Language (BPEL) Version 2.0 [AAA+07]. A basic understanding of the language is needed for the introduction of BPEL^{gold} in Chapter 5.

Activities are the main building block in BPEL. There are structured and basic activities.

Basic activities include the communication activities `<receive>`, `<reply>` and `<invoke>` or the `<assign>` activity that copies data from one location to another—for instance from an XPath expression into a variable.

`<if>`, `<while>`, `<repeatUntil>` and `<forEach>` are structured activities whose semantic is equal to the corresponding constructs of programming languages like Java. Branches of `<forEach>` activities can also be executed in parallel. A `<sequence>` activity is an aggregation

[2]This section is mainly based on [AAA+07, WCL+06], other references are explicitly stated.

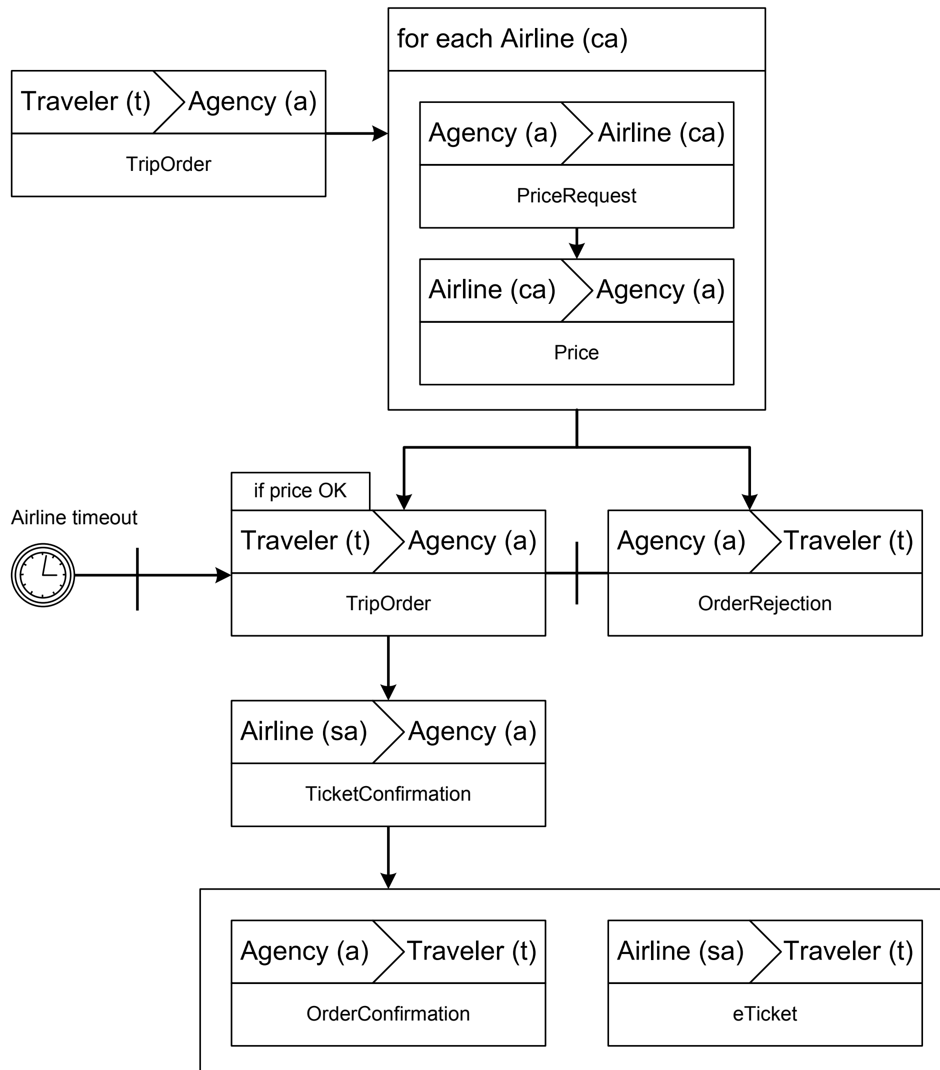


Figure 2.4: The Running Example: Travel Agency Choreography

of other activities that are executed in the specified order. Activities that are nested in a `<flow>` activity are executed in parallel. Conditional control links can be used to partially order them. To ensure a proper execution of the process, possible dead paths are eliminated using Dead Path Elimination (DPE) [LR00, CKLW03].

2.2.1 Extending and Restricting BPEL

BPEL's XML schema allows namespace-qualified attributes on any BPEL element and also elements from other namespaces within BPEL elements. Extensions are declared using a namespace and a definition of whether they are optional or mandatory. If a BPEL processor

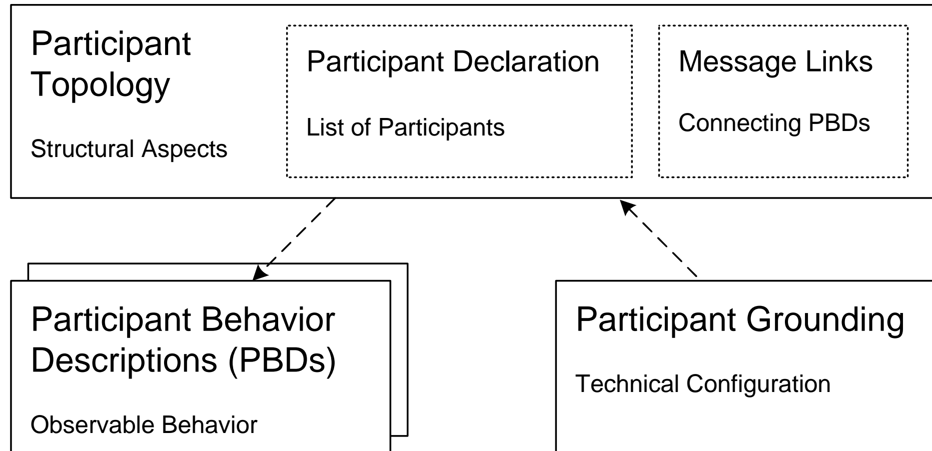


Figure 2.5: BPEL4Chor Artifacts [DKLW08]

does not understand one of the mandatory extensions it must reject the process definition. New activities can be used within an `<extension>` activity.

Language extensibility is discussed in Sections 5.3, 10.9 and 14 of the BPEL specification [AAA+07].

BPEL also provides the possibility to restrict the constructs that can be used in an abstract process description by the means of *Abstract Process Profiles*. In an Abstract Process Profile, one can define which syntactic elements are allowed and which can be omitted. Every Abstract Profile is based on the *Common Base* which allows all of BPEL's syntactic elements and thus the expressive power is not limited compared to an executable process. It also allows activities, expressions, from-specs and all attributes to be omitted. Thereby the strict coupling of BPEL and Web Services Description Language (WSDL) [CCMW01] can be bypassed when omitting partner link references from communication activities as done for BPEL4Chor participant behavior descriptions (see below). Another approach to achieve this decoupling is to prohibit the use of the standard communication activities and replace them with "WSDL-less" alternatives as proposed in [NLKL07].

Abstract processes are discussed in Section 13 of the BPEL specification [AAA+07].

2.2.2 BPEL4Chor

BPEL4Chor is an implementation-specific choreography language that supports the interconnected interface behavior models approach and is based on BPEL.

Figure 2.5 shows the three BPEL4Chor artifacts: *Participant behavior descriptions*, the *participant topology* and the *participant groundings*. The participant behavior descriptions define the observable control flow at a specific participant. BPEL4Chor provides a special abstract profile for BPEL to model these definitions. The participant topology defines the

structural aspects of a choreography by describing the involved participants and the messages flowing between them. Participant groundings bring in the technical configuration such as WSDL port types.

Further details on the latter two artifacts are presented in Chapter 5 as they are reused for BPEL^{gold}.

2.3 WS-Addressing and WS-MetadataExchange [3]

This section describes those parts of Web Services Metadata Exchange (WS-MEX) and Web Services Addressing (WS-Addressing) that are relevant for this work. For a detailed introduction please refer to [Pap08, WCL+06] or the mentioned specification documents.

2.3.1 WS-Addressing

In a transport-protocol-neutral web service world it is not trivial to ensure that messages are correctly delivered to the appropriate destination. *WS-Addressing* [BCC+04] defines two concepts to achieve this goal: *endpoint references* and *message information headers*. Endpoint references provide a way to encode the addressing information needed to reach an endpoint. We focus on message information headers which allow messages to be directed to endpoints. They also include information necessary for a bidirectional and asynchronous interaction.

Only two of the message headers are mandatory: the *To*-header that contains the URI of the target endpoint and the *Action*-header to identify the purpose of the message sent (such as the related operation). The other headers are (all optional): *ReplyTo*, *FaultTo*, *Source* contain the sender's endpoint or an endpoint where a reply or fault should be sent to; *MessageId* and *RelatesTo* can be used to define relationships between messages. For instance to indicate that one message is the response to another.

The headers containing an endpoint reference require special handling of a CSB as discussed in Section 4.2.3.

2.3.2 WS-MetadataExchange

WS-MEX [BBB+06] basically defines a simple WSDL interface that allows metadata to be queried directly from an endpoint. This metadata describes what other endpoints need to know to interact with it. The interface describes two request-response operations: *GetMetaData()*—which is the only operation that has to be provided by a WS-MEX-compliant endpoint—and *Get()*.

A *dialect* is a specific XML language which is usually identified by the URI of the target namespace of the corresponding XML Schema Definition. In Section 4.3.2 we introduce a new

[3] This section is mainly based on [Pap08, WCL+06], other references are explicitly stated.

dialect that is used by the CSB to gather metadata from participants and also show a sample response to a *GetMetadata()*-request.

A *GetMetadata()*-request can either be generic, dialect-specific or identifier-specific. For a generic request all relevant metadata is returned. For instance, the WSDL dialect provides the WSDL description of the endpoint. *Identifiers* allow the requester further constrain the request to ask for specific descriptions. They are not used in this work.

The response to a *GetMetadata()*-request either contains all the metadata embedded or a reference to another endpoint or URL. This is useful for large amounts of data and involves the *Get()*-operation. This topic is not further discussed as it is not relevant for this work.

2.4 Enterprise Service Bus

There is no consensus on a general definition for the term Enterprise Service Bus (ESB). For this work we use one from [Cha04]: "An ESB is a standards-based integration platform that combines message oriented middleware, web services, data transformation and intelligent routing to reliably connect and coordinate the interaction of significant numbers of diverse applications across extended enterprises with transactional integrity." Where an extended enterprise "represents an organization and its business partners, which are separated by both business boundaries and physical boundaries".

To give a better understanding, we list a summary of the main characteristics of an ESB according to [Cha04] below:

C1. Pervasiveness: An ESB can be adapted to suit the needs of general-purpose integration projects across a variety of integration situations. It is capable of building out integration projects that can span an entire organization and its business partners.

C2. Highly distributed, event-driven SOA: Loosely coupled integration components can be deployed on the bus across widely distributed geographic deployment topologies, yet are accessible as shared services from anywhere on the bus.

C3. Selective deployment of integration components: Adapters, distributed data transformation services, and content-based routing services can be selectively deployed when and where they are needed, and can be independently scaled.

C4. Security and reliability: All components that communicate through the bus can take advantage of reliable messaging, transactional integrity, and secure authenticated communications.

C5. Orchestration and process flow: An ESB allows data to flow across any applications and services that are plugged into the bus, whether local or remote.

C6. Autonomous yet federated managed environment: An ESB supports local autonomy at a departmental and business unit level and is still able to integrate in a larger managed integration environment.

C7. Incremental adoption: An ESB can be used for small projects: Each individual project can build into a much larger integration network, which can be remotely managed from anywhere on the bus.

C8. XML Support: An ESB can take advantage of XML as its "native" datatype.

C9. Real-time insight: An ESB provides the underpinnings to enable real-time insight into live business data.

In Chapter 4 we show how an ESB can be extended to support choreography-related features such as conformance checking and enforcement. A prototypical implementation of this concept, based on an open source ESB, is presented in Chapter 6. An evaluation of existing open-source ESBs can be found in [RD08, BHR].

[Ley05] sketches the vision of a "ubiquitous service bus", emphasizing *virtualization* as the main feature of a service bus. Choreography descriptions could also be used for an enhanced service selection but this is out of the scope of our work.

3 Related Work

This chapter gives an overview on related work in the area of choreography languages and the validation of choreographies.

3.1 Choreography Languages

We already introduced work related to choreography languages in Section 2.1.1.

An approach to replace WS-CDL with WSFL and BPEL 1.0 is presented in [Kip06]. There, choreographies are modeled in a style similar to interconnected interface behavior models—with all roles in one process model. BPEL^{gold} (Chapter 5) provides a true interaction model and is based on BPEL 2.0.

3.2 Validation of Choreographies

Choreographies can either be validated statically, through simulation or during run-time.

There are several techniques to analyze choreographies in the design and verification phase (see Section 2.1). They primarily address compatibility and conformance checking. [LKLR07] presents approaches using Petri nets exemplarily for BPEL4Chor choreographies.

3.2.1 Simulating (WS-CDL) Choreographies

Although the standard states that "WS-CDL is not an 'executable business process description language' or an implementation language" [KBR+05], there is some work dealing with the execution or simulation of WS-CDL choreographies.

[Fre06] introduces a prototypical implementation that allows "debugging and simulating" existing WS-CDL documents. As there are no implementations of participants involved and thus no messages exchanged, this is far from really "executing" or monitoring a choreography.

In contrast, the implementation presented in [KWH07] supports handling message exchanges. It also supports fault handling and a coordination mechanism to propagate exceptions. However, the input format is not plain WS-CDL but WS-CDL+, an extension they have built to overcome some of the limitations they see in WS-CDL. The new features include a timer, an implicit finalization mechanism and enhanced debugging and exception options.

Both solutions can be used to test WS-CDL choreographies during the design and verification phase of the choreography lifecycle.

3.2.2 Run-time Validation

There is few work that targets validation of choreographies during the execution.

[ACG+06] introduces a language based on computational logic to formally describe choreographies. Such descriptions are utilized by a tool that is based on a inference engine to validate choreographies during run-time. A similar approach based on the event calculus [KS86] is introduced in [CMMT09].

A way to validate choreographies that can be formalized in Linear Temporal Logic (LTL) is presented in [HV09]. Notations that are translatable to LTL include Let's Dance and Message Sequence Charts (MSCs). The work defines a mapping of LTL to XQuery expressions. These expressions are used by a central XQuery processor that validates the message sequences against them.

The work presented so far is focused on the formalisms to describe choreographies and the proposed implementations are not suited for use in a web service environment. The following approaches also address implementation challenges such as correlation.

[RR09] proposes a "reliable monitoring framework" to validate choreographies that are modeled in WS-CDL. This means that a copy of each message that is routed through an ESB (WSO2 ESB[1]) is forwarded to a central logging service. The data that is collected by this service can then be used to monitor and validate the running conversations. In order to correlate the messages to conversations, the participants have to provide an "instance identifier" in every message header. In contrast, our approach is to keep the validation transparent for the participants (see Section 4.2.2).

The work presented in [KMM08] is also based on an ESB implementation (Mule ESB[2]). They use a variant of MSCs to model choreographies. The models are used to generate one runtime monitor implementation for each role. These are then attached to each participant using aspect-oriented technologies. The monitors are based on a state machine and can reject messages that do not fit to the current state.

An ESB that "natively supports complex Message Exchange Patterns (MEPs)" has been proposed in [Tos08]. Such an ESB can check if exchanged messages fit to a specified MEP during run-time. When breaking choreographies down to MEPs this solution could also be used to verify them.

In contrast to the work at hand, the presented solutions only concentrate on the detection of faulty behavior in a conversation. Our approach also proposes concepts to actively deal with the faults when they occur and also to enforce a choreography under certain circumstances (see Section 4.5).

[1] http://wso2.org/projects/esb/java
[2] http://www.mulesource.org

4 Concept of a Choreography-aware Service Bus

So far we have introduced the foundations in the domain of choreographies as well as the concept of a (traditional) Enterprise Service Bus (ESB). In this chapter we propose a concept to add choreography-related functionality to an ESB: the *Choreography-aware Service Bus (CSB)*.

First, we introduce the basic idea (Section 4.1) and discuss challenges that arise from such a concept (Section 4.2). Then the *Choreography-aware Participant (CAP)* is presented as an additional component of this concept (Section 4.3). Finally, we discuss exception scenarios (Section 4.4) and how they can be handled by a CSB (Section 4.5).

CASmix, a prototypical implementation of a CSB, is presented in Chapter 6.

4.1 The Idea

The fundamentals of the idea of a Choreography-aware Service Bus (CSB) presented in this work are based on [KLN08].

A traditional ESB as introduced in Section 2.4 is not choreography-aware. That means it does not know if and how the messages it routes are part of a conversation. Thus it is not possible to check if a message conforms to a choreography during run-time. There exist approaches to check conformance afterwards using audit logs [SM05]. Such a solution is not feasible for long-running conversations. If an early message exchange is faulty, it should be detected directly and not when the conversation has ended—possibly after years.

In contrast, a Choreography-aware Service Bus (CSB) checks every message it routes against a given choreography description as depicted in Figure 1.1 on page 5. The message is only routed to the receiver if it conforms to the choreography. Thereby the faulty message does not trigger further processing which would have to be reverted. Think of an e-ticket that is sent to the traveler although the travel agency has not confirmed the order (see our running example in Section 2.1.4). Detection of a faulty message can either lead to enforcement of a choreography or to exception handling (see Section 4.5).

Another scenario is the following: The traveler sends its order to the agency. The agency receives the message and both processes continue. Then the agency's process is terminated for some reason. Later, the traveler waits for a message—either a rejection or a confirmation. This message will never be sent as the agency is no longer "alive". If there is no timeout defined, the traveler will probably wait forever. A CSB can detect such a situation and inform the traveler about the exception.

We discuss these scenarios and how a CSB can handle them in detail in Section 4.4 and Section 4.5.

In addition to conformance checking, such a CSB provides choreography-specific functionality related to virtualization. For instance, participants could be discovered and bound based on their role in a choreography. This thesis focuses on conformance checking.

We argue that the choreography-awareness of the bus should be transparent for the participant. This means that it can send a request without additional choreography-related information to the bus.

4.2 Challenges

This section provides an overview on challenges that have to be considered when the presented concept is realized. A CSB must identify messages that are actually related to a choreography. It then must correlate the message to a specific conversation. Finally, it has to be made sure that all interactions of a conversation are routed through the CSB.

4.2.1 Identifying Conversation-related Messages

As we assume that messages not related to any conversation can be sent via the CSB, there has to be a mechanism to identify the messages that are conversation-related. The CSB has to achieve this identification based on the information contained in the message—within the header or the payload. For the sake of generality, we do not regard implementation-specific properties that might be provided by the underlying messaging infrastructure. This information can be used for an optimized approach.

Ideally, there would be a special token in the message that refers to the choreography description and probably also the role of the sender and receiver. The requirement to keep the choreography-awareness transparent to the participant inhibits this solution. Therefore we have to work with the information that is already available. Because an ESB is usually based on messaging middleware we can only be sure about the availability of two information bits: the endpoint of the recipient and the message type. The sender is usually not available. Headers like ReplyTo from WS-Addressing do not necessarily point to the original sender.

For our approach we presume that the CSB has access to the following information: a list of all choreographies that an endpoint can take part in; a list of all roles an endpoint can play in a specific choreography; a list of all message types (or operations) that are used for a specific role.

Together with the data from the message, the CSB can identify a conversation-related message via the following rule: *the message can only be part of a conversation if the message type is used for at least one role that can be played by the receiver.*

The CSB still needs to check if the message can be related to a specific conversation to definitely decide if it is conversation-related. This check is required because the message type might also

be used outside of a choreography context. How a message can be related to a conversation is described in the following.

CASmix handles this problem as presented in Section 6.2.1.

4.2.2 Correlation

When a message has been identified to be part of some conversation, it also must be related to the correct conversation.

Choreographies are usually instantiated more than once in their lifetime. It is also quite possible that multiple conversations are active at the same time. So it must be possible to correlate a single interaction to its appropriate conversation. Consider the transmission of the price in the travel agency example.

Similarly, the participants have to correlate a message to the correct process instance. There the standard mechanism to achieve this correlation is to apply a correlation token—such as an order number—to the messages. This token can also change during the interactions among processes. For example, the travel agency sends a message to the airline with an order number. This order number is added to the airline's response so that it can be routed to the correct instance at the agency. Additionally, the airline adds a ticket number to the response. From then on, the two processes can communicate using only the ticket number as correlation token.

The correlation tokens that are used by the participants can—in general—also be used by the CSB to relate a message to a conversation. This becomes clear when the CSB is regarded as a process that is placed between the communicating participants and receives and forwards all messages.

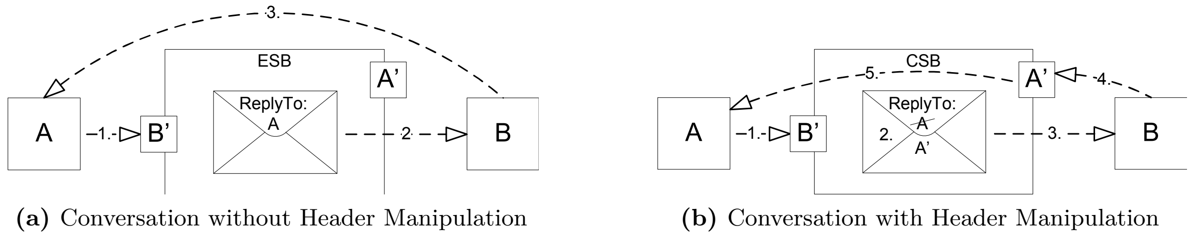
An exception to this rule occurs when stateless participants are part of a choreography. Consider a notification that is sent to a participant that is not involved in the conversation afterwards. Then the CSB needs a correlation token although it is not needed by the participants. The message could be routed correctly but the CSB could not relate it to a conversation.

BPEL^{gold}'s correlation support is presented in Section 5.4.4.

4.2.3 Direct Interactions Between Participants

In order to completely monitor a conversation, it must be made sure that all related messages are routed through the CSB. Therefore the CSB must prevent the participants from communicating directly. For two-way message exchanges the sender probably adds a return address, referring to its endpoint, to its request. The ReplyTo header of a WS-Addressing header can be used for example. Depending on the implementation, there can also be other, application-specific headers that lead to a situation where the CSB is by-passed.

A CSB must be aware of all these headers and manipulate them before routing the message to the receiver. The endpoint reference to the original sender can be exchanged with a reference



(a) Conversation without Header Manipulation

(b) Conversation with Header Manipulation

Figure 4.1: CSB: Preventing Direct Interactions

to an endpoint provided the bus. That endpoint forwards messages to the original sender as depicted in Figure 4.1.

Section 6.2 describes how this is implemented in CASmix.

4.3 The Choreography-aware Participant

Although we argue that a CSB should be transparent for the participants, there are scenarios where participants with knowledge about the choreography-awareness of the bus allow enhanced solutions. We refer to such participants as *Choreography-aware Participants (CAPs)*. They can, for example, push status information on conversations they take part to the CSB. Further use cases are discussed in Section 4.5.

A CAP provides an interface that the CSB can use to request or send information on choreography descriptions and ongoing conversations. The CSB provides a similar interface, so that the CAP can also actively start an information exchange. This section introduces these interfaces after a discussion on the need of a conversation identifier for information exchanges between the two parties.

In this work, we assume that not necessarily all participants of a conversation are choreography-aware. In cases where implementations for participants already exist, it might just not be feasible to adapt them. Thus there have to be alternatives for standard participants in every scenario.

How a CAP can be realized using BPEL events is presented in Section 6.3.

4.3.1 The Need for a Conversation Identifier

When a CAP and the CSB exchange information on conversations, they need a way to specify the respective conversation. In Section 4.2.2 we discuss correlation as a mechanism to correlate a (regular) message to a specific conversation. In theory, both parties can use this information also in the context of information exchanges. In practice, this can become complicated. Consider the participant's implementation is based on a process engine. Then the CAP-functionality would normally be implemented separately, so that the process engine does

not have to be adapted. In this case the correlation information of a local process is probably not fully available to the CAP-implementation. Furthermore, the CAP must also be able to identify a local process after this has already been completed—faultily or successfully.

There might also be cases where the correlation is modeled in a way that allows to route every message to the correct process instance but the information cannot be used to specify the conversation at any point in time [KE09]. These scenarios have not been further investigated in this work.

To overcome these difficulties, we propose to use a special identifier that uniquely specifies a conversation: the Conversation identifier (CID). In contrast to the work presented in Chapter 3 we do not require this identifier to be attached to all message exchanges but only to those related to information exchanges between CAP and CSB.

Detailed information on the usage is presented in the description of the interfaces. First, we discuss how the identifier is distributed to the participants.

A CAP has to know the identifier as soon as it takes part in the conversation. This can either be by sending or by receiving a message. In the latter case the receiver can read the CID from a special field in the header of the message. For simplification, the CSB can add this field to every message of a conversation—instead of evaluating the need individually for each interaction. We propose a header with the name *ConversationID* in the namespace *http://www.bpel4chor.org/BPELgold/csbheader*.

When the CAP enters the conversation by *sending* a message there are different solutions depending on the Message Exchange Pattern (MEP). In the case of a request-response interaction, the sender can read the CID from the response as described above. For a one-way interaction the CSB has to transmit the CID separately. Therefore the CAP must provide a message identifier in a special header field of the request. We introduce *MessageID* in the above-mentioned namespace. Alternatively, existing message identifiers, such as those of WS-Addressing, could be reused. The CSB sends the CID together with the message identifier to the CAP using the *setConversationId* operation of the interface described below. The CAP can use this information to map the the CID to its internal process identifier.

4.3.2 Interfaces for CAP and CSB

In the following we introduce the operations that must provided by a CAP and those provided by the CSB. They cover the exchange of metadata as well as information on conversation status.

WSDL definitions of the presented interfaces are available with the source code of our implementation.

Listing 4.1 Example Response from CAP to a WS-MEX GetMetadata Request

```
<?xml version="1.0" encoding="UTF-8"?>
<env:envelope xmlns:env="..." xmlns:wsx="...">
  <env:header> ... </env:header>
  <env:body>
    <wsx:MetadataSection Dialect="http://www.bpel4chor.org/BPELgold/mex/ChorList">
      <gldmex:choreographyList xmlns:tt="http://example.org/travel/topology"
        xmlns:travel="http://example.org/travel/" xmlns:ot="http://example.org/order/topology"
        xmlns:order="http://example.org/order">
        <gldmex:choreography name="travel:travelScenario"
          myRole="tt:traveler" myEndpoint="http://cap.example.org/travelerport"
          myInfoEndpoint="http://cap.example.org/choraware" />
        <gldmex:choreography name="order:orderScenario"
          myRole="ot:seller" myEndpoint="http://cap.example.org/sellerport"
          myInfoEndpoint="http://cap.example.org/choraware" />
        <gldmex:choreography name="order:orderScenario"
          myRole="ot:buyer" myEndpoint="http://cap.example.org/buyerport"
          myInfoEndpoint="http://cap.example.org/choraware" />
      </gldmex:choreographyList>
    </wsx:MetadataSection>
  </env:body>
</env:envelope>
```

Provision of MetaData

We assume that the CSB knows the endpoints of all CAPs. We leave it open to the implementation how this is achieved. For example an interface of a registry can be used or the information is deployed to the bus via configuration files. The solution for CASmix is presented in Section 6.2.4.

The CSB also needs detailed information on the choreographies a CAP takes part and which roles it can play. Therefore, every CAP must provide an implementation of Web Services Metadata Exchange (WS-MEX) (see Section 2.3) at its endpoint. We define a new WS-MEX dialect (with the URL *http://www.bpel4chor.org/BPELgold/mex/ChorList*) for the retrieval of a *choreography list*. The related XML schema definition is listed in Appendix A.

An example response to a GetMetadata request is shown in Listing 4.1. A choreography list contains several choreography elements that provides the following information: a *name* that refers to a choreography description; *myRole* that refers to a role in this choreography; *myEndpoint* refers to the endpoint where the operations for this role a provided; *myInfoEndpoint* is the endpoint of the implementation of the interface for the exchange of conversation status (see next).

Exchange of Conversation Status

The interface that can be used by the CSB to request and send status information consists of three operations: *setConversationId*, *getStatus* and *notify*.

Status	Description
ConversationStatusUnknown	No information is available for the given CID
ConversationStatusRunning	The conversation's process is currently active
ConversationStatusFinished	The conversation has completed without an exception
ConversationStatusFaulted	The conversation has completed with an exception
InteractionStatusUnknown	No information is available for the given IID
InteractionStatusInitiated	The interaction was initiated but not yet finished
InteractionStatusFinished	The interaction has completed without an exception
InteractionStatusFaulted	The interaction has completed with a fault
InteractionStatusWaiting	The process is waiting to receive the interaction
InteractionStatusSkipped	The interaction was skipped

Table 4.1: Status Types for the Information Exchange between CAP and CSB

Event	Description
ConversationFaulted	The conversation has completed with a fault
ConversationChorViolation	A message violated the choreography description
InteractionFaulted	The interaction has completed with a fault
InteractionWontInitiate	The interaction will not be initiated by the sender

Table 4.2: Event Types for the Information Exchange between CAP and CSB

The operations that are introduced in the following all require a CID to relate the call to a conversation. The CID is send to the CAP (see Section 4.3.1) through *setConversationId(messageID, CID)*. This operation has two parameters: a message identifier and a CID. The message identifier must relate to a message that was sent by the CAP in the context of the conversation that is defined by the CID. After the operation has successfully completed, the CAP must be able to relate the given CID to its process instance when one of the other operations is called.

Status information on a given conversation or interaction can be requested via the *getStatus(CID, IID?)* operation (optional parameters are indicated via "?"). This is a request-response operation which returns one of the values depicted in Table 4.1. The optional parameter *IID* can be used to refer to an interaction within the choreography description that is related to the given CID.

The operation *notify(CID, IID?, event, participantsList)* can be called by the CSB to notify the CAP about an event (see Table 4.2). The IID is again optional. The *participantsList* is a list of all participants that take part in the conversation together with the role they play.

The operations presented so far are all part of the CAP's interface. The CSB's interface is similar except for the parameters:

The operations a CSB must provide are *notify(CID, IID?, event, participant)* and *getStatus(CID, IID?)*. The semantic is the same as for the CAP. Instead of a participants list the CAP must provide a reference to itself when calling *notify*.

The set of status and event types is meant to be extensible. The event types only cover a subset of the status types because not all of them are useful for notification purposes. The exchange of every status change would be an overhead that is not acceptable in most cases. An *initiated interaction* was handed over to the bus by the sender but is not yet delivered to the receiver. For instance, this can happen when the message is blocked for enforcement. A *finished interaction* from the CSB's view has also been forwarded to the receiver—from the view of the sender it has only been initiated.

4.4 Exception Scenarios

A variety of problems can occur when participants communicate during a conversation. For example a message can be sent too late or not at all. The root cause can either be technical, such as a failure in the underlying infrastructure, or it can be on the application level, such as an incorrect modeled process that is executed by a participant. There exists a lot of work in the area of classification and terminology of exceptional behavior. A comprehensive overview on this topic with focus on choreographies is given in [Bis09].

In this work we only consider the symptoms of an exception—regardless of the cause. We focus on those scenarios that are relevant and detectable from the view of the CSB. Therefore we have identified scenarios that can be grouped into two categories: interactions that are not initiated at all and interactions that are unexpected. The initiation of an interaction refers to the point in time when a message is handed over from the sender to the CSB.

We assume that no messages can be lost within the bus and the communication between a participant and the CSB is reliable. Also, messages are handled in the order they were handed over—they cannot overtake each other inside the bus.

In the remainder of this section we discuss concrete scenarios and how they can be detected by the CSB. The possibilities to handle them are presented in Section 4.5.

4.4.1 Category 1: Missing Interactions

Exception Class 1: An interaction (without explicit timeout) is not initiated

When there is no timeout modeled, the receiver has to wait until an interaction finally occurs. Thus it will wait forever if the message is not sent. For example, this can happen when the sender's process is terminated as discussed in Section 4.1.

There are several solutions to detect such a situation: CAPs pushing status information to the CSB, the CSB pulling information from the CAPs and finally default timeouts.

If the CAP detects that an interaction will not occur it can notify the CSB. The sender can raise this event for example when its related process faults. Alternatively the CSB can—either periodically or on demand—pull the status for the conversation from the CAP. Relevant operations for these status exchanges were introduced in Section 4.3.

The pulling solution requires that the sender has already taken part in the conversation. Otherwise the CSB usually only knows the role but not the endpoint of the sender. One exception would be the case that there is only one participant that can play the role. If an endpoint reference has been included in one of the already exchanged messages the CSB can possibly also relate it to the sender. The sender, for its part, does not know the conversation identifier before it has taken part.

A solution that works in all cases, even if the sender is not choreography aware, is to configure default timeout values. This information is additional to timers that can be explicitly modeled in a choreography. The settings can either be configured per choreography (provided at deployment-time) or for all choreographies on the bus in general. When the threshold is hit, the bus can assume that the interaction will no longer occur and initiate the appropriate fault handling. Feasible values can be determined by log-file analysis for example.

The last solution does not guarantee that the interaction will not occur later on. This can lead to a problem similar to 2.2.c exceptions.

4.4.2 Category 2: Unexpected Interactions

An unexpected interaction is an interaction that is not allowed—or expected—at the current state of the conversation when it occurs. This includes interactions that occur too early or too late and also dupes. The different causes of this exception and how they can be detected are discussed in the following.

Please note that we only consider interactions that can actually be related to an active or finished conversation. Other messages are already filtered out before (see Section 4.2.1).

Exception Class 2.1: Early Interactions

Interactions can occur too early with respect to the order in the control flow or with respect to time constraints:

Exception Subclass 2.1.a: Early Interactions - Control Flow This sub-class covers interactions that are initiated too early with respect to their order in the control flow. That means at least one other interaction should have occurred before. Imagine the agency in our running example sends the order confirmation to the traveler before the agency has received the ticket confirmation from the airline. It is also very likely that such a situation occurs for locally unenforceable choreographies.

When the CSB receives such a message, it must check if a state of the conversation can be reached where this message is allowed.

Exception Subclass 2.1.b: Early Interactions - Time An interaction that is initiated too early with respect to time differs from a 2.1.a exception because there is no other interaction that should have occurred before, only a time constraint (such as a `<wait>` in BPEL). Thus it is easier to detect such an exception because it cannot be on a branch of the control flow that will not be reached.

Imagine an investment scenario where the customer receives an offer but must not accept it before the expiration of a time limit of 24 hours due to legal reasons.

(An interaction that occurs too early with respect to time *and* order in the control flow is categorized as a 2.1.a exception.)

Exception Class 2.2: Late Interactions

Late interactions can also be divided into sub-classes:

Exception Subclass 2.2.a: Late Interactions - Control Flow An interaction can be late with respect to the control flow for two reasons. Either the interaction is optional or another branch had been taken. Imagine the agency sends a ticket order to the airline after it already has send a rejection to the traveler.

A CSB must be able to tell if the interaction related the received message was skipped. As one message type can be used in several interactions of a choreography the related interaction can possibly not be identified. In this case the CSB must also make sure no message of the same type is expected later in the control flow (to distinguish this from an early interaction).

Exception Subclass 2.2.b: Late Interactions - Time An interaction that is late with respect to time occurs when there is a timeout modeled (such as `<onAlarm>` in BPEL). In our running example this can occur when the agency sends the ticket order to the airline after the timeout.

Exception Subclass 2.2.c: Late Interactions - Finished Conversation This exception refers to interactions that occur although their related conversation is no longer running. Either the conversation has been aborted or the message is also late in the sense of 2.2.a or 2.2.b. The CSB can detect these cases by checking the status of the conversation that is defined by the related correlation tokens.

Exception Class 2.3: Dupes

Messages with exactly the same content usually only occur in exception scenarios. Consider the airline sending two e-tickets to the customer. To distinguish this exception from a late interaction (class 2.2) the CSB needs access to a history of the messages it routes. These can either be logged with the original context or alternatively a hashsum can be used.

4.5 Exception Handling Strategies

In the following we discuss how the exceptions that are described in the previous section can be handled.

The main rule is to not route faulty messages to the receiver. The message either has to be hold back for an enforcement of the choreography (see below) or discarded which means it is put on a dead letter channel [HW03].

A fault should usually lead to the termination of a conversation unless it can be enforced or fault handling is modeled in the choreography descriptions as it is possible with BPEL^{gold}. If a conversation is terminated this has to be propagated to all participants as discussed in Section 4.5.2.

Enforcement is only an option if it does not interfere with legal requirements such as the time constraint of the investment scenario discussed for 2.1.b exceptions.

4.5.1 Enforcement

Enforcement of choreography constraints is applicable to cases where an interaction occurs too early (exception class 2.2). The CSB can keep the message in a queue and wait to deliver it to the receiver until the conversation is in a state that allows the interaction.

It is not feasible to allow enforcement for every interaction without any restrictions. Therefore we propose enforcement related configuration parameters that can be defined either per choreography or for the CSB in general. There should be a step and a time limit. The former means that a message should not be hold back longer than x steps in the control flow. Another option is to allow enforcement only for those constraints that are locally unenforceable.

Every act of enforcement must be protocoled somewhere, for example in log files. The gathered information can then be used to detect participants that regularly violate the choreography. Permanent enforcement is only acceptable for locally unenforceable choreographies.

4.5.2 Fault Propagation

When a fault occurs, enforcement is not applicable 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 CAPs is achieved using the respective operation introduced in Section 4.3.2. For “regular” participants we have to distinguish: fault messages can be send; erroneous messages can be send and future interactions can be blocked.

In case the participant is expecting a message and the respective operation supports fault messages, one of them can be send by the bus. If no fault message is expected, the bus can send a message of the expected type and fill all parameters with erroneous values (such as empty strings or 0) which will lead to a fault in the participants process. We assume that a participant can handle the message even if its local process is not yet ready to receive it.

Participants that are only involved by sending messages cannot be informed until they initiate the next interaction. This is then rejected by the CSB.

4.6 Summary

In this chapter we motivated the need for a Choreography-aware Service Bus (CSB). We presented possible fault scenarios and how a CSB can handle them. Challenges such as correlation were discussed and the Choreography-aware Participant (CAP) was introduced as an optional enhancement of the concept.

The following assumptions were made:

1. The choreography-related functionality of the bus is transparent for the participant.
2. The communication between the CSB and the participants must be reliable.
3. Messages cannot overtake each other within the CSB.
4. There can be non-choreography related messages on the bus.

To make the CSB a visible part of a choreography we need a choreography language that allows modeling of CSB actions. Such a language is introduced in the next chapter.

A prototypical implementation of a CSB is presented in Chapter 6.

5 BPEL^{gold}

The previous chapters presented relevant terms and concepts in the field of choreography languages as well as examples of existing languages. We also motivated the need for a Choreography-aware Service Bus. This chapter introduces an implementation-specific language that can be used as the input format for a Choreographie-aware Service Bus: BPEL^{gold}— “gold” as in **glo**ba**l** **d**efinition. The language is an extension to BPEL that allows modeling choreographies using the interaction modeling style.

First, the requirements for the language are specified (Section 5.1), then the developed solution is presented (Sections 5.2 to 5.5) and finally BPEL^{gold} is evaluated against the requirements (Section 5.6).

5.1 Requirements

This section presents the basic ideas and requirements that influenced the decisions made during the design of BPEL^{gold}. They are also used to evaluate the result as presented in Section 5.6.

As choreographies described with BPEL^{gold} should be used by the Choreography-aware Service Bus, the language must be machine-readable with a clear semantic.

The service interaction patterns (introduced in Section 2.1.2) are a good benchmark for choreography languages. The language should support easy modeling of the patterns. Some refined requirements that can be extracted from the patterns are presented later in this section.

Let’s Dance (see Section 2.1.3) supports all of the patterns and is used to check the control flow abilities of BPEL^{gold}. Thus, one of the goals for the work at hand is to provide a mapping from Let’s Dance to BPEL^{gold}.

[DKLW08] defines ten requirements that a choreography language should fulfill. Those were extracted from the results of an analysis of the service interaction patterns. The following list provides a summary of the requirements:

R1. Multi-lateral interactions. Choreographies typically consist of interactions among more than two participants.

R2. Participant topology. The choreography must specify the roles that are involved and how participants are interlinked. The service topology provides an essential structural view on the choreography from a global perspective.

R3. Participant sets. A choreography language must support a (potentially unknown) number of participants of the same role (e.g. the airlines in the travel agency example).

R4. Selection of participants and reference passing. Selection of participants can be done at design-, deployment- or at runtime. In any case participants must probably be made aware of the selection (e.g. passing the "successful bidder" to the seller in an auction).

R5. Message formats. The definition of message formats has to be supported natively so that it can be made sure the participants can process the messages they receive.

R6. Interchangeability of technical configurations. To enhance the reusability of choreographies, the technical configuration, such as the port types or operations used, should be easily exchangeable.

R7. Time constraints. Time is an important aspect of choreographies. It must be possible to define in which time period certain messages are allowed.

R8. Exception handling. Exceptions can occur because of technical or any other reason. There should be a way to model the handling of these situation, like cancellation messages or the termination of a conversation.

R9. Correlation. As it is likely that a participant is involved in more than one conversation at a time, a correlation mechanism must be provided to distinguish between different conversations.

R10. Integration with service orchestration languages. Choreography languages must allow an integration with orchestration languages such as BPEL. This means it should be possible to easily generate process definitions out of choreographies and also to extract choreographies from existing interacting processes.

The assessment of current choreography languages in [DKLW08] has shown that none of them—besides BPEL4Chor— covers all of the ten requirements. WS-CDL for example does not support defining an unknown number of participants of the same role (R3). It also does not support (R6) due to its tight coupling to WSDL.

5.2 Overview

Every BPEL^{gold} choreography consists of the following artifacts that are explained in the following sections:

- A *participant topology* describing the involved participants and the messages flowing between them.
- A *process interaction description* based on a new abstract process profile for BPEL.
- A *grounding* as the process interaction description is decoupled from the technical configuration (such as WSDL port types).

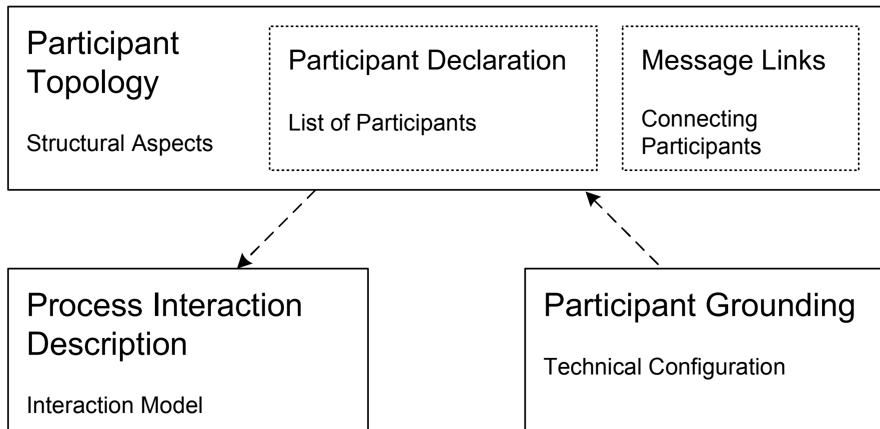


Figure 5.1: BPEL^{gold} Artifacts

The relationship between the three artifacts is also depicted in Figure 5.1. The participants and message links defined in the participant topology are referenced by the process interaction description. The grounding adds information about the technical configuration.

The standard methodology to create a BPEL^{gold} choreography is to first create the participant topology. Then the process interaction description is modeled using the message links defined in the topology. Finally, technical details are added through the grounding.

The partition of a choreography description into these artifacts is inspired by BPEL4Chor which is introduced in Section 2.2.2. The decision to reuse artifacts of BPEL4Chor was made to not reinvent the wheel and also to support a possible future integrated approach using BPEL4Chor and BPEL^{gold}. The more concepts both languages share, the easier it is to develop tools that allow conversion between the two formats. When modeling top-down, one possible use case for this approach is to first model the choreography using BPEL^{gold} and then automatically generate participant behavior descriptions for BPEL4Chor. From there executable BPEL files can be generated. This process is discussed in Section 5.6.3.

5.3 Participant Topology

The participant topology provides a view on the participants of the choreography and how they are connected. The constructs are the same as for BPEL4Chor. We also have adopted the XML schema definition (see [B4C]) without any changes. Setting optional attributes that don't give an extra value for BPEL^{gold} (like `sendActivity`) can be helpful when generating BPEL4Chor models from BPEL^{gold}.

Listing 5.1 shows the participant topology for the travel agency scenario (introduced in Section 2.1.4).

Each topology description defines *participant types*, *participants* and *message links*. A participant type consists of a unique name, a reference to its process interaction description and an optional reference to the language of the behavior description. Here, the URL http://www.bpel4chor.org/BPELgold/ must be used for BPEL^{gold} participant topologies as the default are BPEL4Chor participant behavior descriptions.

Participants are instances of participant types and can be described using either the *participant* (also referred to as *participant reference*) or the *participant set* construct. When using a participant set the actual number of instances is not defined during design-time. Participant references can be contained within a participant set to define special instances like the selected airline in the example topology (Listing 5.1, Line 15).

In the example, there is a participant type for the traveler, the agency and the airline. Given the participants definition, there is at most one traveler and at most one agency taking part in any conversation, but there can be an arbitrary number of airline instances. The traveler *selects* the agency when starting a message exchange whereas the airlines are selected by the agency.

The attribute `forEach` (Listing 5.1, Line 13) on the set of airlines references the `<forEach>` of the process interaction description that should iterate over this set. The `forEach` attribute (Listing 5.1, Line 14) on the contained participant "currentAirline" defines that the current value of the iterator should be stored in this reference.

The "paths" for a potential interaction between participants are described using *message links*. The *sender* and *receiver* of a message link refer to participants which must be listed in the topology. When using *senders* instead of *sender* a list of possible senders can be defined. This is useful to support interactions that can be initiated by different participant types. Examples are given in the models for the service interaction patterns (Section 5.6.2). Defining a *name* for message links is mandatory in BPEL^{gold} participant topologies because they are used to relate it to an interaction (see next section).

The attribute `participantRefs` is used to support reference passing. For example on the message link `agencyTicketOrderAirline` (Listing 5.1, Line 25), a reference to the traveler is passed from the agency to the airline. Another attribute that is related to reference passing is `bindSenderTo`. This forces the sender to add a reference to itself to the message and may be used to indicate that the sender is stored in a different participant reference.

5.4 Process Interaction Description

The Process Interaction Description (PID) is the main part of a BPEL^{gold} choreography. It is based on abstract BPEL processes and two extension activities which are described in the following. BPEL's extension mechanisms are described in Section 2.2.1.

The namespace for the BPEL^{gold} extension is http://www.bpel4chor.org/BPELgold/ and the prefix *gld* is used throughout this document. The extension must be declared as mandatory (`mustUnderstand="true"`).

Listing 5.1 Example Participant Topology

```
1  <topology name="booking_participanttopology" ... >
2        <participantTypes>
3              <participantType name="Traveler"
4                    participantBehaviorDescription="chor:booking_choreography"
5                    processLanguage="http://www.bpel4chor.org/BPELgold/" />
6              <participantType name="Agency" ... />
7              <participantType name="Airline" ... />
8        </participantTypes>
9        <participants>
10             <participant name="traveler" type="Traveler" selects="agency" />
11             <participant name="agency" type="Agency" selects="airlines" />
12             <participantSet name="airlines" type="Airline"
13                   forEach="chor:fe_RequestPrice">
14                   <participant name="currentAirline" forEach="chor:fe_RequestPrice" />
15                   <participant name="selectedAirline" />
16             </participantSet>
17       </participants>
18       <messageLinks>
19             <messageLink name="travelerTripOrderAgency" sender="traveler"
20                   receiver="agency" messageName="tripOrder" />
21             <messageLink name="agencyPriceRequestAirline" sender="agency"
                 receiver="currentAirline"
22                   messageName="priceRequest" />
23             <messageLink name="airlinePriceAgency" sender="currentAirline"
24                   receiver="agency" messageName="price" />
25             <messageLink name="agencyTicketOrderAirline" sender="agency"
26                   receiver="selectedAirline" messageName="ticketOrder"
                         participantRefs="traveler" />
27             <messageLink name="airlineConfirmationAgency" sender="selectedAirline"
28                   receiver="agency" messageName="confirmation" />
29             <messageLink name="agencyConfirmationTraveler" sender="agency"
30                   receiver="traveler" messageName="confirmation" />
31             <messageLink name="airlineEticketTraveler" sender="selectedAirline"
32                   receiver="traveler" messageName="eTicket" />
33             [...]
34       </messageLinks>
35 </topology>
```

We use the notion of the "global observer" to refer to the process—or "participant"—that is modeled by a PID.

5.4.1 The Interaction Activity

The `<interaction>` activity (see Listing 5.2) is the main building block of a BPEL^{gold} choreography. It represents a message exchange between two participants.

The standard attributes (`name`, `suppressJoinFailure`) and elements (`targets`, `sources`) are those of BPEL.

Listing 5.2 Syntax specification of the BPEL^{gold} `<interaction>` element

```
<gld:interaction wsu:id="xs:id"? variable="BPELVariableName"? messageExchange="NCName"?
       partnerLink="NCName"? portType="QName"? operation="NCName"? createInstance="yes|no"?
       standard-attributes>
       standard-elements
       <correlations>?
              <correlation set="NCName" gld:senderInitiate="yes|join|no"?
                  gld:receiverInitiate="yes|join|no"? />+
       </correlations>
</gld:interaction>
```

The attribute `wsu:id` is used to reference a message link in the related participant topology. A message link must be defined for every interaction that is used in the PID. (The namespace wsu (http://schemas.xmlsoap.org/ws/2003/06/utility/) contains definitions for utility data structures.) `wsu:id` defaults to the `name`, so at least one of them must be specified.

In the examples in this work we use the following naming convention for the id: *senderMessagenameReceiver*. This helps to understand the PID without having the participant topology at hand.

The optional attribute `variable` refers to a BPEL variable that contains the message data after the interaction is completed. Defining the message variable is also a way to specify the message format for the interactions. The format can also be implicitly be specified via the grounding (see Section 5.5).

The `messageExchange` attribute can be used to pair two interactions for a two-way operation. This is analogues to receive-reply pairs in BPEL, but always required for these cases because it is the only way to define this relationship between two interactions as no operation and partner link is specified.

`partnerLink`, `portType` and `operation` are analogue to BPEL but should not be filled when modeling the PID. They are set when it is transformed to an executable process using the data from the grounding (Section 5.5).

The `createInstance` also has the same semantic as in BPEL: when set to “yes”, the occurrence of the interaction causes the creation of a new process instance if it does not already exist. The default is “no” to overwrite the BPEL behavior that all `<extensionActivity>` activities can cause an instantiation.

The `correlations` element is discussed in Section 5.4.4.

5.4.2 The onInteraction Activity

The `onInteraction` element (see Listing 5.3) is a replacement for BPEL’s onMessage and onEvent that can be used within `pick` and `eventHandlers`. The occurrence of the specified interaction triggers the execution of the associated child activity.

Listing 5.3 Syntax specification of the BPEL^{gold} `<onInteraction>` element

```
<gld:onInteraction wsu:id="xs:id" variable="BPELVariableName"? messageExchange="NCName"?
       partnerLink="NCName"? portType="QName"? operation="NCName"?>
       <correlations>?
              <correlation set="NCName" gld:senderInitiate="yes|join|no"?
                  gld:receiverInitiate="yes|join|no"? />+
       </correlations>
       activity
</gld:onInteraction>
```

5.4.3 Abstract Process Profiles for BPEL^{gold}

We propose two new Abstract Process Profiles for BPEL that allow modeling interaction models: the *Abstract Process Profile for Basic Interaction Models* and the *Abstract Process Profile for Extended Interaction Models*. The basic profile provides a true global view on only the interactions between participants and the extended profile also allows additional activities that are executed by the global observer.

Abstract Process Profile for Basic Interaction Models

This profile is to be used when plain interaction models should be described. It is based on the Abstract Process Common Base. None of the standard communication activities may be used in this profile. Thus only the interaction between participants can be modeled and the modeled process remains a passive observer.

Subset of the Processes Allowed in the Common Base The following restrictions to the Abstract Process Common Base apply:

- Expressions: The use of opaque expressions is not allowed.
- Activities: The use of opaque activities is not allowed. None of the standard communication activities may be used. The `<interaction>` activity is allowed to be used instead. The `<onInteraction>` element may be used as a replacement for `<onEvent>` and `<onMessage>`.

The attribute `gld:participant` can be added to `<if>`, `<while>`, `<repeatUntil>` and `<foreach>`-activities to indicate the participant that evaluates the condition. If the PID is to be used during execution, the `<condition>` of these activities may only be based on information that is visible to the global observer such as the content of previously exchanged messages.

In every executable completion, `partnerLink`, `portType` and `operation` attributes must be added to `<interaction>` activities using the grounding information. Further additions are not allowed.

The URL for this profile is:
http://www.bpel4chor.org/BPELgold/AbstractProfileForBasicInteractionModels

Abstract Process Profile for Extended Interaction Models

Similar to the basic profile, the Abstract Process Profile for Extended Interaction Models allows modeling the interactions between participants. Furthermore, it allows the observing process to actively participate by receiving and sending messages. This can be useful when explicitly modeling for a Choreography-aware Service Bus (see Chapter 4). For example, messages sent from the global observer to a participant in case of faults could be modeled. Another use case is described in the mapping of Let's Dance's inhibits relationship to BPEL^{gold} in Section 5.6.

This profile is based on the Abstract Process Profile for Basic Interaction Models and removes some of the restrictions: Standard communications activities may be used to model interactions between the global observer and other participants. The use of opaque activities is allowed but requires the attribute `gld:participant` to be set. They can be used to model internal behavior of participants (see Section 5.4.6).

The use of the attributes partnerLink, portType, and operation on communication activities is not allowed. Communication activities and onMessage branches require the new attribute `wsu:id` so that they can be referenced in the participant topology. `wsu:id` defaults to the `name` attribute.

The global observer must also be defined in the participant topology as a participant of the participant type *GOLDobserver*, which also must be added. This participant must not be used in message links that are used by `<interaction>` activities, only in those for standard communication activities.

In every executable completion, `partnerLink`, `portType` and `operation` attributes must be added to communication activities using the grounding information. It is also allowed to add new activities for the global observer. Opaque activities must be removed and must not be replaced by other activities.

The URL for this profile is:
http://www.bpel4chor.org/BPELgold/AbstractProfileForExtendedInteractionModels

5.4.4 Correlation

Correlation is required to relate a message to a specific conversation. The need for correlation in choreography descriptions is discussed in Section 4.2.2.

BPEL uses *correlation sets* to relate messages to a specific process instance. A correlation set is a set of properties that must be part of every message that is sent in the set's context. These properties can then be used to correlate a message to a specific process instance. Correlation sets can be declared on process or scope level. Once a correlation set has been initiated, its

values may no longer be changed. This concept is reused in BPEL^{gold} to correlate messages to conversations.

A `<correlationSet>` is used to declare correlation sets and a `<correlation>` to assign it to a communication activity. When used for `<interaction>` activities and `<onInteraction>` activities, `senderInitiate` and `receiverInitiate` replace the `initiate` attribute, see Listing 5.2 and Listing 5.3.

The semantic of the values is analogue to that in BPEL: If the value of the `senderInitiate` attribute is set to "yes", the correlation set must be initiated at the senders side and may not have been initiated before. The value "join" means it must be initiated if it has not been initiated before. And when the value is set to "no", the set must not be initiated. The same applies to the receiver's side for `receiverInitiate`.

As soon as a correlation set has been initialized, the values of the properties must be the same for all interactions that carry this set. From the global observer's point of view only the first initialization is important. The differentiation between sender and receiver is needed when generating participant behavior descriptions. How this information is used in our implementation is described in Section 6.2.

In BPEL, the properties of a correlation set refer to properties in WSDL definitions via their QName. In order to allow untyped properties, in BPEL^{gold} the properties are interpreted as names without reference and the typing is done in the grounding (see Section 5.5—analogue to BPEL4Chor).

5.4.5 Exception Handling

In order to model the behavior in exceptional situations, special constructs are needed. For BPEL^{gold} we can reuse those that are already provided by BPEL. This means that all constructs that are related to *fault*, *compensation* and *termination* handling can be used in BPEL^{gold} without restrictions.

We define two new standard faults that can be caused by `<interaction>` and `<onInteration>`: *gld:interactionCompletionFault* and *gld:interactionInitiationFault*. The former is raised when the message is sent by the sender and cannot be delivered or is not accepted by the receiver. The later occurs when an active interaction is not even initiated by the sender. How this situation is detected depends on the implementation. See Section 6.2 on how this is handled in our implementation.

When using the profile for basic interaction models, a (fault) handler can only be used to model the interactions that the global observer monitors. It cannot take actively part in the process. The use cases for extended interaction models are more comprehensive. In principle, they are those discussed in Section 4.5. For example, the global observer can send messages to inform participants about the error.

Listing 5.4 BPEL^{gold}: Syntax for Grounding Files

```
<grounding topology="QName">
       <messageLinks>
              <messageLink name="NCName" portType="QName" operation="NCName" />*
       </messageLinks>
       <participantRefs>
              <participantRef name="NCName" WSDLproperty="QName" />*
       </participantRefs>
       <properties>
              <property name="NCName" WSDLproperty="QName" />*
       </properties>
</grounding>
```

5.4.6 Modeling Internal Behavior

Internal behavior relates to activities of participants that are not observable from the outside, that means they are not communication activities. Although choreography languages in general target the observable behavior of participants, in some cases it might be helpful to model local activities. The WS-CDL specification [KBR+05] gives an example, where the fact, that the inventory level affects the decision of a "Buyer" should not be observable by other participants but should be specified in the choreography.

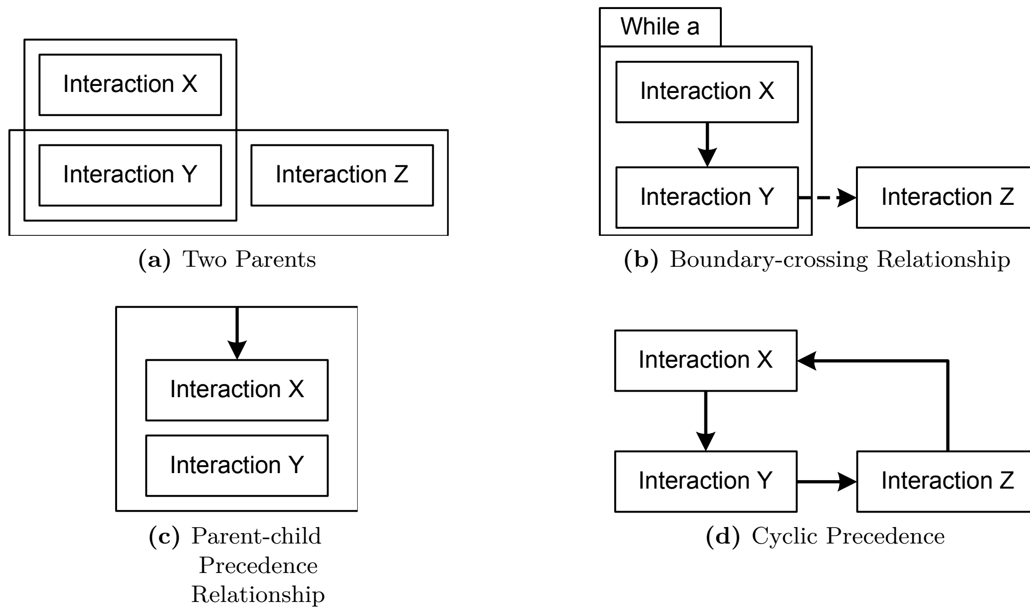
The question if a choreography language should offer constructs to model those internal activities remains open and depends on the use case [KL09]. The evaluation of prominent choreography languages in [KL09] shows that modeling internal behavior is more common with interconnected models. This seems reasonable, as modeling local behavior from a global view requires a change of perspective from the modeler when switching between interactions and internal actions.

BPEL^{gold} supports modeling internal behavior through the use of opaque activities when using the Abstract Process Profile for Extended Interaction Models. The attribute `gld:participant` must be specified and refer to a participant in the participant topology. This information is meant for documentation purposes and not used during execution of a process interaction description.

5.5 Grounding

The artifacts presented so far—participant topology and PID—are free of technical configuration details. This means that no port types, operations and XML Schema types for messages have to be defined. A grounding is used to bring in this information. Again, we adopt the syntax (Listing 5.4) and XSD ([B4C]) from BPEL4Chor.

In the grounding every message link of a topology must be mapped to a WSDL operation. Participant references are mapped to service references via WSDL properties. The same applies to the untyped properties of correlation sets (see Section 5.4.4).



(a) Two Parents

(b) Boundary-crossing Relationship

(c) Parent-child Precedence Relationship

(d) Cyclic Precedence

Figure 5.2: Let's Dance Mapping: Forbidden Models

5.6 Evaluation

This section evaluates the solution presented in the previous sections against the requirements listed in Section 5.1.

5.6.1 Mapping Let's Dance to BPEL^{gold}

First we discuss how the Let's Dance constructs can be represented using BPEL^{gold}. These constructs are described in Section 2.1.3.

Let's Dance as introduced in [ZBDH06] allows modeling incorrect choreographies [ZDH^{+}06]. For example, there could be cyclic precedence dependencies. For the mapping to BPEL^{gold} we assume choreographies are "well-formed" by adding some constraints from [ZDH^{+}06]. Examples that violate the constraints are given in Figure 5.2.

- Every interaction can only be part of one composite interaction. (Figure 5.2a)
- No relation that starts inside a repeated (composite) interaction crosses the boundary of this interaction. (Figure 5.2b)
- There are no precedence dependencies between interactions and their sub-interactions. (Figure 5.2c)
- There are no cyclic precedence dependencies. (Figure 5.2d)

Listing 5.5 Mapping Let's Dance to BPEL^{gold}: `precedes`

```
<flow>
	<links>
		<link name="precedesLink" />
	</links>
	<extensionActivity>
		<gld:interaction name="travelerTripOrderAgency">
			<sources>
				<source linkName="precedesLink" />
			</sources>
		</gld:interaction>
	</extensionActivity>
	<extensionActivity>
		<gld:interaction name="agencyETicketTraveler">
			<targets>
				<target linkName="precedesLink" />
			</targets>
		</gld:interaction>
	</extensionActivity>
</flow>
```

Let's Dance interactions are basically equivalent to BPEL^{gold} `<interaction>` activities and thus can be directly mapped. To model composite interactions in BPEL^{gold}, `<flow>` activities can be used in general.

An `<if>` activity can be used to model a Let's Dance guard. Repeat, While and Foreach are analog to `<repeatUntil>`, `<while>` and `<foreach>`.

BPEL's counterpart to the timer construct is an `<onAlarm>` event handler.

Mapping Let's Dance's relationship types requires a detailed investigation and is discussed in the following.

precedes

The precedes relationship can be mapped to BPEL^{gold} using a `<flow>` activity with a link from the source to the target activity. In the example in Listing 5.5, the agency may not send an eTicket to the traveler before the traveler has placed his order.

weak precedes

To map a weak precedes relationship, a BPEL link in conjunction with Dead Path Elimination (DPE) (see Section 2.2) can be used. As the joinCondition of the target interaction cannot be evaluated before the status of all incoming links is set, it will not be executed before the source interaction has either been finished or skipped. In the latter case, DPE will set the status of the outgoing link to false. Therefore the joinCondition of the target activity should be

Listing 5.6 Mapping Let's Dance to BPEL^{gold}: `weak precedes`

```
<flow suppressJoinFailure="yes">
	<links>
		<link name="weakPrecedesLink" />
	</links>
	<if>
		<condition>[...]</condition>
		<extensionActivity>
			<gld:interaction name="travelerTripOrderAgency">
				<sources>
					<source linkName="weakPrecedesLink" />
				</sources>
			</gld:interaction>
		</extensionActivity>
	</if>
	<extensionActivity>
		<gld:interaction name="agencyETicketTraveler">
			<targets>
				<joinCondition>true()</joinCondition>
				<target linkName="weakPrecedesLink" />
			</targets>
		</gld:interaction>
	</extensionActivity>
</flow>
```

evaluated independently of the (weak precedes) link status. The example given in Listing 5.6, suggests that the agency could send an eTicket even if the traveler had not sent an order.

inhibits

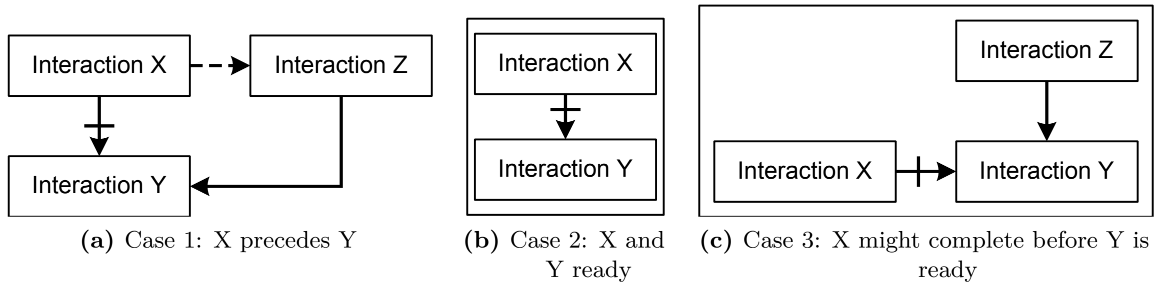
Assume interaction X inhibits interaction Y. The occurrence of X does not only prevent Y from starting, it also forced Y to stop its execution if it had already been started. The latter can be ignored in cases where there is also a precedence relationship between X and Y. This precedence could also be transitive or implicit, for example via their parent interactions.

Thus, we propose mappings for different cases:

1. Y is never ready to start before X has been finished, that means X precedes Y. (Y preceding X would obviously make the inhibit relationship superfluous.)
2. X and Y are ready to start at the same time.
3. X and Y can be ready to start at the same time, but X might also complete before Y is ready to start.

Examples for these cases are depicted in Figure 5.3.

For the first case, cancellation of activities has not be considered. To prevent Y from being started if X was executed, we can use a link from X to Y. The join condition must be defined so that it evaluates to false if X was executed.



(a) Case 1: X precedes Y **(b)** Case 2: X and Y ready **(c)** Case 3: X might complete before Y is ready

Figure 5.3: Let's Dance Mapping: Different Cases for Inhibits

Listing 5.7 sketches the first idea that came up to support cancellation of activities as required for the second case. A `<while>` activity that is active parallel to Y throws a fault as soon as the value of a special variable is "true". That fault is then silently caught by a fault handler that is attached to the scope that encloses Y. Incoming and outgoing links of X and Y must be moved to the enclosing `<sequence>` activities. The problem with this solution is that most BPEL engines would not be able to efficiently handle the while loop ("busy waiting").

Another idea was to attach an event handler to Y's scope that throws a fault as soon as X occurs (using onInteraction). This fault could then be silently caught. Assuming the `<interaction>` activity and the `<onInteraction>` branch should be mapped to existing BPEL activities, this would involve identical `<receive>` activities for both. Thus, this approach leads to a situation that is similar to the BPEL standard fault `conflictingReceive`, which is thrown "when more than one inbound message activity is enabled simultaneously for the same partner link, port type, operation and correlation set(s)" [AAA+07].

The final solution that can be universally applied for the second case requires the use of the Abstract Process Profile for Extended Interaction Models and is presented in Listing 5.8. After X has occurred, the process sends an "inhibit message" to itself. This message then triggers a fault in the `onEvent` branch of an event handler that is attached to a scope containing Y. The fault is caught and ignored in the fault handler. A possible fault that occurs when the "inhibit message" is sent after Y has completed should also be caught, but is omitted from the example. Incoming and outgoing links of X must be moved to the enclosing `<sequence>` activity. Message links from *GOLDobserver* to *GOLDobserver* for the `<invoke>` activity and `<receive>` activity must be defined in the participant topology.

This approach is also the base for the solution for the third case. There we also have to make sure that Y does not get ready to start if X has already finished before Y is activated and its event handlers are installed. Therefore we declare a Boolean variable z that is visible to both activities and initialized to "true". We protect Y with an `<if>` activity that evaluates the value of z. If X is executed, z will be set to "false" afterwards. All incoming and outgoing links must be moved from Y to the enclosing `<if>` activity and from X to the enclosing `<sequence>` activity. An example is given in Listing 5.9.

Listing 5.7 Mapping Let's Dance to BPEL^{gold}: `inhibits` - busy waiting

```
<flow>
	<sequence>
		<extensionActivity>
			<gld:interaction name="X" />
		</extensionActivity>
		<assign>
			<copy><from>true()</from><to variable="inhibit" /></copy>
		</assign>
	</sequence>
	<scope>
		<faultHandlers>
			<catch faultName="inhibited"><empty /></catch>
		</faultHandlers>
		<flow>
			<sequence>
				<extensionActivity>
					<gld:interaction name="Y" />
				</extensionActivity>
				<assign>
					<copy><from>false()</from><to variable="while" /></copy>
				</assign>
			</sequence>
			<while>
				<condition>$while = true()</condition>
				<if>
					<condition>$inhibit = true()</condition>
					<throw faultName="inhibited" />
				</if>
			</while>
		</flow>
	</scope>
</flow>
```

two-way inhibit

In general, a two-way inhibit relationship can be modeled using the proposed mapping for inhibit relationships for both directions like in the example in Listing 5.10. For simple cases, when the two-way inhibit does not cross any composite interaction boundaries, an additional solution is the usage of a `<pick>` activity. In the example in Listing 5.11, either the traveler sends its order to the agency or the agency informs the traveler of a price change.

5.6.2 Service Interaction Patterns with BPEL^{gold}

This section describes how the service interaction patterns can be modeled using BPEL^{gold}. The patterns are documented in detail in [BDH05b].

Given the mapping from Let's Dance to BPEL^{gold} and the fact that Let's Dance supports all of the patterns [ZBDH06], we could actually state that BPEL^{gold} is also capable to express

Listing 5.8 Mapping Let's Dance to BPEL^{gold}: `inhibits` - Case 2

```
<flow>
        <sequence>
                <extensionActivity>
                        <gld:interaction name="X" />
                </extensionActivity>
                <invoke name="sendInhibit" />
        </sequence>
        <scope>
                <faultHandlers>
                        <catch faultName="inhibited"><empty /></catch>
                </faultHandlers>
                <eventHandlers>
                        <onEvent wsu:id="receiveInhibit">
                                <scope><throw faultName="inhibited" /></scope>
                        </onEvent>
                </eventHandlers>
                <extensionActivity>
                        <gld:interaction name="Y" />
                </extensionActivity>
        </scope>
</flow>
```

Listing 5.9 Mapping Let's Dance to BPEL^{gold}: `inhibits` - Case 3

```
<flow>
        <sequence>
                <extensionActivity>
                        <gld:interaction name="X" />
                </extensionActivity>
                <assign>
                        <copy><from>false()</from><to variable="z" /></copy>
                </assign>
                <invoke name="sendInhibit" />
        </sequence>
        <sequence>
                <extensionActivity>
                        <gld:interaction name="Z" />
                </extensionActivity>
                <scope>
                        <faultHandlers>
                                <catch faultName="inhibited"><empty /></catch>
                        </faultHandlers>
                        <eventHandlers>
                                <onEvent wsu:id="receiveInhibit">
                                        <scope><throw faultName="inhibited" /></scope>
                                </onEvent>
                        </eventHandlers>
                        <if><condition>$z</condition><extensionActivity>
                                <gld:interaction name="Y" />
                        </extensionActivity></if>
                </scope>
        </sequence>
</flow>
```

Listing 5.10 Mapping Let's Dance to BPEL^{gold}: `two-way-inhibit`

```
<flow>
	<sequence>
		<extensionActivity>
			<gld:interaction name="O" />
		</extensionActivity>
		<scope>
			<faultHandlers>
				<catch faultName="inhibited"><empty /></catch>
			</faultHandlers>
			<eventHandlers>
				<onEvent wsu:id="receiveInhibitX">
					<scope><throw faultName="inhibited" /></scope>
				</onEvent>
			</eventHandlers>
			<sequence>
				<extensionActivity>
					<gld:interaction name="X" />
				</extensionActivity>
				<invoke name="sendInhibitY" />
			</sequence>
		</scope>
	</sequence>
	<sequence>
		<extensionActivity>
			<gld:interaction name="P" />
		</extensionActivity>
		<scope>
			<faultHandlers>
				<catch faultName="inhibited"><empty /></catch>
			</faultHandlers>
			<eventHandlers>
				<onEvent wsu:id="receiveInhibitY">
					<scope><throw faultName="inhibited" /></scope>
				</onEvent>
			</eventHandlers>
			<sequence>
				<extensionActivity>
					<gld:interaction name="Y" />
				</extensionActivity>
				<invoke name="sendInhibitX" />
			</sequence>
		</scope>
	</sequence>
</flow>
```

Listing 5.11 Mapping Let's Dance to BPEL^{gold}: `two-way inhibit` using a Pick Activity

```
<flow>
       <pick>
              <gld:onInteraction name="travelerTripOrderAgency">
                     <gld:Interaction name="agencyItineraryTraveler" />
              </gld:onInteraction>
              <gld:onInteraction name="agencyTripPriceChangeTraveler">
                     <gld:Interaction name="travelerCancelTripAgency" />
              </gld:onInteraction>
       </pick>
</flow>
```

all of the patterns. But, as Let's Dance is an implementation-independent and BPEL^{gold} an implementation-specific language problems can arise when dealing with implementation-specific details of the patterns. For example, this is the case with the atomic multicast notification pattern (see below).

Single-transmission Bilateral Interaction Patterns

Send and *receive* are supported by using the `<interaction>` activity (or alternatively `<onInteraction>`). Actually the `<interaction>` activity comprises a send and receive at the same time - but for different participants. *Send/receive* can be modeled using two `<interaction>` activities. One for a message exchange from participant X to participant Y and one for Y to X. Of course, there must be corresponding message links in the participant topology (which holds true for all interactions mentioned in this evaluation).

Single-transmission Multilateral Interaction Patterns

Racing incoming messages: One participant expects to receive one specific message among a set of messages which might have different types and can be sent by different roles. BPEL^{gold} supports this pattern through a `<pick>` activity containing arbitrary `<onInteraction>` elements. A participant topology for this pattern is listed in Listing 5.12 and the PID in Listing 5.13. As the order of the `<onInteraction>` elements within the pick does not imply any ranking it is not possible to model a ranking amongst them.

One-to-many send: One participant sends an arbitrary number of messages of the same type to several receivers. This can be modeled using a participant set together with a `<forEach>` activity that contains the interaction. See Listing 5.14 for an example participant topology and Listing 5.15 for the PID.

One-from-many receive: One participant receives a number of logically related messages that arise from autonomous events occurring at different participants. The arrival of messages needs to be timely so that they can be correlated as a single logical request. The solution for this pattern is shown in Listing 5.16 and Listing 5.17. As the participant reference `y` is limited to the `<scope>` within the `<while>`, it is bound to the actual sender in each iteration.

Listing 5.12 Racing Incoming Messages: Topology

```
<participants>
       <participant name="x" type="receiver" />
       <participantSet name="senders" type="sender">
              <participant name="y" />
       </participantSet>
       <participantSet name="otherSenders" type=otSender">
              <participant name="z" />
       </participantSet>
</participants>
<messageLinks>
       <messageLink name="sendersMsgTypeAX" senders="senders" receiver="x" bindSenderTo="y"
           messageName="msgTypeA" />
       <messageLink name="sendersMsgTypeBX" senders="senders" receiver="x" bindSenderTo="y"
           messageName="msgTypeB" />
       <messageLink name="othersendersMsgTypeBX" senders="otherSenders" receiver="x"
           bindSenderTo="z" messageName="msgTypeB" />
</messageLinks>
```

Listing 5.13 Racing Incoming Messages: Process Interaction Description

```
<pick>
       <gld:onInteraction name="sendersMsgTypeAX">activity</gld:onInteraction>
       <gld:onInteraction name="sendersMsgTypeBX">activity</gld:onInteraction>
       <gld:onInteraction name="othersendersMsgTypeBX">activity</gld:onInteraction>
</pick>
```

Listing 5.14 One-to-many Send: Topology

```
<participants>
       <participant name="x" type="sender" selects="receivers" />
       <participantSet name="receivers" type="receiver" forEach="ns:fe">
              <participant name="y" forEach="ns:fe" />
       </participantSet>
</participants>
<messageLinks>
       <messageLink name="xMsgTypeY" sender="x" receiver="y" messageName="msgType" />
</messageLinks>
```

Listing 5.15 One-to-many Send: Process Interaction Description

```
<forEach name="fe" parallel="yes">
       <scope><gld:interaction name="xMsgTypeY" /></scope>
</forEach>
```

Listing 5.16 One-from-many Receive: Topology

```
<participants>
	<participant name="x" type="receiver" />
	<participantSet name="senders" type="sender">
		<participant name="y" scope="ns:scp" />
	</participantSet>
</participants>
<messageLinks>
	<messageLink name="sendersMsgTypeX" senders="senders" receiver="x" bindSenderTo="y"
	    messageName="msgType" />
</messageLinks>
```

Listing 5.17 One-from-many Receive: Process Interaction Description

```
<while><condition>[...]</condition>
	<scope name="scp"><gld:interaction name="sendersMsgTypeX" /></scope>
</while>
```

One-to-many send/receive: One participant sends a request to several other participants. Responses are expected within a given timeframe but do not need to be sent. This pattern is quite similar to the *One-to-many send* pattern and can also be modeled using a participant set and a `<forEach>` activity containing both interactions. The gathering of prices from the airlines in our running example is an application of this pattern (see Appendix B). The timing constraint can be handled using an `<onAlarm>` event.

Multi-transmission Interaction Patterns

Multi-responses: Participant X sends a request to participant Y. Subsequently, X receives an arbitrary number of responses from Y. X does no longer expect any responses after one or more of the following events: (i) X sends a notification to stop; (ii) a relative or absolute deadline has been reached; (iii) X has waited for a defined time interval without receiving a response; (iv) Y sends a message indicating it will no longer send responses. The core of this pattern can be modeled using a `<pick>` activity within a `<while>` activity. The stop condition can be handled as follows: (i) using an interaction from X to Y; (ii) an onAlarm handler on a scope that is enclosing the `<while>` activity; (iii) an onAlarm handler on the `<pick>` activity; (iv) an onInteraction on the `<pick>` activity with a pre-agreed message type. An example for (iv) is given in Listing 5.18 and Listing 5.19.

Contingent requests: Participant X sends a request to participant Y. If X does not receive a response within a given timeframe it sends the request to another participant Z and so on. Late responses can either be accepted or discarded. The example for this pattern is listed in Listing 5.20 and Listing 5.21. The second `<pick>` activity is used to allow late receives. Depending on the concrete scenario, the solution must be adapted to allow more than one late response per iteration.

Listing 5.18 Multi-responses: Topology

```
<participants>
	<participant name="x" type="initiator" />
	<participant name="y" type="responder" />
</participants>
<messageLinks>
	<messageLink name="xInitiateY" sender="x" receiver="y" messageName="initiate" />
	<messageLink name="yResponse1X" sender="y" receiver="x" messageName="response1" />
	<messageLink name="yResponse2X" sender="y" receiver="x" messageName="response2" />
	<messageLink name="yStopX" sender="y" receiver="x" messageName="stop" />
</messageLinks>
```

Listing 5.19 Multi-responses: Process Interaction Description

```
<sequence>
	<gld:interaction name="xInitiateY" />
	<scope>
		<eventHandlers><gld:onInteraction
		    name="yStopX">$while=false</gld:onInteraction></eventHandlers>
		<while><condition>$while</condition>
			<pick>
				<gld:onInteraction name="yResponse1X"><empty
				    /></gld:onInteraction>
				<gld:onInteraction name="yResponse2X"><empty
				    /></gld:onInteraction>
			</pick>
		</while>
	</scope>
</sequence>
```

Listing 5.20 Contingent Requests: Topology

```
<participants>
	<participant name="x" type="requestor" />
	<participantSet name="responders" type="responder" forEach="ns:fe" >
		<participant name="currentResponder" forEach="ns:fe" />
	</participantSet>
	<participant name="y" type="Responder" />
</participants>
<messageLinks>
	<messageLink name="xRequestY" sender="x" receiver="currentResponder"
	    messageName="request" />
	<messageLink name="yResponseX" sender="y" receiver="x" messageName="response" />
	<messageLink name="respondersResponseX" sender="responders" receiver="x"
	    messageName="response" />
</messageLinks>
```

Listing 5.21 Contingent Requests: Process Interaction Description

```
<forEach name="fe"><scope><sequence>
	<gld:interaction name="xRequestY" />
	<flow>
		<pick>
			<gld:onInteraction name="yResponseX" /><empty /></gld:onInteraction>
			<onAlarm><for>1m</for><empty /></onAlarm>
		</pick>
		<pick>
			<gld:onInteraction name="respondersResponseX" /><empty
			    /></gld:onInteraction>
			<onAlarm><for>1m</for><empty /></onAlarm>
		</pick>
	</flow>
</sequence></scope></forEach>
```

Listing 5.22 Request with Referral: Topology

```
<participants>
	<participant name="x" type="X" selects="y p"/>
	<participant name="y" type="Y" />
	<participantSet name="p" type="P" forEach="ns:fe">
		<participant name="pi" forEach="ns:fe" />
	</participantSet>
</participants>
<messageLinks>
	<messageLink name="xMsgType1Y" sender="x" receiver="y" messageName="msgType1"
	    participantRefs="p" />
	<messageLink name="yMsgType2pi" sender="y" receiver="pi" messageName="msgType2" />
</messageLinks>
```

Atomic multicast notification: One participant sends notifications to several other participants such that a certain number of parties are required to accept the notification within a certain timeframe. It must be made sure that all parties receive the notification if one of them receives it. This pattern cannot be fully supported as BPEL does not provide support for transactional atomicity. Please refer to [BDH05b] for further details.

Routing Patterns

Request with referral: Participant X sends a request to participant Y indicating that any follow-up response should be sent to a number of other participants (P1, P2, .., Pn). Reference passing is supported in BPEL^{gold} using the `participantRefs` attribute of a message link in a participant topology (see Listing 5.22). A PID for this pattern is shown in Listing 5.23.

Relayed request: Participant X sends a request to participant Y. Y delegates the request to other participants (P1, ..., Pn). These participants then continue interactions with participant X while Y receives a “view” of the interactions. This “view” can include all messages or only a subset of them. The reference can be passed as proposed above. The “view” can be sent

Listing 5.23 Request with Referral: Process Interaction Description

```
<sequence>
	<gld:interaction name="xMsgType1Y" />
	<forEach name="fe" parallel="yes"><scope>
		<gld:interaction name="yMsgType2pi" />
	</scope></forEach>
</sequence>
```

Listing 5.24 Relayed Request: Topology

```
<participants>
	<participant name="x" type="X" />
	<participant name="y" type="Y" />
	<participant name="p" type="P" />
</participantSet>
</participants>
<messageLinks>
	<messageLink name="xMsgType1Y" sender="x" receiver="y" messageName="msgType1" />
	<messageLink name="yMsgType2P" sender="y" receiver="p" messageName="msgType2"
	    participantRef="x" />
	<messageLink name="pMsgType3X" sender="p" receiver="x" messageName="msgType3" />
	<messageLink name="pMsgType3Y" sender="p" receiver="y" messageName="msgType3" />
</messageLinks>
```

using two `<interaction>` activities with X and Y as recipients within a `<flow>` activity. See Listing 5.24 and Listing 5.25 for an example.

Dynamic routing: A request is required to be routed to several participants based on a routing condition. The routing order is flexible and can be subject to dynamic conditions based on data contained in the original request or obtained in one of the “intermediate steps”. The dynamic Part of this pattern cannot completely be realized with BPEL^{gold}. It is possible to model possible routes with all their alternatives, but this is not as “dynamic” as the pattern requires it to be. An approach to specify SOAP message routing via BPEL processes is presented in [Juc06, SKL09].

Listing 5.25 Relayed Request: Process Interaction Description

```
<sequence>
	<gld:interaction name="xMsgType1Y" />
	<gld:interaction name="yMsgType2P" />
	<flow>
		<gld:interaction name="pMsgType3X" />
		<gld:interaction name="pMsgType3Y" />
	</flow>
</sequence>
```

Service Interaction Pattern	BPEL^{gold}	BPEL4Chor	BPEL	WS-CDL	iBPMN	Let's Dance
1. Send	+	+	+	+	+	+
2. Receive	+	+	+	+	+	+
3. Send/receive	+	+	+	+	+	+
4. Racing incoming messages	+	+	+	+	+	+
5. One-to-many send	+	+	+/-	+/-	+/-	+
6. One-from-many receive	+	+	+	+	+	+
7. One-to-many send/receive	+	+	+/-	+/-	+	+
8. Multi-responses	+	+	+	+	+	+
9. Contingent requests	+	+	+/-	+/-	+/-	+
10. Atomic multicast notification	-	-	-	-	-	+
11. Request with referral	+	+	+	+	+	+
12. Relayed request	+	+	+	+	+	+
13. Dynamic routing	-	-	-	-	-	+

Table 5.1: Service Interaction Patterns: Support in Languages

Overview

Table 5.1 provides an overview on which patterns are supported by BPEL^{gold} and other choreography languages. BPEL is included although it is actually not a choreography language to show where BPEL^{gold} adds functionality. Direct support is depicted by "+", partial support by "+/-" and "-" means that the pattern is not supported. The data for BPEL4Chor, BPEL and WS-CDL is taken from the evaluation in [DKLW08]. The former two lack direct support for patterns 5 and 7 because unknown number of participants can not be modeled. BPEL^{gold} only lacks support for patterns 9 and 13, which are supported by none of the languages, besides Let's Dance. The evaluation of iBPMN is presented in [DB08].

5.6.3 Support of Choreography Language Requirements

This section evaluates if and how BPEL^{gold} supports the general requirements for a choreography language that are listed in Section 5.1. (Ri) is used to refer to Requirement i.

It is obvious that BPEL^{gold} supports modeling interactions among more than two participants (R1).

Participant topologies (R2) are one of the artifacts of BPEL^{gold}. They also allow the definition of participant sets (R3) and participant selection / reference passing (R4).

Message formats can either be specified in the process interaction description using BPEL variables or in the grounding. The concept of groundings enables support for interchangeability of technical configurations (R6).

Time constraints (R7) can be modeled using the constructs provided by BPEL. The correlation mechanism of BPEL is reused to support requirement (R9). The same applies to exception handling (R8).

The integration with an orchestration language (R10) can be realized as depicted in Figure 5.4. (A dashed line indicates that manual refinement is required. The other steps can be performed automatically.) First the choreography is modeled using BPEL^{gold}. Creating a grounding is optional in this step and depends on whether the BPEL^{gold} model is used during execution. A process interaction description can be transformed into BPEL4Chor participant behavior descriptions by replacing `<interactions>` activities with `<invoke>` or `<reply>` activities for the sender and `<receive>` activities for the receiver. `<onInteraction>` activities can be replaced by `<onMessage / onEvent>` activities. If optional attributes, such as `sendActivity`, are omitted in the BPEL^{gold} participant topology this information has to be added for the BPEL4Chor participant topology. The steps from BPEL4Chor to executable BPEL processes have been described in [Rei07].

5.7 Summary

In this chapter we introduced BPEL^{gold} as an interaction modeling-based choreography language. This introduction included the list of requirements, a presentation of the language's constructs and also the evaluation against the requirements.

It has been shown that BPEL^{gold} supports all the requirements for this work and can also be used to model most of the service interaction patterns.

BPEL^{gold} is used as the input format for our implementation of a Choreography-aware Service Bus as presented in Chapter 6.

A full example with the participant topology, process interaction description and grounding for the travel agency example is listed in Appendix B.

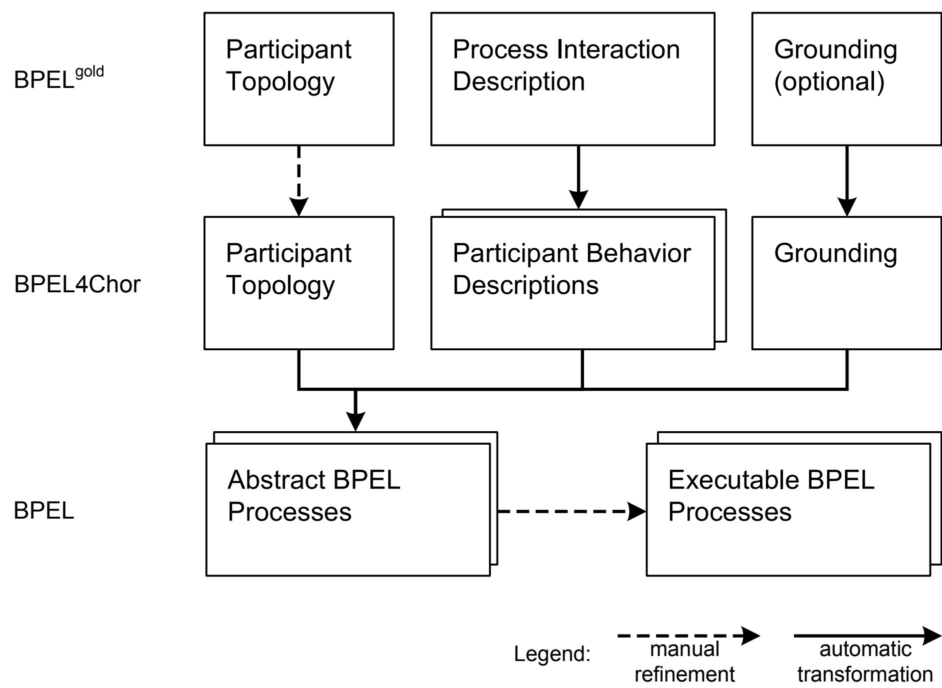


Figure 5.4: From BPEL^{gold} to Executable BPEL Processes

6 CASmix: A Choreography-aware Service Bus

So far we introduced the concept of a Choreography-aware Service Bus (CSB) and BPEL^{gold} as a choreography language. In this chapter we present the prototypical implementation of a CSB that was realized during this work and uses BPEL^{gold} choreography descriptions: *CASmix*. The name is derived from "**C**horeography-**A**ware **S**ervice**mix**".

First, we provide an overview on CASmix' architecture (Section 6.1). Then we present the implementation and how the challenges discussed in Chapter 4 are realized (Section 6.2). An implementation of a Choreography-aware Participant (CAP) using BPEL events is described in Section 6.3.

6.1 Architecture

CASmix' is based on an ESB and a BPEL engine. Basically, BPEL^{gold} choreographies are "orchestrated" by routing all messages through the BPEL engine.

Before introducing CASmix' components and how the message routing is done in detail, we introduce the architecture of the Apache Orchestration Director Engine (ODE) [Apae] and ServiceMix [Apaf]. We chose ServiceMix as the base for our implementation because it is one of the few open source ESBs that support the majority of the features introduced in Section 2.4 and also the integration of ODE.

Actually, the solution is portable with few effort to other ESBs that can be integrated with ODE and allow interception of messages. Our adaption of ODE can be reused.

An alternative to the "orchestration"-approach is to use an automaton that tracks the state of a conversation. This state is checked by the bus before routing the message directly to the recipient. Due to time-constraints, we concentrated on one solution. Further research is required to evaluate the pros and cons of both approaches.

6.1.1 ODE Architecture[1]

An overview on ODE's architecture is given in Figure 6.1. The main components are the *BPEL Compiler*, the *BPEL Runtime* and the *Integration Layer*.

The BPEL Compiler converts the source BPEL artifacts (BPEL process definitions, WSDL and schema files) into a compiled representation. During compilation named references (such as

[1] This section is mainly based on [Apaa], other references are explicitly stated.

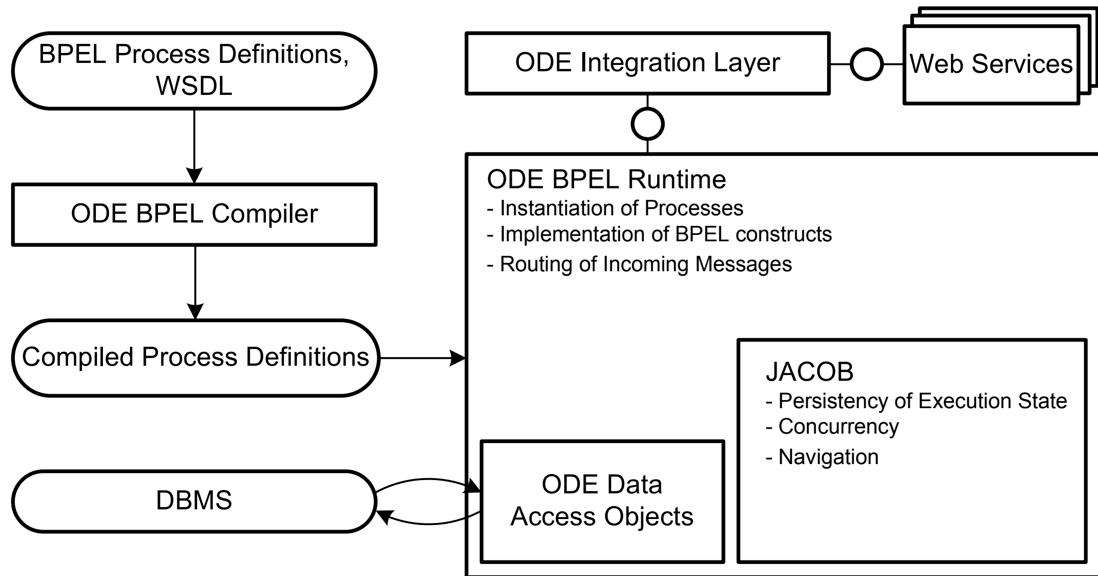


Figure 6.1: ODE Architecture [Apaa]

variables) are resolved, the WSDL and type information is internalized and various constructs such as default compensation handlers are generated. The compiled representation is the only input required by the BPEL Runtime.

The BPEL Runtime is the core of ODE. It provides implementations of the BPEL constructs and also handles the handling of incoming messages. This means it decides when to create a new process instance and to which instance a message is routed.

Data Access Objects (DAOs) are used as an abstraction layer for an underlying datastore. This is usually a relational database but custom DAO implementations can be created. ODE persists the following information to achieve a reliable execution in an unreliable environment: active instances; message routing data; variable and partner link values for each instance and the process execution state.

BPEL constructs are implemented using ODE's Java Concurrent Objects (JACOB) framework that "provides an application-level concurrency mechanism—without using threads—and a transparent mechanism for interrupting execution and persisting execution state" [Apab].

An integration layer is responsible for the communication with the "outside world". There exist implementations for Axis2 [Apac] and Java Business Integration (JBI). We use the JBI integration layer to deploy ODE as a Service Engine (SE) on ServiceMix. The main task of the integration layer is to transform JBI normalized messages into ODE's internal message format and hand it over to the runtime. Lice-cycle management (such as starting and configuring the runtime) is also done by the integration layer.

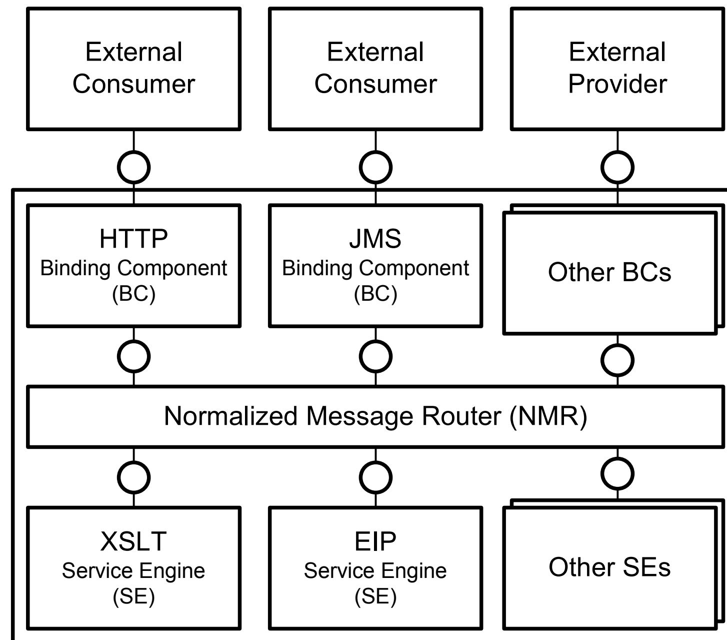


Figure 6.2: ServiceMix Architecture

In this work, we use the current trunk of ODE (which will be released as Version 2.0). The concrete revision number and a list of all applied changes (as unified diff) is available with the source code of our implementation.

6.1.2 ServiceMix Architecture[2]

ServiceMix' implementation is based on the Java Business Integration (JBI) specification JSR 208 [JBI05]. The main concepts described by JBI are *Service Engines (SEs)*, *Binding Components (BCs)* and the *Normalized Message Router (NMR)* as depicted in Figure 6.2.

An SE provides functionality that can be used by other SEs (*consumers*) and can also consume services provided by other components (*providers*). SEs that come with ServiceMix include XLST transformation, a routing component that supports the enterprise integration patterns [HW03], a rules engine and others. We use ODE as an SE in this work. An SE can only be accessed by other components within the JBI container.

BCs are components that provide connectivity to services and applications located outside. They are used to access external services and also to expose internal JBI services. ServiceMix provides BCs for HTTP and JMS bindings amongst others.

[2]This section is mainly based on [RD08, JBI05], other references are explicitly stated.

All internal components (SE and BC) communicate through the NMR using *normalized messages*. Therefore, one of the tasks of a BC is to transform external messages to and from normalized messages.

A normalized message consists of three parts: the *payload*, *message properties* and *message attachments*. The payload is an XML document that conforms to an abstract WSDL message type, that means without protocol encoding or format. Message properties hold meta data such as component-specific information. Message attachments are referenced from the payload and can be used to store non-XML data.

To expose an external service as an external endpoint on ServiceMix two endpoints must be configured for a BC. One external endpoint that receives the message and sends it to a internal endpoint which forwards it to the external service (see Listing 6.1 on page 65 for an example).

We use version 3.3.1 of ServiceMix which was the most recent stable version when the work was started. For version 4.0 the architecture has been significantly revised. But as it still provides a JBI container (although not fully compliant) and interfaces for message interception, our work should be portable with few effort. We did not apply any changes to ServiceMix' code—all extensions were realized through extension mechanisms and JBI components.

6.1.3 CASmix' Components

Figure 6.3 gives an overview on CASmix' architecture. In the following, the components are introduced and the routing of messages is discussed.

Implementation details are presented in Section 6.2.

CASmix MessageInterceptor

The MessageInterceptor's main task is to identify messages that are part of a conversation. It is plugged into ServiceMix' NMR and inspects every message that is put on the bus with regard to the receiver and message type (see Section 4.2.1). This data is compared with information from the ChoreographyManager. If the data does not match any choreography it is routed directly to the receiver. Else the message is routed to the ODE^{gold} component.

CASmix ChoreographyManager

The ChoreographyManager provides information about the currently deployed choreography descriptions. It also provides the implementation of the CSB interface (Section 4.3.2). Therefore it keeps track of the currently running conversations and forwards events to ODE^{gold}.

Additionally, the ChoreographyManager is responsible for the fault propagation to CAPs (see Section 4.5.2).

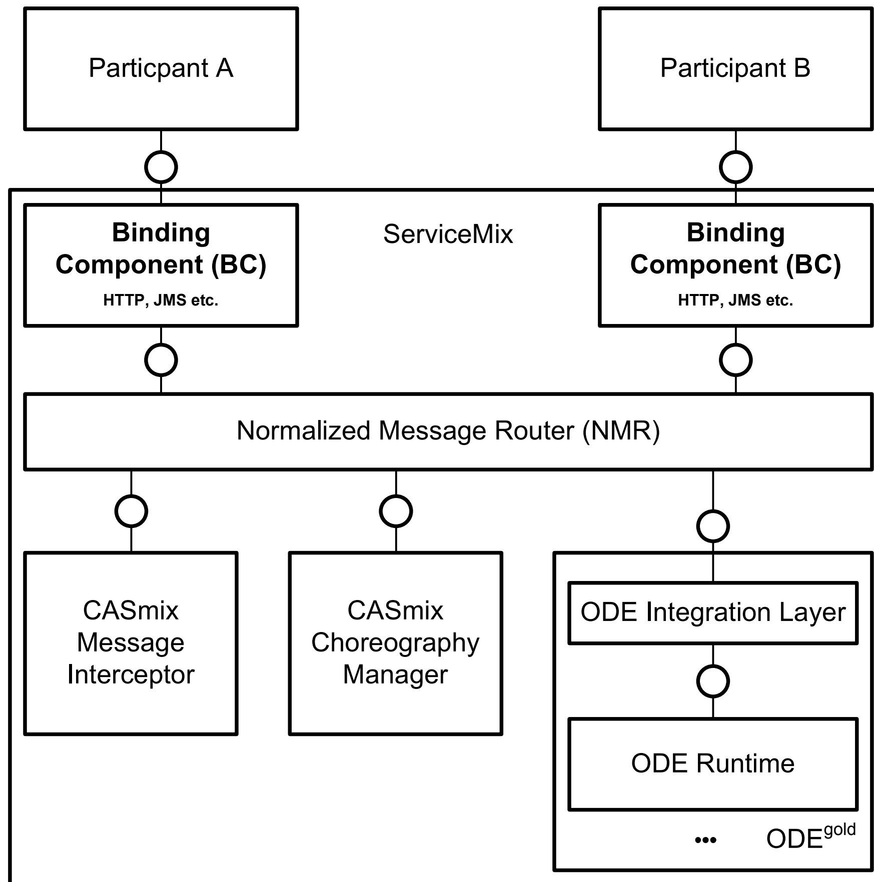


Figure 6.3: CASmix' Components

ODE^{gold}

ODE^{gold} is the core of CASmix. It is based on ODE. Based on BPEL^{gold} choreography descriptions it "orchestrates" choreographies. Its tasks also include correlation handling and fault detection.

6.1.4 Message Routing

Figure 6.4 shows how a message is routed in CASmix:

1. Participant A sends a message to B's external endpoint on CASmix.
2. The BC hands the (normalized) message over to the NMR.
3. The MessageInterceptor inspects the message. If it is choreography-related the target endpoint is exchanged with the corresponding endpoint at ODE^{gold} (continue with 4a.) else the message stays unchanged (continue with 4b.).

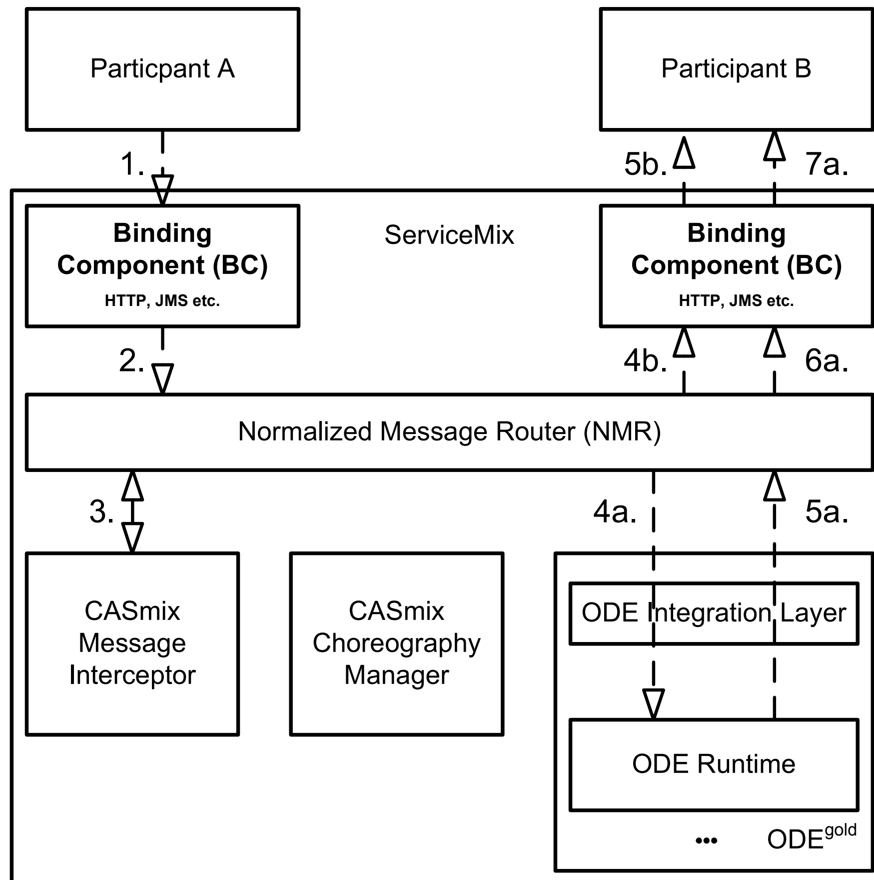


Figure 6.4: CASmix' Message Routing

4a. The message is received by ODE^{gold}'s runtime (via the integration layer).

5a. The message (precisely: a copy of it) is put back on the NMR with the original recipient (B's internal endpoint on CASmix).

6a. / 4b. The message is picked up by the BC endpoint.

7a. / 5b. The message is forwarded to B.

6.2 Implementation

In this section we discuss the main challenges of the implementation. A detailed documentation of the implementation also comes with the source code.

Listing 6.1 Example Endpoint Configurations

```
1  <http:consumer service="travel:AirlineService"
2                 endpoint="AirlinePortConsumer"
3                 targetService="travel:AirlineService"
4                 targetEndpoint="AirlinePort"
5                 wsdl="http://localhost:8080/travelScenarioPT.wsdl" />
6
7  <http:provider service="travel:AirlineService"
8                 endpoint="AirlinePort"
9                 locationURI="http://localhost:8082/ode/processes/AirlineService"
10                wsdl="http://localhost:8080/travelScenarioPT.wsdl" />
```

6.2.1 CASmix MessageInterceptor

The CASmix MessageInterceptor is an implementation of the ServiceMix *ExchangeListener* interface (*org.apache.servicemix.jbi.event.ExchangeListener*). This interface has two methods: *exchangeSent()* which is called when a JBI MessageExchange is handed over to the NMR and *exchangeAccepted()* which is called when the target component has accepted the message. Our logic to reroute messages and manipulate the headers is triggered when *exchangeSent()* is called.

First, the message properties are checked to decide whether the exchange must be rerouted. If the “de.uni.stuttgart.iaas.casmix.odegold.passed” property is set, no further inspection is required because the exchange had already been rerouted before and was handled inside ODE^{gold}. Else we have to decide if the exchange is related to a choreography as discussed in Section 4.2.1. The required information on deployed choreography descriptions is provided by the ChoreographyManager. If the exchange matches a choreography, we change the target endpoint to the corresponding endpoint provided by ODE^{gold}. The original endpoint is stored in a message property so that ODE^{gold} can retrieve it from there. The correlation to a specific conversation is done by ODE^{gold}.

Then we have to check if there are any headers set that can lead to a direct interaction between participants as discussed in Section 4.2.3. If one of the headers exists, we have to replace the endpoint it refers to by one that is exposed by CASmix. Such an endpoint can be discovered as follows: we need to find a provider endpoint of a BC where the target is the original endpoint; then we need a consumer endpoint that uses this provider endpoint. Example endpoint configurations are given in Listing 6.1. Consider a WS-Addressing ReplyTo header pointing to *http://localhost:8082/ode/processes/AirlineService* (Line 9). The endpoint of this provider is *AirlinePort* (Line 8) which is the target endpoint of the *AirlinePortConsumer* endpoint (Line 2). The latter is the endpoint we have to set in the ReplyTo header.

Application specific headers that are currently removed are the ODE *callback* and *session* headers. This list can be expanded if required.

The implementation cannot be deployed to ServiceMix like BCs or SEs. It must be added to ServiceMix’ classpath (for instance via a JAR file in the lib directory) and the configuration file (conf/servicemix.xml).

6.2.2 ODE^{gold}

In this section we show how support for BPEL^{gold}'s new activities (which also envolves fault detection) and enforcement is added to ODE. One of our main design goals is to minimize the changes to existing code. Therefore we use extension mechanisms such as event handlers where possible.

All messages leaving ODE^{gold} have the "de.uni.stuttgart.iaas.casmix.odegold.passed" property set, so that the MessageInterceptor can identify which messages were already handled.

Supporting gld:interaction

We identified three ways to provide support for the `<gld:interaction>` activity and `<gld:onInteraction>` (see Section 5.4.1) in ODE: using ODE's mechanism for `<extensionActivity>`; implementing the activity directly in the runtime; transforming `<gld:interaction>` into standard BPEL activities.

The first solution is not suitable because it is currently not possible to use JACOB constructs in `<extensionActivity>` implementations. Thus, basic actions like receiving or sending a message would have to be reimplemented. Also, this is no solution for the `<gld:onInteraction>` construct (as it is not enclosed in an `<extensionActivity>` activity).

Directly adapting the runtime would be against our goal to minimize changes to ODE and would have lead to code duplication (to reuse send and invoke for instance).

Thus, mapping `<gld:interaction>` and `<gld:onInteraction>` to standard BPEL constructs is the cleanest solution in our case. A simplified result of such a transformation for `<gld:interaction>` is shown in Listing 6.2. For the sake of clarity details such as correlation are left out.

The `<onMessage>` in Line 3 is used to receive the regular interaction. In this case, the message is forwarded to the original recipient. If the ChoreographyManager sends a message to indicate that the interaction will no longer occur, this is handled by the `<onMessage>` in Line 16. The default timeout which is needed for non-choreography-aware participants is realized through the `<onAlarm>` in Line 21.

Enforcement

To support enforcement of choreographies, all messages that do not have a matching active `<receive>` activity when they arrive are kept in a local queue. They are removed from the queue and the status of the JBI message exchange is set to error when the configured timeout or the number of steps (see Section 6.2.4) is exceeded.

A limitation of this approach is, that the actual class of the unexpected message (see Section 4.4.2) can not be identified. They are all handled equally.

Listing 6.2 gld:interaction mapped to Standard BPEL Activities (simplified)

```
<pick>
	<!-- regular interaction -->
	<onMessage partnerLink="inboundPL" operation="someOperation"
				portType="ns:somePT" variable="someRequest">
		<sequence>
			<assign><!-- copy original target epr to outboundPL --></assign>
			<invoke partnerLink="outboundPL" operation="someOperation"
				portType="ns:somePT" variable="someRequest">
				<catchAll>
					<throw>gld:interactionCompletionFault</throw>
				</catchAll>
			<invoke>
		</sequence>
	</onMessage>
	<!-- notification from ChoreographyManager -->
	<onMessage partnerLink="chorManagerInbound" operation="interactionFailed"
				portType="ns2:chorManagerPT">
		<throw>gld:interactionInitiationFault</throw>
	</onMessage>
	<!-- default timeout -->
	<onAlarm>
		<throw>gld:interactionInitiationFault</throw>
		<for>timeout for interactions</for>
	</onAlarm>
</pick>
```

6.2.3 CASmix ChoreographyManager

To keep track of conversation status, the ChoreographyManager has an interface to receive BPEL events from ODE^{gold}. These events are forwarded by an event handler in ODE^{gold}. In case of an exception the ChoreographyManager propagates this to the CAPs via the interface described in Section 4.3.2.

The ChoreographyManager also provides the implementation of the CSB interface for CAPs (see Section 4.3.2). Events are forwarded to ODE^{gold} either by sending messages that are received by the `<onMessage>` handlers discussed in Section 6.2.2 or via the ODE's Process Management API [Apad]. The latter can be used to terminate a conversation, the former to indicate faults of specific interactions.

6.2.4 Deployment Artifacts

Besides the BPEL^{gold} artifacts, the following information is required:

- A standard *ODE deployment descriptor*. The maximum timeout for interactions (see Section 4.4.1) and enforcement settings (see Section 4.5) can be configured via the existing *property* element.
- All *WSDL descriptions* that are referenced in the grounding.

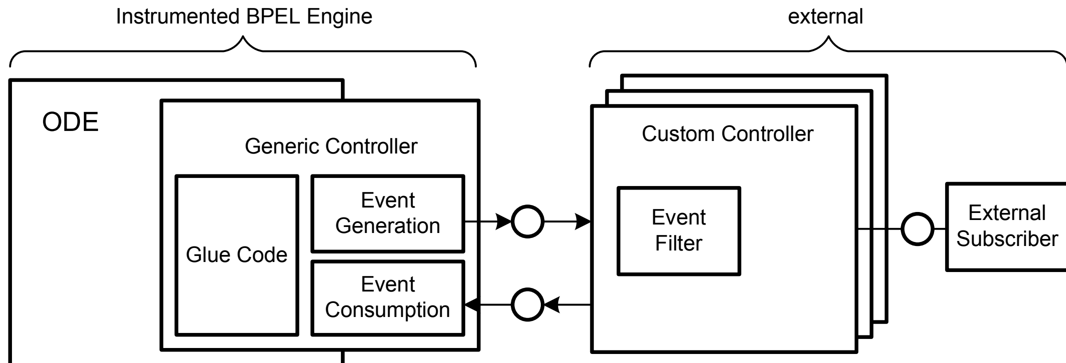


Figure 6.5: Pluggable Framework for BPEL Events ([Ste08])

- A list of all endpoints / participants with the roles (here a $\text{BPEL}^{\text{gold}}$ participant) they can play and whether they are choreography-aware. This information is needed by the ChoreographyManager to allow endpoints on the bus that implement a port type used in a choreography but are never used in a conversation.

Detailed information on the relationship between the artifacts and how they are wrapped (into a *service assembly*) for ServiceMix are given in the setup documentation that comes with the implementation.

6.3 Implementing a CAP using BPEL Events

The CAP is introduced as an optional enhancement to the CSB in Section 4.3. We implemented a prototype using the *Pluggable Framework for BPEL Events* introduced in [Ste08].

The architecture of the pluggable framework is based on [KKL07] and depicted in 6.5. Events in the instrumented BPEL engine (ODE) are caught and made visible to external applications by the *Generic Controller*. It also can block processes and handle incoming events. The communication with external subscribers (such as our CAP-implementation) is done via a *Custom Controller* which filters the events.

Table 6.1 lists the events of [Ste08]'s event model. Events that are relevant for our CAP-implementation are marked. Outgoing (from the view of the BPEL engine) events are used to keep track of the status of a conversation and its interactions. To forward events sent by the CSB we use incoming events.

6.4 Summary and Limitations

In this chapter we showed how to implement a Choreography-aware Service Bus (CSB) by “orchestrating” choreographies in an adapted BPEL engine. An overview on the architecture

Requirement	Blocking	Incoming	CAP-relevant
Process_Deployed			
Process_Undeployed			
Process_Instantiated			X
Instance_Running			
Instance_Suspended			X
Instance_Terminated			X
Instance_Completed			X
Instance_Faulted			X
Activity_Ready	X		X
Activity_Executing			X
Activity_Executed	X		X
Activity_Complete			X
Activity_Dead_Path			X
Activity_Terminated			X
Activity_Faulted	X		X
Evaluating_Transition-Condition_Faulted	X		
Scope_Compensating	X		
Scope_Compensated			
Scope_Handling_Event			
Scope_Event_Handling_Ended			
Scope_Handling_Termination	X		
Scope_Complete_With_Fault	X		
Scope_Handling_Fault	X		
Loop_Condition_True	X		
Loop_Condition_False	X		
Loop_Iteration_Complete	X		
Link_Ready			
Link_Evaluated	X		
Link_Set_True			
Link_Set_False			
Variable_Modification			
CorrelationSet_Modification			X
PartnerLink_Modification			
Compensate_Scope		X	
Fault_To_Scope		X	X
Start_Activity		X	
Complete_Activity		X	
Terminate_Activity		X	X
Continue_Loop		X	
Continue_Loop_Execution		X	
Finish_Loop_Execution		X	
Continue		X	
Read_Variable		X	X
Write_Variable		X	
Suppress_Fault		X	
Set_Link_State		X	
Suspend_Instance		X	
Resume_Instance		X	

Table 6.1: BPEL Event Model [Ste08]

of CASmix, ODE and ServiceMix was given and implementation challenges were discussed. Furthermore, an implementation of a Choreography-aware Participant (CAP) using BPEL events was presented.

In addition to the limitations regarding unexpected messages (see Section6.2.2), there is an issue when one endpoint plays more than one role in a conversation and one specific operation is used in both roles. Consider the following: participant p1 sends a message m1 of type x to role r1 played by participant p2 (interaction i1) and participant p1 sends a message m2 of type x to role r2 played by participant p2 (interaction i2). If both interactions can occur at the same time, the bus can not distinguish them. ODE^{gold} could provide one partnerlink/endpoint for each role, but the MessageInterceptor could not determine the correct one because the role is not part of the message due to our transparency requirement. No further investigation regarding these cases was done in this work.

Unfortunately, the configuration of endpoints in ServiceMix' is not always as straightforward as it could be. The documentation is outdated in many cases. Therefore we created a detailed documentation on how we set up our testing scenario. This is an implementation of our running example (see Section 2.1.4) using external ODEs and our implementation of a CAP as participants.

7 Summary and Outlook

In this work, we presented means to check and enforce choreographies during run-time. Therefore we introduced the Choreography-aware Service Bus (CSB) and BPEL^{gold}, an extension to model choreographies with the Business Process Execution Language (BPEL).

In contrast to a traditional Enterprise Service Bus (ESB), a CSB checks every choreography-related message it routes (Chapter 4). Exceptions can directly be detected and either the corresponding exception handling is triggered or the choreography is enforced. We also showed that a CSB can detect missing messages and how exceptions are propagated to participants.

An optional enhancement to this concept is the Choreography-aware Participant (CAP). A CAP can exchange information on conversations with the CSB. This allows easier detection and handling of exceptions. Nevertheless, we presented solutions for scenarios where not all of the participants are choreography-aware.

BPEL^{gold} is a choreography language that supports interaction models (Chapter 5). It provides all constructs to model choreographies for a CSB such as correlation and fault handling.

Finally, we presented our prototypical implementation of a CSB (Chapter 6). It is based on open source implementations of an ESB and a BPEL engine. A CAP has been implemented using BPEL events.

Outlook

During our work several interesting questions turned up—for both, the CSB and BPEL^{gold}.

Elaborating the integrated approach using BPEL^{gold} and BPEL4Chor as presented in Section 5.6.3 and providing a tool chain is one possible task in the area of BPEL^{gold}. It is also open if and how complex Message Exchange Patterns (MEPs) can be expressed in a compact form. There, the relationship to BPEL^{light} MEPs [NLL08] and how BPEL^{gold} choreographies can be recursively composed need further research.

With regard to the CSB, alternative implementation strategies must be investigated and compared to our solution. For instance, the approach using an automaton to track the conversation status. Here, [LW09] is a good starting point for theoretical aspects on how to transform a choreography description into a finite state automaton.

A Dialect for the Metadata Exchange

The XML Schema Definition for the WS-MEX dialect introduced in Section 4.3.2 is listed below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.bpel4chor.org/BPELgold/mex/ChorList"
    elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.bpel4chor.org/BPELgold/mex/ChorList">

    <xs:element name="choreographyList" type="tChoreographyList" />
    <xs:complexType name="tChoreographyList">
        <xs:sequence>
            <xs:element name="choreography" type="tChoreography"
                    minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="tChoreography">
        <xs:attribute name="name" type="xs:QName" />
        <xs:attribute name="myRole" type="xs:QName" />
        <xs:attribute name="myEndpoint" type="xs:anyURI" />
        <xs:attribute name="myInfoEndpoint" type="xs:anyURI" />
    </xs:complexType>
</xs:schema>
```

B BPEL^{gold} Travel Agency Scenario

The following sections show sample BPEL^{gold} (Chapter 5) artifacts based on our running example (Section 2.1.4).

B.1 Participant Topology

```
<?xml version="1.0" encoding="UTF-8"?>
<topology name="travelParticipantTopology" targetNamespace="http://example.org/travel/topology"
    xmlns:chor="http://example.org/travel/"
    xmlns="urn:HPI_IAAS:choreography:schemas:choreography:topology:2006/12"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:HPI_IAAS:choreography:schemas:choreography:topology:2006/12
    http://www.iaas.uni-stuttgart.de/schemas/bpel4chor/topology.xsd">

    <participantTypes>
        <participantType name="Traveler"
            participantBehaviorDescription="chor:travelScenario"
            processLanguage="http://www.bpel4chor.org/BPELgold/" />
        <participantType name="Agency"
            participantBehaviorDescription="chor:travelScenario"
            processLanguage="http://www.bpel4chor.org/BPELgold/" />
        <participantType name="Airline"
            participantBehaviorDescription="chor:travelScenario"
            processLanguage="http://www.bpel4chor.org/BPELgold/" />
    </participantTypes>

    <participants>
        <participant name="traveler" type="Traveler" selects="agency" />
        <participant name="agency" type="Agency" selects="airlines" />
        <participantSet name="airlines" type="Airline"
            forEach="chor:fe_RequestPrice">
            <participant name="currentAirline" forEach="chor:fe_RequestPrice" />
            <participant name="selectedAirline" />
        </participantSet>
    </participants>

    <messageLinks>
        <messageLink name="travelerTripOrderAgency" sender="traveler"
            receiver="agency" messageName="tripOrder" />
        <messageLink name="agencyPriceRequestAirline" sender="agency"
            receiver="currentAirline" messageName="priceRequest" />
        <messageLink name="airlinePriceAgency" sender="currentAirline"
            receiver="agency" messageName="price" />
        <messageLink name="agencyTicketOrderAirline" sender="agency"
```

```
            receiver="selectedAirline" messageName="ticketOrder" participantRefs="traveler" />
        <messageLink name="airlineTicketConfirmationAgency" sender="selectedAirline"
            receiver="agency" messageName="ticketConfirmation" />
        <messageLink name="agencyOrderRejectionTraveler" sender="agency"
            receiver="traveler" messageName="orderRejection" />
        <messageLink name="agencyConfirmationTraveler" sender="agency"
            receiver="traveler" messageName="confirmation" />
        <messageLink name="airlineEticketTraveler" sender="selectedAirline"
            receiver="traveler" messageName="eTicket" />
    </messageLinks>
</topology>
```

B.2 Process Interaction Description

```
<?xml version="1.0" encoding="UTF-8"?>
<process name="travelScenario"
    abstractProcessProfile=
    "http://www.bpel4chor.org/BPELgold/AbstractProfileForExtendedInteractionModels"
    targetNamespace="http://example.org/travel/" suppressJoinFailure="yes"
    xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
    xmlns:gld="http://www.bpel4chor.org/BPELgold" xmlns:tns="http://example.org/travel/"
    xmlns:wsu="http://schemas.xmlsoap.org/ws/2003/06/utility/">

    <extensions>
        <extension namespace="http://www.bpel4chor.org/BPELgold"
            mustUnderstand="yes" />
    </extensions>

    <messageExchanges>
        <messageExchange name="priceRequest" />
    </messageExchanges>

    <correlationSets>
        <correlationSet name="orderID" properties="tripOrderID" />
    </correlationSets>

    <sequence>
        <!-- traveler sends order -->
        <extensionActivity>
            <gld:interaction name="travelerTripOrderAgency">
                <correlations>
                    <correlation set="orderID" gld:senderInitiate="yes"
                        gld:receiverInitiate="yes" />
                </correlations>
            </gld:interaction>
        </extensionActivity>

        <!-- agency collects prices -->
        <forEach wsu:id="fe_RequestPrice" parallel="yes">
            <scope>
                <sequence>
                    <extensionActivity>
                        <gld:interaction name="agencyPriceRequestAirline"
```

```
                    messageExchange="priceRequest">
                    <correlations>
                        <correlation set="orderID" />
                    </correlations>
                </gld:interaction>
            </extensionActivity>
            <extensionActivity>
                <gld:interaction name="airlinePriceAgency"
                    messageExchange="priceRequest">
                    <correlations>
                        <correlation set="orderID" />
                    </correlations>
                </gld:interaction>
            </extensionActivity>
        </sequence>
    </scope>
</forEach>

<pick>
    <gld:onInteraction name="agencyTicketOrderAirline">
        <correlations>
            <correlation set="orderID" />
        </correlations>
        <scope>
            <sequence>
                <!-- agency orders ticker -->
                <extensionActivity>
                    <gld:interaction name="agencyTicketOrderAirline">
                        <correlations>
                            <correlation set="orderID" />
                        </correlations>
                    </gld:interaction>
                </extensionActivity>
                <!-- selectedAirline confirms -->
                <extensionActivity>
                    <gld:interaction name="airlineTicketConfirmationAgency">
                        <correlations>
                            <correlation set="orderID" />
                        </correlations>
                    </gld:interaction>
                </extensionActivity>


                <!-- traveler receives confirmation and eTicket -->
                <flow>
                    <extensionActivity>
                        <gld:interaction name="agencyConfirmationTraveler">
                            <correlations>
                                <correlation set="orderID" />
                            </correlations>
                        </gld:interaction>
                    </extensionActivity>
                    <extensionActivity>
                        <gld:interaction name="airlineEticketTraveler">
                            <correlations>
```

```
                                        <correlation set="orderID" />
                                    </correlations>
                                </gld:interaction>
                            </extensionActivity>
                        </flow>
                    </sequence>
                </scope>
            </gld:onInteraction>
            <onAlarm>
                <for><![CDATA['PT30S']]></for>
                <scope>
                    <!-- agency sends order rejection -->
                    <extensionActivity>
                        <gld:interaction name="agencyOrderRejectionTraveler">
                            <correlations>
                                <correlation set="orderID" />
                            </correlations>
                        </gld:interaction>
                    </extensionActivity>
                </scope>
            </onAlarm>
        </pick>
    </sequence>
</process>
```

B.3 Grounding

```
<?xml version="1.0" encoding="UTF-8"?>
<grounding
    xmlns="urn:HPI_IAAS:choreography:schemas:choreography:grounding:2006/12"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:props="http://example.org/travel/properties/"
    xmlns:agency="http://example.org/travel/agency/interfaces/"
    xmlns:airline="http://example.org/travel/airline/interfaces/"
    xmlns:traveler="http://example.org/travel/traveler/interfaces/"
    xsi:schemaLocation="urn:HPI_IAAS:choreography:schemas:choreography:grounding:2006/12
        http://www.iaas.uni-stuttgart.de/schemas/bpel4chor/grounding.xsd">

    <messageLinks>
        <messageLink name="travelerTripOrderAgency" portType="agency:agencyPT"
            operation="receiveOrder" />
        <messageLink name="agencyPriceRequestAirline" portType="airline:airlinePT"
            operation="requestPrice" /> <!-- request-response operation! -->
        <messageLink name="airlinePriceAgency" portType="airline:airlinePT"
            operation="requestPrice" /> <!-- request-response operation! -->
        <messageLink name="agencyTicketOrderAirline" portType="airline:airlinePT"
            operation="orderTicket" />
        <messageLink name="airlineTicketConfirmationAgency"
            portType="agency:agencyPT" operation="receiveOrderConfirmation" />
        <messageLink name="agencyOrderRejectionTraveler" portType="traveler:travelerPT"
            operation="receiveRejection" />
        <messageLink name="agencyConfirmationTraveler" portType="traveler:travelerPT"
            operation="receiveConfirmation" />
```

```
        <messageLink name="airlineEticketTraveler" portType="traveler:travelerPT"
            operation="receiveTicket" />
    </messageLinks>

    <participantRefs>
        <participantRef name="traveler" WSDLproperty="props:travelerRef" />
    </participantRefs>

    <properties>
        <property name="tripOrderID" WSDLproperty="props:tripOrderID" />
    </properties>

</grounding>
```

Bibliography

[AAA+07] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guízar, N. Kartha, C. K. Liu, R. Khalaf, D. König, M. Marin, V. Mehta, S. R. Thatte, D. van der Rijn, P. Yendluri, A. Yiu. Web Services Business Process Execution Language: OASIS Standard 11 April 2007, 2007. URL `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`. (Cited on pages 12, 14, 46 and 87)

[ACG+06] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, M. Montali, S. Storari, P. Torroni. LNCS 4184 - Computational Logic for Run-Time Verification of Web Services Choreographies: Exploiting the SOCS-SI Tool. *LECTURE NOTES IN COMPUTER SCIENCE*, (4184):58–72, 2006. (Cited on page 20)

[All09] T. Allweyer. Kollaborationen, Choreographien und Konversationen in BPMN 2.0: Erweiterte Konzepte zur Modellierung übergreifender Geschäftsprozesse, 2009. URL `http://kurze-prozesse.de/?page_id=194`. (Cited on page 9)

[Apaa] Apache. Apache ODE - Architectural Overview. URL `http://ode.apache.org/architectural-overview.html`. (Cited on pages 59, 60 and 88)

[Apab] Apache. Apache ODE - Jacob (Java Concurrent Objects). URL `http://ode.apache.org/jacob.html`. (Cited on page 60)

[Apac] Apache. Axis2 Website. URL `http://ws.apache.org/axis2/`. (Cited on page 60)

[Apad] Apache. ODE Management API Specification. URL `http://ode.apache.org/bpel-management-api-specification.html`. (Cited on page 67)

[Apae] Apache. ODE Website. URL `http://ode.apache.org/`. (Cited on page 59)

[Apaf] Apache. ServiceMix Website. URL `http://servicemix.apache.org/`. (Cited on page 59)

[B4C] BPEL4Chor Schemas. URL `http://www.iaas.uni-stuttgart.de/schemas/bpel4chor/`. (Cited on pages 35 and 42)

[BBB+06] K. Ballinger, B. Bissett, D. Box, F. Curbera, D. Ferguson, S. Graham, K. L. Canyang, F. Leymann, B. Lovering, R. McCollum, A. Nadalin, D. Orchard, S. Parastatidis, C. Riegen, J. Schlimmer, J. Shewchuk, B. Smith, G. Truty, A. Vedamuthu, S. Weerawarana, K. Wilson, P. Yendluri. Web Services Metadata Exchange: Version 1.1, 2006. URL `http://www.ibm.com/developerworks/library/specification/ws-mex/`. (Cited on pages 15 and 87)

[BCC+04] D. Box, E. Christensen, F. Curbera, D. Ferguson, J. Frey, M. Hadley, C. Kaler, D. Longworthy, F. Leymann, B. Lovering, S. Lucco, S. Millet, N. Mukhi, M. Nottingham, D. Orchard, J. Shewchuk, E. Sindambiwe, T. Storey, S. Weerawarana, S. Winkler. Web Services Addressing: W3C Member Submission 10 August 2004, 2004. (Cited on pages 15 and 87)

[BDDW07] A. Barros, G. Decker, M. Dumas, F. Weber. Correlation Patterns in Service-oriented Architectures. *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering (FASE), LNCS*, pp. 245–259, 2007. (Cited on page 10)

[BDH05a] A. Barros, M. Dumas, A. H. M. ter Hofstede. Service Interaction Patterns. *LECTURE NOTES IN COMPUTER SCIENCE*, 3649:302, 2005. (Cited on page 10)

[BDH05b] A. Barros, M. Dumas, A. H. M. ter Hofstede. Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection. 2005. (Cited on pages 10, 47 and 54)

[BDO05] A. Barros, M. Dumas, P. Oaks. A critical overview of the web services choreography description language. *Business Process Trends*, 2005. (Cited on page 10)

[BHR] A. Buchholz, R. Hohloch, T. Rathgeber. Vergleich von Open Source ESBs: Fachstudie. URL `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=FACH-0081&engl=0`. (Cited on page 17)

[Bis09] M. Bischof. Modeling and Runtime Support of Faults in Interaction Choreography Models: Diploma Thesis, 2009. URL `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2885&engl=0`. (Cited on page 28)

[BPM06] Business Process Modeling Notation (BPMN) Specification: Final Adopted Specification: Technical Report. 2006. URL `http://www.www.bpmn.org`. (Cited on page 9)

[CCK+01] J. Clark, C. Casanave, K. Kanaskie, B. Harvey, J. Clark, N. Smith, J. Yunker, K. Riemer. ebXML Business Process Specification Schema Version 1.01. *UN/CEFACT and OASIS*, 2001. (Cited on pages 9 and 87)

[CCMW01] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana. Web Services Description Language (WSDL) 1.1: W3C Note 15 March 2001, 2001. URL `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`. (Cited on pages 14 and 87)

[Cha04] D. A. Chappell. *Enterprise Service Bus*. Theory in practice. O'Reilly, Sebastopol, Calif., 1. ed. edition, 2004. (Cited on page 16)

[CKLW03] F. Curbera, R. Khalaf, F. Leymann, S. Weerawarana. Exception Handling in the BPEL4WS Language. *LECTURE NOTES IN COMPUTER SCIENCE*, 2678:276–290, 2003. (Cited on page 13)

[CMMT09] F. Chesani, P. Mello, M. Montali, P. Torroni. Verification of Choreographies During Execution Using the Reactive Event Calculus. pp. 55–72, 2009. (Cited on page 20)

[DB08] G. Decker, A. Barros. Interaction Modeling Using BPMN. *Business Process Management Workshops: Bpm 2007 International Workshops, Bpi, Bpd, Cbp, Prohealth, Refmod, Semantics4ws, Brisbane, Australia, September 24, 2007, Revised Selected Papers*, p. 208, 2008. (Cited on pages 9 and 56)

[DBKL08] G. Decker, A. Barros, F. Kraft, N. Lohmann. Non-desynchronizable Service Choreographies. In *Proceedings of the 6th International Conference on Service Oriented Computing (ICSOC): LNCS 5364*, pp. 331–346. Springer Verlag, Sydney, Australia, 2008. (Cited on page 8)

[Dec09] G. Decker. Design and Analysis of Process Choreographies: Doctoral Thesis, 2009. (Cited on page 8)

[DKB08] G. Decker, O. Kopp, A. Barros. An Introduction to Service Choreographies. *it - Information Technology*, 50(2):122–127, 2008. (Cited on pages 7, 8, 9 and 88)

[DKLW08] G. Decker, O. Kopp, F. Leymann, M. Weske. Interacting Services: From Specification to Execution, 16.10.2008. (Cited on pages 10, 14, 33, 34, 56 and 88)

[DM08] G. Decker, J. Mendling. Instantiation Semantics for Process Models. *Proceedings of the 6th International Conference on Business Process Management (BPM), LNCS*, pp. 164–179, 2008. (Cited on page 10)

[DOZ06] G. Decker, H. Overdick, J. M. Zaha. On the Suitability of WS-CDL for Choreography Modeling. *Proceedings of Methoden, Konzepte und Technologien fur die Entwicklung von dienstebasierten Informationssystemen (EMISA 2006), Hamburg, Germany, October*, 2006. (Cited on page 10)

[DW09] G. Decker, M. Weske. Interaction-centric Modeling of Process Choreographies: in submission. 2009. (Cited on page 8)

[DZD06] G. Decker, J. Zaha, M. Dumas. Execution Semantics for Service Choreographies. In *Web Services and Formal Methods*, pp. 163–177. 2006. (Cited on page 10)

[Fre06] L. Fredlund, editor. *Implementing ws-cdl*. 2006. (Cited on page 19)

[HV09] S. Hallé, R. Villemaire. Runtime monitoring of web service choreographies using streaming XML. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pp. 2118–2125. ACM, New York, NY, USA, 2009. (Cited on page 20)

[HW03] G. Hohpe, B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley, Amsterdam, 2003. (Cited on pages 31 and 61)

[IT00] ITU-T. Message Sequence Chart. Recommendation Z.120, 2000. (Cited on page 9)

[JBI05] JSR-000208 Java Business Integration 1.0: Final Release, 2005. URL http://jcp.org/aboutJava/communityprocess/final/jsr208/index.html. (Cited on page 61)

[Juc06] F. Juchart. Entwicklung eines Routing-Verfahrens für SOAP-Nachrichten: Diploma Thesis, 2006. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2460&engl=0. (Cited on page 55)

[KBR+05] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, C. Barreto. Web Services Choreography Description Language Version 1.0: W3C Candidate Recommendation 9 November 2005, 2005. URL http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/. (Cited on pages 19, 42 and 87)

[KE09] O. Kopp, L. Engler. Discussion on Special Cases of Correlation Modeling: Personal Conversation, September 16, 2009. (Cited on page 25)

[Kip06] A. Kipp. Ablösung von WS-CDL durch BPEL und WSFL Global Model: Diploma Thesis, 2006. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2379&engl=1. (Cited on page 19)

[KKL07] R. Khalaf, D. Karastoyanova, F. Leymann. Pluggable Framework for Enabling the Execution of Extended BPEL Behavior. *Proceedings of the 3rd International Workshop on Engineering Service-Oriented Application (WESOA 2007)*, 2007. (Cited on page 68)

[KL09] O. Kopp, F. Leymann. Do We Need Internal Behavior in Choreography Models? In Oliver Kopp, Niels Lohmann, editors, *Proceedings of the 1st Central-European Workshop on Services and their Composition, ZEUS 2009, Stuttgart, Germany, March 2–3, 2009*, volume 438 of *CEUR Workshop Proceedings*, pp. 68–73. CEUR-WS.org, Stuttgart, 2009. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2009-31&engl=0. (Cited on page 42)

[KLN08] O. Kopp, T. van Lessen, J. Nitzsche. The Need for a Choreography-aware Service Bus. In *YR-SOC*, pp. 28–34. 2008. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2008-38&engl=. (Cited on page 21)

[KMM08] I. H. Kruger, M. Meisinger, M. Menarini. Interaction-based Runtime Verification for Systems of Systems Integration. *Journal of Logic and Computation*, 2008. (Cited on page 20)

[KMWL08] O. Kopp, D. Martin, D. Wutke, F. Leymann. On the Choice Between Graph-Based and Block-Structured Business Process Modeling Languages. In *Modellierung betrieblicher Informationssysteme (MobIS 2008)*, volume P-141 of *Lecture Notes in Informatics (LNI)*, pp. 59–72. Gesellschaft für Informatik e.V. (GI), 2008. (Cited on page 10)

[KS86] R. Kowalski, M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986. (Cited on page 20)

[KWH07] Z. Kang, H. Wang, P. C. Hung. WS-CDL+: An Extended WS-CDL Execution Engine for Web Service Collaboration. *Web Services, IEEE International Conference on*, 0:928–935, 2007. (Cited on page 19)

[Ley01] F. Leymann. Web services flow language (WSFL 1.0), 2001. (Cited on pages 9 and 87)

[Ley05] F. Leymann. The (Service) Bus: Services Penetrate Everyday Life. *LECTURE NOTES IN COMPUTER SCIENCE*, (3826):12–20, 2005. (Cited on page 17)

[LKLR07] N. Lohmann, O. Kopp, F. Leymann, W. Reisig. Analyzing BPEL4Chor: Verification and Participant Synthesis. In Marlon Dumas, Reiko Heckel, editors, *Web Services and Formal Methods, Forth International Workshop, WS-FM 2007 Brisbane, Australia*, pp. 46–60. Springer-Verlag, 2007. URL `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2007-81&engl=0`. (Cited on page 19)

[LR00] F. Leymann, D. Roller. *Production Workflow: Concepts and Techniques.* Prentice Hall, Upper Saddle River, NJ, 2000. (Cited on page 13)

[LW09] N. Lohmann, K. Wolf. Realizability is Controllability. In Cosimo Laneve, Jianwen Su, editors, *Web Services and Formal Methods, 6th International Workshop, WS-FM 2009, Bologna, Italy, September 4-5, 2009, Revised Selected Papers*, Lecture Notes in Computer Science. Springer-Verlag, 2009. (Cited on page 71)

[NLKL07] J. Nitzsche, T. van Lessen, D. Karastoyanova, F. Leymann. BPEL light. In *Proceedings of the 5th International Conference on Business Process Management (BPM 2007)*, volume 4714 of *Lecture Notes in Computer Science*, pp. 214–229. Springer-Verlag, 2007. URL `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2007-24&engl=0`. (Cited on page 14)

[NLL08] J. Nitzsche, T. van Lessen, F. Leymann. Extending BPELlight for Expressing Multi-Partner Message Exchange Patterns. In *12th International IEEE Enterprise Distributed Object Computing Conference, ECOC 2008, 15-19 September 2008, Munich, Germany*, pp. 245–254. IEEE Computer Society, 2008. (Cited on page 71)

[Pap08] M. P. Papazoglou. *Web services: Principles and technology.* Pearson/Prentice Hall, Harlow, 2008. (Cited on page 15)

[RD08] T. Rademakers, J. Dirksen. *Open Source ESBs in action: Example implementations in Mule and ServiceMix.* Manning, Greenwich, Conn., 2008. (Cited on pages 17 and 61)

[Rei07] P. Reimann. Generating BPEL Processes from a BPEL4Chor Description: Studienarbeit, 2007. URL `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=STUD-2100&engl=0`. (Cited on page 57)

[RR09] M. Riegen, N. Ritter. Reliable Monitoring for Runtime Validation of Choreographies. In *The Fourth International Conference on Internet and Web Applications and Services (ICIW 2009)*. IEEE Computer Society, 2009. (Cited on page 20)

[Sil09] B. Silver. BPMN 2.0 Status Update, 2009. URL `http://www.brsilver.com/wordpress/2009/07/06/bpmn-20-status-update-2/`. (Cited on page 9)

[SKL09] T. Scheibler, D. Karastoyanova, F. Leymann. Dynamic Message Routing Using Processes. In *Proceedings of 16th Fachtagung Kommunikation in Verteilten Systemen (KiVS 09)*. Springer, 2009. URL `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2009-02&engl=0`. (Cited on page 55)

[SM05] M. Sailer, M. Morciniec. Monitoring and execution for contract compliance: Technical Report HPL-2001-261 (R.1). 2005. (Cited on page 21)

[Ste08] T. Steinmetz. Ein Event-Modell für WS-BPEL 2.0 und dessen Realisierung in Apache ODE: Diploma Thesis, 2008. URL `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2729&engl=0`. (Cited on pages 68, 69 and 88)

[Tos08] M. Tost. Konzeption und Implementierung eines Enterprise Service Bus mit nativer Unterstützung von komplexen Message Exchange Patterns: Diploma Thesis, 2008. URL `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2791&engl=0`. (Cited on page 20)

[WCL+06] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D. F. Ferguson. *Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more*. Prentice Hall/PTR, Upper Saddle River, N.J., 4. printing. edition, 2006. (Cited on pages 12 and 15)

[Wes07] M. Weske. *Business process management: Concepts, languages, architectures*. Springer, Berlin, 2007. (Cited on page 7)

[ZBDH06] J. M. Zaha, A. Barros, M. Dumas, A. t. Hofstede. Let's Dance: A Language for Service Behavior Modeling. *LECTURE NOTES IN COMPUTER SCIENCE*, (4275):145, 2006. (Cited on pages 10, 43 and 47)

[ZDH+06] J. M. Zaha, M. Dumas, A. t. Hofstede, A. Barros, G. Decker. Service Interaction Modeling: Bridging Global and Local Views. *Enterprise Distributed Object Computing Conference, IEEE International*, pp. 45–55, 2006. (Cited on pages 8 and 43)

All links were last followed on September 30, 2009.

Abbreviations

The number on the right indicates the page number of the first occurrence of the abbreviation in this document. The reference in brackets points to related standards or documentation.

Abbreviation	Meaning	Page
ODE	Apache Orchestration Director Engine	59
BC	Binding Component	61
BPEL	Business Process Execution Language ([AAA+07])	5
BPSS	Business Process Schema Specification ([CCK+01])	9
BPMN	Business Process Modeling Notation	9
CAP	Choreography-aware Participant	21
CID	Conversation identifier	25
CSB	Choreography-aware Service Bus	6
DAO	Data Access Object	60
DPE	Dead Path Elimination	13
ESB	Enterprise Service Bus	5
JBI	Java Business Integration	60
LTL	Linear Temporal Logic	20
MEP	Message Exchange Pattern	20
MSC	Message Sequence Chart	9
NMR	Normalized Message Router	61
PID	Process Interaction Description	36
SE	Service Engine	60
SOA	Service Oriented Architecture	5
WS-Addressing	Web Services Addressing ([BCC+04])	15
WS-CDL	Web Services Choreography Description Language ([KBR+05])	10
WSDL	Web Services Description Language ([CCMW01])	14
WSFL	Web Services Flow Language ([Ley01])	9
WS-MEX	Web Services Metadata Exchange ([BBB+06])	15

List of Figures

1.1 A Choreography-aware Service Bus 5

2.1 Locally unenforceable Choreography 8
2.2 Categorization of Choreography Languages [DKB08] 9
2.3 Let's Dance Relationship Types 11
2.4 The Running Example: Travel Agency Choreography 13
2.5 BPEL4Chor Artifacts [DKLW08] 14

4.1 CSB: Preventing Direct Interactions 24

5.1 $\text{BPEL}^{\text{gold}}$ Artifacts 35
5.2 Let's Dance Mapping: Forbidden Models 43
5.3 Let's Dance Mapping: Different Cases for Inhibits 46
5.4 From $\text{BPEL}^{\text{gold}}$ to Executable BPEL Processes 58

6.1 ODE Architecture [Apaa] 60
6.2 ServiceMix Architecture 61
6.3 CASmix' Components 63
6.4 CASmix' Message Routing 64
6.5 Pluggable Framework for BPEL Events ([Ste08]) 68

List of Tables

4.1 Status Types for the Information Exchange between CAP and CSB 27
4.2 Event Types for the Information Exchange between CAP and CSB 27

5.1 Service Interaction Patterns: Support in Languages 56

6.1 BPEL Event Model [Ste08] 69

Listings

4.1 Example Response from CAP to a WS-MEX GetMetadata Request 26

5.1 Example Participant Topology 37
5.2 Syntax specification of the BPEL^{gold} `<interaction>` element 38
5.3 Syntax specification of the BPEL^{gold} `<onInteraction>` element 39
5.4 BPEL^{gold}: Syntax for Grounding Files 42
5.5 Mapping Let's Dance to BPEL^{gold}: `precedes` 44
5.6 Mapping Let's Dance to BPEL^{gold}: `weak precedes` 45
5.7 Mapping Let's Dance to BPEL^{gold}: `inhibits` - busy waiting 47
5.8 Mapping Let's Dance to BPEL^{gold}: `inhibits` - Case 2 48
5.9 Mapping Let's Dance to BPEL^{gold}: `inhibits` - Case 3 48
5.10 Mapping Let's Dance to BPEL^{gold}: `two-way-inhibit` 49
5.11 Mapping Let's Dance to BPEL^{gold}: `two-way inhibit` using a Pick Activity . . 50
5.12 Racing Incoming Messages: Topology 51
5.13 Racing Incoming Messages: Process Interaction Description 51
5.14 One-to-many Send: Topology 51
5.15 One-to-many Send: Process Interaction Description 51
5.16 One-from-many Receive: Topology 52
5.17 One-from-many Receive: Process Interaction Description 52
5.18 Multi-responses: Topology 53
5.19 Multi-responses: Process Interaction Description 53
5.20 Contingent Requests: Topology 53
5.21 Contingent Requests: Process Interaction Description 54
5.22 Request with Referral: Topology 54
5.23 Request with Referral: Process Interaction Description 55
5.24 Relayed Request: Topology 55
5.25 Relayed Request: Process Interaction Description 55

6.1 Example Endpoint Configurations 65
6.2 gld:interaction mapped to Standard BPEL Activities (simplified) 67

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Lasse Engler)