

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

Diplomarbeit Nr. 2885

Modeling and Runtime Support of Faults in Interaction Choreography Models

Marc Bischof

Course of Study: Computer Science

Examiner: Prof. Dr. Frank Leymann

Supervisor: Dipl.-Inf. Oliver Kopp

Commenced: January 18, 2009

Completed: July 17, 2009

CR-Classification: C.4, D.1.3, D.2, D.2.2, D.2.5, D.2.11,
D.3.2, D.4, D.4.1, D.4.5, D.4.7, H.4.1, K.1

Contents

Lists of Figures, Tables, Listings and Abbreviations	7
1 Introduction	11
2 Background	15
2.1 Modeling the Business Process Collaboration	15
2.2 Process Modeling Languages	16
2.2.1 BPEL—Business Process Execution Language	16
2.2.2 BPEL4Chor—BPEL for Choreographies	18
2.2.3 BPMN1.2—Business Process Modeling Notation	18
2.2.4 BPMN2.0—Business Process Model and Notation	19
2.2.5 iBPMN—interaction BPMN	20
2.3 WS Coordination	20
2.4 Summary	21
3 Related Work	23
3.1 Terms and Definitions in the Context of Exceptions	23
3.1.1 Definition of Exceptions, Errors and Faults	23
3.1.2 Definition of Fault Tolerance	24
3.2 Exception Classification	25
3.2.1 Exception Classification in Real-Time Systems	25
3.2.2 Exception Classification According to Different Perspectives	26
3.2.3 Exception Classification According to the Source of Occurring	26
3.2.4 Exception Classification According to the Basis of Integration	27
3.2.5 Exception Classification According to Organizational Boundaries	27
3.3 Exception Handling	28
3.3.1 Three Phases of Exception Handling	29
3.3.2 Exception Handling According to Functional Views	30
3.3.3 Exception Handling According to the Continuation Behavior	31
3.3.4 Rule Based Exception Handling	32
3.3.5 Cross-Organizational Exception Handling	32
3.4 Interaction-centric Exception Modeling	34
4 Motivation of the Approach and Running Example	37
4.1 From Orchestration to Choreography	37
4.2 Process Choreography Design	38
4.3 Running Example	40

4.4	Intermediate Events in Interaction Models	41
5	Exceptions and their Handling in Interaction Choreography Models	45
5.1	Classification of Exceptions in Interaction Models	45
5.2	Granularity of Exceptions in Interaction Models	47
5.3	Requirements of Exceptions in Interaction Models	47
5.3.1	On the Need of <i>Silent Actions</i>	48
5.3.2	On the Need of <i>No Action</i> Constructs	49
5.3.3	On the Relevance of Intermediate Events	49
5.3.4	Advantages of Intermediate Events Implementing the Inter Process Communication	50
5.3.5	Simultaneous Raised Exceptions and Exception Graphs	50
5.3.6	Exception Propagation in Choreographies Using Dominance	52
5.4	Cross-Partner Scopes	56
5.4.1	Motivation and Definition of Cross-Partner Scopes	56
5.4.2	Semantics of Cross-Partner Scopes	57
5.4.3	Runtime Considerations on Cross-Partner Scopes	58
5.4.4	Nesting and Overlapping Considerations of Cross-Partner Scopes	60
5.4.5	Classification of Cross-Partner Scopes According to the Type of Overlapping	62
5.4.6	Runtime Considerations on Overlapping Cross-Partner Scopes	62
5.4.7	Shortcomings of Cross-Partner Scopes	64
5.5	Exception Handling in Interaction Models	65
5.5.1	Choreography Life-Line	66
5.5.2	Membership of Exception Handlers	67
5.5.3	Number of Coordinators	67
5.6	Top-Down vs. Bottom-Up Choreography Design using Interaction Exceptions	68
5.7	Summary	69
6	Modeling a Choreography with Interaction Exceptions	71
6.1	Exceptions in Business Process Modeling Languages	71
6.2	Graphical Integration of Intermediate Events	73
6.3	Graphical Integration of Cross-Partner Scopes	75
6.4	Graphical Integration of Other Artifacts	76
6.5	Usage of Controlling Roles	77
6.6	Evaluation of Modeling Artifacts According to Exception Handling Strategies	78
6.6.1	Modeling Exception Handling According to Functional Views	78
6.6.2	Modeling Exception Handling According to the Continuation Behavior	79
6.6.3	Modeling Exception Handling Using the Detached Execution Mode	79
6.7	Summary	79
7	From a Choreography with Interaction Exceptions to the Runtime	81
7.1	Deriving Interconnection Models	81
7.2	Mapping iBPMN to BPEL4Chor	85
7.2.1	Exception Handler for Overlapping Scopes	88
7.2.2	Representation of Exception Graphs in BPEL4Chor	88

7.2.3	Representation of Dominance Trees in BPEL4Chor	89
7.2.4	Summary	89
7.3	Runtime Support of Interaction Exceptions: The BPEL Engine	90
7.4	Runtime Support of Interaction Exceptions: The Coordinator	92
7.4.1	Overview on the Overall Coordination of a Choreography	92
7.4.2	Generation of the Custom Controller	96
7.4.3	The Local Coordination Protocol	97
7.4.4	Mapping Regular Expressions to BPEL	99
7.4.5	The Global Coordination Protocol	99
7.5	Realization of Simultaneous Raised Exceptions	101
7.6	Realization of Exception Propagation	102
7.7	Realization of Scope Hierarchies	102
7.8	Summary	102
8	Prototype Realization	105
8.1	Tools	105
8.1.1	The Oryx Editor	105
8.1.2	Apache Orchestration Director Engine (ODE)	106
8.2	Architecture Overview	106
8.3	Modeling Interaction Exceptions with Oryx	106
8.4	Runtime Support of Interaction Exceptions	108
9	Summary and Outlook	111
	Bibliography	115
A	Overview of Coordination Collaboration	131
A.1	Initialization Phase	131
A.2	Completion Phase	132
A.3	Encountering Fault Phase	133
A.4	Blocking Phase	133
A.5	Termination Phase	134
A.6	Compensation Phase	135
A.7	Closing Phase	135
A.8	Compensate within Fault Handler Phase	135
A.9	Resuming from Fault Handler Phase	136
A.10	Faulting Fault Handler Phase	136
A.11	Terminating Fault Handler Phase	137

List of Figures

1.1	A view on the web services stack	11
1.2	Analogy data, process and choreography management adapted from [LKP08]	12
2.1	Three BPEL4Chor artifacts according to [DKLW07]	18
2.2	Interconnection model of the car rental scenario modeled in BPMN	19
2.3	Overview of iBPMN language constructs according to [Dec08]	20
2.4	Interaction model of the car rental scenario in iBPMN	21
4.1	Deadlock of interacting process orchestrations according to [Wes07]	38
4.2	The choreography design process of the travel scenario	39
4.3	State of the art simplified travel scenario choreography with embedded exceptions	40
4.4	Simplified travel scenario with cross-partner scope adapted from [Kop08a]	41
4.5	Desynchronizability: race condition and deadlock resolving according to [Dec09a]	43
4.6	Modeling attached intermediate events in interaction models	44
5.1	Combined classification of interaction exceptions	46
5.2	Granularities of interaction exceptions	47
5.3	Exception graph and coordinated exception handling according to [XRR98]	52
5.4	Two-dimensional exception propagation according to [XRR98]	53
5.5	Control flow graph and dominance tree of running example	54
5.6	Complex control flow graph and dominance tree according to [Bis08]	55
5.7	BPEL scope fragments according to [Ruf07]	58
5.8	Coordination alternatives of CPSs	60
5.9	Detailed view on running example involving overlapping CPSs	61
5.10	Overlapping types of CPSs: simple scope overlapping	62
5.11	Overlapping types of CPSs: complex scope overlapping	63
5.12	Example overlapping graphs for complex overlapping	64
5.13	Scenarios leading to deadlocks due to CPS fragmentation presented in [Ruf07]	65
5.14	Projection of Figure 5.13 using imposed CPSs	65
5.15	Buyer manufacturer scenario	66
5.16	Number of coordinators needed	67
6.1	Travel scenario as self-adapting recovery net	72
6.2	Modeling alternatives of non-interrupting events	73
6.3	Overview of CPSs in extended iBPMN	76
6.4	Silent- and no action construct in extended iBPMN	77
6.5	Usage of controlling roles	77

7.1	Transformation rules for the conversation phase adapted from [Dec09a]	82
7.2	The transformation phase of interaction models to interconnection models	83
7.3	Transformation rules for the conversation phase: additional artifacts and events	84
7.4	Transformation rules for the conversation phase: Cross-Partner Scopes	86
7.5	Invalid BPMN models in rearrangement phase according to [Dec09a]	87
7.6	Architecture overview of the pluggable framework according to [Ste08]	91
7.7	Simplified state-chart of a CPS adapted from [Ste08]	91
7.8	Modeling alternatives for the collaboration between BPEL engine and coordinators	94
7.9	Global view on the interaction between coordinating entities	96
7.10	Acyclic coordination protocol graph of local coordinator	99
7.11	Cyclic coordination protocol graph of global coordinator	101
7.12	Mapping Cross-Partner Scopes to WS-Coordination	103
8.1	Architectural overview on the global interplay	107
A.1	Initialization phase	132
A.2	Completion phase	132
A.3	Encountering fault phase	133
A.4	Blocking phase	134
A.5	Termination phase	134
A.6	Compensation phase	135
A.7	Closing phase	136
A.8	Compensate within fault handler phase	136
A.9	Resuming phase	137
A.10	Faulting phase	137
A.11	Termination phase (subactivities)	137

List of Tables

6.1	Overview on intermediate events in extended iBPMN	74
7.1	Overview of events used for CPS coordination in the pluggable framework according to [Ste08]	92

List of Listings

2.1	The simplified structure of the car rental process modeled in BPEL	17
7.1	The structure of choreography scope fragment definition in participant behavior of BPEL4Chor presented in [Ruf07]	85
7.2	The structure of choreography scope fragment in the topology of BPEL4Chor presented in [Ruf07]	87
7.3	The proposed structure of Cross-Partner Scopes in the topology of BPEL4Chor	87
7.4	The structure of fault handlers for <i>overlapping CPSs</i>	88
7.5	The structure of exception graphs	88
7.6	The structure of dominance trees	89
7.7	Simplified BPEL code for regular expressions according to [Mie06]	100

List of Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
BP	Business Process
BPEL	see WS-BPEL
BPEL4Chor	BPEL for Choreographies
BPM	Business Process Management
BPMN	Business Process Modeling Notation
BPMN2.0	Business Process Model and Notation
BTP	Business Transaction Protocol
CA	Child Activities
CC	Custom Controller
CPG	Coordination Protocol Graph
CPS	Cross-Partner Scope
DBMS	Database Management System
DCS	Direct Child Scopes
DFA	Deterministic Finite Automaton
DPE	Death Path Elimination
EA	Enclosing Scope Activities
ECA	Event Condition Action rules

ERD	Entity-Relationship Diagram
FCT handling	Fault, Compensation and Termination handling (BPEL terminology)
GC	Generic Controller
GCo	Global Coordinator
IPC	Inter Process Communication
IPN	Interaction Petri Net
LCo	Local Coordinator
OASIS	Organization for the Advancement of Structured Information Standards
ODE	Orchestration Director Engine
OMG	Object Management Group
PBD	Participant Behavior Description
RDF	Resource Description Framework
SARN	Self-adapting Recovery Net
SOA	Service-Oriented Architecture
SQL	Structured Query Language
UDDI	Universal Description, Discovery and Integration
UML	Unified Modeling Language
URI	Universal Resource Identifier
WAMO	Workflow Activity Model
WfMS	Workflow Management System
WS-BA	Web Services Business Activity
WS-BPEL	Web Services Business Process Execution Language
WS-C	Web Service Coordination
WS-CAF	WS-Composite Application Framework
WS-CDL	WS-Choreography Definition Language
WSDL	Web Service Description Language
WSFL	Web Services Flow Language
XLANG	XML LANGuage
XSL	Extensible Stylesheet Language
XSLT	XSL Transformations

1 Introduction

Today, there is an increasing acceptance of *Service-Oriented Architectures* (SOA) as a paradigm for systems development and integration. Thereby independently developed and interoperable applications are loosely coupled within and across organizational boundaries. These applications can be represented as (web) services in business process modeling languages. The de facto standard for this purpose is the *Web Services Business Process Execution Language* (WS-BPEL, or BPEL for short) [BPE07]. The *Business Process Modeling Notation* (BPMN) [BPM09] provides a graphical interface. The interconnections between services are realized during a stack of standards shown in Figure 1.1, including *SOAP* [SOA07], *Web Service Description Language* (WSDL) [WSD01] or *Universal Description, Discovery and Integration* (UDDI) [UDD05].

Web services can be orchestrated into a single business process while choreographies are used to describe the interaction among such processes from a global view. There exist two modeling approaches for choreographies: In the case of interconnection models the control flow is defined per participant whereas interaction models provide a truly global perspective [DKB08]. Realistic service interactions become increasingly complex since they go beyond simple sequences of requests and responses. Furthermore, they involve large numbers of participants which can be enabled „to create a context needed to propagate an activity to other services“ [WC09]. For this purpose, „the *WS-Coordination* [(WS-C)] specification describes an extensible framework for providing protocols that coordinate the actions of distributed applications“ [WC09].

Beyond the technical realization of services the conceptual modeling can be regarded focusing on „high level features demanded by business modelers“ [LKP08, p. 22]. Additionally, conceptual modeling decrease the complexity of choreographies to essential units, so that non-IT users can also cope with choreography modeling. [LKP08] describes an analogy of data modeling to process modeling using three layers of abstraction such as shown in Figure 1.2a and the left part of Figure 1.2b.

The conceptual layer provides an abstract overview using graphical notations such as *Entity-Relationship Diagram* (ERD) [Che76], *Unified Modeling Language* (UML) [UML05] on the data

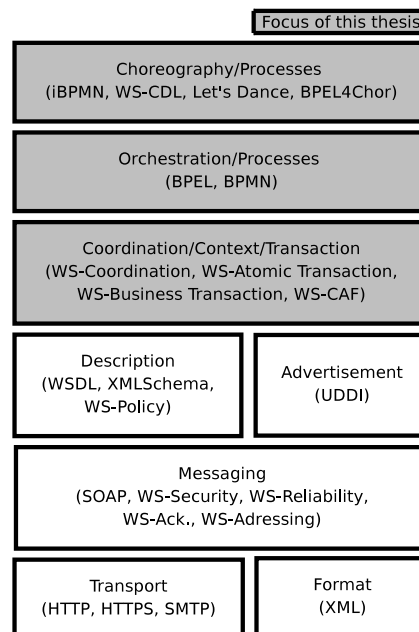


Figure 1.1: A view on the web services stack according to [Wee+05] and [BDO05]

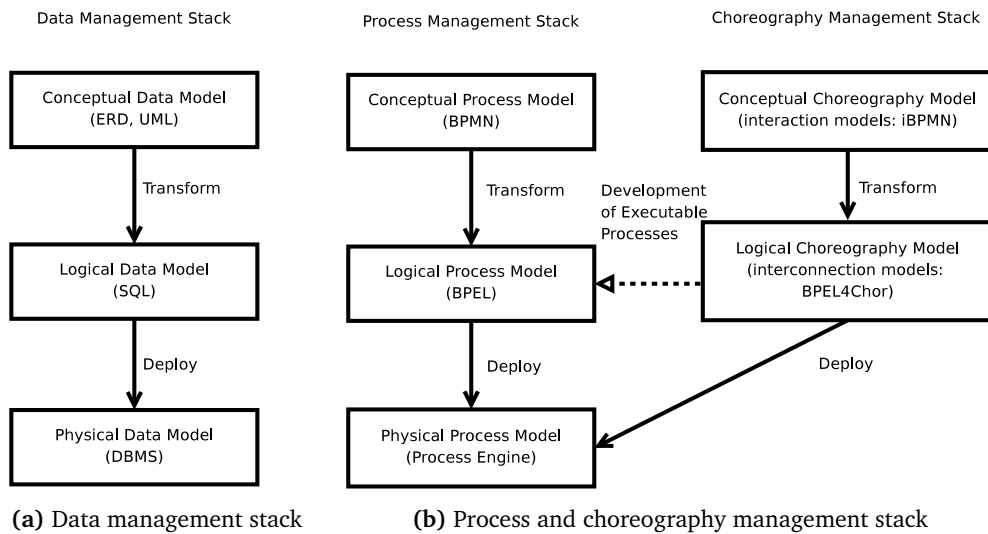


Figure 1.2: Analogy data, process and choreography management adapted from [LKP08]

management stack or BPMN on the process management stack. The logical layer describes the technical perspective using a full featured programming interface such as *Structured Query Language* (SQL) [CB74] or BPEL. Finally, the physical layer describes the concrete representation in the *Database Management System* (DBMS) [HR01] or the process engine.

According to this idea of different abstraction levels a *choreography management stack* can be deduced which is shown in Figure 1.2b. This time, the conceptual layer is represented by interaction-centric notations such as iBPMN [DB07] or Let's Dance [Zah+08]. The logical layer is described by interconnection-centric notations such as *BPEL for Choreographies* (BPEL4Chor) [DKLW07]. Out of the BPEL4Chor description the deployment can be achieved by the participant grounding. Finally, the physical layer is the process engine.

Process models cover at least three parts: a) the essential components of processes, b) the nature of their interaction and c) the manipulation of their independent and collective behavior, especially in erroneous situations. Part a) can be achieved by business process modeling languages. For b) there are two paradigms already mentioned above: interconnection and interaction models. Part c) refers to exception management.

Exception management on choreography level provides a means to ensure fault tolerance in the presence of distributed concurrency and cross-partner interaction and their faults. Therefore exception handling can be applied to interaction models to achieve increased reliability of choreographies. Present interaction-centric languages support this in a rudimentary and deficient way or even ignore this requirements at all.

However, exceptional requirements for cross-organizational workflow interaction are often independent of a particular process implementation. They correspond to the interaction with the outside of the current process which may be even not known to the specific process at

design time. Therefore a model to specify exceptional properties on a higher level is required which can automate coordinated exception handling out of the conceptual specification.

Current approaches discuss similar usage in the means of interconnection models using fault handling of BPEL-based choreographies in BPEL4Chor [Ruf07]. Furthermore, dynamic approaches such as [HB04; Ham05] „adapts the mechanism of the underlying Petri net at run time to handle exceptions while keeping the Petri net design simple and easy“ [HBM08, p. 2]. This thesis specifies exceptional behavior of choreographies on interaction level and describes an automated runtime support out of this enhancement.

Task and Structure of this Thesis

The goal of this thesis is to identify potential exception modeling strategies on interaction choreography level and to describe a possible runtime support. For this purpose, the range of issues leading to exceptions during choreography execution and the various way in which they can be addressed are investigated. The research is guided by a classification of exceptions and handling strategies including an evaluation how they fit into interaction models. Thereby, the thesis focuses on two main requirements: model the proposed solution using interaction models and describe possible runtime support of it.

On the one hand, a detailed discussion of modeling attached intermediate events, especially intermediate error events is needed. Furthermore, the investigation examines how the concept of BPEL-scopes can be included into interaction models and describes their behavior and coordination. This results in a detailed discussion of *Choreography Scopes* or *Cross-Partner-Scopes* (CPS) as they are proposed by [Ley05; Kop08b; KWL09].

In the following, this work describes how choreography exception handling can be achieved using interaction models. Therefore the work is split into 9 chapters:

Chapter 2—Background covers the basics needed to understand the following chapters.

Chapter 3—Related Work identifies related work.

Chapter 4—Motivation of the Approach and Running Example motivates the task of this thesis and introduces the running example.

Chapter 5—Exceptions and their Handling in Interaction Choreography Models introduces the theoretical foundation of the investigation.

Chapter 6—Modeling a Choreography with Interaction Exceptions discusses how the theoretical aspects of Chapter 5 are graphically integrated.

Chapter 7—From a Choreography with Interaction Exceptions to the Runtime discusses how the theoretical aspects of Chapter 5 are technically integrated.

Chapter 8—Prototype Realization shows how the aspects of Chapters 5 and 6 are implemented into a prototype.

Chapter 9—Summary and Outlook finally draws a conclusion and provides an outlook proposing suggestions on how to further prosecute the matter.

2 Background

This chapter introduces necessary related background and technologies needed to understand the content of this thesis. First the foundation of business process management is explained in Section 2.1. Then business process modeling languages and notations are considered in Section 2.2 on the following page. Finally, Section 2.3 on page 20 comprises the WS-Coordination framework. Since an exhaustive introduction is not the task of this thesis, readers not familiar with these backgrounds are advised to look at the referenced literature.

2.1 Modeling the Business Process Collaboration

This section introduces the main concepts to describe business collaborations using web services. Therefore, some terms and definitions are introduced and main approaches are described.

The key observation of business process management is that each product of a company is the outcome of a number of activities performed. These activities can be summarized as *Business Process*. Thus business process management describes the explicit representation and the execution constraints between business processes. [Wes07] Formally, business processes can be defined according to [Wes07, p. 5]:

„*Business Process Management* (BPM) includes concepts, methods, and techniques to support the design, administration, configuration, enactment, and analysis of business processes.“

Modeling business collaborations can be described from two perspectives: (i) the view of a single participant or (ii) from a global perspective. The first is achieved through process *orchestrations* describing the interactions and dependencies within a single organization. Importantly the „services themselves have no knowledge of their involvement in a higher level application [...]“ [BWR08, p. 1]. For this reason, a central coordinator monitors the collaboration of the involved services. In contrast, process *choreographies* focus on externally observable interactions between different business partners in a peer-to-peer collaborations. Without a central coordinator, each participant „[...] knows exactly when to execute its operations and with whom to interact“ [BWR08, p. 1 f.]. Choreographies can be used as a message-exchange-design-time-pattern for processes, leaving gaps for implementation-dependent behavior. Formally, a choreography can be defined according to [W3C04]:

„A *choreography* defines the *sequence and conditions* under which multiple cooperating independent agents *exchange messages* in order to perform a task to achieve a goal state.“

For choreography modeling two approaches exist: (i) interconnection models and (ii) interaction models. *Interconnection* models describe interconnected interface behavior models from

a local point of view. *Interaction* models focus on the *Inter Process Communication* (IPC) as interactions from a global perspective and describe their control flow dependencies. Thus, interaction models are an important concept to narrow the „gap between organizational aspects and the information technology“ [Wes07, p. 4] since they hide implementation details and focus on the global behavior of processes. For a detailed discussion of these approaches, refer to [DW07; Dec08; DKLW09].

Process engines control the process lifecycle. Thereby, orchestrations run on a centralized process engine while processes within choreographies normally are distributed to several engines [Wes07]. Modeling languages can be used to describe processes and their collaboration. A choice of languages relevant in this thesis is described in the next section.

2.2 Process Modeling Languages

To describe business processes and their collaboration different languages and notations can be used. Out of the broad range of the notational landscape four examples are described in this section according to Figure 1.2 on page 12: (i) *BPEL* as de facto standard for specifying business processes, (ii) *BPEL4Chor* as an extension of BPEL for defining choreographies, (iii) *BPMN* as de facto standard for graphical representation of business processes. Finally, (iv) *iBPMN* as an extension of BPMN for interaction modeling.

2.2.1 BPEL—Business Process Execution Language

BPEL is both an XML-based programming language and an export format for business processes using web services which emerged as WS-BPEL 2.0 OASIS Standard [BPE07]. It describes the interaction of multiple services by defining the imported and exported functionality, necessary to achieve the business goal [BG06]. BPEL combines IBMs graph-based *Web Services Flow Language* (WSFL) [Ley01], to describe the composition of web services, with Microsoft’s calculus-based *XML Language* (XLANG) [Tha01]. Furthermore, BPEL provides an interoperable and portable language for both abstract and executable processes. Executable processes describe the internal implementation while abstract processes focus on the external message behavior. BPEL can be viewed as the glue, used to construct business processes with services (from a SOA system) as building blocks [Ley06].

Partner links define the relations to other partners to model peer-to-peer collaborations. Thereby business process instances can be created when receiving messages or events. „BPEL[...] binds Web services into cohesive units encapsulated into activities“ [BG06, p. 7]. These activities are structured using several control flow constructs known from traditional programming languages (i. e. sequential or parallel structuring, branches, loops or (sub) scopes). In addition, BPEL provides the concept of instance routing using correlation sets. Listing 2.1 on the next page shows the structure of a BPEL process. Graphical editors for BPEL are available under [Ory08a; Ecl08], while also programming interfaces was proposed [BKLL09]. The running example in this thesis involves a car rental process which is shown

Listing 2.1 The simplified structure of the car rental process modeled in BPEL

```

1 <process name="CarRental" targetNamespace="http://example.com/car-rental/"
2   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable">
3
4   <partnerLinks>
5     <partnerLink name="customer" partnerLinkType="pns:PartnerType" myRole="rentalService"/>
6   </partnerLinks>
7
8   <variables>
9     <variable name="bookingData"/>
10    <variable name="registrationData"/>
11  </variables>
12
13  <sequence>
14    <receive partnerLink="customer" operation="bookingRequest" variable="bookingData"/>
15    <if><condition>!customer.isRegistered</condition>
16      <sequence>
17        <reply partnerLink="customer" operation="AccountCreationRequest"/>
18        <receive partnerLink="customer" operation="registrationInformation"
19          variable="registrationData"/>
20        <reply partnerLink="customer" operation="registrationConfirmation"/>
21      </sequence>
22    </if>
23    <if><condition>carAvailable</condition>
24      <sequence>
25        <reply partnerLink="customer" operation="confirmReservation"/>
26        <wait><until>bookingData.rentalTime</until></wait>
27        <empty><documentation>rentalProcessing</documentation></empty>
28      </sequence>
29    <else>
30      <sequence>
31        <reply partnerLink="customer" operation="rejectReservation"/>
32      </sequence>
33    </else>
34  </if>
35 </sequence>
</process>

```

in Listing 2.1. A detailed introduction of the overall choreography of the running example is given in Section 4 on page 37, while parts are already shown in this section. For the sake of completeness, exception handling of BPEL is presented.

Exception Handling in BPEL: BPEL provides a structured exception handling mechanism similar to the try-catch mechanism of traditional programming languages [BKLL09]. In contrast to the definition given in Section 3.1 on page 23, exceptions in BPEL are called faults. The equivalent of the try block is the BPEL *scope* while the catch block is represented by a *fault handler*. Thus, a *separation* of normal and exceptional code is achieved as well as a *structured hierarchy* of fault handlers. If the fault can be handled successfully, the fault will not appear outside of the scope. In case of an unsuccessful handling, the fault is propagated to the next level according to the hierarchy. The outer most scope is the (implicit) process scope. If the propagation reaches this

scope, the process itself fails. Thus, BPEL's standard way of handling faults is forward recovery. Further readings coping with fault handling in BPEL can be found in [CKLW03; Kha08].

2.2.2 BPEL4Chor—BPEL for Choreographies¹

BPEL4Chor provides extensions „to lift BPEL from an orchestration language to a fully choreography language“ [KL08, p. 3]. Thereby, BPEL4Chor consists of three artifacts: (i) participant behavior descriptions, (ii) topology behavior description and (iii) participant grounding.

The *Participant Behavior Descriptions* (PBD) are abstract BPEL processes which have to be completed to executable ones. WSDL port types or operations are not used in the participant behavior description since they are brought in during *participant grounding*. This allows a behavior description without the fixed connection to concrete realizations. The *topology description* provides a global view on the choreography by defining participants and message links. The overall structure of BPEL4Chor is summarized in Figure 2.1. A graphical editor for BPEL4Chor is available under [Ory08b].

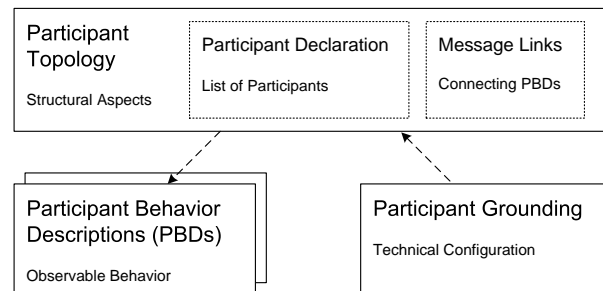


Figure 2.1: Three BPEL4Chor artifacts according to [DKLW07]

2.2.3 BPMN1.2—Business Process Modeling Notation²

BPMN was developed under the coordination of the *Object Management Group* (OMG) [BPM09]. By focusing on design and analysis, BPMN aims at supporting business levels as well as software technology levels. Thus, BPMN provides a graphical interface for modeling business processes from a less technical view or higher level respectively. Although translations from BPMN to BPEL are available (cf. [Whi05; ODBH06; Sch08a]), BPMN lacks expressiveness of event and termination handlers for example [PDKL07].

BPMN expresses business process models in business diagrams containing a set of modeling elements. The building blocks of BPMN are task or subprocesses and events which are structured using gateways and connecting objects *within* swimlanes. An editor for BPMN is available under [Ory08c].

The running example in this thesis involves a *customer* making a journey. Thereby, the *customer* needs to rent a car. Figure 2.2 shows the global view on the rental process as interconnection model. The process contains four main parts: (i) the booking request, (ii) an optional

¹ This section is mainly based on [DKLW07; KL08], other references are explicitly stated.

² This section is mainly based on [Wes07], other references are explicitly stated.

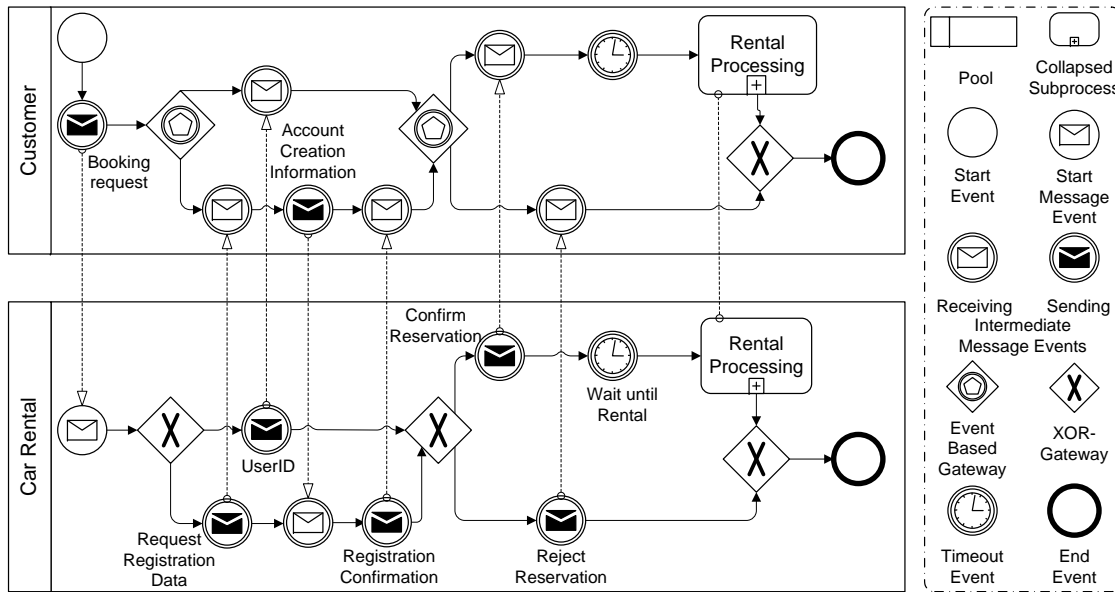


Figure 2.2: Interconnection model of the car rental scenario modeled in BPMN

registration and a choice between a (iii) successful reservation leading to the rental processing or a (iv) failed reservation.

2.2.4 BPMN2.0—Business Process Model and Notation

For a short time there are interesting proposals of the OMG in the context of BPMN2.0 [BPM08]. The key idea is to provide a consistent notation which allows the usage in the whole design cycle of business processes. This means, from the global design perspective of business analysts to the implementation perspective of process engines. Since the information provided with the proposals are quite sparse, only the main ideas relevant in this thesis are presented.

BPMN2.0 allows the traditional specification of abstract and executable processes and their orchestration as in BPMN1.2. In addition to these traditional modeling artifacts two new diagram types are introduced: (i) choreography diagrams and (ii) collaboration diagrams. Thereby, *choreography diagrams* describe the sequence and conditions of expected observable behavior. Hence, answering the question how a choreography proceed [Sch09]. In contrast, *collaboration diagrams* describe a more abstract view on the interactions of a choreography. Hence, answering the question what interactions occur between whom [Sch09].

Within choreography diagrams, BPMN2.0 introduces the notion of *Choreography Task* as atomic activity in a choreography. Furthermore, *non-interrupting events* are introduced, referring to event handling without cancellation of the process (this will be discussed in detail later).

During this thesis, diagram types as well as non-interrupting events are investigated in more detail (cf. Chapter 4 and Chapter 5).

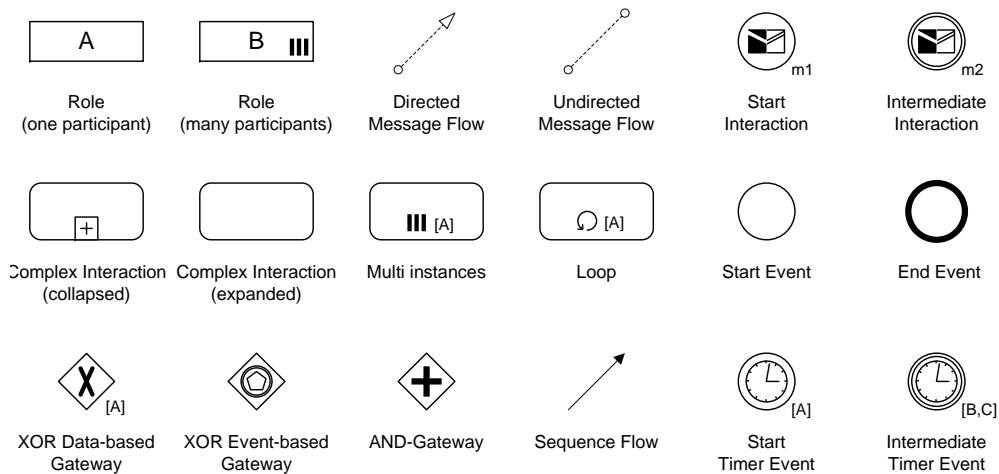


Figure 2.3: Overview of iBPMN language constructs according to [Dec08]

2.2.5 iBPMN—interaction BPMN

While BPEL4Chor provides modeling support for choreographies, iBPMN [DB07; Dec09a] focuses on interaction choreography modeling using BPMN. iBPMN restricts the set of modeling elements of BPMN to *interactions and events* which are structured by control flow elements *between* participants. Figure 2.3 gives an overview of the language constructs of iBPMN. iBPMN provides a means of conceptual modeling on the level of choreography interaction modeling (cf. Figure 1.2). An editor for iBPMN is available under [Ory08d].

For a first illustration, Figure 2.4 shows the registration part of the car rental process of Figure 2.2 on the preceding page. Thereby, the main point of interest lies in both (i) atomic interactions and (ii) controlling roles. *Atomic interactions* refer to the atomic nature of interactions. In contrast to non-atomic interactions, there is only a single symbol *between* roles marking atomic message exchanges, such as the *booking request* in Figure 2.4. *Controlling roles* refer to the owner of control flow- and data flow related decisions. Hence, avoiding redundant specifications of these artifacts. For instance, the decision marked with *[CR]* in Figure 2.4 indicates, that the *car rental* role is responsible for this decision (cf. XOR and event-based gateways in Figure 2.2).

2.3 WS Coordination³

The WS-Coordination (WS-C) [WC09] specification provides an extensible framework for the coordinated outcome of distributed applications. The framework enables application services to create a distributed context needed to operate in a heterogeneous environment. For this purpose, WS-C provides three abstractions: (i) The *coordination protocol* as a set of rules

³This section is based on [ACKM04; WC09], other references are explicitly stated.

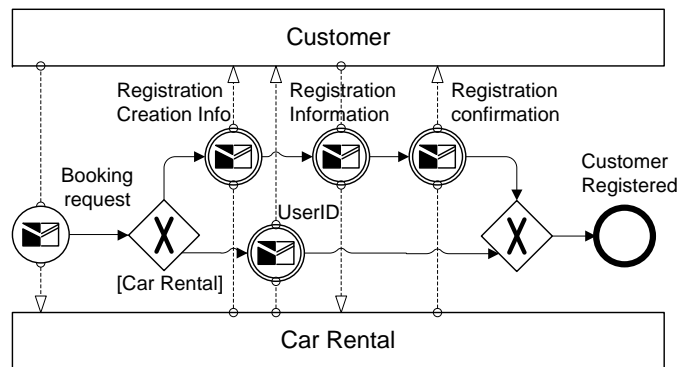


Figure 2.4: Interaction model of the car rental scenario in iBPMN

governing conversations. (ii) The *coordination type* as a set of coordination protocols. Finally, (iii) the *coordination context* as a data structure indicating which messages belongs to the same conversation.

Between a coordinator and its participants WS-C provides three forms of interactions: (i) activation service, (ii) registration service and (iii) protocol specific interactions. Thereby, the first two are independent of the type of coordination. To start the coordination a participant requests a coordinator to create a new coordination context using the *activation service*. Having created a coordination context, participants register with a coordination using the *registration service*. After registration, the coordinator notifies the corresponding participants of a coordination context when the protocol is executed by *protocol specific interactions*. Thereby, WS-C does not assume a concrete coordination protocol. One possible protocol is the *Web Services Business Activity (WS-BA)* [WB09] which is extended during this investigation. WS-BA provides two coordination protocols for compensation-based transactions: *coordinator completion* and *participant completion*. The difference between both is the owner of the decision when a participant has completed its work. With coordinator completion, the coordinator owns the completion decision, while with participant completion the participants decide their completion.

For complex scenarios where hierarchies of coordinators are necessary, the concept of *interposition* allows a participant of a transaction to act as a coordinator (i. e. subcoordinator). Thereby, the top-level coordinator communicates with the subcoordinator as a participant. Thus, the subcoordinators become registered with the *create coordination context* message to an activation service.

2.4 Summary

This introduction give an overall view on the background of business process modeling, especially modeling orchestrations and choreographies. The main business process modeling

languages and notations such as BPEL, BPEL4Chor, BPMN and iBPMN are introduced. Finally a short description of WS-C is given.

BPEL acts as runtime language of business processes which are deployed on workflow engines. In a choreography BPEL processes act as independent participants. With the approached concept of this thesis, these independent participants are observed externally without having to change the particular process implementation. This external observation spans multiple participating processes residing on different workflow engines. Thus, the external observation involves WS-Coordination in order to achieve a coordinated outcome of these distributed applications. In Chapter 7, BPEL4Chor is extended by additional artifacts which are needed to express fault handling capabilities in choreographies. These extensions are derived out of an extended iBPMN model. The theoretical foundation of the extended interaction model using iBPMN is given in Chapter 5. The derivation of the extended interaction model to a interconnection model is presented in Chapter 6. Thereby, BPMN is used as graphical representation of interconnection models. The next chapter gives a view on related work.

3 Related Work

Nowadays people depend daily on services provided by computer controlled systems. Modern automobiles are controlled by at least 18 independent control units which assists driving such as *Anti-lock Braking System (ABS)* and *Electronic Stability Program (ESP)* [Her09]. In addition, they take over health protection such as the air bag. Furthermore, aircrafts, banking systems and even power plants are controlled by computer systems. As a consequence, a loss of control within such a system through a fault would be disastrous. At the best case, this would result in economic losses, while the worst case would be the loss of human lives. [Hil98] Therefore, exceptions are the *bread and butter* of today's information technologies and have been widely recognized in the past 40 years.

Since this thesis deals with interaction fault handling, it is necessary to know which kind of faults can appear in a choreography. Additionally, it is important how exceptions can be handled on the one hand, and how exceptions can be modeled in choreographies. Finally, a clear semantic of such exception modeling has to be given by describing the runtime support. Although there are similar concepts, the domain and purpose of exceptions and their handling are quite different. This chapter first introduces terms and definitions about exceptions in Section 3.1. Then Section 3.2 on page 25 describes the classification of exceptions while Section 3.3 on page 28 discusses handling strategies. Section 3.4 on page 34 investigates exception modeling on interaction level.

3.1 Terms and Definitions in the Context of Exceptions

This chapter figures out recent proposals related to the context of interaction choreography models and their exception handling. Therefore, more precise definitions of exception related terms and reliability of a system are necessary. The following two subsections introduces traditional terms which are then regarded in the context of exception handling in business processes.

3.1.1 Definition of Exceptions, Errors and Faults

This thesis follows the definitions of [AR79]. They define the *reliability of a system* as a measure of the success with which the system conforms to some authoritative specification of its behavior. Furthermore, they propose the following definitions of failures, errors and faults:

When the behavior of a system deviates from that which is specified for it, this is called a *failure*. Failures refer to the *external behavior* of a system. They are the *result of internal errors* in the system. An *error* arises when the computation reaches an internal state that was not anticipated in the design of the system. The internal, technical cause for an error is termed a *fault*. [AR79]

Thereby, „*failures* are generally assumed to be handled at the system and application level, typically by relying on the transactional properties of the underlying [...] platform“ [CCPP99, p. 34], while *exceptions* denote deviations on a higher level. In the context of exception handling in business processes „*exceptions* constitute events which may occur during business process execution and require deviations from the normal business process behavior“ [HBM08, p. 2]. These exceptions can be either generic or application specific [HBM08]. For example, *generic exceptions* can be the unavailability of resources or a timeout, while an *application-specific* exception may be the unavailability of seats for a flight booking service. Having introduced exceptions, errors and faults, the reliability of software systems can be discussed.

3.1.2 Definition of Fault Tolerance

To improve the reliability of software systems, two approaches can be distinguished: (i) fault prevention and (ii) fault tolerance [AL90]. *Fault prevention* attempts to eliminate any possibility of faults in a system at design time. For this purpose it consists of two stages: *Fault avoidance* and *fault removal* using design review, program verification or statical analysis. [BW01] „Unfortunately, system testing can never be exhaustive and remove all potential faults“ [BW01]. For example, a study of large software systems [HH86] found out that „for every million lines of code, 20,000 bugs entered the software; normally 90% of these found out by testing“ [BW01, p. 102].

Due to the limitations of fault prevention, *fault tolerance* has to be considered by designers. Fault tolerance enables a system to continue functioning in the presence of faults. For this reason it provides three levels: (i) full fault tolerance, (ii) graceful degradation and (iii) fail safe. *Full fault tolerance* continues to operate in the presence of faults with no significant loss of functionality or performance. In contrast, *graceful degradation* accepts partial degradation of functionality or performance during recovery or repair. Finally, during a *fail safe* the system maintains its integrity while accepting a temporary halt in its operation. [BW01]

Fault tolerance in transactional environments is achieved through backward and forward recovery which are discussed in detail in Section 3.3 on page 28. In the context of exception handling in business processes [HB04; HBM08] define fault-tolerance as „the property that allows BPs [...] to respond to expected but unusual situations and failures (also called exceptions)“ [HBM08, p. 2]. In the following, Section 3.2 on the facing page discusses exception classifications. Then Section 3.3 on page 28 investigates exception handling strategies based on fault-tolerance.

3.2 Exception Classification

The previous section introduced the notion of exceptions. Failures are exceptions which arise through errors. Errors themselves are the manifestation of faults. Thus, a proper classification of exceptions has to cope with the reasons of exceptions to find out which exceptions can appear in general.

Several classifications of exceptions can be found in related literature. However, no approach provides an overall view on exception handling in the context of business processes. Therefore, four fundamental views are investigated in detail in this section. Finally, Chapter 5 provides a combination of these views which is applicable to exception handling on both, interaction and interconnection modeling.

Naturally, the *cause* of exceptions is a main point of interest but also other viewpoints are possible. Out of the multitude proposals, four are of special interest in this thesis (cf. sections 3.2.2 to 3.2.5 on pages 26–27): Exceptions can be regarded from (i) *different perspectives* of the handling, (ii) the *source of occurring*, (iii) on the *basis of integration* and (iv) according to *organizational boundaries*.

Additionally, exceptions can be further classified according to seven other perspectives which are not mentioned in this thesis: (i) similar to [EL95] by dividing them according to violated consistency constraints [Hei98], (ii) based on different levels of abstraction (i. e. adaptation to context changes, process level, resource level and infrastructure level) [SHB98], (iii) on the basis of time of occurring (i. e. localized or sparse synchronicity or asynchronously) [CP99], (iv) by the degree of difference to the normal case (i. e. established and otherwise exceptions referring to incomplete or missing business rules or true exceptions which are unanticipated situations) and effect on organizational rules (i. e. no effect or causing instance or type level updates) [SW95], or (v) according to the exception type (default exceptions, suboptimal coordination or mechanism failures) [KD00], (vi) using specific exception handling corresponding to whom is responsible for an exception [Ard+06], or who has the same skills as a faulting service [CCP07]. Finally, (vii) based on commitment that exists between participants [MS05].

3.2.1 Exception Classification in Real-Time Systems

Exceptions are a well discussed and accepted topic in real-time systems. (Real-time systems are systems required to react to stimuli from environments within certain time intervals [Ran+95].) Therefore, real-time systems are used as foundation in this section.

In general three types of faults can be distinguished according to [BW01]: (i) *Transient faults* such as external interference, start at a particular time, remains in the system for some period and then disappears. (ii) *Permanent faults* such as software design errors, remain in the system until they are repaired. (iii) *Intermittent faults* are transient faults that occur from time to time.

Having identified the reasons of faults, three general exceptions can be derived, again quoting from [BW01]: (i) *Value failures* such as an incorrect result or out of range arguments, indicate that a value associated with a service is in error. (ii) *Timing failures* signal that services are delivered at the wrong time. For example, this can be too late, too early or never. (iii) *Commission failures* denote that a service is delivered which is not expected.

Also [GR93] centered on causes for exceptions regarding environment, operations, maintenance, hardware and software faults.

Although, providing a good starting point for a structured investigation of exceptions, it turns out that these classifications are too low level (i. e. technical) according to the usage on an interaction level. For instance, the notion of *exception* is not properly regarded in this classification. Thus, a classification of exceptions on interaction level needs a further dimension. This leads to the classification of exceptions according to different perspectives.

3.2.2 Exception Classification According to Different Perspectives

Exceptions can be classified through different perspectives of the handling. In the area of information systems, one of the earliest contributions came from [SM95] which provides five perspectives on exceptions (random-event, error, political system, total quality management and human computer) [SO00].

In the context of *Workflow Management Systems* (WfMS) [EL95; EL96] are the first providing an essential classification of workflow exceptions. They provide a more general classification of exceptions than [SM95]. A key aspect is the classification of exceptions into four types according to the perspective of the handling: (i) *Expected exceptions* describe deviations from the normal behavior as defined in Section 3.1 on page 23. They are to be handled at process level. (ii) *Unexpected exceptions* correspond to not properly modeled (since unexpected) semantics of the business process. They are to be handled at process definition level. (iii) *Basic failures* represent faults of the underlying platform. They are to be handled at the corresponding platform. Finally, (iv) *application failures* constitute faults due to upgrades of services, interfaces or database transaction failures. They are to be handled at application level.

This taxonomy is more general, since it copes with exceptions as well as failures. However, regarding the notion of business processes something is missing again. In business processes exceptions can be thrown due to different sources, such as user, activity or through timeouts. This leads to the classification of exceptions according to the source of occurring.

3.2.3 Exception Classification According to the Source of Occurring

[Sti+98; HB04; HBM08] focus on the source of occurring which is of special interest also in interaction models. They distinguish between user, task and duration exceptions. *User exceptions* are triggered by the user. *Task exceptions* constitute data events and physical or logical unavailability of resources. Finally, *duration exceptions* refer to deadline expiration.

So far, previous classifications identified exceptions from a quite technical view. Since this thesis deals also with the modeling of exceptions, classifications having a more global perspective has to be regarded.

3.2.4 Exception Classification According to the Basis of Integration

[SO00] classifies exceptions from a modeling perspective regarding „when and how useful exceptions can be included in the process model“ [SO00, p. 4]. According to [EL95], the classification distinguishes between (useful) expected and unexpected exceptions.

Expected exceptions are further refined into embedded and separated exceptions. *Embedded exceptions* are the simplest approach to model exceptional behavior since they are embedded into the process logic. Therewith, they „become part of the core process and generally are not even considered as exceptions“ [SO00, p. 7]. To avoid this camouflage and to support maintenance by designing readable process models, the notion of *separate exceptions* is identified. Separate exceptions are not part of the normal control flow and can be achieved through two approaches: (i) *exception rules* using a rule or event base or (ii) *exception workflows* where exceptions can be modeled as processes themselves. The important difference between exception rules and exception workflows is that with exception workflows the initiation of an exception is not dependent on workflow tasks.

In contrast, *unexpected exceptions* are those which cannot be predicted until their first occurrence [SO00]. They are divided into (i) system failures and (ii) semantic failures. *System failures* are those avoiding to meet the process goal. Thereby, the workflow system is able to recover from system failures by program abortion for example. *Semantic failures* lead to an instance which is unable to proceed the process model through application or process level failures. „Thus the workflow system is unable to cater for [...] special circumstances“ [SO00, p. 7].

This taxonomy is especially useful for the modeling part of this thesis, since it provides distinct possibilities to integrate exceptions into a process model. According to the *quality of software* [Gla92] it is essential to divide the behavior of a process or program into normal behavior and exceptional behavior. Thus, the quality levels of design and specification of business process models can be increased by the *Software ilities* [Fil98; Voa01]. Furthermore, classifications may regard the organizational context in which exceptions appear. This leads to a classification according to organizational boundaries.

3.2.5 Exception Classification According to Organizational Boundaries

Finally, the last of the four discussed proposals for exception classification, is to divide exceptions according to organizational boundaries [CD97; KR98b; KR98a; CLK99; LSKA03; RDB03]. Thereby, two types of exceptions can be distinguished: (i) cross-enterprise or intra-workflow exceptions and (ii) cross-organizational or inter-workflow exceptions.

Cross-enterprise exceptions are further refined into three categories: (i) infrastructure exceptions, (ii) application exceptions and (iii) workflow exceptions. *Infrastructure exceptions* are caused

by the underlying system through hardware, network or communication errors. *Application exceptions* are also referred to as *logical exceptions*. They are application-specific within the workflow such as database login errors. *Workflow exceptions* are also referred to as *business exceptions*. They include *systems* and *user exceptions*. System exceptions (or *runtime exceptions*) are those not expected by the workflow, while user exceptions are specified by the workflow designer. „Due to the heterogeneous nature of exceptions in applications and infrastructure“ [LSKA03, p. 14], the corresponding exceptions are mapped to workflow exceptions.

Cross-organizational exceptions can be one of the above mentioned exceptions. A key aspect of them is, that they can „affect the outsourcing fulfillment, and [. . .] may not be handled alone in one outsourcing partner“ [LSKA03, p. 14].

This taxonomy provides a general view on exceptions which can be used also in interaction modeling. Although, some of the aspects mentioned in previous sections are missing, an essential insight is achieved according to cross-organizational exceptions: they may not be handled alone in a single participant!

The former sections introduced several exception classifications. Although every taxonomy has its special advantage, it turns out that none of them provide an overall view on exceptions in the context of choreographies. Thus, Chapter 5 combines these approaches to a more general picture of exceptions.

3.3 Exception Handling

After the previous section discussed exception classifications, this section investigates possible exception handling strategies. In the area of exception handling there are also different approaches.

Most of them, dealing with *expected exceptions*, (i) mainly look at individual processes, while (ii) some research is more related to cross-organizational settings. Out of the first category three approaches are of special interest in this thesis (cf. sections 3.3.2 to 3.3.4 on pages 30–32): (i) [LSKA03] discusses exception handling from a *functional view*. (ii) [Sti+98; GFJK03] focus on the different *continuation behavior* while (iii) [SW95; CCPP99] focus on a more declarative approach using *rules*. Further, a declarative approach using several *patterns* is proposed by [RH06a; RH06b]. The second category dealing with cross-organizational exception handling is discussed in Section 3.3.5 on page 32.

Although, not exhaustive discussed in this thesis, finding handling strategies for *unexpected exceptions* is a significant problem [HT04]. For this purpose *knowledge based solutions* [KD00] and approaches using *case based reasoning* [LSKM00] are proposed. Especially in the area of unexpected exceptions investigations cope with adaptive or evolutionary workflows. [CLK99; CLK00] use *ECA rules* for adaption which provide the opportunity for a reuse in multiple scenarios. Furthermore, [HBM08] uses *petri nets* which are adapted using a separate exceptional flow. [AHE05] uses worklets and *ripple-down rules* [CJ88] for dynamic workflow evolution. *Worklets* are small, self-contained processes handling specific tasks in a larger process [AHE05].

Ripple down rules „comprise a hierarchical set of rules with associated exceptions“ [AHE05, p. 4]. Due to the fact that organizations lack handling strategies for unexpected exceptions [Mou+03; MA07] propose unstructured human intervention to overcome these situations. They use the concept of *map guidance* which provides users with contextual information about the WfMS and environment, further it allows the interruption of model control and supports collaborative exception handling [MA07].

3.3.1 Three Phases of Exception Handling

Before the discussion of exception handling strategies a short introduction into the three phases of exception handling is given. A key insight of the discussed exception classification (cf. Section 3.2 on page 25) is that „it is only possible to specify handlers for expected types of exception“ [RH06b, p. 4]. But nevertheless, there are possibilities to cope with unexpected exceptions as well.

According to [BM99] and [MA05] three functions of exception handling can be distinguished: (i) exception detection, (ii) situation diagnosis and (iii) exception recovery actions.

Exception detection is clearly event driven, aiming on the occurrence of an event as a given exception [SO00]. Thereby, three levels of occurrence can be designed according to [SO00]: (i) *task level*, (ii) *block level* and (iii) *process level*. Focusing on task level failure conditions of completed states, *control flow* can be used (cf. Section 3.2.4 on page 27). „Semantic failure occurs when an instance is unable to proceed according to the given workflow model“ [SO00, p. 7]. If workflows end in the terminated state, usually semantic exceptions can be diagnosed by *exception rules*. In cross-organizational settings a single workflow process cannot completely provide the fault data. Therefore, „exception workflow may be triggered through external notification“ [SO00, p. 9]. Exception detection can be either manual or automatic and is usually integrated in the workflow engine. Further investigations dealing with exception detection can be found in [SO00; MA04].

Exception diagnosis establishes information about possible exceptions which are then handled using recovery actions. Thereby, the scope can be either process specific, when only a small set of instances is affected, or cross-organization specific, when large or different sets of instances are affected [MA05]. „The assessment of the event type is mandatory, because it directly impacts the handling phase“ [MA05, p. 7]. Similar to the discussed classification in Section 3.2, these events can be data, temporal, workflow, external, noncompliance¹ and system/application events. These events can have different impacts on the organization, their rules and goals as well as different reaction times and complexities of the solution. „Finally, the event type dimension does not have a clear relationship to the recovery strategy. For instance, a timeout does not imply the reaction time should be quick“ [MA05, p. 10].

¹„Correspond to situations where the [...] process [...] deviates from the model [...] or the model is not applicable to a particular context.“ [MA04, p. 6]

Recovery actions handle exceptions by modifying task assignments or skipping a task as presented in [CLK99]. The following sections describe different perspectives of such exception handling strategies.

3.3.2 Exception Handling According to Functional Views

According to a functional point of view exception handlers are divided into four categories as presented in [KD00; LSKA03]: (i) trivial, (ii) basic functionality, (iii) special situations and (iv) transactional environment.

Trivial is further refined into (i) *ignore* an exception by taking no actions to handle it, or (ii) *record* it when it occurs.

Basic functionalities are the termination, suspension and resumption of processes. „They provide the basic functionality for supporting workflow exception handling“ [LSKA03, p. 11].

Special situations usually work with poor performance, but in special situations they are available with good performance. For instance, solutions working only within transactional environments or a retried request to a database server that is restarting [LSKA03]. Thus, exception handler in special situations are: (i) compensation, (ii) procedural exception handler, (iii) propagation, (iv) re-execution (or rework), (v) pipeline or (vi) reallocation.

Compensation can only be used within completed state when an action compensating the effects is available. According to [LSKA03] compensation can appear in three flavors: (i) *complete* compensation applies when the effects of an operation can be totally removed, (ii) *partial* compensation allows effects only to be undone partially. Finally, (iii) *no* compensation indicates that no action can be taken to decrease the effects. *Procedural exception handlers* involve a series of steps to handle exceptions. *Propagation* is used, „if no local handlers are available then exception is routed to another workflow component, e. g. an exception handling coordinator“ [LSKA03, p. 11]. *Re-execution* allows a complete or partial *retry* of a repeatable action. Instead of retrying the execution of a task, an alternative task can be executed. *Pipelining* allows a partial release of results to increase concurrency. Using pipelining therefore increase parallelization by increasing the runtime overhead to ensure the previous sequential execution. *Reallocation* is similar to re-execution with the difference that for example an activity is assigned to another provider.

Finally, *transactional environments* such as forward and backward recovery has „proved successful in enforcing fault tolerance in closed distributed systems“ [TIRL03, p. 3]. Early solutions of exception handling are based on variations of transactional mechanisms. *Atomic transactions* are introduced by [Lam81]. They provide the same results as concurrent executing transactions by a serial execution order. The first who identified the compensatable, retrievable and pivot transaction was [MRKS92]. They divide each global transaction into a number of subtransactions. Thereby, the *compensatable transaction* is a subtransaction whose execution can be undone while the *retrievable transaction* is a subtransaction that can be retried [MRKS92]. The *pivot transaction* is subtransaction that is neither retrievable nor compensatable. According to [AL90; LR00; ACKM04] there are two main classes of error recovery: backward and forward

recovery. *Backward recovery* is based on rolling system components back to a previous correct state to undo partial execution if it is not possible to complete a workflow. In contrast, *forward recovery* tries to transform the system into any correct state. For this purpose, the „WfMS maintains the workflow instance execution state in a persistent storage“ (logging) [ACKM04]. A *coordinated nested transaction* model for workflow execution allowing forward, backward and partial recovery is proposed by [RS95]. Also [Ley95; LR97; LR00] use a transactional model based on the notion of *compensation* and *atomicity spheres*. In their *Workflow Activity Model* (WAMO) [EL95; EL98] enrich the conceptual workflow model by transactional properties for tasks. The exception handling is achieved using these transactional properties. *Flexible exception handling* is proposed by [HA98; HA00] using an integration of programming language concepts with advanced transaction models. In [HA99] a subscription mechanism is used for inter process communication. Further, advanced transactional concepts such as *Sagas* [GMS87] and *nested transactions* [Mos81], not discussed here, are explained in detail in [LR00].

Especially in the context of web services a number of exceptional situation cannot be managed with backward or forward recovery. They are too expensive in terms of lost work and usually „the WfMS is incapable of discriminating among the different causes of semantic failure“ [CCPP99, p. 37]. Thus, transactional environment has to be extended to deal with expected exceptions as in [Sti+98; WWJP01]. For instance, [WWJP01] introduces three kinds of atomicity: (i) strict atomicity, (ii) alternative atomicity and (iii) exception atomicity. In contrast to [Ley95] who approaches atomicity from a persistence perspective, [WWJP01] defines atomicity as a property of the workflow. *Strict atomicity* specifies the basic atomic behavior, i. e. the *all-or-nothing* semantic. *Alternative atomicity* defines exclusive execution of atomicity spheres, i. e. the *exactly-one* semantic. Finally, *exception atomicity* allows the violation of the atomicity constraint. Thereby, an exception task is executed when one or more tasks do not execute.

This statement is also grounded by earlier investigations in the database context of long-lived computations [RS95]. One possibility for an extension of transactional environments is the exception handling according to the continuation behavior.

3.3.3 Exception Handling According to the Continuation Behavior

Exception handling in terms of the continuation behavior is addressed by [Sti+98; GFJK03]. Both divide the exception handling into two categories: (i) continuation exception handling (or non-cancellation) and (ii) abortion exception handling (or cancellation). The key insight of this classification is the fact, that long running processes involving many collaborators should only be canceled as a last resort [GFJK03].

Continuation exception handling „describes the types of behavior required when an event causes a business process to deviate from normal process but in a less serious way that does not require cancellation“ [GFJK03, p. 4]. For instance, pausing or returning to normal processing, the execution of alternative activities, rework (i. e. retry as well as rollback and redo) and continue processing with an additional process is possible.

Abortion exception handling describes the required behavior of a deviation that results in a cancellation. Handlers of this category are mainly related to transaction environment using backward and forward recovery. For instance, termination and compensation handling as well as executing independent activities or activities dependent on state are in this category. Furthermore, human intervention is used as last resort.

Concluding, this taxonomy addresses the need of two distinct exceptional strategies. This understanding is especially useful regarding the modeling part of this thesis. However, exception handling according to the continuation behavior focuses on constraints local to a special object (e. g. scopes). Another possibility for an extension of transactional environments is especially useful for handling violations of global constraints using *Event Condition Action* (ECA) rules.

3.3.4 Rule Based Exception Handling

ECA rules are well discussed in the area of database technology. Thus several attempts are made to use this concept within the workflow context [BBKZ93; GFV96; Cha97; KR98c; CCPP99; CLK99]. The key idea of this area is that exception handling „should interfere as little as possible with regular workflow processing“ [CCPP99, p. 3]. For this purpose, the workflows are guarded by events which trigger rules having a statically defined priority.

In [CCPP99] rules are executed in transactional contexts. These contexts are different to the context in which the exceptional event was generated (i. e. *detached execution mode* [Elm92]). Rules can be either *targeted* to a workflow (i. e. their side effects are only propagated to the tasks and cases of that workflow) or *untargeted*. The side effects of *untargeted rules* (or global rules) affect all the cases of all workflows managed by the WfMS. [CCPP99]

Exceptions can be rather common in a WfMS. Hence, reusing exception handlers increase effectiveness, user-friendliness and efficiency of the WfMS. ECA rules in the context of exception handling are especially useful for handling violations of global constraints. [CLK99] The interesting point with ECA rules is twofold: On the one hand, ECA-events are similar to events in BPEL and BPMN. Thus, the concept can be easily adapted to these contexts. On the other hand, the idea of the detached execution mode of exception handlers seems to be perfect for interaction exception handling, since (i) exceptions spanning a whole choreography can be handled and (ii) exception handlers (e. g. as exception handling processes) can be reused in other choreographies. Thus reducing cloned exception handling code.

3.3.5 Cross-Organizational Exception Handling

Having discussed three approaches dealing with exception handling of individual processes, this section focuses on cross-organizational related literature. More precise, this section should be named „Exception Handling in Interaction Models“, but due to the missing attention of this problem in current research, the focus has to be widen also to (i) process oriented exception handling and (ii) decomposition of processes and (iii) transactional workflows.

In contrast to cross-enterprise exceptions as introduced in Section 3.2.5, the exception handling strategy of cross-organizational settings is not as simple as the basic try-catch mechanism known from traditional programming languages. For this reason, the notion of *process oriented exception handling* is introduced by [LSKA03]. Process oriented exception handling uses an *exception handling coordinator* to achieve a flexible exception propagation through the processes.

Although, not tightly related to choreographies, there are approaches dealing with *decomposition of processes* [KL06]. Common insights of such investigations are interesting in the focus of this thesis since decomposed processes have many similarities with choreographies in general. Instead of joining processes together into a choreography, processes are split into different fragments which are then able to run on different process engines [Ruf07]. This fragmentation can be achieved through two approaches according to the coordination: (i) distributed (or decentralized) coordination and (ii) centralized coordination. Using *distributed coordination*, the coordination code is split into chunks distributed inside the fragments [NCS04; CCKM05]. In contrast, a *central coordinator* as in [KL07; Pal07] can also control the decomposition. [Ruf07] mentions, that process fragmentation (i. e. the resulting group) has many similarities with choreographies on the one hand, while on the other hand many new problems arise. The main problem is guided by the observation that split processes belong to one company while a choreography usually involves many companies.

Again, notable work has been done in the context of *transactional workflows* [SR93]. [KR98a; KR98b] use opportunistic compensation and re-execution to eliminate unnecessary recovery overheads based on a rule driven approach. They distinct between *complete* and *partial* compensation and *complete* and *incremental* re-execution. Thereby, a complete compensation or re-execution is necessary when previous execution steps are useless. Otherwise, incremental re-execution or partial compensation is sufficient for an equivalent effect. For dependencies between compensation, *compensation dependent sets* customize the order of step compensation. The difference to *compensation spheres* [Ley95] is that compensation spheres result in a compensation of all steps, including unnecessary ones. [TIRL03] uses *coordinated atomic actions* (CA actions) [Xu+95], based on *distributed transactions* [GR93] and *atomic actions* [AL81; CR86], as natural structuring unit for complex exception handling in distributed object systems [XRR98]. „Distributed transactions use backward error recovery as the main fault tolerance measure in order to satisfy completely or partially the *ACID* (atomicity, consistency, isolation, durability) properties“ [TIRL03, p. 3]. Thereby, atomic actions control cooperative concurrency and implement coordinated error recovery while ACID transactions maintain consistency of shared resources [TIRL03].

Regarding coordinated exception handling in the context of business processes [Ruf07] discusses a concept of distributed fault handling in choreographies. Thereby, the concept of *Cross-Partner Scopes* (CPS) [Ley05; Kop08b; KWL09] is introduced to BPEL4Chor. CPSs are discussed in detail in Section 5.4 on page 56. In combination with a coordinating entity CPSs are used to provide choreography-wide exception handling by simulating CPSs through coordinated BPEL scopes. Thus, [Ruf07] uses fragmented distributed coordination while the approach taken in this thesis uses non-fragmented coordination in combination with a kind of distributed-centralized coordination.

Notice, that in related literature the concept of *scope* is used as a synonym of *sphere* or *region*. For the sake of consistency, the term *scope* is used in the rest of this thesis.

Interested in cross-organizational settings in erroneous situations, it turns out that current WfMSs lack proper Fault, Compensation and Termination handling for choreographies (FCT handling) [GFJK03]. Since this thesis focuses on *exceptions* in interaction choreography models (using BPEL terminology), a detailed investigation of compensation and termination handling in cross-organizational settings is not provided. However, compensation and termination handling in such contexts is discussed. Readers interested in these topics are referenced to [Wan09] who focuses on compensation handling using a generic scope concept. Termination handling in inter-organizational workflows is discussed by [Lud99; LSBG99].

3.4 Interaction-centric Exception Modeling

Having discussed what kind of exceptions can appear and how exceptions can be handled, a view on modeling solutions is needed to give an idea on how exceptions can be modeled on interaction level. As a consequence of the missing attention of interaction exception handling, also modeling interaction exceptions is marginal targeted in literature.

Modeling interaction-centric exceptions is influenced also by traditional programming languages (such as *Java* [Jav05]) as well as software process modeling languages (such as *SPELL* [Con+92] or *APPL/A* [SHO95]). However, due to time and space limitations only business process modeling languages are considered in this thesis. Thereby, this section evaluates which interaction-centric language is suitable for an extension to interaction exception modeling.

Section 2.2 on page 16 introduces BPMN [BPM09] and BPEL4Chor [DKLW07] as languages supporting the interconnection modeling style. Furthermore, iBPMN [DB07] was introduced as an extension for interaction modeling using BPMN. In addition to iBPMN, there are several notations supporting the interaction modeling style. For instance, the *WS-Choreography Definition Language* (WS-CDL) [WC05b] and *Let's Dance* [Zah+08]. For a detailed discussion of other (interaction) modeling languages not regarded in this thesis refer to [Dec06; KL08]. As formal model for describing and analyzing interaction models *Interaction Petri Nets* (IPN) [DW07] are proposed. For the special purpose of *event-based exception handling* [HA98], [HBM08] extends petri nets to *Self-adapting Recovery Nets* (SARN). Thereby, exception handling is modeled by using special *recovery transitions* associated with exceptional events within the business process. SARNs are discussed in detail in Section 6.1 on page 72.

Chapter 1 already mentions the three parts process models cover: (i) the essential components of processes, (ii) the nature of their interaction and (iii) the manipulation of their independent and collective behavior, especially in erroneous situations. Especially with part (iii) it turns out that exception handling is mostly disregarded in interaction models. For this purpose, only WS-CDL provides support for exception handling but lacks a graphical representation. Although WS-CDL is a W3C standardized, „expressive and rich language for modeling global view of the composition“ [Ast08, p. 17], there are a lot of open problems with it (cf. [BDO05; DOZ06]). According to [Ast08] these problems affect the usage of coordination protocols, the

check of information alignment and choreography coordination, correctness and conformance of choreographies as well as the generation of correct behavioral interfaces. For instance, [LHZP07] proposes formal semantics for WS-CDL, but the couple of problems lead to a wide spread rejection.

Let's Dance, as second presented interaction language, focuses on control flow specification by abstracting from concrete message formats [Wes07]. Although having a graphical interface, „it is very different to that of established process modeling languages“ [DB07, p. 2].

Conceptually, exception handling in choreography diagrams may be possible with BPMN2.0. Especially with respect to non-interrupting events, this may be interesting. However, BPMN2.0 does not provide any information about exception handling in these diagram type. Furthermore, BPMN2.0 does currently not provide a theoretical foundation by having a translation to a formal model. Additionally, a clear description is missing dealing with the generation of observable behavioral models. Thus, BPMN2.0 provides a graphical interface but lacks (currently) a semantic for this purposes.

As a consequence this thesis chooses iBPMN as preferred language of investigation for the following reasons: iBPMN provides a graphical interface and therefore is qualified also for high level modeling. iBPMN has a clear theoretical foundation using the proposed translation to IPNs of [DB07; Dec09a]. Observable behavioral models using BPMN can be generated out of iBPMN.

4 Motivation of the Approach and Running Example

After the previous chapter gave an overview on related work, this chapter motivates the task of this thesis before the next chapter discusses exception handling in interaction choreographies. „Choreographies have a central role in ensuring interoperability between process orchestrations“ [Wes07, p. 227]. Furthermore, they „facilitate and automate business process collaborations both inside and outside enterprise boundaries“ [Yen03]. Several industry initiatives try to establish standardized choreographies in special domains. For this purpose, they define *collaboration rules* that companies need to comply to collaborate with each other. For instance, *RosettaNet* (<http://www.rosettanet.org>) deals with supply chain management, *SWIFTNet* (<http://www.swiftnet.co.uk/>) copes with financial services and *Health Level Seven (HL7)* (<http://www.hl7.org/>) focuses on health care services.

Collaboration rules serve as reference for collaborations and are specified by process choreographies. Thereby, „costs for the individual companies are reduced“ [Wes07, p. 227], since the overall interaction behavior does not need to be designed for every single business partner. Knowing these rules, new companies can join the market more easily. Although, standardized choreographies are important in their particular domains, they cannot cope with the definition of new types of business-to-business collaborations in a flexible way [Wes07]. Thus, new concepts for process choreography design and implementation are defined (cf. chapter 5 of [Wes07]). Nevertheless, these concepts do not include the notion of exceptions spanning multiple participants or intermediate events in the context of choreographies. For instance, it is not possible to handle an exception in *company 1* raised in *company 2* in a well-defined manner, or to use intermediate events in interaction models. Thus, choreography models still lack flexibility although they are more suitable than standardized choreographies in general. Additionally, standardized choreographies can benefit from exceptions and intermediate events on interaction level.

4.1 From Orchestration to Choreography

The collaboration design can be achieved through two approaches (cf. Section 2.1 on page 15). *Process orchestrations* describe collaborations from the view of single participants which are related by message flow. Thereby, several modeling errors can appear which are referred to as *anti-pattern* [Dec09a, p. 118]. Anti-pattern can be distinguished into three categories dealing with (i) *decision making* such as incompatible branching behavior, (ii) *ordering constraints* such as contradicting sequence flow (deadlock) and (iii) *process instance creation and termination*

such as optional participation or not-guaranteed termination. For instance, Figure 4.1 shows a deadlock of interacting process orchestrations, referring to the second category.

The first activity of *company 1* is a message receive activity waiting on a message from *company 2*. However, *company 2* itself waits on a message from *company 1*. As a consequence, each of the process orchestrations cannot proceed—a classical deadlock—(assuming synchronous communication, i. e. the receive activities are blocking). Although, each process provides a valid behavior on its own. The deadlock arises due to links between activities forming the interaction between process orchestrations. As the example illustrates, the viewpoint of *individual process orchestrations* is unsuitable to model the interaction between process orchestrations. It is important to notice, that each participant only has its local view on the collaboration, while the figure shows the global view to illustrate the problem. Thus, *company 2* is colored gray. To overcome these problem, interaction choreography models provide a truly global perspective on the interactions between process orchestrations. [Wes07, p. 228 ff.]

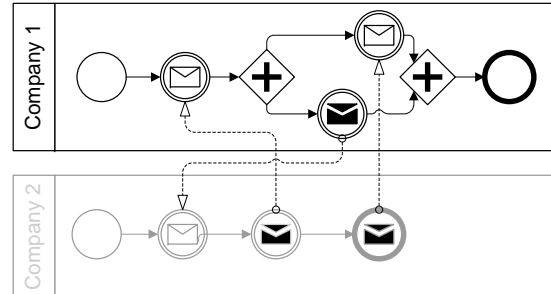


Figure 4.1: Deadlock of interacting process orchestrations according to [Wes07]

The most prominent anomalies of interconnection models are compatibility and conformance. *Compatibility* [Mar03] answers the question of whether a number of process models can interact successfully. In contrast, *conformance* [Aal+06] answers the question of whether a process model is valid according to a given specification. Since modeling decisions are only reflected once in interaction models, incompatibility is largely avoided. However, new problems arise when using interaction models. *Realizability* [FBS04] covers the possibility to construct a set of observable behavior models that exactly show the interaction behavior specified in the choreography. A relaxed criterion is given through *local enforceability* [DW07]. Thereby, only a set of observable behavior models is required to show a subset of the specified interaction behavior in combination with a reachability requirement for all interactions. To avoid a contradicting view on the global conversation state *desynchronizability* [DBKL08] requires a set of observable behavior models to be compatible under the assumption of asynchronous communication. [Dec09a, p. 144 ff.]

4.2 Process Choreography Design

In today's business landscape, companies increasingly provide their products to the market as services using the SOA paradigm. These independently developed and interoperable services are loosely coupled within and across organizational boundaries. Thereby, they can form quite complex collaborations having a big impact on the operational business of the company. As a consequence, collaborations should be designed very carefully.

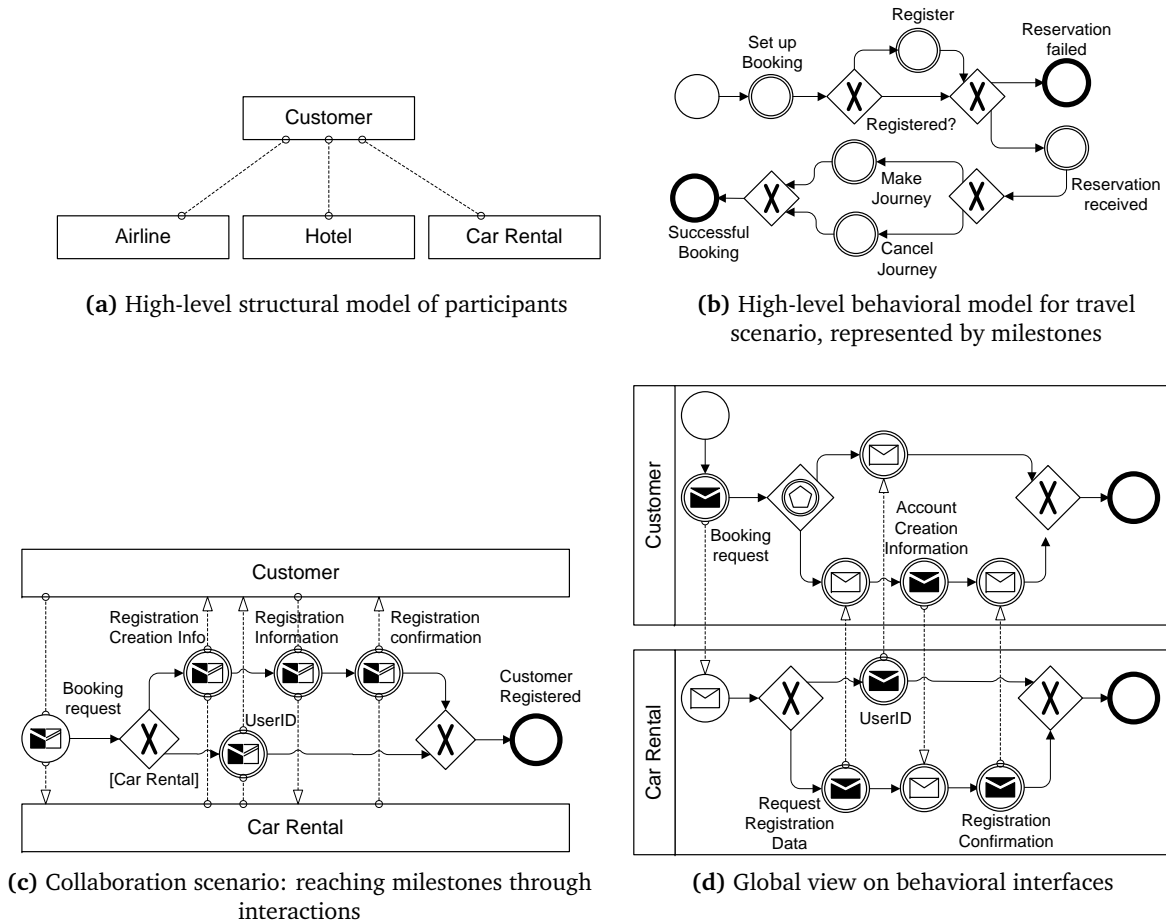


Figure 4.2: The choreography design process of the travel scenario

According to [DD04; Dec06; BDD06; Wes07] the top-down approach of process choreography design involves four kinds of activities developing several design artifacts: (i) *high-level structure models* identify the collaborating participant roles and their communication dependencies. (ii) *high-level behavioral models* specify the global synchronization points (*milestones*) of the collaboration and their ordering constraints. (iii) *collaboration scenarios* „focus on behavioral dependencies between interactions“ [Dec06, p. 56]. They are also called *global interaction models*. Scenarios are kept small by representing only single milestones. (iv) *behavioral interfaces* represent the individual view of specific participants in the choreography. Thereby, „the internal aspects of the own process orchestration, as well as the interactions involving only other participants, are disregarded“ [Wes07, p. 237]. Figure 4.2 shows the complete choreography design process of the running example which is explained next.

4.3 Running Example

The running example in this thesis is a fictitious *travel scenario* where a *customer* wants to make a journey. Thereby, the *customer* needs three participants providing the necessary services for his journey: (i) the *airline* providing the flight, (ii) the *hotel* where he can sleep and finally (iii) the *car rental* to drive around.

Having identified all participant roles and their communication dependencies, the structural model can be derived as in Figure 4.2a on the preceding page. Next, the milestones of the collaboration and their ordering constraints can be figured out. Figure 4.2b shows a behavioral model with the different outcomes: *successful booking* and *failed reservation*. Further, five intermediate milestones (surrounded by a double circle such as BPMN events) are identified. These are (i) the booking request itself, (ii) an optional registration if the *customer* does not have an account yet. Having requested a booking as registered *customer*, (iii) the reservation is either confirmed or not. If the reservation is confirmed, the *customer* can either (iv) *make* the journey or (v) *cancel* it for some reasons. As next, collaboration scenarios can be designed by specifying the dependencies between interactions. For instance, Figure 4.2c shows a collaboration scenario to achieve the *register* milestone. Finally, the behavioral interfaces can be derived out of the collaboration scenarios for the involved participants. For illustration, Figure 4.2d shows a part of the global view of the behavioral interfaces of the *customer* and the *car rental*.

Concluding, the *customer* has to book a *flight*, a *hotel* and a *car* before he can make his journey. Additionally, the *customer* can cancel the journey for some reason. Using traditional choreography modeling without a special exception handling mechanism involving multiple participants, the cancellation has to be modeled explicitly by sending a message to every involved participant as shown in Figure 4.3 for the *car rental*. As the example illustrates, this leads to more complicated and hard to understand and maintain process models, since exceptions as defined in Section 3.1 on page 23 are *embedded* in the normal control flow.

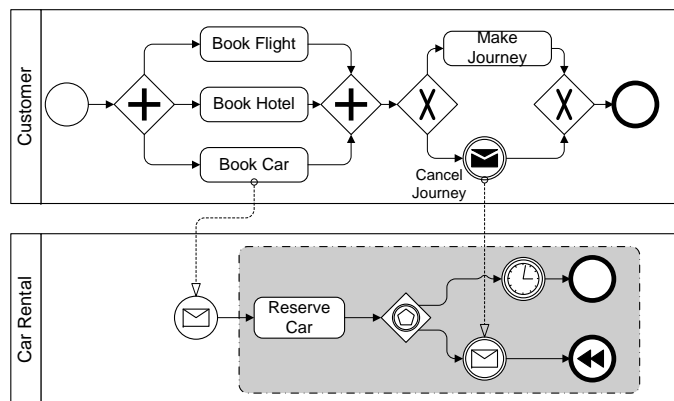


Figure 4.3: State of the art simplified travel scenario choreography with embedded exceptions

The general idea of collaboration models is to couple closely together parts of different choreography participants sharing some sort of common ground. Thus, collaboration models avoid anti-pattern. At this point, the interesting observation is that exceptions also introduce anti-pattern, since they span multiple participants in choreographies. Hence, a concept of exception on this level would simplify choreography models and follow the general idea of collaboration models. For instance, the simplified choreography may look like shown in

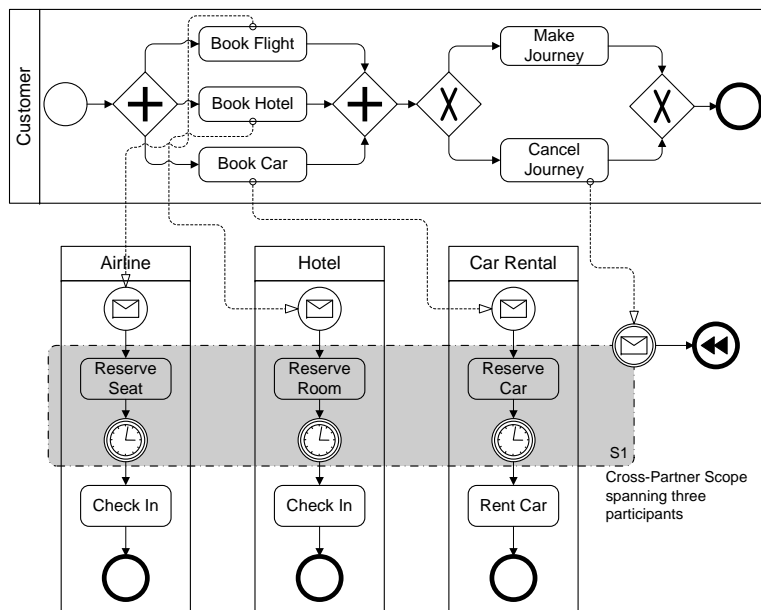


Figure 4.4: Simplified travel scenario with cross-partner scope adapted from [Kop08a]

Figure 4.4. A *Cross-Partner Scope (CPS)* (discussed in detail in Section 5.4) span multiple participants to gain the possibility of an easy understandable exception handling. As the figure illustrates, the *cancel* message is send to the scope itself. Thereby, using the mechanisms of the underlying runtime support of an externalized controlling entity. This frees modelers from specifying exception handling manually. Figure 4.4 uses the representation of interconnection models. When trying to move exception handling to interaction models several problems appear. All in all, this thesis investigates possibilities to gain such an exception handling in interaction choreography models to benefit from advantages of local exception handling on the global level.

4.4 Intermediate Events in Interaction Models

According to the definition of exceptions in Section 3.1, events constitute deviations from the normal behavior. Thereby, an interesting observation is that intermediate events affect the interaction behavior of process collaborations. Thus, they should be represented in interaction models. To avoid an increased complexity in the choreography design process of Figure 4.2, this may introduce an additional step between (c) and (d). Thereby, step (c) either remains as is and step (c₂) is enriched with exceptional constraints, or step (c) is reduced to high-level collaborations and therefore (c₂) is enriched with exceptional constraints. With respect to the collaboration and choreography diagram of BPMN2.0, the second approach is recommended.

For instance, take the *Buyer-Seller-Scenario* shown as *Interaction Petri Net (IPN)* representation (cf. Section 3.4 on page 34) in Figure 4.5a on page 43. IPNs are a petri net extension of

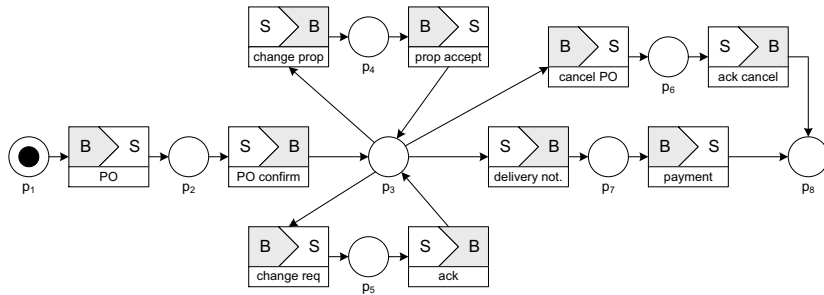
Let's Dance. Thereby, circles represent places which contain tokens marking a certain state. Rectangles represent transitions (interactions in Let's Dance) marking message exchanges. For example, the first transitions in Figure 4.5a represent an interaction from the *buyer* to the *seller* exchanging the *purchase order PO*.

In the *Buyer-Seller-Scenario*, a *buyer* orders some products from a *seller*. After the *seller* confirms the request, the normal control flow schedules the delivery notification and the payment. Additionally, three intermediate events are specified after the confirmation: (i) a change-request from the *buyer*, (ii) a change-proposal from the *seller* and finally (iii) a cancellation of the request by the *buyer*. These events may occur as long as the purchase is not finished (i. e. *delivery* and *payment* are executed). For simplicity reasons, these three events are omitted at place p_7 .

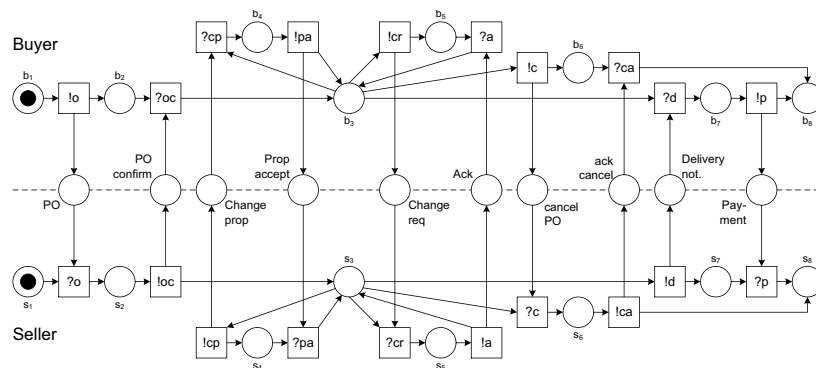
As the figure illustrates, intermediate events lack a proper representation in IPNs (and thus Let's Dance), since the semantic of intermediate events cannot be represented due to both missing notational difference of events and runtime support according to the BPEL semantic of event handling. For instance, *cancel* and the two *requests* have event character, but can only be modeled as normal interactions. Hence, the only fact indicating an event is the fact that the flow of control is split at a certain place. However, regarding the *cancel* event, this may be also a normal control flow branch.

As a consequence, this camouflage of events leads to race conditions (i. e. which event comes first) regarding asynchronous communication. For this purpose, desynchronizability considerations try to resolve these problems by weakening the main property of interaction models: the interaction atomicity. This leads to even more confusing and hard to understand models. Figure 4.5b shows the desynchronized variant of the scenario. Therein, a deadlock appears since weak termination is broken: when a token is in places b_6 and *delivery notification* no other transition can fire. After resolving the deadlock by introducing additional transitions, the scenario looks even more confusing as shown in Figure 4.5c.

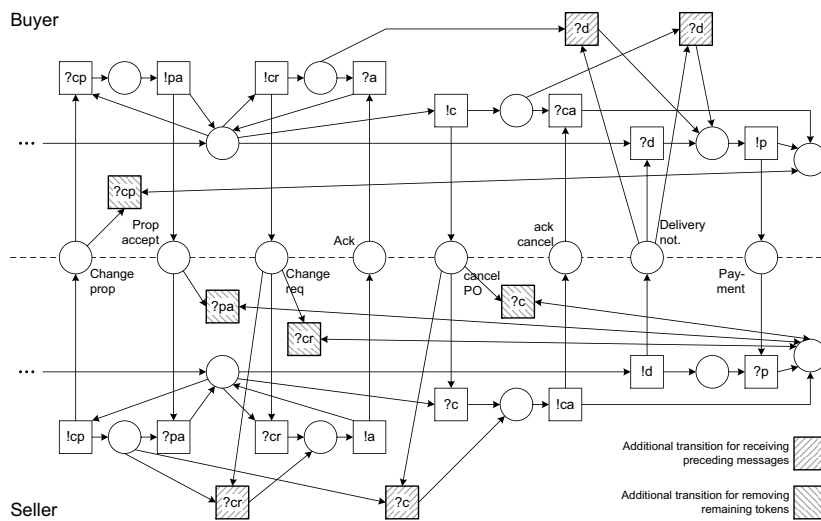
Since intermediate events form deviations from the normal control flow (also known as *happy path*), they should have a distinct notational difference. Thus, one task in this thesis is to investigate the usage of intermediate events in interaction models. To demonstrate the goal of the thesis, the model of Figure 4.5a is transformed into a simplified iBPMN model shown in Figure 4.6a, which is then refined in Figure 4.6b using *backward associations*. Backward associations are not part of iBPMN and are only used to demonstrate to intended semantic of the model in this refinement step. Finally, Figure 4.6c uses a BPEL/BPMN-like semantic of scopes and intermediate events. As the figure illustrates, the intended semantic is represented in a more elaborate way as before. Thereby the intermediate events are represented at the bottom of the scope and the error event is shown at the right of the scope. Thus, the model represents the semantics of the original model in an easy understandable manner. As a consequence to this observation, this thesis provides an investigation of possible exceptions, their handling and runtime semantics. Thereby, the following chapter introduces theoretical aspects of exceptions and their handling in interaction choreography models guided by a combined classification of exceptions in interaction models.



(a) Buyer-seller-scenario in interaction petri net notation using events (simplified)

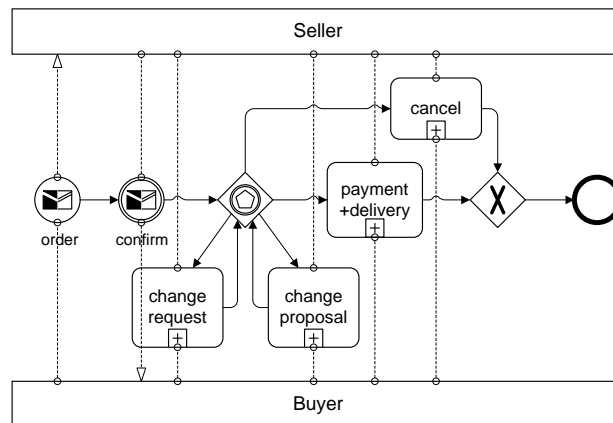


(b) Desynchronized buyer-seller-scenario: atomic interaction property destroyed

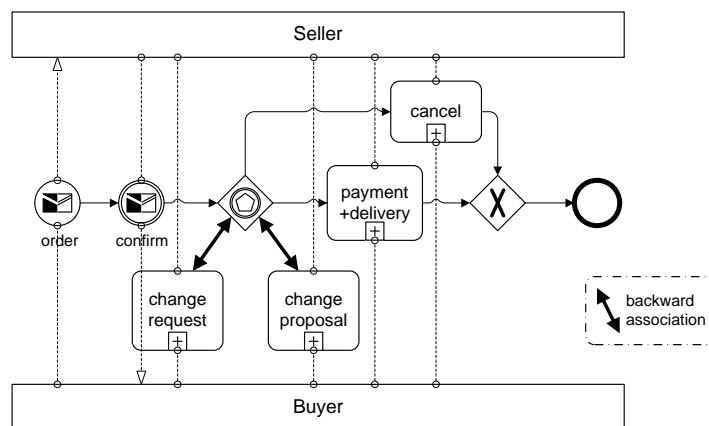


(c) Resolved buyer-seller-scenario: additional transitions added

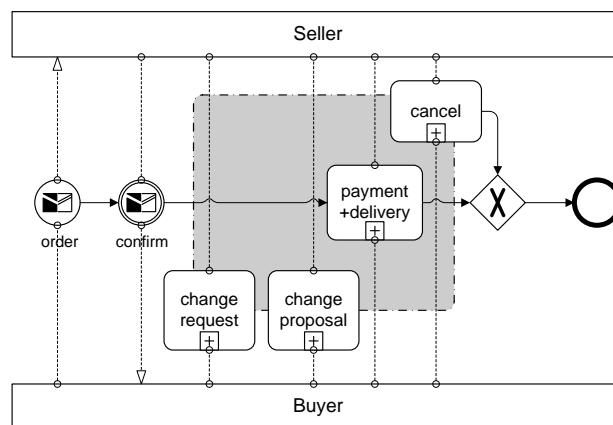
Figure 4.5: Desynchronizability: race condition and deadlock resolving according to [Dec09a]



(a) Simplified buyer-seller-scenario in iBPMN



(b) Simplified buyer-seller-scenario using additional backward associations



(c) Simplified buyer-seller-scenario using attached intermediate events

Figure 4.6: Modeling attached intermediate events in interaction models

5 Exceptions and their Handling in Interaction Choreography Models

The previous two chapters discuss how exceptions can be captured and give a motivation of the task of this thesis. However, exceptions arise not only within single processes or orchestrations, but also between them. This is the case if they interact with each other in a business-to-business collaboration, typically by sending and receiving messages. Further, exception mechanisms of traditional sequential programs are not applicable without changes in the context of distributed exception handling, since „different components may raise different exceptions and exceptions may be raised simultaneously“ [XRR98, p. 2].

The goal of this chapter is to provide a theoretical foundation of exceptions in interaction choreography models. According to the requirements identified in the previous chapters, this chapter is organized as follows: First, a classification of exceptions in interaction models is given to provide a foundation of the investigation. Thereby, the granularity of exceptions is concerned with respect to the modeling artifacts. Second, requirements of exceptions as the need of *silent actions*, intermediate events and exception propagation in interaction models are discussed. Due to the importance of choreography-wide scopes, they are discussed in a separate section. Having defined a classification of exceptions as well as the requirements, the next section deals with exception handling considerations. Thereby, focusing on strategies to gain a coordinated way of exception handling. Finally, the last section of this chapter evaluates the proposals according to their usage in both top-down and bottom-up choreography modeling.

5.1 Classification of Exceptions in Interaction Models

Exceptions are the *bread and butter* to achieve fault tolerant processes. In order to gain fault tolerance also in combination with interaction choreographies, this section attempts a general classification of exceptions in interaction choreography models based on the discussed proposals in Section 3.2 on page 25.

The classification shown in Figure 5.1 is based on the fundamental proposal of [EL95] using perspectives. Thereby, *deviations from the normal behavior* are split into two distinct categories: (i) exceptions and (ii) failures.

Exceptions may be solved using a particular strategy or allow the choreography modeler to react to failures. According to organizational boundaries, exceptions in interaction models can be either handled totally internal to a process, referring to *intra-process exceptions*, or signaled to the choreography environment, referring to *inter-process exceptions*. Exceptions are

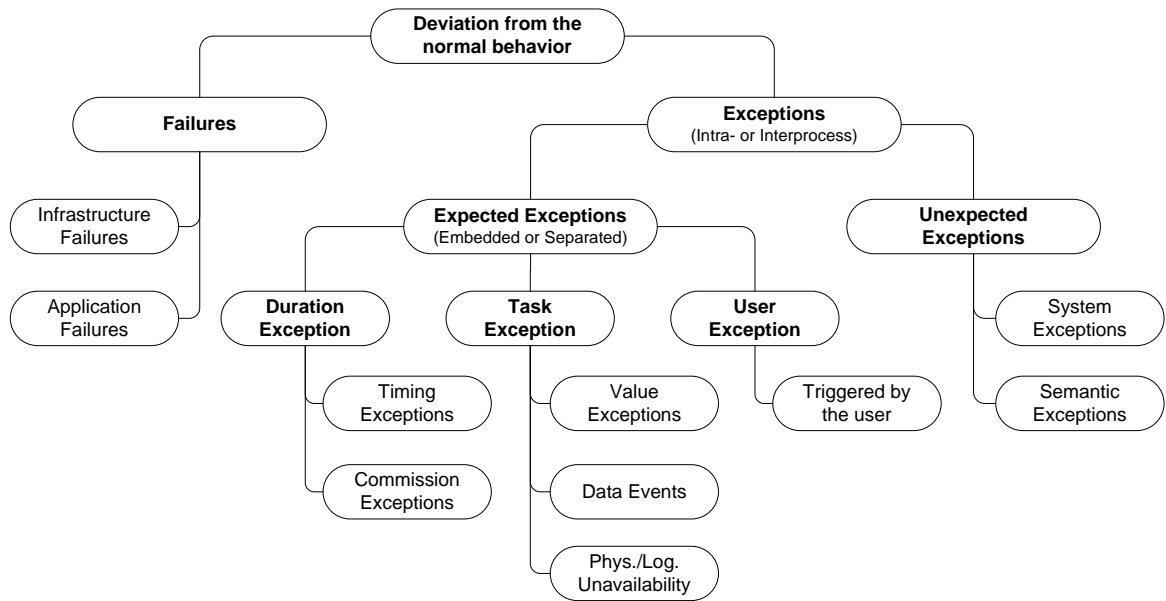


Figure 5.1: Combined classification of interaction exceptions

further refined into those already known to the modelers, referring to *expected exceptions* and those totally unknown to the modelers, referring to *unexpected exceptions*. With respect to the modeling, only expected exceptions can be modeled. This modeling can be achieved due to both *embedding* exceptions into the process model or *separating* them by a distinct exception logic (cf. basis of integration [SO00]). Since exception handling is part of the business logic of a business process it may end up dominating its normal behavior [HA00] and thus be separated instead. Expected exceptions are refined into three categories according to the source of occurring [Sti+98; HB04; HBM08]: (i) duration, (ii) task and (iii) user. *Duration exceptions* are those referring to timing constraints or commission exceptions. *Task exceptions* are those referring to value exceptions, data events or physical/logical unavailability. Finally, *user exceptions* are those triggered by the user. *Unexpected exceptions* include all aspects which cannot be predicted until their first occurrence [SO00]. Thereby, the most common exceptions are (i) system exceptions and (ii) semantic exceptions. *System exceptions* are those avoiding to meet the process goal while *semantic exceptions* lead to an unproceedable process instance.

Failures in the context of this classification, refer to problems that cannot be solved at modeling level, since they arise due to infrastructure or application problems. *Infrastructure failures* refer to failures handled at infrastructure level of the *underlying* infrastructure. *Application failures* refer to programming failures such as unexpected input in components of application programs [EL95].

5.2 Granularity of Exceptions in Interaction Models

When an exception is detected, exceptional execution replaces the normal control flow using exception handling mechanisms. For any given exception mechanism, *exception contexts* have to be defined. Exception contexts are „regions in which the same exceptions are treated in the same way“ [XRR98, p. 3]. *Exception handlers* associated with each context specify the reaction to a certain exception. According to the particular continuation behavior of the exception (cf. Section 3.3.3 on page 31), the corresponding process either continues or terminates. In cases where no particular handler is specified or a specified handler is not able to handle the exception, the exception will be propagated through a hierarchy. In traditional programming languages, this hierarchy is formed by procedure calls or nested blocks while in process choreographies this is somehow difficult.

The granularity of exceptions in interaction models is tightly related to the modeling artifacts of the used notation. For instance, exception contexts of BPEL are *single activities*, *groups of activities* (also known as *scopes*) and *the process itself*. In general, *scopes* are the main region of exception handling in BPEL. Regarding process choreography settings these three artifacts do not suffice, since collaborations involve at least two participants. A scope may affect only one participant (i. e. the scope is *local* to a given participant) or multiple participants (i. e. the scope is *global* to a given participant). In an analogous way, an exception may affect not only a single process but a group of processes. This is similar to a scope spanning a whole process. As last instance, an exception may not only affect a group of processes but all processes of a given choreography (i. e. the whole choreography). Out of these granularities, several requirements of interaction exceptions can be derived.

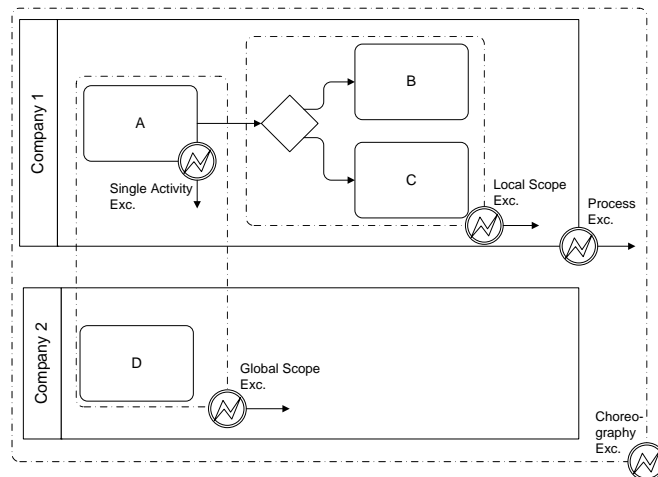


Figure 5.2: Granularities of interaction exceptions

As last instance, an exception may not only affect a group of processes but all processes of a given choreography (i. e. the whole choreography). Out of these granularities, several requirements of interaction exceptions can be derived.

5.3 Requirements of Exceptions in Interaction Models

The previous section introduces a classification of interaction exceptions and discusses their granularities. Thereby, five requirements can be derived out of the granularities that need a detailed discussion since they become difficult in interaction models and choreographies: (i) exceptions involving single activities, (ii) exception contexts, (iii) the representation of exceptions and finally, (iv) concurrent exceptions and (v) their propagation.

Exceptions involving single activities in the context of interaction models are interesting, since interaction models normally do not involve this kind of modeling artifact: they restrict the

modeling artifacts to atomic interactions and their control flow dependencies and ignore the internal behavior completely. For this purpose, Section 5.3.1 discusses so called *silent actions* and Section 5.3.2 focusing on *no action* constructs. *Exception contexts* refer to scopes on interaction level. Thus, exception contexts form the main point of interest in an investigation focusing on exceptions in interaction choreography models. Thus, this requirement is moved to a separate Section 5.4 on page 56. The *representation of exceptions* refers to the continuation behavior. For this reason, *intermediate events* are discussed in Section 5.3.3. *Concurrent exceptions* arise due to the interaction of independent participants and are discussed in Section 5.3.5. Due to the independent participants, additional mechanisms are needed to coordinate multiple exceptions representing a system-wide exception. *Exception propagation* in traditional processes is achieved through a chain of nested callers. However, regarding independent participants exception propagation may require additional mechanisms.

5.3.1 On the Need of *Silent Actions*

During the discussion of exception granularities in Section 5.2, the notion of single activity exceptions are investigated. Single activities in choreographies refer to the internal behavior of the choreography. Thus, single activities introduce a conflict in choreography languages since internal behavior is out of the scope of observable behavior. This conflict manifests in „an open question whether a choreography language should offer constructs to model internal behavior“ [KL09a, p. 1].

Out of the regarded notations for interaction modeling, only the standardized WS-CDL provides information: „The *silentAction* activity is an explicit designator used for marking the point where participant specific actions with non-observable operational details MUST be performed“ [WC05b, Sec. 6.5]. For instance, the way a *buyer* checks the inventory of a warehouse is normally not observable to other participants. Nevertheless, the inventory level influences the observable behavior and needs to be specified in the choreography definition [WC05b]. Additionally, Section 4.4 on page 41 discusses a *Buyer-Seller-Scenario*. Thereby, an IPN of this scenario is presented in Figure 4.5a on page 43. Changing the requirements of the scenario as follows, gives a second hint on the usage of *silent actions* in interaction models. If the participant events are limited to the state of the IPN where the purchase order is confirmed but the delivery has not started (p_3), an artifact is needed to represent this state. Also in the context of interconnection modeling using BPMN or BPEL4Chor, internal behavior can be modeled [KL09a].

As conclusion, [KL09a, p. 6] figured out that „it depends on the use-case of the choreography whether internal actions are needed“. This conclusion also manifests in an intermingled survey in [KL09b], where some people prefer the usage of *silent actions* while others refuse them. According to the conclusion of [KL09a], this thesis does not answer this question. However, this task is discussed in the modeling part of this thesis (cf. Chapter 6).

5.3.2 On the Need of *No Action* Constructs

When no activity is applicable at a point where an activity is syntactically required, the *no action* activity can be used [WC05b]. Analogous to *silent actions*, actions representing no activity are not targeted satisfyingly in literature. Again, out of the interaction modeling notations, only WS-CDL provides information: „The *no action* activity is an explicit designator used for marking the point where a participant does not perform any action“ [WC05b, Sec. 6.6].

Once again, the need of the existence of *no actions* is not answered in this thesis, but this task is discussed in the modeling part of this thesis (cf. Chapter 6). However, a shortcoming is given at this point for further investigation: introducing *silent* and *no actions* into interaction choreographies leads to both, the weakening of the atomic interaction property and the danger of mixing choreographies with orchestrations. Thus, these artifacts should be used carefully, with respect to a high-level choreography modeling (cf. Section 5.6 on page 68).

5.3.3 On the Relevance of Intermediate Events

Until this point, intermediate events are regarded multiple times in this thesis. Sections 3.1 and Section 4.4 on page 41 discuss the need of intermediate events in interaction models due to exceptional events and modeling considerations. The main motivation is the fact that events denote deviations from the normal behavior which affect the observable behavior of the process collaboration. Additionally, a detailed example is discussed in Section 4.4.

However, no interaction modeling language provides the possibility to use intermediate events yet. Although, their need has been recognized in several proposals, „the interleaving of [...] business logic with [...] exception logic makes [...] business process[es] very complex and the original business logic hardly recognizable“ [HBM08, p. 5]. Furthermore, this mixture complicates the verification of business processes as shown in Section 4.4, as well as their maintenance.

[HBM08] proposes Self-Adapting Recovery Nets (SARN) which extend petri nets to model exception handling, by using special *recovery transitions* associated with exceptional events within the business process. SARNs are discussed in detail in Section 6.1 on page 72.

In addition, the latest proposal of BPMN 2.0 [BPM08] uses non-interrupting intermediate events, referring to the continuation behavior of exception handling as discussed in Section 3.3.3 on page 31. Thereby, the key idea is the treatment of cancellation and exceptions as events, rather than representing them „under a separate and inflexible fault-handling regime“ [GFJK03, p. 8]. Thus introducing both the notion of *event-based exception handling* [HA98] and intermediate events implementing the Inter Process Communication.

5.3.4 Advantages of Intermediate Events Implementing the Inter Process Communication

Chapter 3 mentions variations of transactional environments as basis of early solutions of exception handling. This section investigates the advantages of events over transactional environments according to [HA98; HA99]:

- Events allow an easy tool-integration using a common event notification mechanism.
- Events allow to model asynchronous situations.
- The event mechanism allows to simplify control flow specification.
- „The handling of exceptions can be greatly improved when an event-like mechanism is used [. . .], thereby giving a chance to fix the failure without having to abort the whole process“ [HA99, p. 4].

Thus events are predestinated to be used in process choreographies, especially in combination with exception handling. Events are already used in interconnection models and should be migrated to interaction models. For this reason, their usage in interaction models is recommended, especially to model different kinds of continuation behavior. Nevertheless, by the introduction of intermediate events, problems arise also with respect to their termination and revocation.

Regarding intermediate events in combination with exception handling, it appears that events have to be *terminated* due to exceptions. For this, BPEL states that „the behavior of a fault handler [. . .] begins by disabling the scope’s event handlers and implicitly terminating all activities enclosed within [the scope] that are currently active (including all running event handler instances)“ [BPE07, Sec. 12.6]. Thus, in process choreographies using intermediate events termination coordinators are necessary. Since this thesis deals with exception handling and not with termination, it is not possible due to time and space limitations to cover this aspect in the necessary detail. However, further investigation is needed and interested readers are referred to [Lud99] for the purpose of termination handling in inter-organizational workflows.

The second problem introduced by intermediate events refers to the *revocation* of events in cases of compensation. Thereby, manifesting in the question what the behavior of an already executed event is, if the ancestor (choreography-wide) scope aborts. Again, this thesis cannot provide an investigation of this task in the necessary detail. Thus, readers are reference to [HA98; HA99] for event revocation in event based exceptions handling and [Wan09] for an investigation of a generic scope concept especially regarding compensation.

5.3.5 Simultaneous Raised Exceptions and Exception Graphs

Process choreographies form *concurrent distributed systems* [TS02] including the notion of simultaneous raised events. This becomes especially interesting when these events represent exceptional situations since multiple concurrently raised exceptions may represent a system-wide exception. To handle these concurrent exceptional situations several ways are proposed.

Simple models (i) only handle the first occurrence of an exception, ignoring following exceptions or (ii) prioritize exceptions. For instance, the first model is used by [Ruf07] to gain fault handling in BPEL-based choreographies. In contrast to [Ruf07], there are also proposals in the context of distributed systems, representing the concurrent distributed semantic of exceptions in a more elaborate manner. According to the definition of choreographies (cf. Section 2.1 on page 15), participants exactly know the order of their operations and involved participants without involving a central coordinator. Thus, participants have no knowledge of exceptions in other participants until they are informed in a particular manner, i. e. by explicit modeling or implicit exception propagation which is discussed in Section 5.3.6 on the next page. Regarding exceptions in those settings lead to two interesting observations [XRR98]:

- (i) different participants may raise different exceptions and,
- (ii) these exceptions are raised simultaneously

Additionally, „concurrently raised exceptions may be merely a manifestation in multiple components of a system-wide exception“ [XRR98, p. 2]. As a consequence, a proper exception handling in choreographies cannot be achieved using these simple models, but through so called exception graphs. *Exception graphs* [XRR98] are an essential hierarchy-based approach to find a higher-order exception covering all exceptions raised simultaneously.

In order to overcome the problems of the simple models discussed before, [XRR98] proposes the notion of *exception graphs* for the representation of exception hierarchies. Exception graphs are the generalization of *exception trees* [CR86], due to the observation that exception hierarchies in distributed settings are often more complicated than simple trees. In exception graphs, „multiple exceptions are resolved into the exception that is the root of the smallest subtree containing all the raised exceptions [CR86]“ [XRR98, p. 7]. Exception graphs are defined by [XRR98, p. 8] as follows:

Definition 1 (Exception Graph) *An exception graph is a directed graph $G(E, R)$ where E denotes the exception set $E = \{e_1, \dots, e_n\}$ and R denotes a relationship $R \ni (e_i, e_j)$ in which $e_i \in E$ is the direct high-level node, or parent node of $e_j \in E$.*

Corresponding to the in- and out-degree $d_{in,out}(e_i)$ of a node e_i , they define three types of nodes: (i) primitive exceptions, (ii) resolving exceptions and (iii) universal exceptions. *Primitive exceptions* cover no other exceptions, therefore $d_{out}(e_i) = 0$ (cf. e_1, e_2, e_3 on level 0 in Figure 5.3a). *Resolving exceptions* cover other exceptions, therefore $d_{in}(e_i) \neq 0$ and $d_{out}(e_i) \neq 0$ (cf. level 1 and 2 in Figure 5.3a). Finally, the *universal exception* builds the root of the hierarchy, therefore $d_{in}(e_i) = 0$.

Furthermore, [XRR98] uses two types of exceptions called (i) undo exception and (ii) failure exception. *Undo exceptions* refer to actions that have been aborted and can be undone (i. e. compensatable actions). *Failure exceptions* refer to actions that have been aborted and cannot be undone (i. e. not compensatable actions).

[XRR98] proposes a four-step process of coordinated exception handling:

- (i) propagate the exception to all participants,

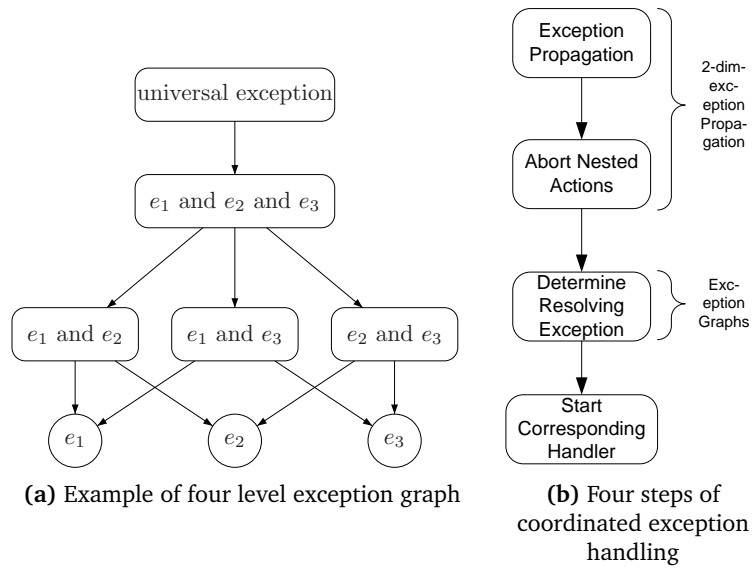


Figure 5.3: Exception graph and coordinated exception handling according to [XRR98]

- (ii) aborting nested actions,
- (iii) determine the resolving exception and
- (iv) start the corresponding exception handlers

As illustrated in Figure 5.3b, exception graphs are used in step three to determine the resolving exception in order to start the corresponding exception handler in step four. Steps one and two are discussed in the next section.

5.3.6 Exception Propagation in Choreographies Using Dominance

The previous section discusses that participants in choreography settings have no knowledge of exceptions raised in other participants until they are informed. This section discusses this information between participants, which is also called *exception propagation*.

Sections 5.3.1 and Section 5.3.3 on page 49 discuss the notion of silent actions and intermediate events. The next section discusses *exception contexts* in terms of its cross-partner property. Due to this artifacts, interaction models allow nested scoping by using different exception contexts, thereby introducing an extra dimension of exception propagation.

To achieve exception propagation between independent participants a coordinating instance is needed, knowing (i) where to pass exceptions to and (ii) which nested activities to abort. Since a simple chain of nested callers does not suffice, the notion of *two-dimensional exception propagation* is introduced by [XRR98]. As illustrated in Figure 5.4, not every exception context can be aborted by just passing the exception to it. The exception E raised in $A2$ „may need to be propagated upward to the enclosing action from a nested action“ [XRR98, p. 4]. Additionally,

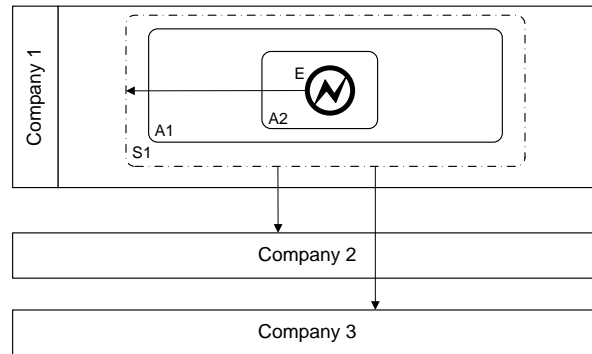


Figure 5.4: Two-dimensional exception propagation according to [XRR98]

S_1 may have nested actions in other participants. Thus, these actions have to be aborted by propagating exceptions also to other participants of the choreography. Furthermore, interacting participants „cooperate not only when they execute the normal [...] functions but also when they recover [...], i. e. during abnormal situations“ [XRR98, p. 4]. In the four-step process for coordinated exception handling, two-dimensional exception propagation affects step one and two, while the exception graphs affect steps three and four (cf. Figure 5.3b).

To describe common control flow situations in process orchestrations, control flow patterns have been proposed (cf. [AH07] and Chapter 4 in [Wes07]). Due to the difference of process orchestrations to process choreographies, common situations of process choreographies are described in a different way as *service interaction patterns* [BDH05] (cf. Chapter 5.5 in [Wes07]). During the investigation of several choreography models and service interaction patterns, an interesting property of exception propagation can be figured out. This observation manifests in the fact that every choreography contains starting and end interactions of particular participants, marking the begin and end of a choreography. These interactions must not be necessarily distinct. „A choreography is *initiated*, establishing a collaboration when an interaction, explicitly marked as an *choreography initiator*, is performed. [...] Two or more interactions *MAY* be marked as *choreography initiators*, indicating alternatives for establishing a collaboration“ [WC05b, Sec. 5.7]. To gain superior start and end markers, special *entry* and *exit* nodes can be injected.

Thus, interactions and control flow dependencies in process choreographies induce graph structures similar to those used in *control flow analysis* (cf. [NNH99]). Choreographies have particular initiators involving a number of participants providing services. These participants may involve other participants to provide their services. Since these participants are independent collaborating with each other, their knowledge of a concrete choreography-goal, they are involved in, decreases with increasing calling-depth. Thus, following assumption can be observed: Nodes having a low distance to the *entry-node* have more knowledge to cope with exceptions which are related to the choreography setting than nodes near to the *exit*. To gain a structured hierarchy, a tree structure can be derived using the concept of *dominance* (cf. [Muc97]). This tree structure forms a structured hierarchy to propagate exceptions through a choreography. Thus, it provides a more flexible approach than an propagation to every

participant, since only the initiator has the global view on its choreography semantic. Participants only providing specific services are not aware of their global usage and its corresponding exceptions.

*Terms and Definitions of Dominance*¹ A control flow graph $G(V, E)$ is a directed graph with vertices V representing interactions and edges E representing the control flow. The structure of such graphs can be simplified to point out the overall dependencies and responsibilities between participants. Branching and looping constructs have to be removed to gain a hierarchy of participant interaction. This can be achieved through the concept of *dominance*. In the context of dominance the graph is extended by two special nodes: *entry* marking the begin of a control sequence and *exit* marking the end of a control sequence. For instance, Figure 5.5a shows an exception graph and dominance tree for the running example presented in Section 4.3. Thereby, the *customer* plays a central role, interacting with the *airline*, *hotel* and *car rental*. To derive the dominance tree out of the control flow graph, the *dominance relation* is explained.

If an occurrence X is on every path from *entry* to Y , then X dominates Y ($X \text{ dom } Y$). Thus, dominance forms a *directed partial order* and can be arranged as *dominance tree*. $X \text{ dom } Y$ with $X \neq Y$ is termed *strict dominance* ($X \gg Y$). The *direct dominator* of Y ($\text{idom}(Y)$) is the next strict dominator of Y . Hence, it is the direct predecessor in the tree, or the intersection in the case of multiple predecessors. The dominance tree can be build by a top-down approach, using the *topological order*.

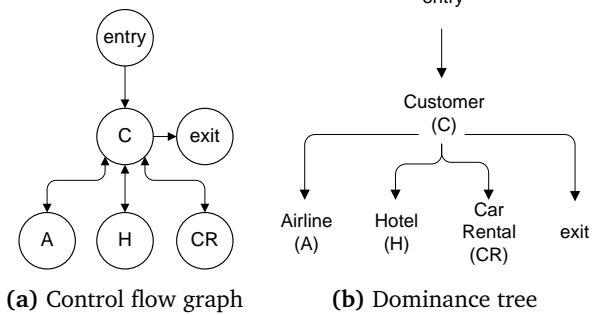


Figure 5.5: Control flow graph and dominance tree of running example

Using Dominance to Gain a Default Exception Propagation Hierarchy Having introduced the concept of dominance trees, they can be used to provide a default exception propagation hierarchy in choreographies. The propagation hierarchy can be derived statically out of the interaction model without having to know a particular implementation of the participants. This statically determined propagation hierarchy is more effective than (i) a propagation to all participants of the choreography as done in WS-CDL [WC02, Sec. 5.10] and (ii) a direct propagation along the calling sequence induced by the interactions, especially when looping structures, optional participation or delegations are involved. For this reason, the assumption, that nodes having a low distance to the entry-node have more knowledge to cope with exceptions which are related to the choreography setting than nodes near to the exit, has to be discussed further.

For instance, an exception is raised in the *car rental* participant of the running example. Thereby, it is unnecessary to propagate this exception to the *airline* and *hotel* participants, since they cannot handle this exception. Only the *customer* as initiator of the choreography has the

¹ The principles behind this section are based on [Muc97; NNH99; Bis08].

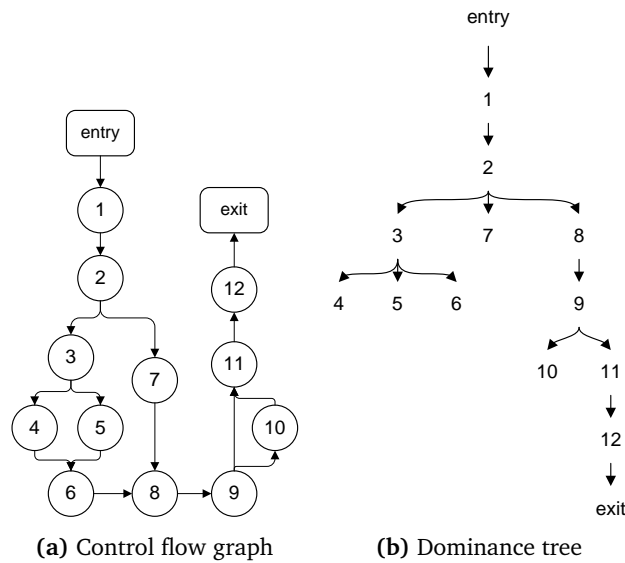


Figure 5.6: Complex control flow graph and dominance tree according to [Bis08]

necessary knowledge to handle this exception. Hence, the *customer* may request an alternative provider. Only, as a last resort the complete journey has to be canceled.

Lemma 1 *Nodes having a low distance to the entry are more suitable for the global interplay than nodes near to the exit or leaves in the dominance tree.*

Nodes having a low distance to the *entry* can only appear through interactions near to the initiators. Thus, they represent a kind of callers, asking for some service in a particular context. Thereby, the invoked participant only provides its service without knowing this particular context. If the invoked participant raises an exception which cannot be handled by itself, it has to propagate this exception somewhere. Since calling-participants know the particular context, they are a good point for such a propagation. For this, either a calling sequence or direct dominance can be used. Since the existence of a caller can only be decided at runtime, a calling sequence is more complex than direct dominance which can be determined statically. For instance, Figure 5.6a shows a more complex control flow graph. If thereby an exception is raised by participant 8, the existence of both branching predecessors has to be checked. This may result in three results: (i) + (ii) exactly one branch was taken, or (iii) both branches are taken. Thus, the propagation depends on this dynamically determined decision. In contrast, the statically determined dominator is participant 2, which in turn is aware of the particular branch and its context. Thus, this idea of statically determined exception propagation hierarchies using dominance may be used similar to BPELs default FCT-handling: as a default way of propagating faults, allowing the modeler to use both the default way of propagation or to do their own propagation such as a special order of compensation in BPEL.

5.4 Cross-Partner Scopes

Section 5.2 on page 47 discusses granularities of exceptions in interaction models. Thereby the notion of scopes local or global to a given participant is introduced. Due to the nature of interaction models these scopes form a main point of interest since they provide the needed context for exceptions and events. The relevance of scopes in cross-process settings has also been reflected in several proposals dealing with *cross-process transactions* [LR08] or *choreography transactions* [KWL09]. These transactions run across multiple process instances without requiring BPEL engines [LR08]. Thereby, arbitrary activities of different processes are grouped together into an all-or-nothing group, forming a transaction with atomic cross-process behavior. Section 3.3.2 on page 30 explains that atomic transactional behavior is a too strong requirement in the context of exceptions in web services. For this reason, [Ley05] introduces *Cross-Partner Scopes* (CPSs) which are applicable to an *extensible set of policies* indicating the behavior of the enclosed activities. Especially in the context of *interaction* choreography models CPSs are a useful artifact to model choreography-wide scopes since they somehow profit from the global view.

Corresponding to its implementation, CPSs can be (i) simulated through BPEL constructs [Ruf07], or (ii) used in existing processes without having to modify them using the *pluggable framework* [Ste08]. Furthermore, through CPSs it is possible to automate the coordination of the outcome (i. e. freeing the modeler from this aspect). Thus, CPSs can be imposed on existing processes as well as used in new processes. As a consequence, CPSs cannot only be used for exception handling in interaction choreographies, but also for ensuring *quality of services* in process orchestrations and choreographies.

In the following, this section discusses CPSs in detail. CPSs are defined in Section 5.4.1 and their semantic is investigated in Section 5.4.2. Subsequently, Section 5.4.3 discusses runtime considerations of CPSs and Section 5.4.4 deals with nesting and overlapping considerations. Finally, Section 5.4.7 copes with shortcomings of CPSs.

5.4.1 Motivation and Definition of Cross-Partner Scopes

The relevance of CPSs in this thesis is given through the observation, that exception contexts are needed on interaction level. These exception contexts cannot be provided by *normal BPEL scopes* due to two reasons: normal BPEL scopes used in interaction models (i) limit the expressiveness of the notation and (ii) introduce ambiguities among the semantic of a normal BPEL scope construct on interaction level, since these normal scopes are only related to the internal behavior of one single participant. Thus, a construct similar to BPEL scopes is needed covering cross-process behavior—Cross-Partner Scopes. Figure 4.4 on page 41 motivates the usage of CPSs on a high-level. According to [LR00; Ley05; Ruf07; Kop08b; Wan09; KWL09] CPSs can be defined as follows:

Definition 2 (Cross-Partner Scope (CPS)) *Let Ch be a choreography and P_i its participants, where participants are a set of activities a_j . Thus, $Ch = \bigcup_i P_i$ is a choreography model with*

participants P_i . Let $CPS \subseteq Ch$ be a Cross-Partner Scope of the choreography Ch . A set of activities a_j is called CPS $\iff \exists a_1, a_2, a_1 \neq a_2 : a_1 \in P_1, a_2 \in P_2, P_1 \neq P_2$.

Thus, a CPS contains at least two distinct activities of two distinct participants. Thereby, CPSs ensure common behavior of all enclosed activities that can be part of multiple processes. In general, CPSs are not required to satisfy any transactional properties. However, they can be annotated with different *policies* indicating their behavior of completion.

[Kop08b] formulates three requirements for CPSs: (i) BPEL users should not be aware of completely new scope semantics. This can be achieved due to (ii) an extension of BPEL scope semantics to cross-partner settings. For a broad acceptance, (iii) CPSs should be used within existing processes without modifications in them. Especially the third requirement of [Kop08b] can be stated more precisely, regarding (a) transaction behavior of CPSs, (b) exceptions (including events) and (c) overlapping considerations. As discussed in Section 3.3.2 on page 30 the *transaction behavior*, CPSs need to support at least seven types of transactions: *strict atomicity* (all-or-nothing), *alternative atomicity* (exactly once) and *exception atomicity* as well as *ACID Transactions*, *Business Transactions*, *Mixed Transactions* and *Autonomous Decisions* [GR93; BN97; LR00; HR01]. This can be achieved by coordinating CPSs through WS-Coordination (WS-C). Although WS-C has no direct support of all transactional requirements, WS-C suffices since [Vet06] presents a simulation of *Business Transaction Protocol* (BTP) [BTP04] and *WS-Composite Application Framework* (WS-CAF) [WC05a].

CPSs have to support *exception and event handling*. Thus, some kind of exception and event handling coordinator is needed. Due to the generic structure of CPSs, which allows them to be imposed on processes without modification, *overlapping* becomes interesting. Overlapping is normally forbidden in the context of transactions and scopes due to several difficulties. Nevertheless, overlapping increases the power of CPSs and is therefore investigated in detail in Section 5.4.4 on page 60.

5.4.2 Semantics of Cross-Partner Scopes²

Since CPSs have no transactional properties per definition, policies specify the *requirements* of completion. Thus, CPSs are a general scoping concept that can be used in BPM for several use-cases. For CPSs, an extensible set of policies can be provided through particular protocols with respect to the discussed atomicity constraints of exception handling in Section 3.3.2 on page 30 (i. e. strict, alternative and exception atomicity) and the requirements of CPSs of Section 5.4.1 on the preceding page (i. e. ACID, business, mixed transactions and autonomous decisions). The standard case provided with this thesis is the atomic policy providing an all-or-nothing semantic. Activities specify the effects of non-completion (i. e. Fault, Compensation or Termination handling (FCT)). In addition to [Kop08b], activities should also specify the effects of non-interrupting completion (i. e. events having BPEL's onEvent semantics).

For instance, if a CPS S_i is not completed and (i) the policy states that the transaction is not completed, and (ii) a scope S_j starts execution afterwards. Then S_j is not started at all, but set

² This section is mainly based on [Kop08b], other references are explicitly stated.

to „completed with fault“. This also applies for all activities nested in S_j (i. e. the intended side effect is triggering *Death Path Elimination* (DPE) [CKLW03]). If the enclosed scope faults or terminates, termination is triggered. In contradiction to fault semantics, this information is not propagated to the partner. Hence, it counts as non-completed scope due to the semantics of termination, since the reason of the termination cannot be propagated. If a CPS faults, the specified fault handling is triggered and the CPS behaves as specified in the BPEL semantics. Thereby, the default behavior is the propagation to the enclosing scope and termination of all subactivities. Thus, it enables conversion of faults and fault data and ignoring faults.

5.4.3 Runtime Considerations on Cross-Partner Scopes

Due to the distributed nature of CPSs, a coordinating entity is needed to gain a distributed outcome in cases of exceptions, compensation or termination. In contrast to process orchestrations, this becomes even more difficult in process choreographies, since each participant is independent without any knowledge about the overall choreography state. For this reason, several proposals are made, with the main difference in the representation of CPSs and their coordination.

CPSs are introduced as scopes spanning multiple participants. These participants are independently collaborating in a process choreography. Thus, a representation of cross-process artifacts in independent participants is needed. In general, two alternatives exist for representing CPSs corresponding to the degree of its access to the process model: (i) extend the process model, or (ii) leave the process model as is (i. e. do not touch its implementation).

Extending the process model needs a simulation of CPSs through normal BPEL artifacts. For this purpose, [Ruf07] proposes a solution using BPEL scope fragments representing CPS elements of a particular process. This is shown in Figure 5.7. Figure 5.7a shows a CPS enclosing a nested activity A . Figure 5.7b shows the corresponding representation using a fragment F which is strictly nested.

Leaving the process model untouched prohibit changes of the process model. Thus, a mechanism is needed to observe processes externally. This can only be achieved by somehow externalizing the behavior of processes as proposed by [Mie06; PML07] in combination with the transaction behavior of scopes. Hence, this form of representing CPSs mainly depends on the kind of coordination. As mentioned in previous sections, a main goal of this investigation is to leave the process model untouched and therefore impose CPSs on processes. Thus, the first alternative is not recommended.

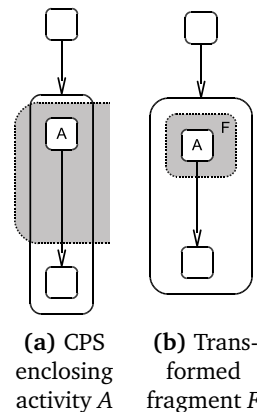


Figure 5.7: BPEL scope fragments according to [Ruf07]

Coordination of Cross-Partner Scopes Having discussed the representation of CPSs, a second interesting question is the placement of the coordination code, which is necessary due to

the independent nature of process choreographies. Corresponding to the different kinds of representation two general types of coordination code placement can be distinguished:

- internal coordination and
- external coordination.

Internal coordination refers to the placement of coordination code *within* the process engine, while external coordination refers to the placement of coordination code *external* to the process engine. *Internal coordination* can be further refined into two subcategories: (i) engine-based coordination, and (ii) process-based coordination. *Internal engine-based coordination* uses a choreography-aware BPEL engine, inducing exceptions, compensation and termination. Hence, it can use both representations of CPSs. The main advantage is that process models remain as is and exceptions can be monitored. The disadvantage is that (1) an extended BPEL engine, and (2) standardized protocol messages between engines of different vendors are needed. Furthermore, it is (3) difficult to provide flexible coordination with this approach since every choreography needs an interface to somehow use its coordination logic in the process engine. Thus, this approach is not recommended since it requires all coordination-code to reside in an inflexible way in the BPEL engine. Figure 5.8a shows an overview of internal engine-based coordination of CPSs simulated by two fragments *F1* and *F2*. The coordination code completely remains in the BPEL engines.

In contrast, *internal process-based coordination* changes the process model by embedding all coordination code into them. Thereby, event handlers are used for coordination messages, forcing a registration at the entry of a CPS. Figure 5.8b illustrates this approach which is implemented by [Ruf07]. Although, this approach is more flexible as the previous one, it comes with the main disadvantage of extending the process model. Thus, introducing the contradiction between weakening simplicity and maintainability requirements of process models and the usage of exception handling to improve fault tolerance, simplicity and maintainability. As a consequence of this observation and the recommendation of the previous section, internal coordination is not recommended for exception handling in choreographies.

Finally, *external coordination* involves CPSs that can be imposed on process models without changing their implementation. Thereby, the behavior of processes is externalized using a *pluggable event framework* as proposed by [KKL07; Ste08]. As illustrated in Figure 5.8c, processes are no longer involved in their coordination with respect to its independent nature. Instead, processes (i. e. their runtime events in the engine) are observed by controllers having the possibility to block particular activities. These blockades are resolved by external subscribers—the coordinators.

With respect to the discussed top-down and bottom-up approach of choreography design in Section 5.6 on page 68, it seems to be unsuitable to require changes of the process model. Thus, this thesis follows the second approach—external coordination—to be able to impose CPSs on processes without changing the process model, using external coordination. A detailed discussion of the runtime support of CPSs is given in Chapter 7.

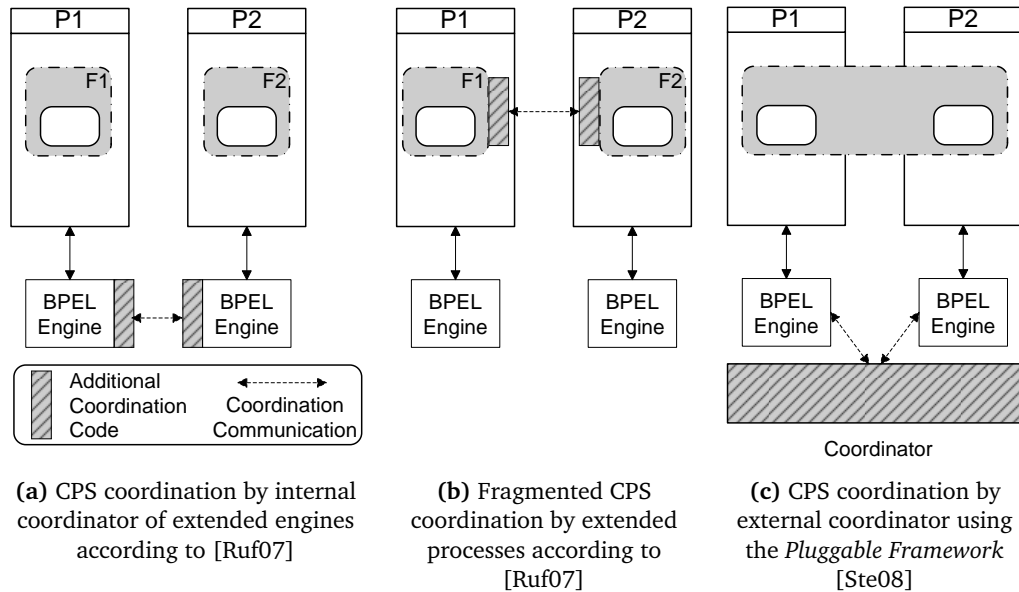


Figure 5.8: Coordination alternatives of CPSs

5.4.4 Nesting and Overlapping Considerations of Cross-Partner Scopes

Previous approaches of transactions, regions, scopes or even CPSs have the requirement that scopes must not overlap. For instance, scopes in BPEL are constructs that introduce a hierarchy into the process, similar to nested blocks in traditional programming languages [BKLL09]. Thus, an activity may not be part of two distinct CPSs.

Regarding scopes from a global perspective, especially with CPSs in mind, it turns out that these restrictions may be insufficient with exception-enabled interaction choreographies. For instance, Figure 5.9 shows a detailed view on the ticket ordering between *customer* and *airline* of the running example. In addition to the CPS S_1 introduced in Figure 4.4 on page 41, there is a CPS S_2 referring to technical failure events of the *carrier*. The *carrier* participant is introduced to the choreography since it is possible that an *airline* provides its flights by involving different *carriers*. Thereby, it is possible, that a *carrier* is not fit for service due to technical failures. Thus, the *airline* has to reschedule the corresponding flights. Hence, if overlapping is forbidden, such situations cannot be modeled with its anticipated semantic (i. e. forcing modelers to simulate this through embedded exception modeling).

The observation at this point is, that CPSs allow global structuring of activities which is not necessarily strict hierarchic at first glance, but again induces a graph structure. Notice, that there are existing concepts of dealing with graph based hierarchies (cf. Section 5.3.5). Thereby, the similarity lies in the simultaneous occurrence of activities in multiple scopes. For example, the *reserve seat* activity in Figure 5.9 lies in scope S_1 as well as in scope S_2 .

A detailed investigation of CPSs is needed since CPSs are used in this thesis on a global interaction level providing event and exception support. No literature can be found dealing

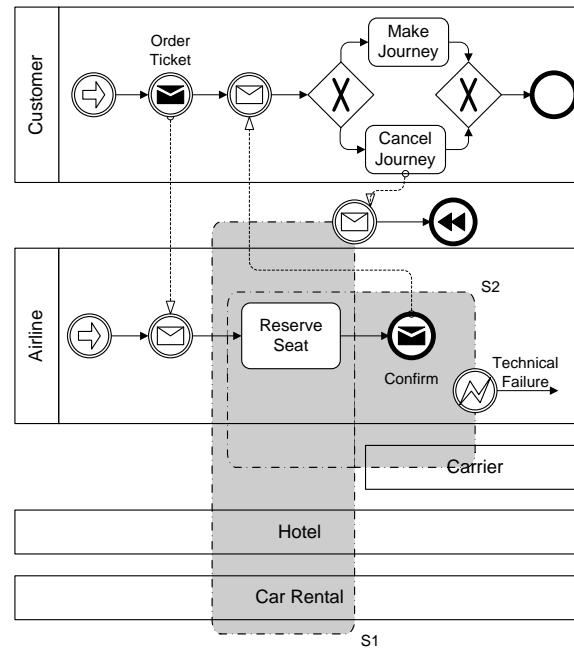


Figure 5.9: Detailed view on running example involving overlapping CPSs

with overlapping considerations in combination with exception contexts. However, overlapping is discussed in combination with generic scope compensation:

Definition 3 (Overlapping Scopes adapted from [Wan09, p. 18]) Let S_1 and S_2 be two scopes in a process model graph (assuming that scopes are sets of activities). If $S_1 \cap S_2 \neq \emptyset$ and $S_1 \cap S_2 \neq S_1$ and $S_1 \cap S_2 \neq S_2$, S_1 and S_2 intersect with each other or they are overlapping and $S_1 \cap S_2$ is the intersection of S_1 and S_2 .

The interesting questions that come in mind when regarding such an overlapping mainly deals with the semantics of the overlapping, the responsibilities of the different handlers (if there are any) and of course the allocation and propagation of exceptions. For instance, the intended *semantic* of Figure 5.9 is that CPS S_1 allows the *customer* an easy cancellation of his journey due to some reasons. Furthermore, CPS S_2 allows the *airline* to reschedule its flights in case of technical failures of the *carriers*. Thus, the illustrated overlapping forms an overlapping where the two scopes are somehow independent of each other. This independence manifests in the case that on *technical failures* within S_2 , S_1 is not triggered (assuming successful failure handling through involving another provider of the *carrier* role). In Figure 5.9 there are no conflicts between the exceptions handlers, but there may be problematic situations. For example, the *responsibility* of handlers becomes ambiguous when there are either (a) no handlers, or (b) two handlers for an particular exception. At first glance, the expected behavior in this cases and the allocation of exceptions and their coordinators is not clear.

Thus, an investigation of overlapping types in combination with CPSs is needed.

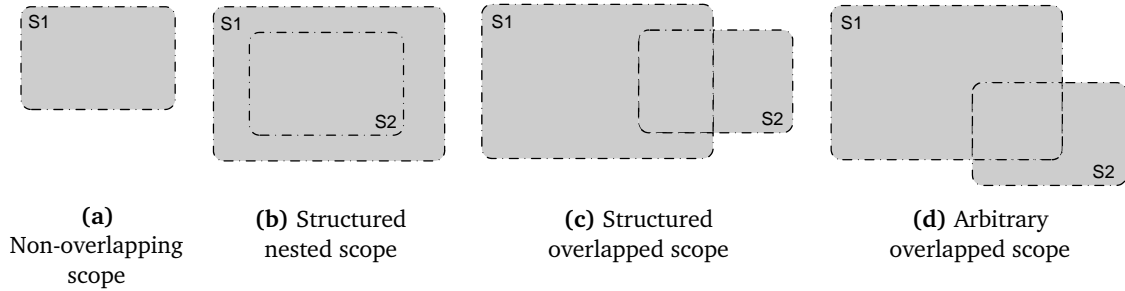


Figure 5.10: Overlapping types of CPSs: simple scope overlapping

5.4.5 Classification of Cross-Partner Scopes According to the Type of Overlapping

In general, two categories and five groups of overlaps can be distinguished.

Thereby, the first category is shown in Figure 5.10 and forms (i) *simple scope overlapping* between one or two participants. The first group marks a scope having no overlap at all. This kind of scope is called (a) *non-overlapping scope*. Further, scopes can be arbitrarily but structured nested. This kind of scope is called (b) *structured nested scope*. The first new group marks scopes having an overlap and are structured. This kind of scope is called (c) *structured overlapped scope*. Finally, scopes can overlap arbitrary and thus called (d) *arbitrary overlapped scopes*. This can be further refined into two subcategories. (d.1) *Shared scopes* or *equal scopes* may trigger each other. In contrast, (d.2) *independent scopes* are handled as two independent scopes having no knowledge of each other (i. e. these scopes cannot trigger each other).

The second category of overlaps is shown in Figure 5.11 on the next page and involves more than two participants forming (ii) *complex scope overlapping*. Thereby, the difficulty arises due to the responsibilities of particular scopes. Figure 5.11a illustrates an overlap spanning a single participant. The corresponding intended semantic of this overlap is given in Figure 5.11b. Thereby, the overlap of S_1 and S_2 results in three scopes, representing the CPSs S_1 and S_2 as well as the area where both scopes are aware of. This becomes even more complex when the area of overlapping is widened to span multiple scopes as shown in Figure 5.11c. As a consequence, this results in three CPS projections S_1 , S_2 and $S_1 + S_2$ shown in Figure 5.11d. How the handlers and responsibilities between these CPS projections can be derived is discussed in the next section.

5.4.6 Runtime Considerations on Overlapping Cross-Partner Scopes

Section 5.4.3 on page 58 presents general runtime considerations on CPSs. This section focuses on runtime consideration of the previous discussed overlapping.

Possible solutions to handle overlapping are similar to those discussed in Section 5.3.5 on page 50. These solutions mainly deal with (i) precedence of one scope to another or (ii) a leader election. In addition, (iii) a simple union, intersection or XOR-semantic can be

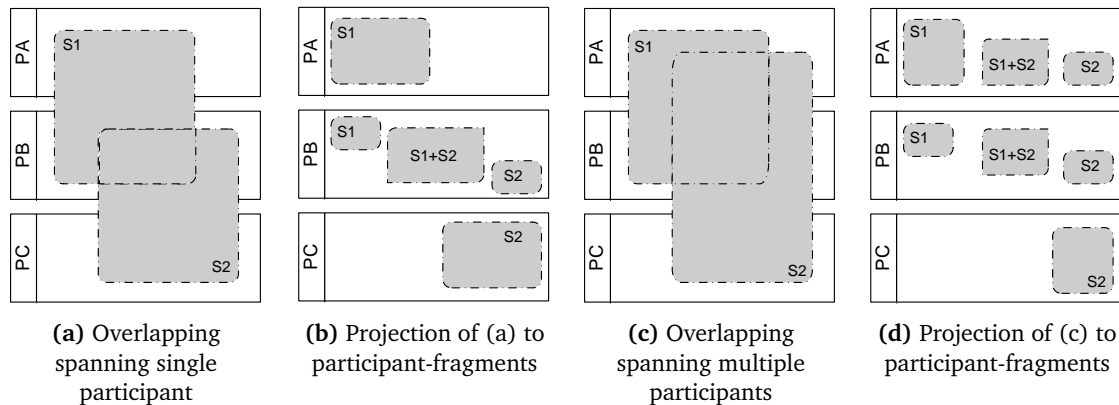


Figure 5.11: Overlapping types of CPSs: complex scope overlapping

provided. Further, solutions may involve (iv) an external single decision or use (v) coordinating approaches.

Solutions using different *precedence* levels of scopes provide a good starting point for investigating overlapping scopes. Thereby, each scope is associated with a precedence indicating its level of responsibility. For instance, out of Figures 5.10b and Figure 5.10c on the facing page an implicit precedence of scope S_2 over S_1 can be derived for all activities lying in the intersection. More difficult is such a derivation with *arbitrary overlapped scopes* shown in Figure 5.10d. *Shared scopes* as well as *independent scopes* must not have precedence over each other per definition. Thus, it is not clear which handler is responsible with arbitrary overlapping.

Leader election [CF98] deals with a competitive or cooperative election of a leader having more precedence over other participants. Thus, this leader is responsible for all decisions. Especially with process choreographies this seems to be unnecessary since this information can be derived at design-time (i. e. the initiator of the choreography is the implicit leader). Furthermore, this solution also lacks the possibility of handling arbitrary overlapping scopes due to the same reasons as before.

Solutions dealing with *set operations* are also unsatisfying. A simple *union* or *intersection* of CPSs and their handler becomes difficult when both scopes have a handler for the same exception. Thus, it is not clear how to join or intersect these specifications and what is the intended semantic. For this, the *XOR-semantic* provides help. Thereby, only one handler may be used in the overlapping area. However, this solutions also lacks a clear representation of arbitrary overlapping scopes.

External single decision is similar to the leader election solution with the difference that the decision at a certain point is not done by a participant. Thus, this decision can only be achieved through either human intervention or an external coordinator. As a consequence this is not practicable.

The last solution refers to *coordinating approaches* as they are discussed in Section 5.4.3 on page 58. Hence, it is somehow a combination of the previous solutions. Thereby, an

additionally coordinator is introduced for the each overlapping area (cf. Figure 5.11d). The responsibilities of the different coordinators is given through a graph hierarchy similar to exception graphs of Section 5.3.5 on page 51.

Definition 4 (Overlapping Graph) An overlapping graph is a directed graph $G(E, R)$. E denotes the set of nodes $E = S \cup P$ where S is the set of scopes including their intersections $S_{i \cap j}$ $S = \{S_1, \dots, S_n, S_{i \cap j}\}$ and P denotes involved participants $P = \{P_1, \dots, P_m\}$. R denotes a relationship $R \ni (e_i, e_j)$ in which $e_i \in S$ and $e_j \in P$. Corresponding to the in- and out-degree $d_{in,out}(e_i)$ of a node e_i , there are three types of nodes: (i) $d_{in}(e_i) = 0$ refers to the root of the overlapping, (ii) $d_{in,out}(e_i) \neq 0$ refers to participating scopes or suboverlaps, (iii) $d_{out}(e_i) = 0$ refers to participants P (i. e. $\bigcup_{d_{out}(e_i)=0} = P$).

For instance, Figure 5.12 shows the corresponding overlapping graphs of the complex overlapping of Figure 5.11. If an exception occurs in S_1 of Figure 5.12a and not in $S_1 S_2$ and if this exception cannot be handled successfully, the exception is propagated to PA and PB .

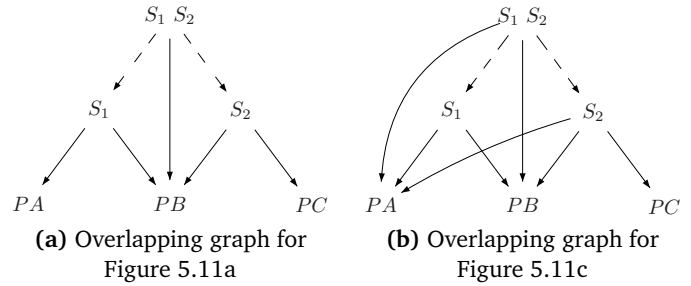


Figure 5.12: Example overlapping graphs for complex overlapping (dashed lines are implicit)

The usage of overlapping graphs lies in the distinction of activities according to a responsible coordinator. Since no solutions can be found to

automate the merging of handlers of overlapped scopes, the *choreography modeler* (i. e. not the process modeler) has to provide a separate handler for overlapping areas. Thereby, the intended semantic of the overlapping has to be provided within these handlers.

5.4.7 Shortcomings of Cross-Partner Scopes

[Ruf07] identifies three scenarios which lead to *deadlocks* due to projected CPSs, which are shown in Figure 5.13. It is important to notice the assumption that fragments in [Ruf07] are left only if all other fragments of the corresponding CPS have completed. Figure 5.13a shows a scenario where F_2 depends on F_1 , while F_1 waits for F_2 to finish. However, deadlocks are not easy to gather in general. For instance, in Figure 5.13b a deadlock occurs when S_1 is isolated, F_4 cannot finish without F_3 , while F_3 depends on F_4 . Again a deadlock. The scenario can be even widen to cross-partner settings as shown in Figure 5.13c. Thereby, the CPSs CS_1, CS_2 are not overlapping each other in the sense of the discussed overlapping of Section 5.4.5 on page 62. In contrast, CS_1, CS_2 „are crossing bounds which is allowed but leads to a deadlock“ [Ruf07, p. 44]. CS_1 waits on activity E of CS_2 , while CS_2 itself waits on activity F of CS_1 . Thus, another deadlock situation appears.

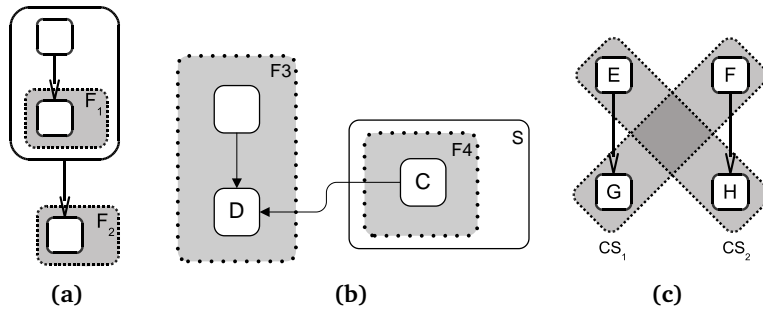


Figure 5.13: Scenarios leading to deadlocks due to CPS fragmentation presented in [Ruf07]

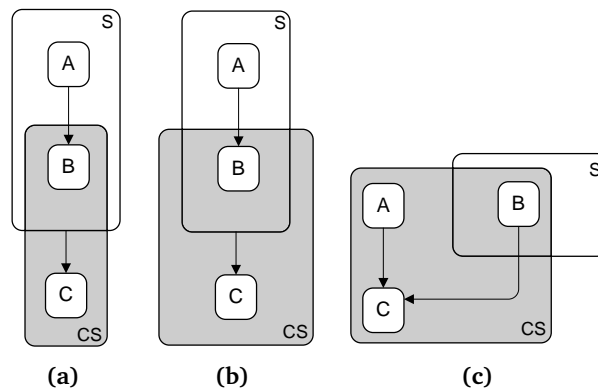


Figure 5.14: Projection of Figure 5.13 using imposed CPSs.

(a) and (b) shows the projection of Figure 5.13a. (c) shows the projection of Figure 5.13c.

All in all, the first two problems only arise due to the nature and execution semantics of fragmented scope representation. Hence, Figure 5.14(a,b) shows two alternatives for modeling Figure 5.13a. Figure 5.14a refers to a projection indicating a lower priority of CS over S , while Figure 5.14b indicates a higher priority of CS over S . Also the second problem shown in Figure 5.13b refers to representation-specific problems. For this purpose, Figure 5.14c provides a CPS which is imposed and thus the deadlock does not appear (i. e. imposed CPSs are independent of permeability problems). However, the problem presented in Figure 5.13c refers to the need of choreography modelers to ensure deadlock-freedom of their models. This problem is already mentioned in [Ruf07] and appears also with CPSs.

5.5 Exception Handling in Interaction Models

The previous sections introduces interaction exceptions and derived requirements of them, motivates the usage of intermediate events and introduces the concept of CPSs to gain choreography-wide exception handling contexts. Finally, a detailed investigation is missing how these artifacts can be used to achieve exception handling in interaction choreography models. The main

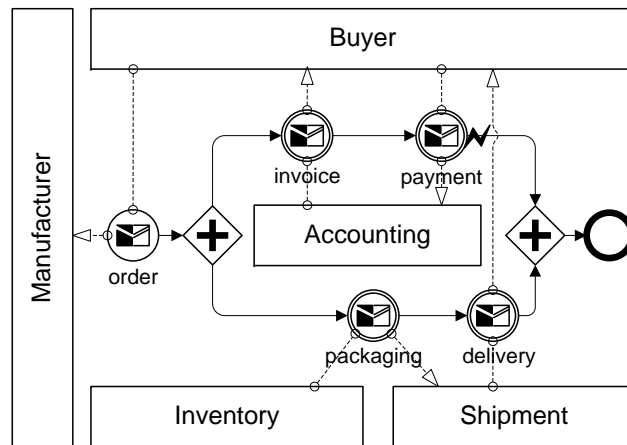


Figure 5.15: Buyer manufacturer scenario

points of interest are thereby the membership of exception handlers, i. e. the question how exceptions are monitored in choreographies and where handling and coordinating code must be placed. Furthermore, the number of coordinators is an interesting point.

5.5.1 Choreography Life-Line

Regarding the choreography life-line WS-CDL is insufficient: „A choreography in an Enabled State MUST complete unsuccessfully when an exception is caused in the choreography and its exceptionBlock, if present, MUST be enabled. This MUST cause the choreography to enter the Unsuccessfully Completed State“ [WC05b, Sec. 5.7]. Coordination of choreographies in WS-CDL includes the propagation of exceptions to all partners [WC05b].

These requirements seems to be too strong, since in some situations strategies as they are proposed in Section 3.3 on page 28 may be sufficient to keep the choreography alive through successful exception handling. Thus, exception handling strategies are able to reduce the time where business goals cannot be fulfilled since they avoid unnecessary complete rollbacks and additional costs due to human intervention. Furthermore, a propagation of exceptions to all participants seems to be unnecessary too (cf. Section 5.3.6). For instance, the buyer manufacturer scenario is shown in Figure 5.15. Thereby, five participants are involved: (1) the *buyer*, (2) the *manufacturer*, (3) the *accounting*, (4) the *inventory* and (5) the *shipment*. After the *buyer* orders some products, payment and delivery are executed in parallel. Thereby, an *invoice* is send to the *buyer* which has to pay the invoice. Additionally, the *inventory* starts *packaging* and informs the *shipment* after completion. If thereby an exception occurs within the *payment*, it is unnecessary to propagate the exception to all participants and close the choreography in general. This can be understand, since the exception may be handled successfully. For example, another payment method can be choosed and thus, the choreography can be kept alive. Only as a last resort, the complete choreography needs to be canceled.

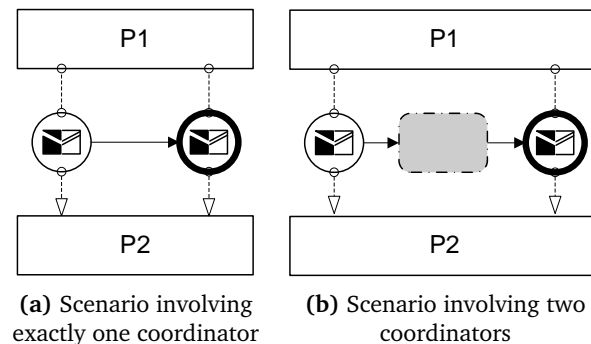


Figure 5.16: Number of coordinators needed

Thus, a choreography must only complete unsuccessfully when an exception is caused in the choreography where (i) no handler is present (i. e. also no *catchAll block*), or (ii) the exception cannot be handled by its handler and no appropriate handler further in the propagation hierarchy. In all other cases the choreography completion depends on both the continuation behavior of the particular exception and the degree of the handling strategy.

5.5.2 Membership of Exception Handlers

Having discussed the choreography life-line, it is an open question where a particular exception handler belongs to. Following the first requirement of Section 5.4.1 on page 56 stating that BPEL users should not be aware of completely new scope semantics, exception handling in interaction models orientates on the BPEL semantic.

Section 5.4 on page 56 introduces CPSs, their requirements and coordination. Thus, providing coordinated exception handling contexts. The strategy of handling exceptions can then be defined in interaction models using intermediate events.

In the context of BPEL, granularities of interaction exceptions are discussed in Section 5.2 on page 47. Due to the independent nature of participants in process choreographies, there is in general no participant having itself a global view on the state of collaboration. Thus, separate coordination code must be embedded somewhere. For this reason, Section 5.4.3 on page 58 discusses the notion of external coordination using the pluggable framework.

CPSs are made for cross-process settings. They can be imposed on existing processes without local changes. CPSs involve external coordination and thus, they are the main artifact of exception handling in interaction choreographies.

5.5.3 Number of Coordinators

Finally, an issue is the number of coordinators needed for exception handling in choreographies. As proposed in Section 5.4.3 on page 58, a coordinator is needed for every CPS. Thereby, an

implicit scope spans all participants in order to gain a final exception handling. This is similar to BPEL's implicit process scope.

Figure 5.16a shows a choreography, involving two interactions. Thereby, exactly one coordinator is needed in order to monitor the specified protocol. If there is no need for exception handling, the coordinator can be omitted. Figure 5.16b, shows a more complex choreography involving an additional CPS. Thus, two coordinators are needed. One to control the choreography protocol and the other the control the CPS.

5.6 Top-Down vs. Bottom-Up Choreography Design using Interaction Exceptions

In general there are two worlds of service composition and modeling. Choreographies can be either designed from the scratch or by using existing processes. Thereby, both approaches have distinct requirements to modeling artifacts.

For instance, a choreography designer specifies a complete new choreography. As described in Section 4.2 on page 38 participants are identified, milestone and collaboration scenarios are developed. Finally, behavioral interfaces are derived out of the collaboration scenarios and executable processes are derived out of the behavioral interfaces. Thus, the designer is able to modify certain aspects in the process implementation.

However, the thought of SOAs is the re-usage of services by composing existing services into new applications. Thus, developing complete choreographies from the scratch is normally unnecessary. „The disadvantage of this approach is the necessity for the process modeler, to know, within the business process, the action that need to be carried out [in exceptional situations]“ [LR08, p. 4]. Additionally, existing processes are not designed as participants of a special choreography. Thus, process modelers should no longer specify exceptional activities which are related with a special usage [LR08], since processes contained in a particular choreography are not aware of exceptions occurring in this collaboration.

Choreography designers combine services out of several already existing services. Since these existing services are independent, choreography designers may not be able to change certain aspects in those services. However, exceptions related to particular choreographies have to be modeled due to fault tolerance reasons. Thus, a concept is needed to gain choreography fault handling *without* changing processes. CPSs as they are used in this thesis provide such a possibility. Hence, the approach taken in this thesis is applicable to both top-down and bottom-up choreography design.

Interaction choreographies provide a global view on the relationship of participants. Thereby, previous approaches mostly disregard exception handling in this context. A detailed investigation of the behavior of participants and their collaboration leads to the observation that independent participants collaborating according to a specific choreography protocol cannot ensure the overall success of the collaboration by local exception handling. This observation is

guided by the fact, that local exception handling mechanisms do not suffice to overcome global exceptions of the choreography, since participants lack of information about the global state.

The global view of interaction choreography models allows the abstraction from internal behavior and thus allows the possibility to describe exceptional situations *between* partners of a particular collaboration. This also belongs to a simple, expressive and complete model.

However, also a global view may have a *functional layer* as well as a *detailed layer*. The functional layer focuses on complex collaboration scenarios answering the question which collaborations appear between whom. Process choreographies describe the sequence of expected interaction behavior. Hence, these sequence is influenced by exceptions. Thus, a detailed layer is a good starting point for an introduction of choreography-wide exception handling. This detailed layer focuses on more concrete collaborations and therefore describes the complete sequence and control dependencies between participants—also involving exceptions.

All in all, interaction choreographies describe the ways of interactions, however, these ways may be unsuccessful. Thus, the identification of exceptional situations on choreography level and abstracting them into handling strategies can be used to: „(i) simplify the design of exception handling and (ii) automate exception handling processes by, e.g., generating exception handling controllers from specifications“ [HBM08, p. 6]. For instance, these specifications may be interaction models.

5.7 Summary

This chapter develops a theoretical foundation of exceptions in interaction choreography models. Thereby, a classification and granularities of interaction exceptions is developed out of the given classifications in Chapter 3. Section 5.3 on page 47 discusses requirements of exceptions in interaction models. Thereby, *silent-* and *no action-construct* as well as intermediate events are discussed. Furthermore, simultaneous raised exceptions and exception propagation are presented due to the concurrent distributed nature of interaction choreographies. To provide exception contexts in choreography settings, *Cross-Partner Scopes* are investigated, especially focusing on their externally coordination. In addition, overlapping of CPSs are considered with respect to the mentioned choreography design process. Finally, Section 5.5 combines the proposed artifacts to a general picture of dealing with exception in choreographies, while Section 5.6 evaluates these artifacts according to the choreography design process. Based on the theoretical foundation, the next two chapters deal with modeling and runtime support of interaction exceptions.

6 Modeling a Choreography with Interaction Exceptions

The previous chapter discusses exceptions and their handling in interaction models from a merely theoretical perspective, to be free from existing limitations. Now this chapter provides a detailed discussion of the realization in modeling. In general, in the field of choreography design, there are three issues to tackle: (i) modeling of a choreography, (ii) verification of the choreography and finally (iii) mapping of the choreography to the runtime [KL08, p. 32]. During this chapter issue (i) is discussed, while issue (iii) is discussed in Chapter 7. Since issue (ii) is out of the scope of this thesis, readers interested in verification issues are referred to [Mar03; Aal+06; DW07; LKLR07; Dec09b; LW09].

Modeling artifacts are the essential building blocks for successful modeling notations. Thus, a detailed investigation of user behavior and their needs as well as a detailed evaluation of the acceptance of new artifacts are the basis of successful modeling notations. The multiplicity of different notations point to different needs of modelers. For instance, there are high-level notations focusing on business analysts modeling abstract behavior. Thereby, it is not necessary to specify each element in detail, leaving gaps which are refined later. For this purpose, implementation-centric notations are provided, coping with different granularities of abstraction. Finally, every model is transformed to its runtime representation.

6.1 Exceptions in Business Process Modeling Languages

This section first discusses different support of exceptions in business process modeling. BPM merely ignores the need of exceptional behavior in interaction languages, while there are solutions introducing these concepts into petri nets which are used for verification purposes.

In the context of this thesis, high-level notations follow the interaction modeling style, while interconnection centric notations are used on a lower level. In general, it is difficult to provide a strict classification, since the constraints are cloudy and depend on particular use-cases. The decreasing level of abstraction is used as indicator as well as an approximation to notations having a programming interface instead of a graphical interface [BKLL09]. In terms of BPMN's typical control flow constructs, such as sequential and parallel execution as well as branches and loops, are well supported. Furthermore, BPMN provides exception modeling. In contrast, exceptions are mostly disregarded in high-level notations following the interaction modeling style such as Let's Dance and iBPMN. WS-CDL provides exceptional constraints, but lacks user acceptance and a graphical representation.

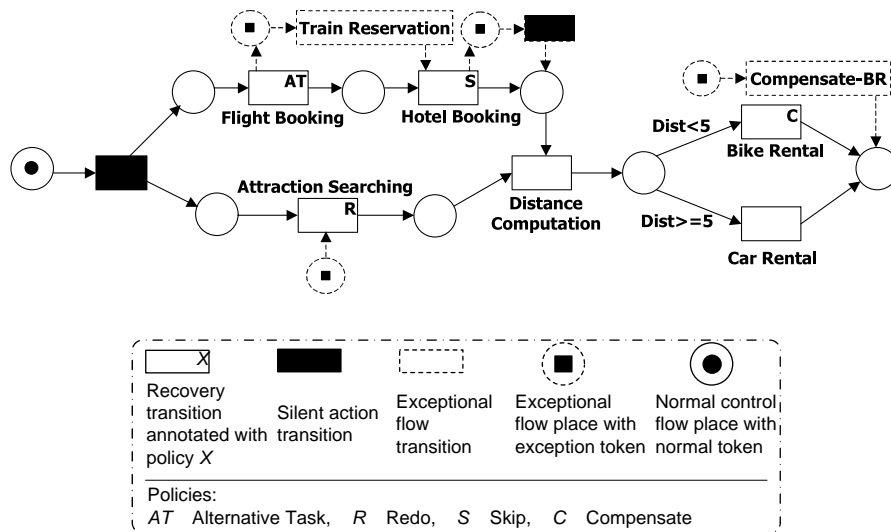


Figure 6.1: Travel scenario as self-adapting recovery net

In addition to modeling needs, verification is an important issue in every modeling notation. Thereby, formal semantics are essential for analyzing modeling notations. Inspired by traditional programming languages and other notations *state machines*, *state-charts* [Sch08b] and especially *petri nets* are used for such purposes. Hence, as discussed in Section 4.4 on page 41, existing notations are not suitable for exceptional behavior.

As a consequence, modelers are forced to use embedded exception modeling, which decreases simplicity as well as understandability and therefore makes it difficult to maintain the model (cf. Section 3.2.4 on page 27). In terms of petri nets, [HB04; HBM08] propose an extension dealing with exceptional events.

Self-Adapting Recovery Nets (SARN) are petri nets extended by *recovery transitions* and *recovery tokens*. They „concentrate on handling exceptions at the instance level [by using dynamic model extensions] and not on modifying the business process schema“ [HBM08, p. 3].

For instance, Figure 6.1 shows the travel scenario as SARN. *Standard transitions* mark business process tasks. The parts drawn in dotted lines mark exceptional flow using a special exception token (black rectangle instead of circle). This exceptional flow is only executed when the corresponding exception event occurs. The black rectangle transition marks silent actions, while recovery transitions are annotated with recovery policies. Recovery policies are described by a generic set of primitive operations such as creation and deletion of arcs, transitions and tokens. For example, the *flight booking* transition is annotated with *AT* referencing to the *alternative task* policy, where another task is executed in case of exception. Hence, trying to reserve a train if an exception event occurs within the flight booking. *R* refers to the *redo* policy, *S* refers to the *skip* policy, *C* refers to the *compensate* policy. „When an exception within a task occurs, an event is raised and a recovery transition will be enabled and fired.“ [HBM08, p. 6].

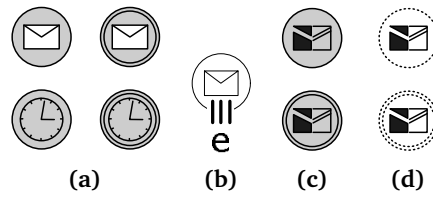


Figure 6.2: Modeling alternatives of non-interrupting events.
 (a,b) BPMN2.0, (c) projection to iBPMN, (d) alternative representation

Concluding, SARNs are an interesting approach for handling exceptional events in petri nets. If they are extended to interaction petri nets, they would provide an interesting approach for formal verifications of interaction exception handling. Due to time and space limitations, this is left to future work.

6.2 Graphical Integration of Intermediate Events

Having discussed the state of the art of exceptions in business process modeling, this section deals with the graphical integration of intermediate events in interaction models. Thereby, the main point of interest is the extension of iBPMN by additional intermediate events as well as non-interrupting events, CPSs and additional artifacts.

Introducing intermediate events to iBPMN is not a complete new task, since some of them are already mentioned. For instance, iBPMN already provides a start event, an end event as well as start timer and intermediate timer. Also atomic interactions, such as the start interaction and intermediate interaction are supported, but without event character. Since their non-atomic counterparts contain a *receive* message task which can be viewed also as an event, interactions can be viewed as events too. However, non-interrupting events, error events, compensation and termination as well as attached intermediate events are not recognized in iBPMN.

Intermediate Events are discussed in Section 4.4 on page 41 and Section 5.3.3 on page 49. BPMN already provides a good solution for a representation of error events, compensation and termination. Thus, these artifacts are introduced to iBPMN in this thesis. The result of the *extension of iBPMN through intermediate events* is shown in Table 6.1.

Non-Interrupting events are currently not mentioned in BPMN1.2, but in the latest BPMN2.0 [BPM08] proposal. Thereby, non-interrupting events have a gray background as shown in Figure 6.2a. Since iBPMN supports the interaction modeling style, these solutions have to be projected to iBPMN as shown in Figure 6.2c. However, this approach is controversy discussed in the community of BPM, since it is hard to write such representations on the blackboard [Gro08]. Thus, a second variant of representing non-interrupting events is proposed in this thesis as shown in Figure 6.2d. Thereby, not the background color is changed. Instead, the surrounding circles are marked with dotted lines, inspired by the exceptional flow of SARNs [HBM08].





























	Start	Catch	Interm.	Throw	End
Event					
Non Interrupting					
Interaction					
Non Interrupting					
Timer					
Non Interrupting					
Error					
Compensation					
Termination					

Table 6.1: Overview on intermediate events in extended iBPMN

In addition to BPMN2.0, also non-interrupting start events and end events are introduced in this thesis, to provide the ability of modeling complex scenarios. These events are especially needed for the usage of an *inline event handling* similar to subprocesses as discussed next. This also redundantize the need of an additional *event marker* [BPM08], which is shown in Figure 6.2b.

Finally, Table 6.1 gives a complete overview on all intermediate events of the extended iBPMN. Additional events may be useful in interaction models too, but due to the context of this thesis only these events are regarded. Start events can be used within normal interaction models as well as inline handlers. Especially, in combination with error-, compensation- and termination-startEvents, customized FCT handling can be specified. Intermediate events are used within normal interaction models as well as *boundary representation* and *inline event handlers*. Boundary representations and inline event handlers are adapted from [BPM08]. The boundary representation refers to the attachment of events on the boundary of an activity and inline event handlers refer to the usage of a special handling constructs within an activity. Both representations are discussed in the next section in detail. Notice, that error, compensation and termination events may be thrown to a particular participant and thus appears as catching and throwing events and not only as end events. End events are used to mark the end of a particular control flow path and are used within normal interaction models, boundary representation and inline handlers. Notice, that non-interrupting control flow paths are ended by their corresponding end events (cf. Figure 6.3c), but may also turn into interrupting events in case of exception (cf. Figure 6.3e). In case of interrupting events, the control flow of the handler may join the normal control flow after the corresponding CPS (cf. Figure 6.3f and Figure 6.3g). This is discussed in the next section.

Section 3.3.2 on page 30 discusses *trivial functionalities* of exception handling and identifies two requirements: (i) ignore an exception and (ii) record an exception. For this, no separate modeling artifacts are introduced to iBPMN, since trivial functionalities can be also achieved using *properties*. Setting the *ignore* property leads to the structural constraint that the exception handler MUST NOT contain any other actions (i. e. only contain a catching error-event directly followed by an end event). The intended semantic of the ignore property will continue the choreography without any exception handling. Setting the *record* property leads to the storage of all related exceptional data in a database. This database can be used to enable knowledge-based exception handling as in [KD00], or for the advancement of the process interaction.

6.3 Graphical Integration of Cross-Partner Scopes

Section 5.4 on page 56 discusses CPSs and its examples give a first hint of their representation. According to BPMN-scopes, CPSs are represented as rectangles, as shown in Figure 6.3a.

To specify intermediate events in combination with scopes, BPMN2.0 proposes two approaches: (i) boundary events and (ii) inline handlers [BPM08]. This is similar to the approach of [Pfi07; PDKL07]. *Boundary events* refer to events attached to the boundary of scopes, as shown in Figure 6.3b and Figure 6.3c. Modeling every event as boundary representation may clutter up the model, since event handling can become rather complex and naturally there are many events. Thus, an additional concept is needed to keep the model simple and clear. For this reason *inline handlers* are proposed similar to subprocesses as shown in Figure 6.3d and Figure 6.3e. Thereby, intermediate events and intermediate end events are used within the boundary representation as shown in Figure 6.3b and Figure 6.3c. Inline handlers begin with start events followed by intermediate events and intermediate end events as shown in Figure 6.3d and Figure 6.3e. In case of non-interrupting events either a non-interrupting end event (cf. Figure 6.3c) or a throwing end event must be used (cf. Figure 6.3e). In case of interrupting events, the control flow may either use an end event (cf. Figure 6.3b and Figure 6.3d) or join with the normal control flow after its corresponding scope (cf. Figure 6.3f and Figure 6.3g). The join with the normal control flow can be arbitrary since *Death Path Elimination* (DPE) [CKLW03] can disable the normal control flow. Thereby, the CPS is interrupted on receipt of the interrupting intermediate event and DPE is triggered to the outgoing links of the CPS. As illustrated in Figures 6.3h and Figure 6.3i, the normal control flow path (1) joins with the exceptional path at a certain point or (2) reaches an end event. In case of overlapping CPSs, separate exception handlers have to be defined for CPS_1 , CPS_2 and their overlap as shown in Figure 6.3j (cf. Section 5.4.6 on page 62). Concluding, inline handlers are recommended since they provide a clear semantic. Due to its cross-partner nature, CPSs possibly have many events associated with it. Thus, both attached event handlers and inline handlers are introduced to iBPMN.

Sections 5.4.1 and Section 5.4.2 on page 57 discuss that with CPSs a general scoping concept is provided that can be used with an extensible set of policies. These policies are selected at mod-

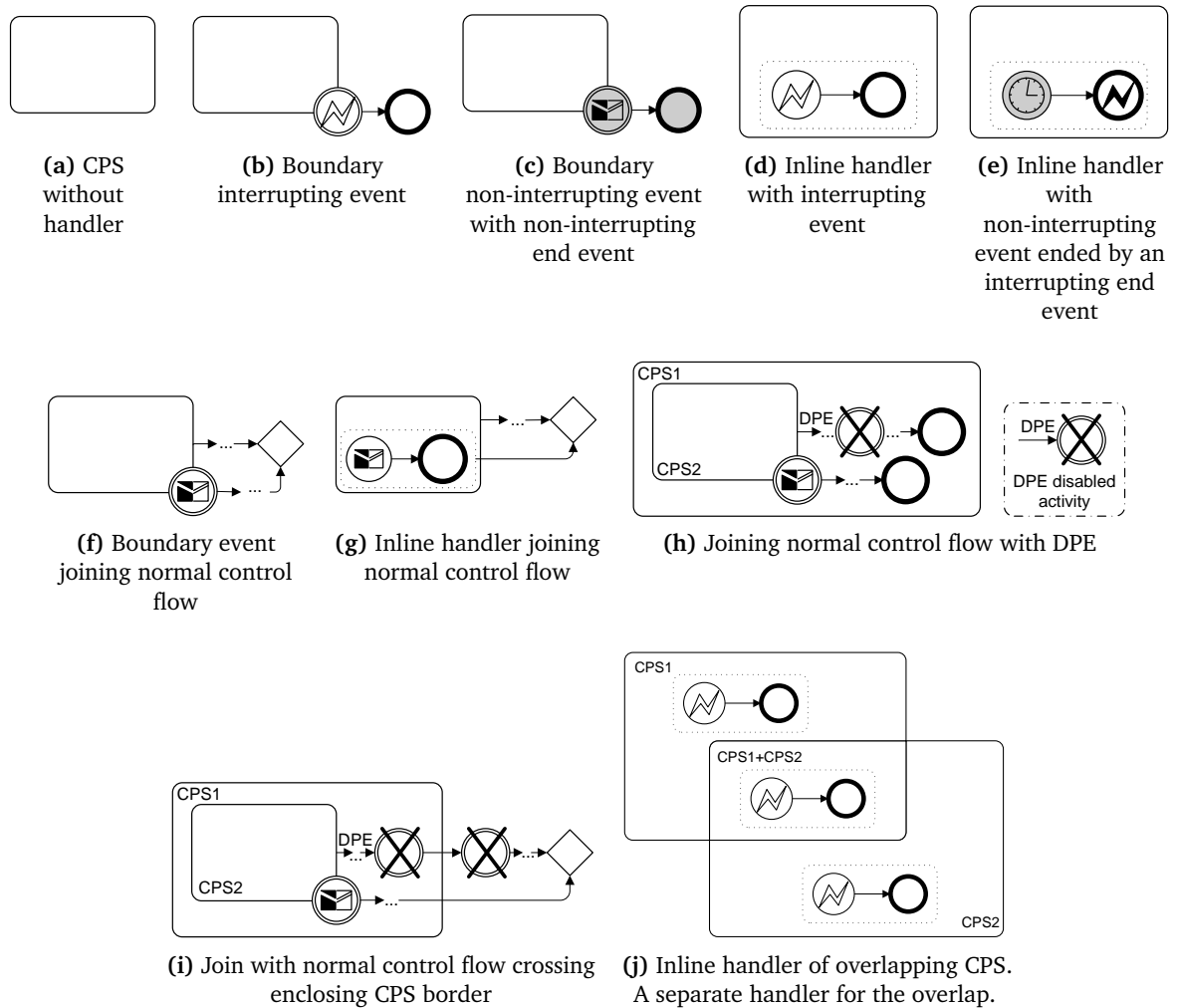


Figure 6.3: Overview of CPSs in extended iBPMN

eling level using a property. Such a policy-selection leads to a particular coordination protocol at runtime, though freeing the modeler to cope with special aspects of the coordination.

6.4 Graphical Integration of Other Artifacts

Having introduced intermediate events and CPSs, there are two artifacts left which are discussed. *Silent Actions* are presented in Sections 5.3.1 and Section 5.3.2 mentions *no action* constructs. BPMN already provides a sufficient representation with the *task* activity which is ported to iBPMN as in Figure 6.4a. Thereby, the distinction between both is given by a property. To gain an unique textual representation the nature of a particular action can be written inside the activity as in Figure 6.4b and Figure 6.4c.

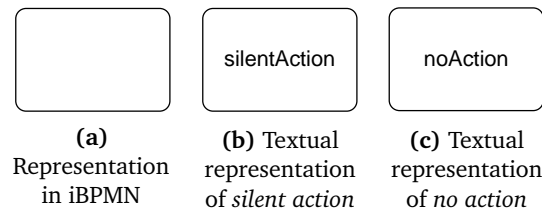


Figure 6.4: Silent- and no action construct in extended iBPMN

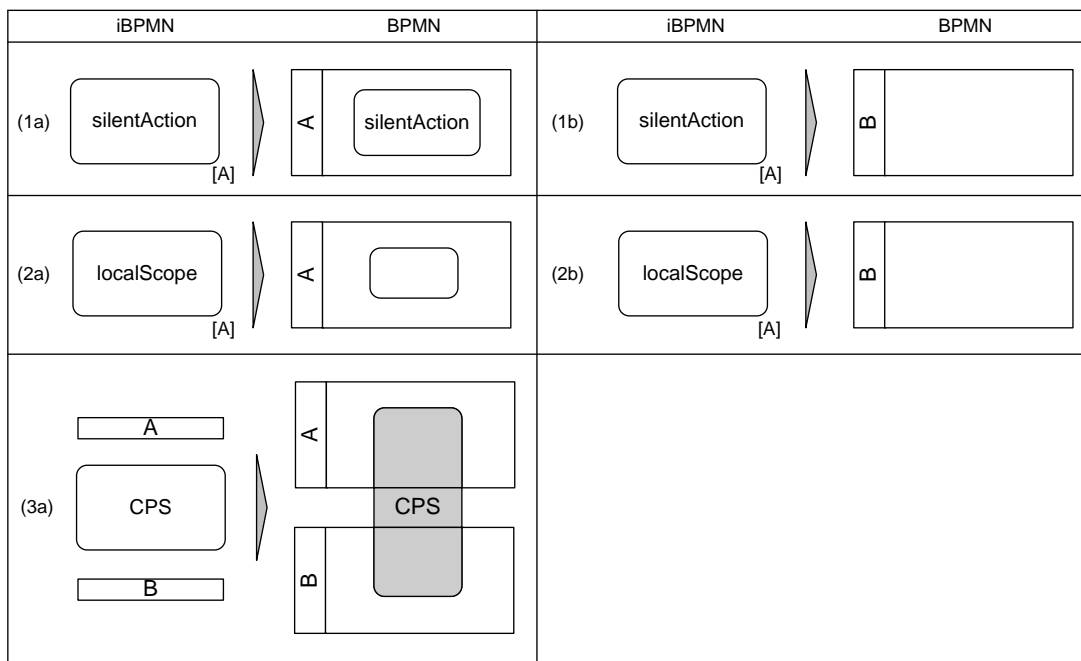


Figure 6.5: Usage of controlling roles

6.5 Usage of Controlling Roles

Section 2.2.5 on page 20 presents controlling roles referring to the owner of a control flow or data flow decision. With interaction models redundant specifications of such roles can be avoided. Thus, the new artifacts introduced to iBPMN provide the ability of controlling roles.

For instance, if a model requires a *silent action* in one single participant *A*, the *silent action* has to be annotated with *[A]* as shown in Figures 6.5(1a) and 6.5(1b). The resulting interconnection model of *A* will then contain the *silent action* as opaque activity, while other participants are not tackled by this modeling. If no controlling role is specified for a certain artifact it is treated like artifacts where all involved participants are specified. As illustrated in Figures 6.5(2a) and 6.5(2b), the scope *S* is annotated with the role *[A]*. Thus, *S* refers to a local scope

appearing in participant *A*. If the annotation of *S* is omitted, the scope turns into a CPS involving participants *A* and *B* as shown in Figure 6.5(3a) (assuming that there are no other participants than *A* and *B*). All other artifacts are handled in a similar way. The particular runtime semantics are discussed in Chapter 7.

6.6 Evaluation of Modeling Artifacts According to Exception Handling Strategies

After Section 3.3 on page 28 discusses several exception handling strategies, this section provides an evaluation of the proposed modeling artifacts of this chapter against the exception handling strategies. Thereby, three aspects are discussed: (1) exception handling according to functional views, (2) exception handling according to the continuation behavior and (3) exception handling using detached execution mode.

6.6.1 Modeling Exception Handling According to Functional Views

Section 3.3.2 on page 30 discussed exception handling from a functional view. Thereby, four main categories are identified: (i) trivial, (ii) basic functionality, (iii) special situations and (iv) transactional environment.

The *trivial functionalities* are fully supported using the *ignore* and *record* properties of Section 6.2 on page 73.

Basic functionalities are partly supported by modeling artifacts. Thus, only the *terminate* activity is reflected in modeling. Additionally, *suspension* and *resumption* are supported implicitly via the pluggable framework which is discussed in Chapter 7.

Out of the *special situation* category, there are multiple alternatives. For instance, *compensation* is reflected using the throwing compensation event. *Procedural exception handlers* are supported using either *boundary events* or *inline handlers*. The *propagation* of exceptions is supported implicitly as well as explicitly. Implicit propagation is triggered when an exception handler is unable to handle the exception using dominance trees, while explicit exception propagation can be triggered using the throwing error event. *Re-execution*, *pipeline* and *reallocation* are not reflected by introducing new artifacts to BPMN for compliancy reasons. They can be achieved, however, using procedural exception handler.

Transactional environments are supported due to an extensible set of policies as discussed in Section 5.4.2 on page 57. During this thesis, the default all-or-nothing policy is discussed.

6.6.2 Modeling Exception Handling According to the Continuation Behavior

Section 3.3.3 on page 31 discusses exception handling regarding especially the continuation behavior of exceptions. Thereby, *continuation exception handling* and *abortion exception handling* are identified. *Continuation exception handling* is supported by the introduction of non-interrupting events, while *abortion exception handling* is supported using interrupting events. Thus, a flexible way of handling exceptions is provided, canceling processes only as a last resort.

6.6.3 Modeling Exception Handling Using the Detached Execution Mode

Section 3.3.4 on page 32 discusses exception handling from a rule-based perspective. Thereby, the *detached execution mode* was discussed, referring to the fact that exception handling contexts are different to those where the exceptional event was generated.

During this thesis, the detached execution mode of exception handlers using exception handling processes is not treated. But due to *process-oriented exception handling* involving an exception handling coordinator, such an extension may be provided by future work. Process-oriented exception handling is discussed in Section 3.3.5 on page 32.

6.7 Summary

This chapter introduces additional modeling artifacts into iBPMN with respect to the discussed theoretical foundation of Chapter 5. Thereby, the main focus are extensions through intermediate events (cf. Section 6.2 on page 73), *Cross-Partner Scopes* (CPSs) (cf. Section 6.3 on page 75) and additional artifacts (cf. Section 6.4 on page 76). Section 6.4 on page 76 evaluates the proposed artifacts against the support of the discussed exception handling strategies of Section 3.3 on page 28.

Concluding, with the introduced modeling artifacts there is a large set of exception handling strategies supported by the introduced exception modeling artifacts. Finally, their runtime support is presented in the next chapter.

7 From a Choreography with Interaction Exceptions to the Runtime

After the previous chapter introduces a graphical representation of the concepts discussed in Chapter 5, this chapter copes with the mapping of this graphical representation to the runtime. Corresponding to the choreography design approach (i. e. top-down or bottom up), there is no need for a direct translation of interaction models to executable BPEL code. Thus, iBPMN is transformed into a logical model using BPEL4Chor. Having generated the logical model, executable BPEL processes can be either grounded using already existing processes that conforms to the choreography or they can be developed by using abstract BPEL processes which can be generated out of the interaction model. The generation of abstract BPEL processes refers to the translation of interaction models to interconnection models which is described in [DB07; DKLW09; Dec09a]. Since Cross-Partner Scopes are not part of particular participant processes, a coordinator has to be generated, referring to the discussed solution of external coordination in Section 5.4.3 on page 58.

7.1 Deriving Interconnection Models

An important task in the top-down choreography design process is the derivation of interconnection models out of interaction models. For this purpose, [Dec09a] proposes an algorithm using the concept of *model reduction* [Zah+08]. Model reduction is common in transformation and optimization of complex models. Thereby, complex models are subsequently refined by eliminating and transforming model constructs.

The algorithm is divided into three phases: (i) conversation phase, (ii) rearrangement phase, and (iii) graphical optimization phase. During the *conversation phase* constructs relevant for the observable behavior are converted into its corresponding constructs of the particular participants. With respect to the specified controlling roles, not every construct is relevant for the observable behavior of a particular participant. Thus, these constructs are converted into τ -nodes which are removed in the rearrangement phase. τ -nodes are placeholders which are used for the model reduction indicating no operations. After the conversation, the *rearrangement phase* removes all τ -nodes as well as empty complex interactions and rearranges the control flow accordingly. Finally, the *graphical optimization phase* validates the compliance with the syntactical constraints of BPMN and rearranges the control flow in case of variations.

For a better understanding, Figure 7.1 shows an overview of the conversation phase. Figures 7.1(1*) and Figure 7.1(2*) show the transformation of interactions. Since interactions

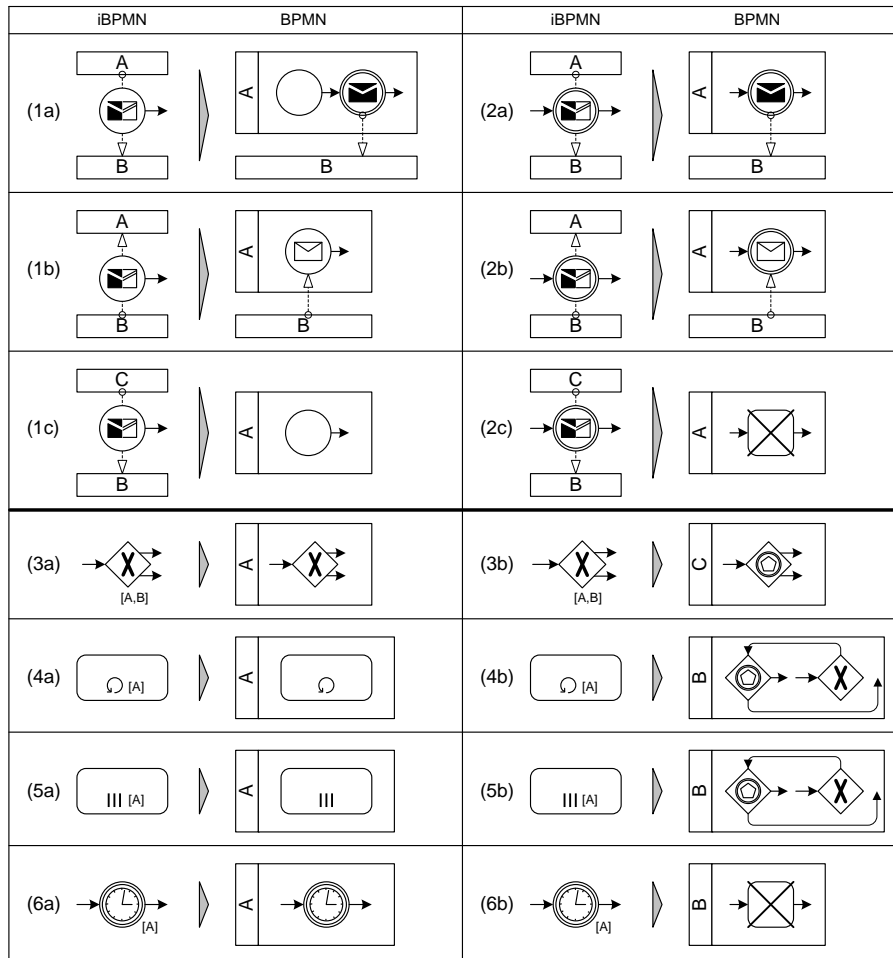


Figure 7.1: Transformation rules for the conversation phase adapted from [Dec09a]

combine sending and receiving messages in iBPMN, they are broken into in a message send activity and a message receive activity in its corresponding participants. For start interactions, an additional start event is inserted. τ -nodes are inserted into participants not involved in interactions. τ -nodes are represented as crossed rectangle. Figure 7.1(3) shows the transformation of gateways. The specified gateway is inserted into participants which are specified within the controlling role statement, while an event-based gateway is inserted into other participants since they are not responsible for this decision. „Due to a lack of support for the workflow pattern ‘multiple instances without apriori knowledge’ [in iBPMN]“ [DW08, p. 17], *loop unrolling*¹ [Muc97] has to be applied in case of multi instances and loops, as shown in Figures 7.1(4*) and 7.1(5*). Finally, Figure 7.1(6) shows the transformation of timers, which is similar to those of interactions. A timer is inserted into a participant if it is specified in the controlling role, if not a τ -node is inserted.

¹ „Loop unrolling replaces the body of a loop by several copies of the body and adjusts the loop-control code accordingly.“ [Muc97, p. 559]

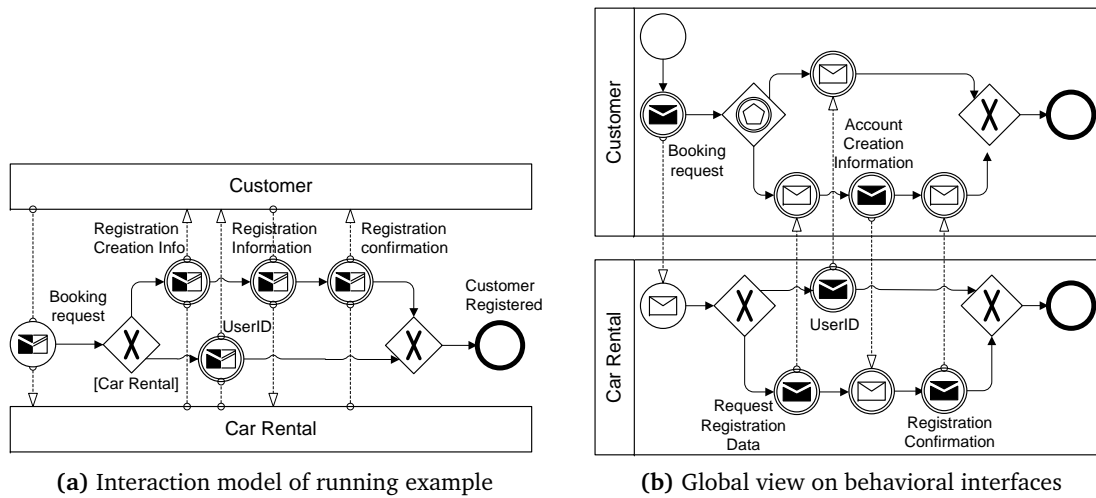


Figure 7.2: The transformation phase of interaction models to interconnection models

For instance, Figure 7.2a shows an interaction model of the running example introduced in Chapter 4. Figure 7.2b presents the result after applying the rules of the conversion phase.

In addition to the artifacts of [Dec09b], *silent actions* and throwing *error events*, *compensation events* and *termination events* are transformed similar to timers, while *no actions* are transformed into τ -nodes. This is shown in Figure 7.3.

Local scopes and their handlers are also transformed directly into the particular process while τ -nodes are inserted into other processes as shown in Figures 7.4(1a) and 7.4(1b). *Intermediate events* used in exception handlers are more difficult than local scopes. Since CPSs are imposed on the process model, they do not appear in the particular process implementation, but their *events* do. Hence, the decision of the outcome of a particular CPS may depend on the externalized decision of another scope. For instance, in the running example (Figure 4.4 on page 41) the decision of the outcome of the scope involving the *airline*, *hotel* and *car rental* may depend on the interaction event of the *customer* if he cancels his journey.

The transformation of a CPS with an intermediate message event is shown in Figures 7.4(2a) and Figure 7.4(2b). In participant *B* of Figure 7.4(2a) an event-based XOR gateway and a message receive activity are added. Figure 7.4(2b) shows that the XOR gateway and a message send activity are added to participant *C*, since *C* is the controlling role of this event. It is important to notice, that on receipt of the message the CPS is aborted. The abortion is based on continuation behavior of exceptions as discussed in Section 3.3.3 on page 31. This abortion is achieved through external coordination of the CPS which is discussed in detail in Sections 7.3 and Section 7.4 on page 92.

Non-interrupting events refer to the continuation behavior of CPSs, thus their handling differs from the handling of intermediate events. Hence, non-interrupting events appear as normal BPMN events in the particular processes. In contrast to interrupting events, non-interrupting

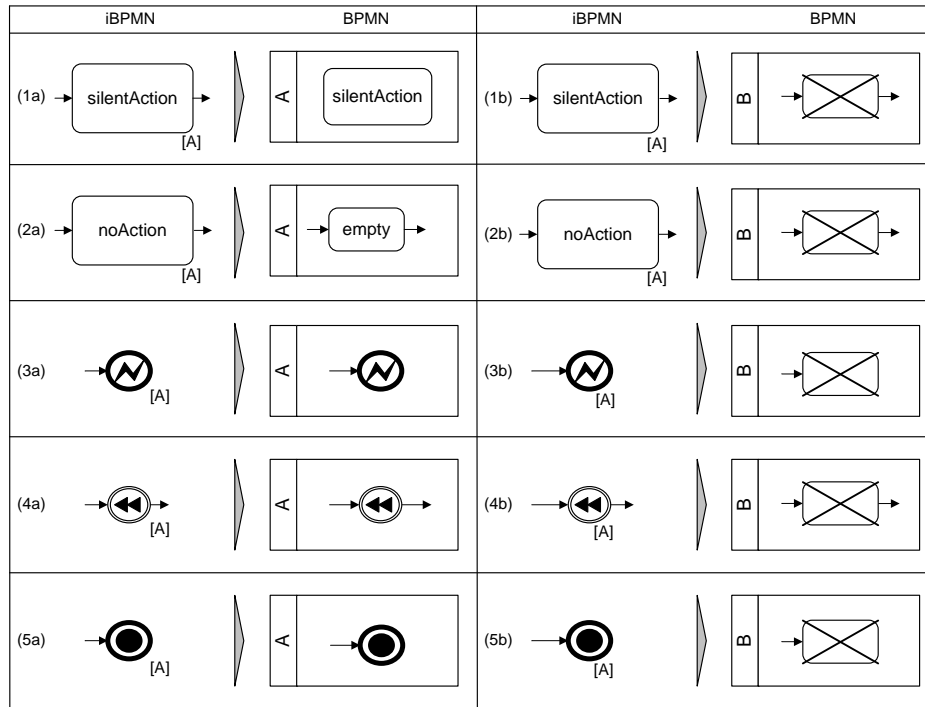


Figure 7.3: Transformation rules for the conversation phase: additional artifacts and events

events may appear multiple times. Thus, Figures 7.4(3a) and Figure 7.4(3b) have an additional control flow link as *back-edge* to the event-based XOR gateway.

It is important to notice, that the presented transformations for Figures 7.4(2) and Figure 7.4(3) are invalid BPMN models, since the event-based XOR gateway is not followed by an event in every branch. Furthermore, there are other possibilities for compliant interconnection models not shown in this transformation. These other possibilities involve looping structures or simple occurrences of the particular events. Since interaction models focus on the *atomic observable behavior* of business process models, both observations are not handled in the transformation phase. Instead, the transformations focus on the semantic of the interaction model and can be refined or simplified in subsequent design phases. Thus, both observations are to be handled in the subsequent transformation or optimization phase as well as the particular process implementation phase adding the non-observable behavior [LKL07]. Especially in the error-prone process implementation phase valid BPMN models are to be ensured referring to the anomalies of interconnection models in Section 4.1 on page 37.

Figure 7.4(4a) provides the view on the observable behavior of A. A is specified in the controlling roles of the CPS and thus, the corresponding activities which are specified inside of the CPS are added. In contrast to Figures 7.4(2a) and 7.4(2b), A is not involved in the intermediate interaction. In addition to the transformation corresponding to controlling roles in Figure 7.4(4a), Figure 7.4(4b) illustrates the transformation of CPSs into other participants. Thereby, τ -nodes are added to other participants.

Listing 7.1 The structure of choreography scope fragment definition in participant behavior of BPEL4Chor presented in [Ruf07]

```

1 <choreographyFragment name="NCName">
2   <fragmentFaultHandlers>?
3     <catch process="NCName" faultName="QName" faultVar="BPELVariableName"
4       (faultMessageType="QName" | faultElement="QName")? />*
5     <catchAll/>?
6   </fragmentFaultHandlers>
7   activity+
8 </choreographyFragment>

```

Furthermore, Figures 7.4(5a) and 7.4(5b) show the transformation using additional subsequent control flow. Thereby, the transformation of inline handlers is similar to those of boundary events. Figure 7.4(5a) shows the usage of default control flow behavior by using an end event in the handler. If an interrupting end event is used in a handler, the CPS proceeds with its outgoing links due to the cancellation behavior of interrupting events. In cases when an exception handler wants to skip certain activities, arbitrary control flow can be joined as shown in Figure 7.4(5b).

The rearrangement phase removes all τ -nodes and empty complex interactions after the conversation phase. Furthermore, structural constraints are validated in the optimization phase. It is important to notice, that the rearrangement phase may encounter situations where a *valid BPMN model cannot be provided*. Such situations occurs due to both *non-free-choices* [DE95] and XOR event-based gateways followed by message send events [Dec09a]. For instance, Figure 7.5a illustrates a non-free-choice and Figure 7.5b an invalid XOR event-based gateway use. Since both phases are not necessary for a general understanding of the concept they are not discussed further. A detailed description of both the rearrangement phase and the optimization phase is given in [Dec09a].

7.2 Mapping iBPMN to BPEL4Chor

Having discussed how interaction models can be derived out of interaction models, a further representation is needed, since CPSs are not part of particular participant processes. According to the logical model discussed in Chapter 1, a representation in the logical model BPEL4Chor is needed. As discussed in Section 2.2.2 on page 18, BPEL4Chor consists of three artifacts: (i) Participant Behavior Descriptions (PBDs), (ii) topology behavior description and (iii) participant grounding. PBDs are either existing or can be derived with the algorithm of Section 7.1. Furthermore, the participant grounding refers to the technical configuration and thus is also beyond the focus. Thus, this section focuses on the representation of CPSs in BPEL4Chor topology descriptions.

For representing CPSs in BPEL4Chor, [Ruf07] proposes an approach using fragmented scopes embedded in the particular participant processes as shown in Listing 7.1. CPSs are then assembled out of the participant specific fragments in the topology, as shown in Listing 7.2.

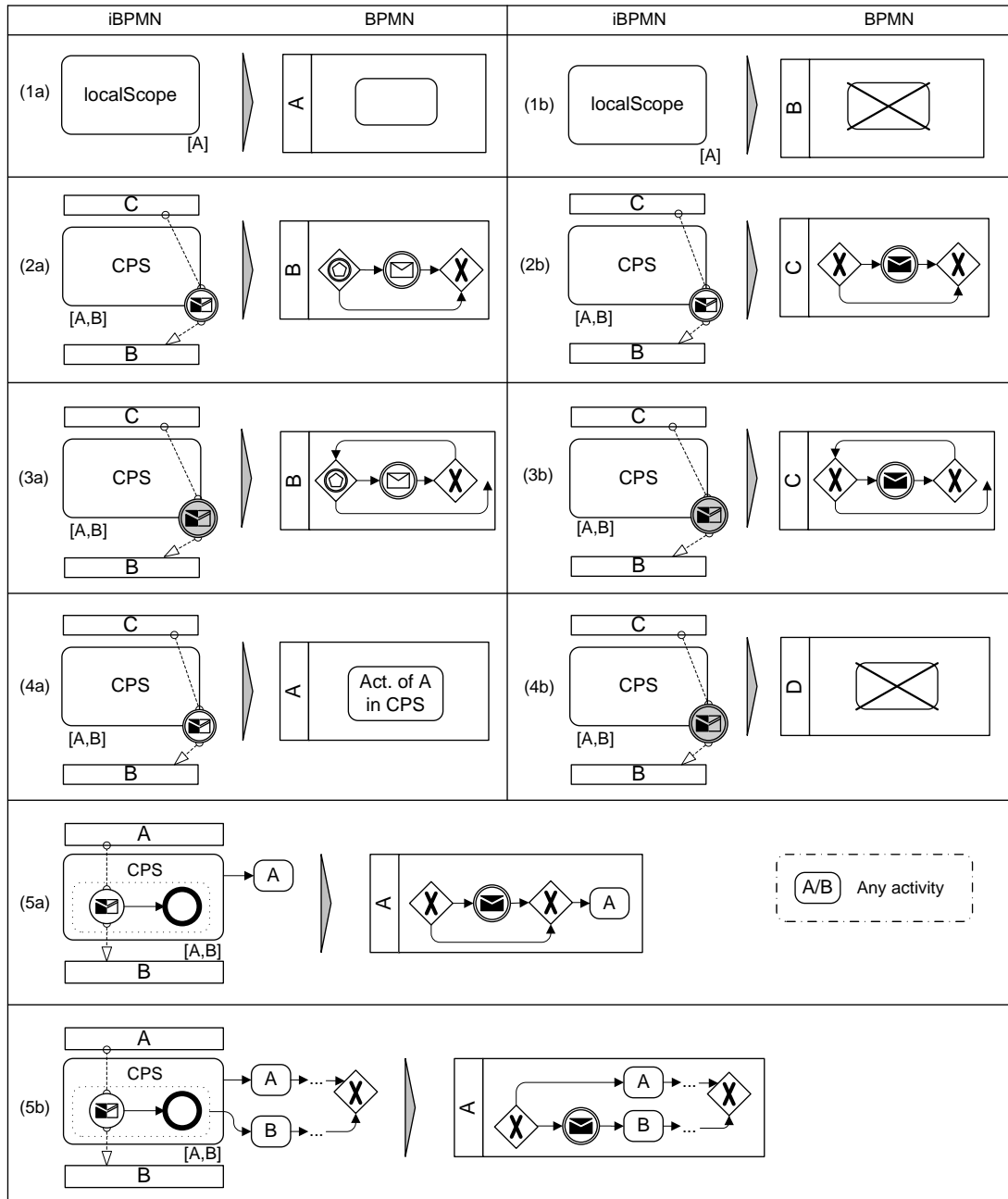


Figure 7.4: Transformation rules for the conversation phase: Cross-Partner Scopes

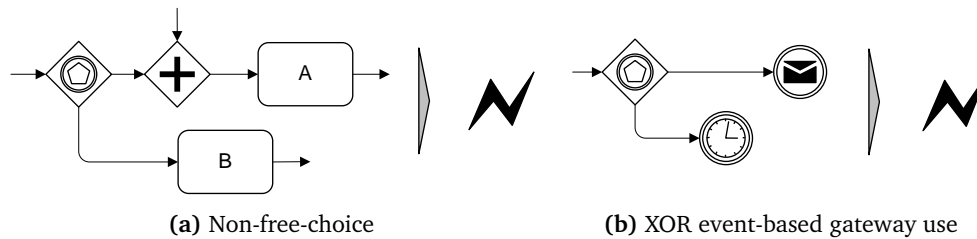


Figure 7.5: Invalid BPMN models in rearrangement phase according to [Dec09a]

Listing 7.2 The structure of choreography scope fragment in the topology of BPEL4Chor presented in [Ruf07]

```

1 <choreographyScopes>?
2   <choreographyScope name="NCName">+
3     <fragment process="NCName">+
4       <requiredVariable process="NCName" name="BPELVariableName" /*
5     </fragment>
6   </choreographyScope>
7 </choreographyScopes>

```

Out of this description exception handlers are generated automatically. Thereby, „all fragments listed in one *<choreographyScope>* are coordinated together“ [Ruf07, p. 48].

In contrast to [Ruf07], the approach taken in this thesis follows the requirement of leaving the process model untouched (cf. Section 5.4.3 on page 58). Thus, the solution of [Ruf07] has to be extended to fulfill the requirement. The obvious approach is to combine both representations to exactly one which is placed in the topology description. Hence, the participant descriptions remain untouched and the necessary representation of CPSs is achieved.

This new representation is shown in Listing 7.3. Thereby, the activities for every CPS process have to be provided as well as the description of the corresponding exception handlers of the particular CPS. Notice that fault handlers of overlapping CPSs areas are *not* defined

Listing 7.3 The proposed structure of Cross-Partner Scopes in the topology of BPEL4Chor

```

1 <choreographyScopes>?
2   <choreographyScope name="NCName">+
3     <fragment process="NCName">+
4       activity+
5     </fragment>
6     <requiredVariable process="NCName" name="BPELVariableName" /*
7     <faultHandlers>?
8       <catch faultName="QName" faultVar="BPELVariableName" (faultMessageType="QName" |
9         faultElement="QName")? /*
10      <catchAll/?>
11    </faultHandlers>
12  </choreographyScope>
13  <faultHandlers/*> <!-- for overlapping areas -->
14 </choreographyScopes>

```

Listing 7.4 The structure of fault handlers for *overlapping CPSs*

```
1 <choreographyScopes>?
2   <choreographyScope name="NCName"/>+
3   <faultHandlers>?
4     <faultHandler choreographyScope="NCNames">+
5       <catch faultName="QName" faultVar="BPELVariableName" (faultMessageType="QName" |
6         faultElement="QName")? />*
7     <catchAll/>?
8   </faultHandler>
9 </faultHandlers>
</choreographyScopes>
```

Listing 7.5 The structure of exception graphs

```
1 <exceptionGraph>?
2   <exception name="NCName">
3     <exception/>*
4   </exception>
5 </exceptionGraph>
```

within the `<choreographyScope>` definition. They are defined in the `<faultHandlers>` part of the `<choreographyScopes>` definition. In Listing 7.3(l. 12) only exception handlers are shown (called *faultHandlers* in order to be compliant to the BPEL terminology). Furthermore, compensation and termination handlers can be included analogously, but this is out of the scope of this thesis. The exception handlers provide either a behavior for particular exceptions or general behavior in case of catching all exceptions.

7.2.1 Exception Handler for Overlapping Scopes

Section 5.4.4 on page 60 discusses the notion of overlapping Cross-Partner Scopes (CPS). In addition, Section 6.3 on page 75 discusses *boundary events* and *inline handlers*. Thereby, the requirement arises to specify separate exception handlers for overlapping areas. Thus, an additional artifact is introduced to the `<choreographyScopes>` part, which is shown in Listing 7.4. After the definition of the particular CPSs, a separate `<faultHandler>` part is injected, specifying the corresponding CPS-overlapping with its intended fault handler.

7.2.2 Representation of Exception Graphs in BPEL4Chor

In Section 5.3.5 on page 50 *exception graphs* are introduced. Thereby, the structure of exception graphs arises due to the particular interplay of the choreography participants and must be specified by the choreography designer. To increase maintainability as well as readability and understandability, Listing 7.5 proposes a structure for exception graphs which can be included into the topology description. This structure can be used by coordinators to determine the resolving exception as discussed in Section 7.5 on page 101.

Listing 7.6 The structure of dominance trees

```
1 <dominanceTree>?  
2   <node name="NCName">  
3     <node/>*  
4   </node>  
5 </dominanceTree>
```

7.2.3 Representation of Dominance Trees in BPEL4Chor

Section 5.3.6 on page 52 mentions exception propagation using the concept of dominance. Thereby, the corresponding control flow graph is implicitly given by the *Participant Behavior Descriptions* (PBD) and the dominance tree can be generated out of these PBDs. To support a clear model, the implicit nature of dominance trees may be represented explicitly. For this reason, Listing 7.6 shows a possible representation of dominance trees. This representation is similar to the previous discussed exception graphs. As a consequence, this structure can be used by coordinators for exception propagation as discussed in Section 7.6 on page 102.

7.2.4 Summary

This section discusses an extension of the BPEL4Chor topology to represent the proposed artifacts of Chapter 6. As discussed in Chapter 1, BPEL4Chor acts as logical model of interaction choreographies. Thus, the extended BPEL4Chor provides all information needed to coordinate a choreography: (i) the *participant topology* describes all participant declarations, message links as well as Cross-Partner Scopes. Furthermore, the induced propagation hierarchy of the Cross-Partner Scopes can be derived statically out of the topology description. (ii) *participant behavior descriptions* can either be transformed out of the interaction model or are already existing. If they are transformed out of the interaction model, they must be refined into executable processes, since internal details are not fully provided within interaction models. Although, abstract issues may be provided when using *silent* and *no actions*. (iii) if all necessary participant behavior descriptions are determined, they can be grounded to concrete realizations.

At this point, an interesting observation arises, referring to the need of *silent* and *no action* constructs of Sections 5.3.1 and Section 5.3.2. This observation comes with the need to decide start and end of a CPS with the information provided from a interaction model and without changing the process model. Without using *silent* and *no actions*, this is a difficult decision, since traditional interaction models only provide atomic interactions as essential building blocks. Using these atomic interactions for the decision of the start and end of a CPS restrict the resulting implementation to processes starting only with message exchanges. Thus, an additional concept is needed to support this decision on interaction level. For instance, if a *silent action* is the first activity of a CPS, this would be reflected in the topology description of the corresponding CPS. Thus, a particular coordinator can be informed by the BPEL engine on instantiation of this activity. How the coordinator is derived out of the BPEL4Chor description and how the information between coordinator and BPEL engine is achieved, is discussed in the next sections.

7.3 Runtime Support of Interaction Exceptions: The BPEL Engine²

CPSs can be implemented through two approaches: (i) using a *pluggable event framework* [KKL07; Ste08] or (ii) using a simulation through BPEL constructs as in [Ruf07]. Using the *pluggable framework* allows a process independent implementation, while a simulation through BPEL constructs follows a process dependent implementation. Since this thesis focuses on CPSs which are imposed on processes without changing their implementation, the pluggable framework is used for implementation. Thus, the pluggable framework is introduced.

The main idea of the pluggable framework is the functional extension of long-running processes without having to modify or restart them. Furthermore, it is a main requirement to add new functionalities without having to change the workflow engine. To achieve such a solution particular events occurring during the process execution have to be externalized. Thus, additional activities can be executed or injected when the particular events occur. For instance, use-cases for the pluggable framework can be found in (i) the monitoring of active processes to control quality of services, or (ii) the execution of *distributed loops* as described in [Pal07] or *split loops and scopes* [KL07]. (iii) especially interested in CPSs, CPSs can be also realized through the pluggable framework, referring to the externalization of transactional behavior as in [Mie06; PML07]. Thereby, the notion of *blocking events* allows the external blocking of processes. Thus, it is possible to use an external coordinator to control the outcome of processes.

The realization of the pluggable framework involves two main spheres of control, which are illustrated in Figure 7.6: (i) the generic controller and (ii) the custom controller. At first events have to be caught in the engine. If an event has to be blocking, the corresponding activity is blocked by the engine. The *generic controller* is responsible for picking events from the engine and the distribution of events to the outside (*event generation*). Furthermore, the generic controller handles *incoming events* and is responsible for resolving blockades through *unblocking events*. To achieve this, the generic controller is glued to the process engine. The *custom controller* is an external application which can register to the generic controller. Thereby, the custom controller can either monitor events or register for a blocking access to particular events. In case of event monitoring, the generic controller informs the custom controller through (*outgoing*) events. For a blocking access to particular events, the generic controller waits for *incoming events* of the custom controller to release blockades of particular activities.

The pluggable framework can be used to realize runtime support for CPSs and thus exception handling in interaction choreographies. For this, an application has to be designed using the custom controller to register to the generic controller. Thus, processes on the workflow engine can be monitored by external applications. As a consequence, the pluggable framework is the essential building block for realizing CPSs, since CPSs are (a) imposed on processes without changing their implementation and (b) have to be controlled externally. As presented in [Ste08], the application needs to know (i) the *child activities* of the particular processes which are to be controlled by the CPS. Furthermore, (ii) the *enclosing scopes* of these child activities as well as *Direct Child Scopes* (DCS) are needed. Direct child scopes are scopes lying

² This section is mainly based on [Ste08], other references are explicitly stated.

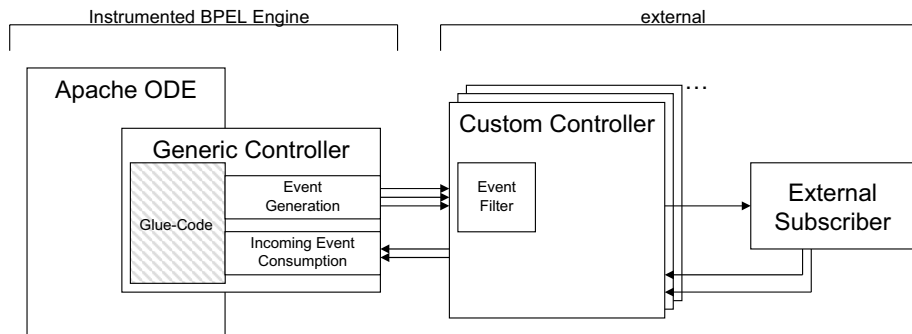


Figure 7.6: Architecture overview of the pluggable framework according to [Ste08]

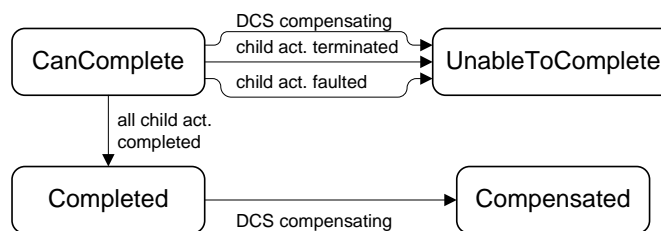


Figure 7.7: Simplified state-chart of a CPS adapted from [Ste08]

directly within a child activity of a CPS. Notice, that child activities may be executed multiple times as well as never in case of branches, loops or event handler. In this cases, the CPS may be also end successful. Since there may exist multiple choreography instances, it must be monitored which process instances belongs to a particular choreography instance, i. e. instance routing. For this purpose, [Ste08] proposes the usage of a *global correlation set*, which has been introduced in [KL06; Kha08]. The semantic of a CPS can be controlled using four states: (1) *canComplete*, (2) *completed*, (3) *unableToComplete* and (4) *compensated*. Figure 7.7 illustrates the simplified state-chart of these states, which are discussed next:

1. *canComplete* The CPS is running in normal mode (i. e. no exception appears)
2. *Completed* The CPS is completed successfully
3. *UnableToComplete* In case of exceptions such as a faulting, compensating or terminating child activity, the CPS cannot complete.
4. *Compensated* After the CPS completed successfully, it may be compensated through compensation of a direct child scope or an enclosing scopes.

Table 7.1 shows an overview of events which have to be used to monitor a process (i. e. these events are used to determine the state of a particular CPS). Thereby, events that may be blocking are marked separately. Furthermore, the right side of Table 7.1 shows the incoming events of the generic controller.

Event	Blocking	
Process_Instantiated	-	
Instance_Faulted	-	
Instance_Completed	-	
Instance_Terminated	-	<u>Incoming Events</u>
Activity_Ready	x	Compensate_Scope
Activity_Executed	x	Fault_To_Scope
Activity_Complete	-	Start_Activity
Activity_Terminated	-	Complete_Activity
Activity_Death_Path	-	Terminate_Activity
Activity_Faulted	x	Continue
Evaluating_TransitionCondition_Faulted	x	Suppress_Fault
Scope_Compensating	x	
Scope_Compensated	-	
Scope_Complete_With_Fault	-	
Scope_Handling_Fault	-	

Table 7.1: Overview of events used for CPS coordination in the pluggable framework according to [Ste08]

7.4 Runtime Support of Interaction Exceptions: The Coordinator

The previous section discusses the engine support of CPSs. This section focuses on the coordinating entity. Thereby, several questions dealing with controlling and coordinating entities are of special interest which are discussed in the following.

During the previous chapter, a graphical representation of exceptions in interaction choreographies is introduced. This graphical representation forms a conceptual model whose transformation to a logical model using an extended version BPEL4Chor is also introduced. According to the discussed choreography management stack of Chapter 1, this logical model forms the starting point for the generation of coordinating entities.

7.4.1 Overview on the Overall Coordination of a Choreography

WS-Coordination (WS-C) (cf. Section 2.3 on page 20) provides an extensible framework for the coordinated outcome of distributed applications. Thereby, WS-C enables applications to create distributed contexts needed to operate in a heterogeneous environment and thus is predestinated for a usage in choreography exception handling. To gain a broad understanding how exception handling can be achieved by using WS-C, the overall collaboration of a BPEL engine with coordinators is investigated. Thereby, this section provides a general overview and discusses alternatives, while the following sections dive into the particular steps dealing with:

- automated collaboration with the BPEL engine
- automated collaboration with the coordinator

Although the previous section discusses the runtime support regarding the BPEL engine, there are alternatives how the pluggable framework can be used for coordination. A first alternative is shown in Figure 7.8a. Thereby, the custom controller is directly collaborating with the coordinator. With respect to the representation of CPSs, especially their activities, there are two alternatives for representing WS-C participants:

- (i) Each CPS-activity is a WS-C participant on its own (cf. Figure 7.8a), or
- (ii) all CPS-activities of a particular choreography-participant which are enclosed by a CPS are summarized by a fragment. Thus, this fragment can act as WS-C participant (cf. Figure 7.8b).

(i) Representing each CPS-activity as a WS-C participant introduces a huge communication overhead, a complex coordinator and a huge delay of the process execution. This results from the necessity to communicate the whole life-cycle of a single activity to the coordinator (cf. incoming events of custom controller in Section 7.3 on page 90). Thereby, the delay of execution remains the main weakness of this architecture, since every event which is blockable needs to be blocked until the coordinator resolves the blockade. Since a high execution delay in the normal execution mode limits the applicability of the approached concept, the execution delay has to be reduced. To reduce this execution delay, a second alternative (ii) represents all CPS-activities of a particular choreography-participant as a fragment. Thereby, it is also possible to reduce the complexity of the coordinator by an increased complexity of the custom controller. I. e. with alternative (i) the custom controller passes every (outgoing) event of the generic controller to the coordinator, while with alternative (ii) the custom controller has to take over coordinating issues as well. Thus, reducing both the number of WS-C participants as well as the process execution delay.

However, the degree of abstraction in the custom controller is a main weakness of such architectures, since the custom controller is completely application-specific. To reduce this complexity and the degree of application-dependence, there is a second alternative possible. This alternative is shown in Figure 7.8c. Here, the coordinator is split into one *global coordinator* and several *local coordinators*. Thereby, the local coordinator runs on the same BPEL engine as its corresponding process. Due to this architecture, the complexity of the coordinators can be reduced. The *global coordinator* is only informed when a fragment is completing or faulting, while the *local coordinators* monitor the fragment activities. Again, such a local monitoring can be either achieved by a different representation of the WS-C participants as discussed earlier.

When the activity-monitoring is left to the custom controller by using the fragment representation, the communication overhead can be reduced and thus the execution delay of the processes. This can be understood by considering the three forms of interactions WS-C provides for its participants: (1) activation service, (2) registration service and (3) protocol specific interactions (cf. Section 2.3 on page 20). Each WS-C participant has to request the creation of a new coordination context using the *activation service*. Having created a coordination context, participants register with a coordination using the *registration service*. After the registration,

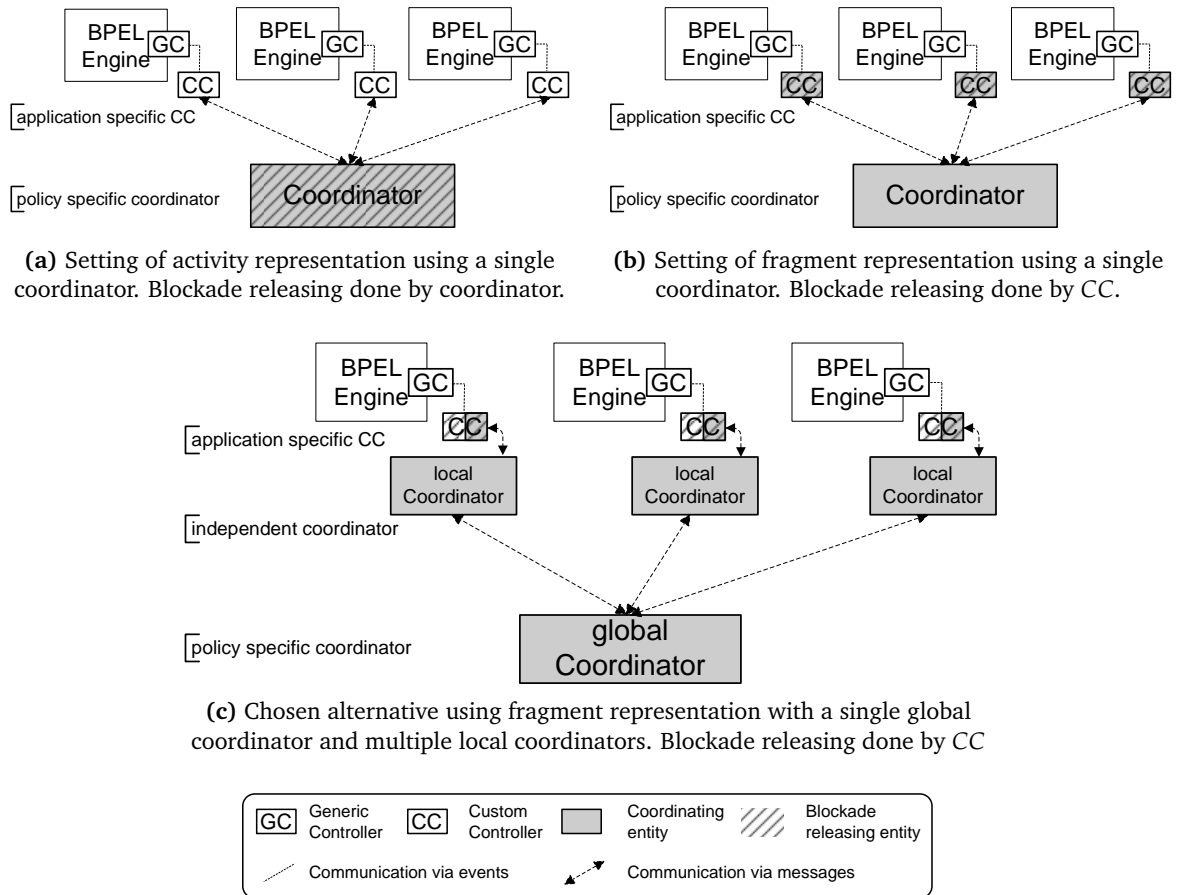


Figure 7.8: Modeling alternatives for the collaboration between BPEL engine and coordinators

the coordinator communicates with its WS-C participants through *protocol specific interactions*. Thereby, (1) and (2) are independent of the type of coordination. However, the number of WS-C participants for a WS-C activity influences the communication overhead. Thus, using the *activity representation* results in a large number of WS-C participants, while the *fragment representation* results in exactly one WS-C participant per CPS fragment. Furthermore, using the *activity/fragment representation* the complete life-cycle of an activity/fragment has to be communicated using protocol specific interactions. Regarding the normal behavior of CPSs where no exceptions occur, this results in *five* interactions using the *activity representation* and *four* interactions using the *fragment representation*:

- For the *activity representation* these interactions are caused by *activity_ready*, *activity_executed* with their unblocking incoming events and a finish-message which can be either *activity_complete*, *activity_terminated*, *activity_death_path* or *activity_faulted* (corresponding to Table 7.1).
- In contrast, using the *fragment representation* leaves the activity-monitoring to the custom controller (i. e. the custom controller blocks every blockable activity and releases

blockades immediately until the coordinator states a different handling). Thus, only fragment-relevant messages have to be communicated to the local coordinator. These are at least two: one for the initialization of the fragment and one for the finishing. Additionally, the local coordinator has to pass through these messages to the global coordinator.

Thus, it is more suitable to use the fragment representation and leave the activity-monitoring to the custom controller. Thereby, reducing both the complexity of the coordinators and the communication overhead for the coordination in the non-exceptional case.

The *reduced complexity* of the coordinators also leads to a generic hierarchy of coordinators. Thereby, the *custom controllers* form the lowest level, since they are application-specific and has to be generated for every particular use case. *Local coordinators* form the next hierarchy level and are completely independent of a particular application. Finally, *global coordinators* form the highest hierarchy level. Similar to local coordinators, global coordinators are completely independent of a particular application. However, global coordinators represent a particular policy as discussed in Sections 5.4.1 and Section 5.4.2 on page 57. Thus, an extensible set of policies can be provided by additional global coordinators, while local coordinators and custom controllers remain untouched.

The *reduced communication overhead* in non-exceptional cases comes with a decreased process execution delay. In Figure 7.8a the coordinator is responsible for releasing blockades. In contrast, using split coordinators as in Figure 7.8c, allow custom controllers to release blockades immediately until the coordinator states a different handling. Thus, providing a fast non-exceptional execution of the choreography, while having more communication overhead and thus a slow execution in exceptional situations.

As a consequence, the second alternative using split coordinators is investigated in detail in the following section for the all-or-nothing policy. Thereby, the following issues are discussed:

- the generation of the custom controller out of the logical model
- the local coordination protocol
- the global coordination protocol

For a better understanding of the overall interaction of the discussed alternative, Figure 7.9 gives a concluding overview of it. Thereby, four actors are involved: (i) the generic controller, (ii) the custom controller, (iii) the local coordinator and (iv) the global coordinator. The *generic controller* uses *outgoing events* which are sent to the custom controller. The *custom controller* responds with *incoming events* to the generic controller. Furthermore, the custom controller is responsible for the creation of the CPS fragment WS-C activity by the local coordinator and for monitoring the life-cycle of the corresponding fragment activities. If exception-relevant events of a fragment occurs, the *local coordinator* is informed which takes over control. The local coordinator itself is a WS-C participant of a corresponding CPS which is controlled by a global coordinator. For the sake of brevity, a detailed overview of the particular interaction between these actors is given in Appendix A.

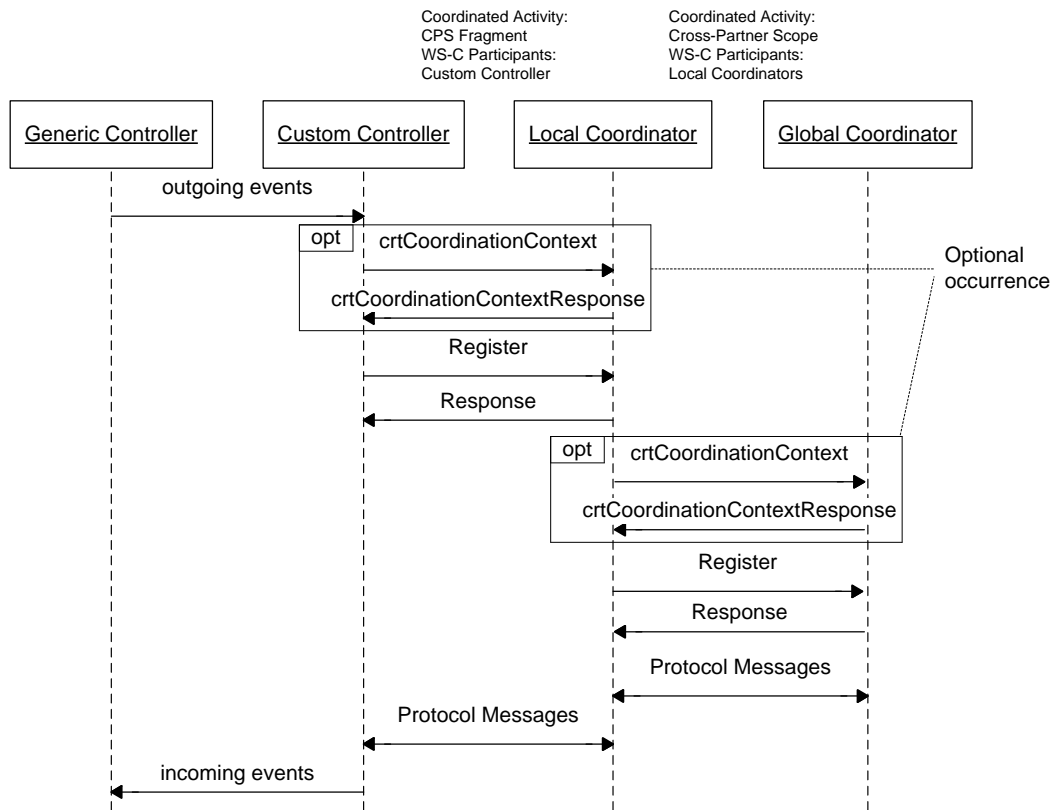


Figure 7.9: Global view on the interaction between coordinating entities

Since runtime support somehow refers to lower level semantic, the following sections use BPEL terminology of *faults* for compliancy reasons.

7.4.2 Generation of the Custom Controller

According to [Ste08], the custom controller needs the following five information for an external control of CPSs:

1. the set of child activities CA_{S_i} for a given CPS S_i
2. the set of enclosing scope-activities EA_{S_i} for a given CPS S_i
3. the set of direct child scopes DCS_{S_i} for a given CPS S_i
4. the set of scope-activities needed to complete the CPS successfully SA_{S_i} for a given CPS S_i
5. the set of loops needed to complete the CPS successfully SL_{S_i} for a given CPS S_i

These information can be derived automatically out of the logical model. For this purpose, the following XML-tree structures provide the information: (i) the CPS description of the BPEL4Chor topology and (ii) the participant behavior descriptions involved in a CPS.

(1) *Child Activities* (CA) can be derived from the CPS description in the BPEL4Chor topology. For instance, Listing 7.3(1.3) shows the fragment definitions of the particular processes. Thus, the set of child activities CA can be derived as *union* of all process activities $a_j \in A$ of a particular CPS S_i . It is important to notice, that a simple concatenation does not suffice, since CPSs as defined in this thesis may overlap.

$$CA_{S_i} = \bigcup_j a_j \in S_i \quad , \text{ the set of child activities for a given CPS } S_i$$

(2) *Enclosing Scope Activities* (EA) can be determined from the *Participant Behavior Description* (PBD) in combination with CA_{S_i} . This information can be achieved through a bottom-up search in the XML-tree of the PBD of the corresponding participants of S_i . Notice, that only activities out of CA must be considered, since EA must contain at least one activity of CA to be enclosing.

(3) *Direct Child Scopes* (DCS) are scopes lying directly within a child activity of a CPS. Thus, they can be achieved through a search in the PBDs containing child activities CA .

(4,5) refer to scope-activities and loops needed to change the state from *CanComplete* to *Completed*. They can be achieved within (2) using a little modification (i. e. the bottom-up search looks not only for the enclosing scope-activity, but also for scope- and loop activities).

Thus, all information needed for the custom controller of [Ste08] can be generated automatically from the logical model.

7.4.3 The Local Coordination Protocol

Having discussed the derivation of the custom controller, the sequence of the CPS-activities can be monitored externally. To validate the correct sequence of these activities, an important aspect is the mapping of events inside the process life-cycle to appropriate coordination protocol messages. Thus, the coordinator can control the participants using the custom controller.

Section 7.4.1 on page 92 discusses different proposals of modeling the collaboration between BPEL engine and coordinators. Thereby, the usage of split coordinators is proposed referring to the need of a local as well a global coordination protocol. In addition, the notion of coordinating fragments instead of single activities is also discussed. Hence, monitoring activities is the responsibility of the custom controller which informs corresponding local coordinators about relevant events. The local coordination protocol given in Figure 7.10 is developed out of the simplified state-chart of CPSs in Figure 7.7 on page 91. To show the relation to Figure 7.7 on page 91, the corresponding states of Figure 7.7 are marked gray. [Kop+08a; Kop+08b] formalize this kind of protocol specification into *Coordination Protocol Graphs* (CPG) and present a model-driven architecture approach to simplify and accelerate the implementation of coordination protocols. Thereby, CPGs are deterministic acyclic state-based graphs with labeled edges and labeled nodes. Nodes denote states of the coordination protocol. Their labels describe the particular semantic. Edges depict protocol-specific messages by both the *coordinator* (solid line) and its *participants* (dashed line). „Each CPG has exactly one node with

no incoming edges (source) and at least one node without outgoing edges (sink)“ [Kop+08a, p. 4].

The starting point of every local coordination is the initialization of a particular process. This issue arises by the need of monitoring events such as *instance terminated* which may appear before particular events of the CPS are executed. Thus, a CPS may transition to the *unableToComplete* state before a particular activity is started. In the *canComplete* state, the custom controller informs the coordinator of *faulting* and *completing* fragments, leading either to the *unableToComplete* state or *completed* state. In addition, the local coordinator may send a *termination* or *blocking* request. In case of termination, the custom controller tries to terminate the corresponding activities, which can either successful result in a *fragment terminated* message, or unsuccessful result in a *fragment faulted* message. In case of blocking, the custom controller stops releasing blockades immediately. If all activities are blocked, the local coordinator is informed. In the *unableToComplete* state, the corresponding exception handling is achieved by the global coordinator. This either leads to successful exception handling (*resuming* the fragment), *compensation* or *faulting* the fragment, as well as *terminating* the fragment. Out of the *completed* state, the custom controller may try to compensate the scope due to a corresponding incoming event. Thus, the local coordinator is informed. Additionally, the local coordinator may decide to compensate the fragment itself, or close the fragment. Appendix A gives a detailed overview of the overall collaboration between custom controller and local coordinator.

From an abstract point of view the coordinator is a business process receiving a sequence of messages from the custom controller, according to the discussion in [Mie06]. On receipt of a message the coordinator changes into a new state depending on the message type. Thus, the control flow of the coordinator depends on the messages it receives, similar to a *finite state machine* [Sch08b] (assuming the sequence of incoming messages as input, while all other activities are ignored). Notice that it is even a *Deterministic Finite Automaton* (DFA) [Sch08b], referring to the fact that „a business process instance MUST NOT simultaneously enable two or more <receive>-activities for the same *partnerLink*, *portType*, *operation* and *correlationSet*“ [BPE07, p. 91].

[Kop+08a; Kop+08b] demand CPGs to be acyclic due to the missing support of unstructured loops by BPEL and announces the support of cyclic CPGs in future work. The generation of the local coordination protocol can be achieved using the concept of [Kop+08a; Kop+08b]. However, the next section involves a cycle in the coordination protocol. As a consequence, the concept of [Kop+08a; Kop+08b] cannot be used (yet). Thus, a way of mapping *regular expressions* [Sch08b] to BPEL structures is explained next as presented in [Mie06]. Since regular expressions are equivalent to DFAs [Sch08b], this mapping can be used to provide a generation of the global coordinators out of the coordination protocol.

Listing 7.7 Simplified BPEL code for regular expressions according to [Mie06]

```
1 #Concatenation
2 <sequence>
3   <receive message="m1"/>
4   <receive message="m2"/>
5 </sequence>
6
7 #Alternation
8 <pick>
9   <onMessage message="m1"/>
10  <onMessage message="m2"/>
11 </pick>
12
13 #Kleene star
14 <sequence>
15   <receive message="m1"/>
16   <while loopCondition="true">
17     <pick>
18       <onMessage message="m2"/>
19       <onMessage message="m3">
20         <assign false to loopCondition/>
21       </onMessage>
22     </pick>
23   </while>
24 </sequence>
```

coordination context is provided through CPSs. The *coordination type* depends on the CPS's policy. For instance, this may be the all-or-nothing policy. Finally, the *coordination protocol* is introduced in this section.

The local coordination protocol of the previous section allows to monitor particular fragments externally in an efficient way. Since the local coordinators have no knowledge of the global state of CPSs, they need to be controlled by a global coordinator. It is important to note, that a global coordinator only focuses on the outcome of one particular CPS. [Mie06; PML07] propose the use of the WS-Business Activity *business agreement with participant completion* (WS-BA BAP) protocol. In addition, [Mie06] mentions that BAP „is not sufficient to map all events that occur in the life-cycle of a BPEL scope“ [Mie06, p. 36]. As a consequence, an extended WS-BA BAP is proposed, introducing new states and messages. Due to different behavior of CPSs in interaction choreographies the extended BAP is adapted to the requirements of CPSs as shown in Figure 7.11.

The starting point of a CPS coordination (as presented in Figure 7.11) is given by its initialization due to initialization of local coordinators. In the *active* state (*canComplete* state of Figure 7.7 on page 91) the CPS coordinator is interested in *completion* and *faulting* of participants (DCS compensating, child activity terminated or faulted of Figure 7.7). In case of a *completed* message from a participant, the coordinator changes into the *completing* state and then *acknowledges* the local coordinator. This behavior is due to the avoidance of signal race as identified by [Mie06]. In *completed* state a scope can be either *closed* if no compensation is needed or *compensated* by its parent scope. Since compensation may also fail, participants

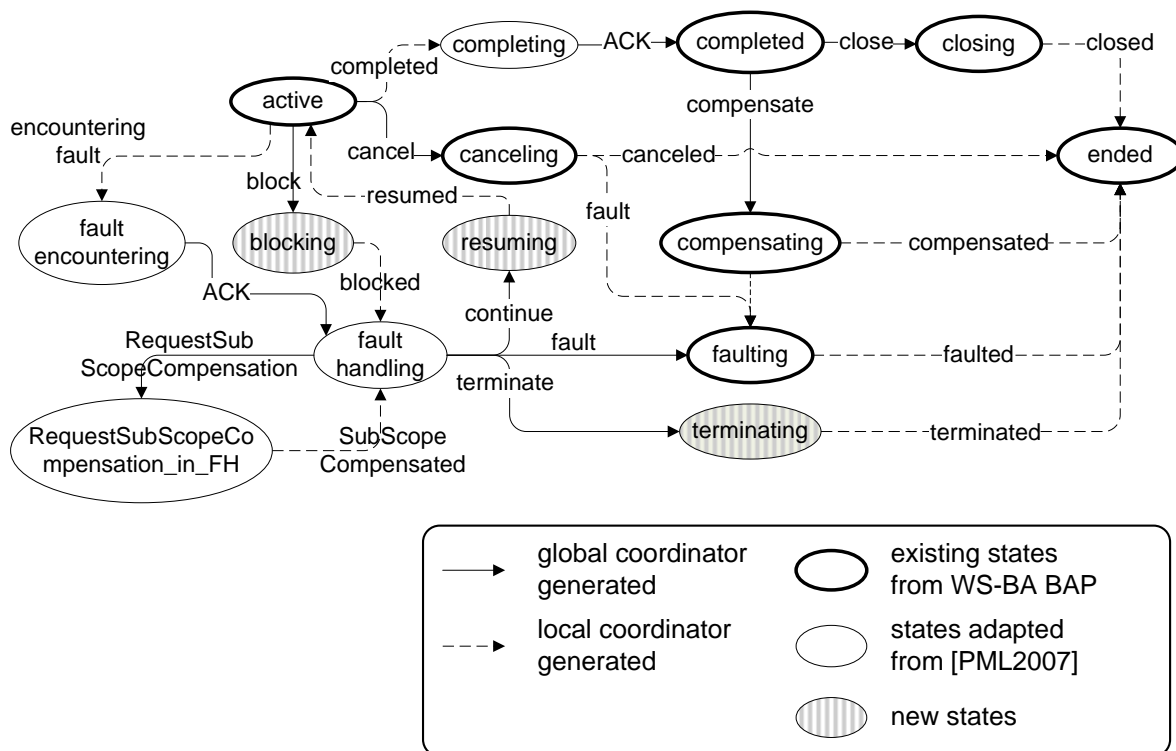


Figure 7.11: Cyclic coordination protocol graph of global coordinator

reply from *compensating* state either with successful *compensated* or *fault*. Since the coordinator may decide to compensate a particular fragment *request sub scope compensation*, „specifies which participants are to be compensated“ [Mie06, p. 37]. Similar to the completion handling participants may indicate faults introducing the need for compensating participants. Thus, the states *fault encountering* and *fault handling* with its corresponding *sub scope compensation* are introduced by [Mie06]. Since fault handlers require cancellation of all nested active subsopes, a coordinator may send a *cancel* message to a participant. Thereby, cancellation may fail due to some reasons. Thus, *fault* and *canceled* messages can be triggered by the local coordinator. Out of the *fault handling* state three alternatives can be taken. If the fault handling states *resumption* of the respective fragment, then *continue* message is triggered. If the fault handling states *faulting* of the respective fragment, then *fault* message is triggered. Finally, fault handling may state *terminate* the respective fragment. The overall collaboration between local coordinator and global coordinator is given in Appendix A.

7.5 Realization of Simultaneous Raised Exceptions

The notion of simultaneous raised exceptions as discussed in Section 5.3.5 on page 50 is supported due to the fact that participants may signal an exception as long as they are not blocked. Notice that a participant is blocked by the custom controller in case of an exception.

Furthermore, the custom controller informs the corresponding local coordinator, which itself informs the corresponding global coordinator. On receipt of a *fragment faulted* message, the global coordinator informs all corresponding fragments by sending a *block* message. The other fragments can raise exceptions until they have not received the *block* message, i. e. until they are in the *encountering fault phase* (cf. Section A.3 on page 133). Thus, in the state *fault handling* multiple exceptions can be signaled and the resulting exception can be determined according to the exception graph (cf. Section 5.3.5 on page 51).

7.6 Realization of Exception Propagation

Section 5.3.6 on page 52 discusses the notion of exception propagation dealing with dominance. The exception propagation hierarchy can be determined statically and placed in the topology description as in Section 7.2.3 on page 89. This information can be used at runtime (i. e. in the case of fault handler faulted) to propagate the exception to its corresponding participant. For this purpose, the corresponding participant can be determined using a *global correlation set* as introduced by [KL06; Kha08].

7.7 Realization of Scope Hierarchies

Section 5.4.4 on page 60 mentions the notion of nested and overlapping scopes. Thereby, a main issue is to use a separate coordinator for the overlapping area and thus introducing the notion of scope hierarchies. Thereby, a discussion is needed how scope hierarchies (i. e. their induced overlapping graphs) can be simulated by WS-C. To support a hierarchy of coordination Section 2.3 on page 20 mentions the concept of *interposition* where participants are enabled to act as coordinators itself. Thereby, the subcoordinator is responsible for the outcome of its coordinated activities, while the top-level coordinator communicates with the subcoordinator as a participant.

[Mie06] identifies „the need for a mechanism that correlates participants with activities“ [Mie06, p. 31]. For this purpose a mapping of scopes to WS-C is needed. This mapping is adapted to CPSs and the overlapping requirement in Figure 7.12. Thereby, each CPS becomes a WS-C activity, where the corresponding fragments register as participants (i. e. for *cps1*: the fragment including *a1* and *a2* and the fragment representing the overlapping). Since overlapping areas are treated as separate CPSs, they are handled as participating subcoordinators via interposition. Furthermore, Figure 7.12 shows that all CPSs are represented as participants too.

7.8 Summary

This chapter discusses the runtime support of interaction exceptions. The main issues of the investigation are the derivation of interconnection models from interaction models in

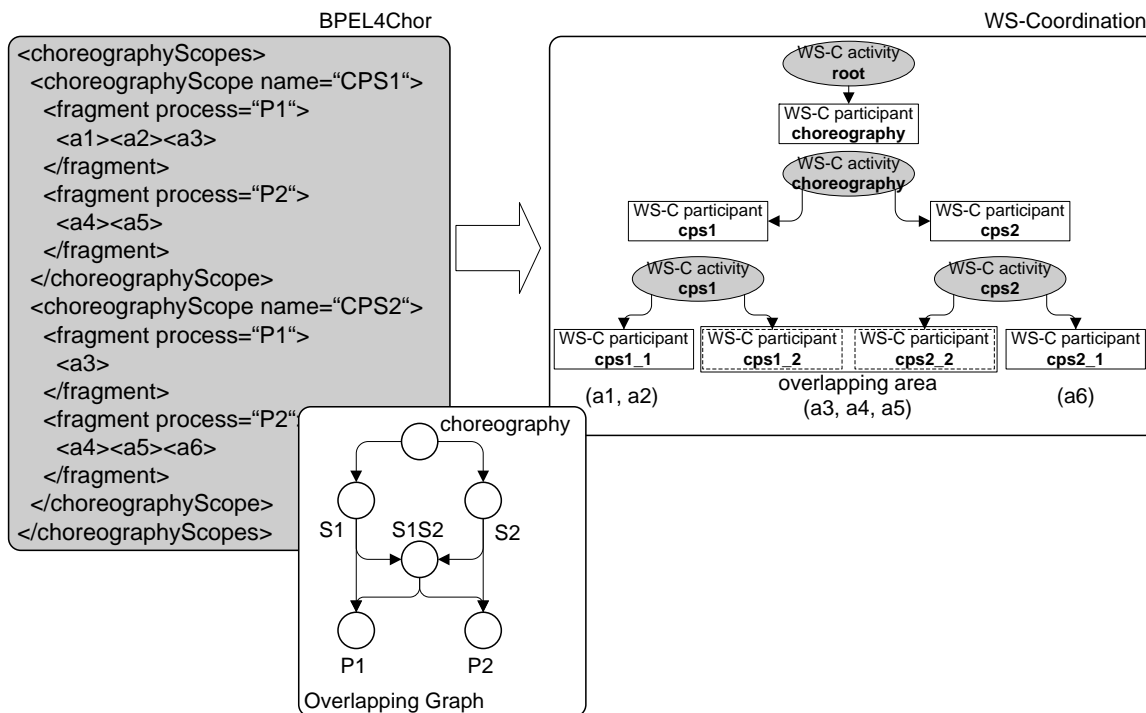


Figure 7.12: Mapping Cross-Partner Scopes to WS-Coordination

Section 7.1 on page 81. The proposed mapping of [Dec09a] is extended by a transformation of the additional iBPMN artifacts to BPMN. Thereby, it is possible to encounter situations in the rearrangement phase where no valid BPMN model can be provided. Thus, the model has to be rejected in such cases. Section 7.2 on page 85 mentions the representation of the conceptual interaction model in the logical interconnection model, especially focusing on the BPEL4Chor representation. Thereby, an extension of the BPEL4Chor topology is proposed which allows the representation of cross-partner scopes, their overlapping and exception handlers. In addition, an explicit representation of exception graphs and dominance trees in BPEL4Chor is proposed to support exception resolution and exception propagation at runtime. Sections 7.3 and Section 7.4 on page 92 discuss possible implementation alternatives of the coordination. Thereby, the BPEL engine as well as the particular external coordinators are regarded. The pluggable framework is used to gain external control of processes without the need to change their particular implementation. To gain simple coordinators without a huge communication overhead, a generic coordinator hierarchy is proposed. For this purpose, the custom controller forms the application-dependent part of the coordination, which is responsible for the activity coordination. The local controller forms a completely independent part in this hierarchy, while the global coordinator is dependent on the particular policy. Thereby, the main focus is a marginal process execution delay in the normal case, while exceptional situations are allowed to delay the process execution. Furthermore, the generation of the particular components is described. Finally, Sections 7.5 to 7.7 on pages 101–102 concludes the investigation

dealing with the realization of simultaneous raised exceptions, exception propagation and CPS-hierarchies. The next chapter discusses aspects of the prototype realization.

8 Prototype Realization

During this thesis Chapter 5 develops a theoretical foundation of exceptions and their handling in interaction choreography models. Chapters 6 and 7 discuss how the theoretical concept can be integrated in (i) modeling and (ii) runtime. To complete the idea of the approach, this chapter provides aspects of the prototypical implementation. This thesis provides a graphical extension to model the approached concept. All generations and transformations are not provided within this thesis, since they related to the interconnection models and not to interaction models.

Not the whole architecture discussed in this thesis should be implemented from the scratch. For this reason, several tools are discussed in Section 8.1. For instance, this is at least (i) an integration into a graphical editor for modeling interaction choreographies, (ii) an integration into a particular workflow engine and (iii) a framework for the externalization of the process behavior is necessary. Having identified a certain tool-chain, Section 8.2 provides an architectural overview of the whole concept. Thereby, a stencil set for modeling interaction exceptions using iBPMN is developed and integrated into the Oryx Editor. Finally, the concrete runtime support of the proposed concept is discussed.

8.1 Tools

For the realization of the approached concept two issues have to be tackled: (i) modeling and (ii) runtime support. For modeling purposes the *Oryx framework* (www.oryx-editor.org) is used. In cases of the runtime support, *Apache Orchestration Director Engine* (ODE) (ode.apache.org) is used as runtime engine. In addition, the *pluggable framework* is used for the externalization of the process behavior. The pluggable framework is discussed in Section 7.3 on page 90, while Oryx and Apache ODE are introduced in this section.

8.1.1 The Oryx Editor

Oryx is an extensible graphical environment for *conceptual modeling* [CAKT99] which was developed at the *Hasso-Plattner-Institute* (www.hpi.uni-potsdam.de) [DOW08]. For this purpose, creating, sharing and documenting conceptual models is provided using a browser-based application which does not require any software installation. The main requirements of Oryx are (i) support of different modeling languages, (ii) meta-information and feature extension and (iii) data portability. The *support of different modeling languages* is achieved via *stencil sets* [Pet07] which provide explicit typing, connection rules and visual appearance. *Meta-information*

enlarge the model elements and *feature extensions* such as *model checking* allow to improve the models. Finally, *data portability* is achieved by using *Universal Resource Identifier* references (URI) and *Resource Description Framework* export (RDF) [RDF98] via *XSL Transformations* (XSLT) [XSL99].

8.1.2 Apache Orchestration Director Engine (ODE)

Apache ODE is an open source BPEL-Engine for the execution of BPEL processes which are conform to the BPEL2.0 specification. Currently, it is available at version 1.3.2. Furthermore, a beta release of version 2.0 is provided. In this thesis version 1.1.1 is used due to the use of the pluggable framework. Apache ODE already has an *event framework* which is used and extended in the pluggable framework. For the execution of processes Apache ODE uses corresponding *ODE-object-models*, which do not access original the BPEL process. The ODE-object-model departs from the BPEL2.0 specification [Apa08]. For instance, from- and toPart syntax is not supported for message exchange activities. Furthermore, version 1.1.1 does not support *termination handler* and *extension activities*.

8.2 Architecture Overview

During the previous chapters, this thesis investigates the use of exception handling in interaction choreography models. For this purpose additional modeling artifacts are introduced and the runtime support is described. The overall architecture to gain exception handling is shown in Figure 8.1 on the facing page and discussed in the following.

During the *choreography design process* (discussed in Section 4.2 on page 38) the need of modeling collaboration scenarios arises. For this purpose interaction models can be used in a conceptual layer and refined by exception handling in the technical model (cf. Section 5.6 on page 68). Out of the technical interaction model, a logical representation can be derived using BPEL4Chor as discussed in Section 7.2 on page 85. Thereby, the *participant declarations*, *message links* and *cross-partner scopes* appear in the *participant topology* (cf. 1,2,3 in Figure 8.1). Furthermore, the *observable behavior descriptions* (PBD) can be derived as discussed in Section 7.1 on page 81 (cf. 4 in Figure 8.1) and has to be *grounded*. Finally, the particular *coordination protocols* and *custom controllers* have to be generated as discussed in Section 7.4 on page 92 (cf. 6,7 in Figure 8.1).

8.3 Modeling Interaction Exceptions with Oryx

The implementation of the modeling part of this thesis follows the discussed requirements of Chapter 6. Thus, *intermediate events*, *cross-partner scopes* and additional artifacts are introduced to iBPMN. For the introduction of these new artifacts, a *stencil set extension* [Pet07] is provided for the existing iBPMN stencil set. This stencil set extension is called *iBPMN Fault*

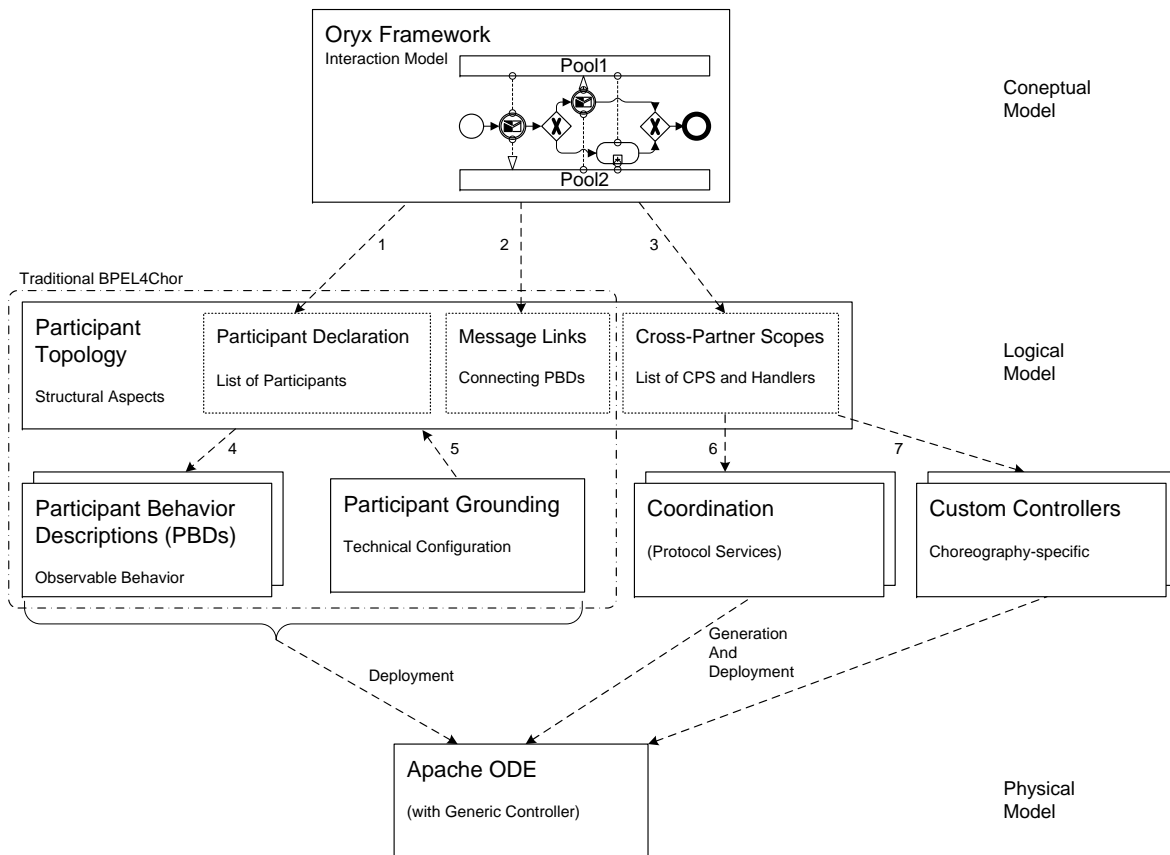


Figure 8.1: Architectural overview on the global interplay

Handling Extension (iBPMN-FH) and resides under <http://oryx-editor.org/stencilsets/extensions/ibpmn-FH#> namespace.

The original iBPMN stencil set consists of three groups: (i) pools and interactions, (ii) control flow and (iii) events. The iBPMN-FH extension consists of five groups: (i) pools and interactions, (ii) control flow, (iii) startevents, (iv) intermediateevents and (v) endevents. To structure the 28 intermediate events which are presented in Table 6.1, they are partitioned into three separate groups according to their usage in modeling (i. e. modeling start, intermediate or end events).

The main changes provided with the iBPMN-FH extension are discussed in the following. In the *Pools and interactions* group the start interaction event is moved to the *startevents* group and the intermediate interaction is moved to the *intermediate evens* group. Furthermore, three activities are added to the group: silent/no action activity, scope activity and handler activity. The *control flow* group remains as is. The *startevents* group contains the start event, start timer event and the start interaction event. Additionally, the non-interrupting counterparts of these events are added. Furthermore, the start error event, start compensation event and start termination event are added. The *intermediateevents* forms the biggest group of the iBPMN-FH

extension. Thereby, interaction and timer events appears in its interrupting as well as its non-interrupting variant. Furthermore, error, compensation and termination events appears in its catching and throwing variant. Finally, the *endevents* group consists of end events, end interactions, end timer events, end error events, end compensation events and end termination events. Furthermore, non-interrupting end events, end interactions and end timer events are added.

Additional to the graphical representation, Chapter 6.2 discusses situations where particular properties are needed to provide additional information. These situations are: (a) error events, (b) Cross-Partner Scopes (CPSs) and (c) silent and no actions. (a) For error events Section 6.2 on page 73 discusses the need for both an *ignore property* and a *record property*. These properties are summarized under the *trivial functionality property* which can be either *none*, *ignore*, *record* or *ignore+record*. Thereby, the default value is set to *none*. Notice that error events can be modeled using both catching and throwing error events. Since it makes no sense to specify an throwing error event which has to be ignored, only catching error events are extended with this properties. (b) For CPSs Section 6.3 on page 75 discusses the need for a *policy property* to support an extensible set of policies. Thus, a *policy property* is introduced to scopes. According to the discussed requirements the value of this property can be either *all-or-nothing*, *alternative atomicity*, *exception atomicity*, *business transaction* or *autonomous decision*. Thereby, the default value is set to *all-or-nothing* which is the discussed policy of this thesis. Finally, (c) Section 6.4 on page 76 discusses the need for a *property* to distinguish between *silent actions* and *no actions*. Hence, the *type property* is introduced to activities, which can be either *silent action* or *no action*. Thereby, the default value is set to *silent action*

8.4 Runtime Support of Interaction Exceptions

The runtime support marks the final step in the *choreography management stack* (cf. Chapter 1), referring to the generation of the *physical model* out of the *logical model*. For this purpose, the following issues are identified:

- deployment of grounded executable processes
- generation and deployment of local and global coordinators
- generation and deployment of custom controllers

The *deployment of executable processes* is out of the focus of this thesis, since it is specific to the particular process engine. The *generation of local and global coordinators* as described in Section 7.4 on page 92 can be achieved using both (i) the generation out of *Coordination Protocol Graphs* (CPG) and (ii) *regular expressions*. (i) The generation of the local coordinators can be achieved by following the approach of [Kop+08a; Kop+08b]. (i) does (currently) not support cyclic CPGs. Thus, (ii) has to be used for the generation of the global coordinator using the proposed mapping to a BPEL representation of [Mie06]. It is important to note, that the local and global coordinators have to be generated *once only* (i. e. the global coordinator once for every policy), while the custom controllers have to be generated for *every* choreography.

This can be understood referring to the discussed generic hierarchy of coordinators of Section 7.4.1. Thereby, the global coordinator is policy-specific, the local coordinator is completely independent and the custom controllers are application-specific since they need to know the activities to register for. Since this thesis focuses on conceptual aspects of modeling and runtime support, the generation of these coordinators out of BPEL4Chor-descriptions is not part of the implementation. If these generations are implemented in future work, the deployment would be similar to those of executable processes. Thereby, the *local coordinators* should reside on the same BPEL engine as its corresponding processes, while the *global coordinators* can be deployed to any BPEL engine. In addition, the *generation of custom controllers* (Section 7.4.3) out of BPEL4Chor-descriptions is also beyond the scope of this thesis. The corresponding deployment of custom controllers is described in [Ste08]

9 Summary and Outlook

For process orchestrations a *process management stack* can be defined introducing a three-layer architecture to business process management . The three-layer architecture decreases the complexity of modeling to different abstraction levels, namely *conceptual model*, *logical model* and *physical model*. For process choreographies this idea can be adapted to a *choreography management stack*. In contrast to process orchestrations, exception management in process choreographies is an open problem marginal targeted in literature. Thus, this thesis investigates *modeling and runtime-support of faults in interaction choreography models* referring to exception management on choreography level. Thereby, fault tolerance is ensured in the presence of distributed concurrency and cross-partner interaction and their faults.

Present interaction-centric languages support exception handling in a rudimentary and deficient way or even ignore this requirements at all. As a consequence a model to specify exceptional properties on a higher level is required which can automate coordinated exception handling out of the conceptual specification. The approach taken in this thesis specifies exceptional behavior of choreographies on interaction level and describes automated runtime support out of this enhancement. Thereby, the range of issues leading to exceptions and the various way they are handled is identified in a detailed discussion of exception classifications in Sections 3.2 and exception handling in Section 3.3 on page 28. Especially regarding modeling issues, it turns out that only *expected exceptions* can be modeled, while unexpected exceptions need superior solutions. A multi-viewpoint approach dealing with different levels of abstraction is provided with the *choreography design process*. However, *interaction-centric* exception classification, handling and modeling on choreography level is marginal targeted. Within this thesis, three interaction notations are identified where interaction exceptions can be applied to: WS-CDL, BPMN2.0 and iBPMN. Out of these notations *iBPMN* turned out as preferred notation for an extension with interaction exceptions.

During the investigation five requirements of exceptions on an interaction choreography level are identified. For this purpose, Chapter 5 discusses *silent actions*, *no actions*, interrupting and non-interrupting intermediate events, simultaneous raised exceptions, exception propagation in choreographies and cross-partner scopes. *Cross-Partner Scopes* (CPSs) form a general scoping concept where arbitrary activities of different processes are grouped together. Policies state the completion behavior of CPSs. Since CPSs provide a choreography-wide exception handling context, they are a main point of interest in this thesis. Cross-process artifacts are out of the scope of the BPEL specification and hence standard BPEL engines do not support them. Thus, CPSs have to be simulated using additional mechanisms. A previous approach simulates CPSs through BPEL constructs. In contrast, this thesis investigates the usage of CPSs without the need to change a particular process implementation. Due to the atomic nature of interaction

models, overlapping becomes an interesting requirement of CPSs and thus is investigated in detail in Section 5.4 on page 56.

Having discussed the requirements of interaction exceptions, additional modeling artifacts can be derived. For this purpose, Chapter 6 investigates the integration of the proposed artifacts into iBPMN on the basis of the theoretical investigation of Chapter 5. Finally, these integration considerations are evaluated against the presented exception handling strategies.

Chapter 7 investigates runtime support of the concept. Thereby, the main point of interest is the derivation of interconnection models out of interaction models. As a result of the introduction of new modeling artifacts, the derivation of these models is extended accordingly. Since, CPSs are not part of the particular process implementations, BPEL4Chor is extended to represent the new artifact. Thereby, a mapping of the conceptual iBPMN model to the logical BPEL4Chor model is provided in Section 7.2 on page 85. Finally, a detailed investigation describes the resulting runtime support on the physical level. Section 7.3 on page 90 presents the runtime support of interaction exceptions in the BPEL engine. Since participants are independently involved in a choreography, they are not aware of exceptions raised in other participants. Thus, coordinators have to monitor the process execution to take over control in case of exceptions. Thereby, main issues of the coordination are:

- (i) a little delay of the process execution in non-exceptional situations and
- (ii) simple coordinators

(i) refers to the communication overhead of the additional coordination. (ii) refers to the gap between application-specific and -unspecific parts of a particular coordinator. The *WS-Coordination framework* is the de facto standard for coordinating web services. Thus, this thesis proposes additional coordination protocols for CPSs by extending *WS-Business Activity with Coordinator Completion*.

Concluding, this thesis provides a complete description of the modeling and runtime-support of interaction exceptions. For the modeling part, this thesis provides a *stencil set extension* for the *Oryx Framework* (www.oryx-editor.org) called *iBPMN Fault Handling Extension* (iBPMN-FH). Thereby, the discussed modeling artifacts are implemented in a graphical editor, which can serve as starting point for a generation of the discussed logical representation in BPEL4Chor. Out of the logical BPEL4Chor model the proposed coordinators can be generated as discussed in Section 7.4 on page 92. The runtime support of interaction choreography models ends at the logical interface (i. e. the interconnection model BPEL4Chor). Thus, the corresponding generations from (i) iBPMN to BPEL4Chor and (ii) from BPEL4Chor to BPEL are not provided within this thesis. For (ii) a generation already exists [RKDL08].

Findings

The proposed concept enables choreography designers to gain a powerful possibility to model complex scenarios with exceptions on interaction modeling level. Thereby, the choreography designer as well as the process designer are freed from particular coordination purposes.

Additionally, modelers are enabled to use exceptions which are not required to interrupt the whole choreography. Thus, the introduced *cross-partner scopes* are not restricted to exceptional situations. Furthermore, *quality of services* can be monitored. Especially, the two layers (i) functional layer and (ii) detailed layer discussed in Section 5.6 on page 68 fits perfectly in the choreography design process. This can be understood, since the existing choreography design process is refined by one step after deriving the collaboration scenario. This additional step is the detailed layer and introduces the notion of exceptions on an interaction level. Thus, choreography designers are enabled to model exceptions before they suffer the loss of the global perspective.

Future Work

During this thesis exception handling in interaction choreographies is investigated. Thereby, compensation and termination handling cannot be discussed in the necessary detail. Thus, future work may investigate compensation and termination handling in interaction choreographies. For *compensation handling*, [Wan09] is a good starting point. It is important to notice, that by compensating intermediate events the issue of *event revocation* arises. For this purpose, a good starting point is provided with [HA98; HA99]. For *termination handling*, [Lud99; LSBG99] is a good starting point.

This thesis focuses on exception modeling and runtime support. Thereby, exception handling strategies are discussed in Section 3.3 on page 28. Section 6.6 on page 78 evaluates the introduced modeling artifacts against these exception handling strategies. Thereby, the *detached execution mode* of exception handlers using exception handling processes is not treated. Thus, future work may investigate this issue with a special focus on the usage of *workflow exception patterns* [RH06a; RH06b].

Every concept and graphical notation depends on its user acceptance. Thus, future work may provide an evaluation of the acceptance of both interaction exceptions and cross-partner scopes. For this purpose, the provided stencil set extension is a good starting point. Furthermore, future work may answer the discussed open questions on the need of both *silent actions* and *no actions*.

For further improvement of usability and acceptance of the approached concept, several translations and generations have to be provided by future work. For instance, there is no implemented translation from the conceptual iBPMN model to the logical BPEL4Chor model. However, the necessary steps of such a translation are described in Section 7.1 on page 81 and [Dec09a]. Furthermore, the automated generation of custom controllers out of the BPEL4Chor description and the implementation of the coordinators are essential issues of future work. These issues are described in detail in this thesis and has to be implemented in future work.

Bibliography

Exception Handling related Literature

- [AHE05] Michael Adams, Arthur H. M. ter Hofstede, and David Edmond. “Facilitating Flexibility and Dynamic Exception Handling in Workflows”. In: *Proceedings of the CAiSE 05 Forum, FEUP*. 2005, pp. 45–50. Cited on page pp. 28, 29.
- [AL81] Tom Anderson and Peter A. Lee. *Fault Tolerance : Principles and Practice*. eng. EUA : Prentice-Hall, 1981. ISBN: 0-13-308254-7. Cited on page p. 33.
- [AL90] Tom Anderson and Peter A. Lee. *Fault Tolerance: Principles and Practice*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1990. ISBN: 0-38782-077-9. Cited on page pp. 24, 30.
- [AR79] Tom Anderson and Brian Randell. *Computing Systems Reliability*. New York: Cambridge University Press, 1979. ISBN: 0-521-22767-4. Cited on page pp. 23, 24.
- [Ard+06] Liliana Ardissono et al. “Fault Tolerant Web Service Orchestration by Means of Diagnosis”. In: *EWSA*. Vol. 4344. Lecture Notes in Computer Science. Springer, 2006, pp. 2–16. ISBN: 3-540-69271-1. DOI: 10.1007/11966104_2. Cited on page p. 25.
- [BM99] Alex Borgida and Takahiro Murata. “Tolerating Exceptions in Workflows: A Unified Framework for Data and Processes”. In: *WACC '99: Proceedings of the international joint conference on Work activities coordination and collaboration*. New York, NY, USA: ACM, 1999, pp. 59–68. ISBN: 1-58113-070-8. DOI: 10.1145/295665.295673. Cited on page p. 29.
- [BBKZ93] Holger Branding, Alejandro P. Buchmann, Thomas Kudrass, and Jürgen Zimmermann. “Rules in an Open System: The REACH Rule System”. In: *Rules in Database Systems*. 1993, pp. 111–126. Cited on page p. 32.
- [BW01] Alan Burns and Andrew J. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN: 0-20172-988-1. Cited on page pp. 24–26.
- [CR86] Roy H. Campbell and Brian Randell. “Error Recovery in Asynchronous Systems”. In: *IEEE Transactions on Software Engineering* SE-12.8 (Aug. 1986), pp. 811–826. Cited on page pp. 33, 51.

- [CCPP99] Fabio Casati, Stefano Ceri, Stefano Paraboschi, and Guiseppe Pozzi. “Specification and Implementation of Exceptions in Workflow Management Systems”. In: *ACM Trans. Database Syst.* 24.3 (1999), pp. 405–451. ISSN: 0362-5915. DOI: 10.1145/328939.328996. Cited on page pp. 24, 28, 31, 32.
- [CP99] Fabio Casati and Giuseppe Pozzi. “Modeling Exceptional Behaviors in Commercial Workflow Management Systems”. In: *COOPIS '99: Proceedings of the Fourth IECIS International Conference on Cooperative Information Systems*. Washington, DC, USA: IEEE Computer Society, 1999, p. 127. ISBN: 0-7695-0384-5. DOI: 10.1109/COOPIS.1999.792164. Cited on page p. 25.
- [CCKM05] Girish Chafle, Sunil Chandra, Pankaj Kankar, and Vijay Mann. “Handling Faults in Decentralized Orchestration of Composite Web Services.” In: *ICSOC*. Vol. 3826. Lecture Notes in Computer Science. Springer, 2005, pp. 410–423. ISBN: 3-540-30817-2. DOI: 10.1007/11596141_31. Cited on page p. 33.
- [Cha97] Sharma Chakravarthy. “SENTINEL: An Object-Oriented DBMS With Event-Based Rules”. In: *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. 1997, pp. 572–575. Cited on page p. 32.
- [CD97] Qiming Chen and Umeshwar Dayal. “Failure Handling for Transaction Hierarchies”. In: *ICDE '97: Proceedings of the Thirteenth International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 245–254. ISBN: 0-8186-7807-0. Cited on page p. 27.
- [CLK99] Dickson K. W. Chiu, Qing Li, and Kamalakar Karlapalem. “A Meta Modelng Approach to Workflow Management Systems Supporting Exception Handling”. In: *Inf. Syst.* 24.2 (1999), pp. 159–184. ISSN: 0306-4379. DOI: 10.1016/S0306-4379(99)00010-1. Cited on page pp. 27, 28, 30, 32.
- [CLK00] Dickson K. W. Chiu, Qing Li, and Kamalakar Karlapalem. “ADOME-WFMS: Towards Cooperative Handling of Workflow Exceptions”. In: *Advances in Exception Handling Techniques*. Vol. 2022. Lecture Notes in Computer Science. Springer, 2000, pp. 271–288. ISBN: 3-540-41952-7. Cited on page p. 28.
- [CCP07] Karelitis Christos, Vassilakis Costas, and Georgiadis Panayiotis. “Enhancing BPEL Scenarios with Dynamic Relevance-Based Exception Handling”. In: *Web Services, IEEE International Conference on 0* (2007), pp. 751–758. DOI: 10.1109/ICWS.2007.86. Cited on page p. 25.
- [CKLW03] Francisco Curbera, Rania Khalaf, Frank Leymann, and Sanjiva Weerawarana. “Exception Handling in the BPEL4WS Language”. In: *International Conference on Business Process Management*. Vol. 2678. LNCS. 2003, pp. 276–290. Cited on page pp. 18, 58, 75.
- [EL95] Johann Eder and Walter Liebhart. “The Workflow Activity Model WAMO”. In: *Proceedings of the 3rd international conference on Cooperative Information Systems (CoopIs)*. 1995, pp. 87–98. Cited on page pp. 25–27, 31, 45, 46.

-
- [EL96] Johann Eder and Walter Liebhart. “Workflow Recovery”. In: *COOPIS '96: Proceedings of the First IFCS International Conference on Cooperative Information Systems*. Washington, DC, USA: IEEE Computer Society, 1996, p. 124. ISBN: 0-8186-7505-5. Cited on page p. 26.
- [EL98] Johann Eder and Walter Liebhart. “Contributions to Exception Handling in Workflow Management”. In: *EDBT Workshop on Workflow Management Systems*. Valencia, Spain 1998, pp. 3–10. Cited on page p. 31.
- [Elm92] Ahmed K. Elmagarmid. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992. ISBN: 1-55860-214-3. Cited on page p. 32.
- [GFV96] Stella Gatzju, Hans Fritschi, and Anca Vaduva. *SAMOS an Active Object-Oriented Database System: Manual*. ifi-96.02. Tue, 21 May 1996 09:34:23 GMT. Department of Computer Science, University of Zurich, 1996. Cited on page p. 32.
- [GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. ISBN: 1-55860-190-2. Cited on page pp. 26, 33, 57.
- [GFJK03] Paul Greenfield, Alan Fekete, Julyan Jang, and Dean Kuo. “Compensation is Not Enough”. In: *EDOC '03: Proceedings of the 7th International Conference on Enterprise Distributed Object Computing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 232. ISBN: 0-7695-1994-6. Cited on page pp. 28, 31, 34, 49.
- [HA98] Claus Hagen and Gustavo Alonso. *Flexible Exception Handling in Process Support Systems*. Tech. rep. 1998. Cited on page pp. 31, 34, 49, 50, 113.
- [HA99] Claus Hagen and Gustavo Alonso. “Beyond the Black Box: Event-based Inter-Process Communication in Process Support Systems”. In: *In Proc. of the 19th Intl. Conference on Distributed Computing Systems*. 1999, pp. 450–457. Cited on page pp. 31, 50, 113.
- [HA00] Claus Hagen and Gustavo Alonso. “Exception Handling in Workflow Management Systems”. In: *IEEE Transactions on Software Engineering* 26.10 (2000), pp. 943–958. ISSN: 0098-5589. DOI: 10.1109/32.879818. Cited on page pp. 31, 46.
- [Ham05] Rachid Hamadi. “Formal Composition and Recovery Policies in Service-based Business Processes”. PhD thesis. The University of New South Wales, Sydney, Australia, 2005. Cited on page p. 13.
- [HB04] Rachid Hamadi and Boualem Benatallah. “Recovery Nets: Towards Self-Adaptive Workflow Systems”. In: *Proceedings of the 5th International Conference on Web Information Systems Engineering (WISE 04), LNCS 3306*. Springer Verlag, 2004, pp. 439–453. Cited on page pp. 13, 24, 26, 46, 72.
- [HBM08] Rachid Hamadi, Boualem Benatallah, and Brahim Medjahed. “Self-adapting Recovery Nets for Policy-driven Exception Handling in Business Processes”. In: *Distrib. Parallel Databases* 23.1 (2008), pp. 1–44. ISSN: 0926-8782. DOI: 10.1007/s10619-007-7020-1. Cited on page pp. 13, 24, 26, 28, 34, 46, 49, 69, 72, 73.

- [HH86] Herbert Hecht and Myron Hecht. *Fault-tolerant software*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986, pp. 658–696. ISBN: 0-13-308222-9. Cited on page p. 24.
- [Hei98] Petra Heidl. “Exceptions During Workflow Execution”. In: *Proceedings of the Sixth International Conference on Extending Database Technology*. 1998. Cited on page p. 25.
- [Hil98] Martin Hiller. *Software Fault Tolerance Techniques from a RealTime Systems Point of View: An Overview*. 1998. Cited on page p. 23.
- [HT04] San-Yih Hwang and Jian Tang. “Consulting Past Exceptions to Facilitate Workflow Exception Handling”. In: *Decis. Support Syst.* 37.1 (2004), pp. 49–69. ISSN: 0167-9236. DOI: 10.1016/S0167-9236(02)00194-X. Cited on page p. 28.
- [KR98a] Mohan Kamath and Krithi Ramamritham. “Failure Handling and Coordinated Execution of Concurrent Workflows”. In: *In Proc. of the 14 th Intl. Conf. on Data Engineering (ICDE’98)*. 1998, pp. 334–341. Cited on page pp. 27, 33.
- [KR98b] Mohan Kamath and Krithi Ramamritham. *Pragmatic Issues in Failure Handling and Coordinated Execution of Workflows in Distributed Workflow Control Architectures*. Tech. rep. Amherst, MA, USA 1998. Cited on page pp. 27, 33.
- [KR98c] Gerti Kappel and Werner Retschitzegger. “The TriGS Active Object-Oriented Database System: An Overview”. In: *SIGMOD Record (ACM Special Interest Group on Management of Data)* 27.3 (1998), pp. 36–42. ISSN: 0163-5808. Cited on page p. 32.
- [Kha08] Rania Khalaf. “Supporting Business Process Fragmentation While Maintaining Operational Semantics : A BPEL Perspective”. English. Dissertation. Universität Stuttgart Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2008, p. 193. Cited on page pp. 18, 91, 102.
- [KD00] Mark Klein and Chrysanthos Dellarocas. “A Knowledge-based Approach to Handling Exceptions in Workflow Systems”. In: *Comput. Supported Coop. Work* 9.3-4 (2000), pp. 399–412. ISSN: 0925-9724. Cited on page pp. 25, 28, 30, 75.
- [Kop08b] Oliver Kopp. *Cross-partner Fault Handling and Transactions*. unpublished. 2008. Cited on page pp. 13, 33, 56, 57.
- [Lam81] Butler W. Lampson. “Atomic Transactions”. In: *Distributed Systems - Architecture and Implementation, An Advanced Course*. London, UK: Springer-Verlag, 1981, pp. 246–265. ISBN: 3-540-10571-9. Cited on page p. 30.
- [Ley95] Frank Leymann. “Supporting Business Transactions Via Partial Backward Recovery In Workflow Management Systems”. In: *BTW*. 1995, pp. 51–70. Cited on page pp. 31, 33.
- [LR97] Frank Leymann and Dieter Roller. “Workflow-Based Applications”. In: *IBM Systems Journal* 36.1 (1997), pp. 102–123. Cited on page p. 31.

-
- [LHZP07] Jing Li, Jifeng He, Huibiao Zhu, and Geguang Pu. “Modeling and Verifying Web Services Choreography Using Process Algebra”. In: *SEW '07: Proceedings of the 31st IEEE Software Engineering Workshop*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 256–268. ISBN: 0-7695-2862-7. DOI: 10.1109/SEW.2007.105. Cited on page p. 35.
- [Lud99] Heiko Ludwig. “Termination Handling in Inter-Organisational Workflows - An Exception Management Approach”. In: 1999. DOI: 10.1109/EMPDP.1999.746655. Cited on page pp. 34, 50, 113.
- [LSKA03] Zongwei Luo, Amit Sheth, Krys Kochut, and Budak Arpinar. “Exception Handling for Conflict Resolution in Cross-Organizational Workflows”. In: *Distrib. Parallel Databases* 13.3 (2003), pp. 271–306. ISSN: 0926-8782. DOI: 10.1023/A:1022827610371. Cited on page pp. 27, 28, 30, 33.
- [LSKM00] Zongwei Luo, Amit Sheth, Krys Kochut, and John Miller. “Exception Handling in Workflow Systems”. In: *Applied Intelligence* 13.2 (2000), pp. 125–147. Cited on page p. 28.
- [MS05] Ashok U. Mallya and Munindar P. Singh. “Modeling Exceptions via Commitment Protocols”. In: *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*. New York, NY, USA: ACM, 2005, pp. 122–129. ISBN: 1-59593-093-0. DOI: 10.1145/1082473.1082492. Cited on page p. 25.
- [MRKS92] Sharad Mehrotra, Rajeev Rastogi, Henry F. Korth, and Avi Silberschatz. “A Transaction Model for Multidatabase Systems”. In: *Proc. 12th Int'l. Conf. on Distr. Computing Sys.* Yokohama, Japan 1992, p. 56. Cited on page p. 30.
- [MA04] Hernani Mourao and Pedro Antunes. “Exception Handling Through a Workflow”. In: *In CoopIS 04*. Springer Verlag, 2004, pp. 37–54. Cited on page p. 29.
- [MA05] Hernani Mourao and Pedro Antunes. “A Collaborative Framework for Unexpected Exception”. In: *Handling, Groupware: Design, Implementation, and Use*. Springer-Verlag, 2005, pp. 168–183. Cited on page p. 29.
- [MA07] Hernani Mourao and Pedro Antunes. “Supporting Effective Unexpected Exceptions Handling in Workflow Management Systems”. In: *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2007, pp. 1242–1249. ISBN: 1-59593-480-4. DOI: 10.1145/1244002.1244271. Cited on page p. 29.
- [Mou+03] Hernani Mourao et al. “Supporting Direct User Interventions in Exception Handling”. In: *in Workflow Management Systems. 9 th CRIWG*. 2003. Cited on page p. 29.
- [Pal07] Michael Paluszek. “Coordinating Distributed Loops and Fault Handling, Transactional Scopes using WS-Coordination protocols layered on WS-BPEL services”. englisch. Diplomarbeit. Universität Stuttgart Fakultät Informatik, Elektrotechnik und Informationstechnik Germany, 2007, p. 155. Cited on page pp. 33, 90.
- [Ran+95] Brian Randell et al. “From Recovery Blocks to Concurrent Atomic Actions”. In: *Predictable Dependable Computing Systems* (1995). Cited on page p. 25.

- [RDB03] Manfred Reichert, Peter Dadam, and Thomas Bauer. “Dealing with Forward and Backward Jumps in Workflow Management Systems”. In: *Inform., Forsch. Entwickl.* 18.3-4 (2003), pp. 132–151. Cited on page p. 27.
- [RS95] Andreas Reuter and Friedemann Schwenkreis. “ConTracts - A Low-Level Mechanism for Building General-Purpose Workflow Management-Systems”. In: *IEEE Data Engineering Bulletin* 18 (1995), pp. 4–10. Cited on page p. 31.
- [Ruf07] Fabian Ruf. “Fault Handling in BPEL-based Choreographies”. englisch. Diplomarbeit. Universität Stuttgart Fakultät Informatik Elektrotechnik und Informationstechnik Germany, 2007, p. 107. Cited on page pp. 6, 8, 13, 33, 51, 56, 58–60, 64, 65, 85, 87, 90.
- [RH06a] Nick Russell and Arthur H. M. ter Hofstede. *Exception Handling Patterns in Process-Aware Information Systems*. BPM Center Report. 2006. Cited on page pp. 28, 113.
- [RH06b] Nick Russell and Arthur H. M. ter Hofstede. “Workflow Exception Patterns”. In: *In: Proceedings of 18th CAiSE*. 2006, pp. 288–302. Cited on page pp. 28, 29, 113.
- [SW95] Heikki Saastamoinen and George M. White. “On Handling Exceptions”. In: *COCS '95: Proceedings of conference on Organizational computing systems*. New York, NY, USA: ACM, 1995, pp. 302–310. ISBN: 0-89791-706-5. DOI: 10.1145/224019.224051. Cited on page pp. 25, 28.
- [SO00] Shazia Sadiq and Maria E. Orlowska. “On Capturing Exceptions in Workflow Process Models”. In: 2000, pp. 3–19. Cited on page pp. 26, 27, 29, 46.
- [SR93] Amit Sheth and Marek Rusinkiewicz. “On Transactional Workflows”. In: *IEEE Data Eng. Bull.* 16.2 (June 1993), p. 37. Cited on page p. 33.
- [Sti+98] Remco Van Stiphout et al. “TRES: Workflow TRansactions by Means of EXceptions”. In: *WFMS'98 EDBT Workshop on Workflow Management Systems*. 1998. Cited on page pp. 26, 28, 31, 46.
- [SM95] Diane M. Strong and Steven M. Miller. “Exceptions and Exception Handling in Computerized Information Processes”. In: *ACM Trans. Inf. Syst.* 13.2 (1995), pp. 206–233. ISSN: 1046-8188. DOI: 10.1145/201040.201049. Cited on page p. 26.
- [TIRL03] Ferda Tartanoglu, Valeeie Issarny, Alexander B. Romanovsky, and Nicole Levy. “Coordinated Forward Error Recovery for Composite Web Services”. In: *In 22nd Symposium on Reliable Distributed Systems (SRDS)*. 2003, pp. 167–176. Cited on page pp. 30, 33.
- [WWJP01] Derks Wijnand, Jonker Willem, Dehnert Julyane, and Grefen Paul. “Customized Atomicity Specification for Transactional Workflows”. In: *CODAS '01: Proceedings of the Third International Symposium on Cooperative Database Systems for Advanced Applications*. Washington, DC, USA: IEEE Computer Society, 2001, p. 140. ISBN: 0-7695-1128-7. Cited on page p. 31.

-
- [XRR98] Jie Xu, Alexander Romanovsky, and Brian Randell. “Coordinated Exception Handling in Distributed Object Systems: From Model to System Implementation”. In: *ICDCS '98: Proceedings of the The 18th International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society, 1998, p. 12. ISBN: 0-8186-8292-2. Cited on page pp. 6, 33, 45, 47, 51–53.
- [Xu+95] Jie Xu et al. “Fault Tolerance in Concurrent Object-Oriented Software Through Coordinated Error Recovery”. In: *FTCS*. 1995, pp. 499–508. Cited on page p. 33.

Web Service Related Literature

- [Aal+06] Wil van der Aalst et al. “Choreography Conformance Checking: An Approach based on BPEL and Petri Nets”. In: *The Role of Business Processes in Service Oriented Architectures*. Dagstuhl Seminar Proceedings 06291. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. Cited on page pp. 38, 71.
- [ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architecture and Applications*. Springer Verlag, 2004. ISBN: 3-540-44008-9. Cited on page pp. 20, 30, 31.
- [BWR08] Adam Barker, Christopher D. Walton, and David Robertson. “Choreographing Web Services”. In: *Transactions on Services Computing* (Sept. 2008). Cited on page p. 15.
- [BDD06] Alistair Barros, Gero Decker, and Marlon Dumas. *Multi-staged and Multi-viewpoint Service Choreography Modelling*. July 2006. Cited on page p. 39.
- [BDH05] Alistair Barros, Marlon Dumas, and Arthur H.M. ter Hofstede. *Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection*. QUT Technical report, FIT-TR-2005-012. Queensland University of Technology, Brisbane 2005. Cited on page p. 53.
- [BDO05] Alistair Barros, Marlon Dumas, and Phillipa Oaks. *A Critical Overview of the Web Services Choreography Description Language*. 2005. URL: www.bptrends.com. Cited on page pp. 11, 34.
- [BKLL09] Marc Bischof, Oliver Kopp, Tammo van Lessen, and Frank Leymann. “BPELscript: A Simplified Script Syntax for WS-BPEL 2.0”. Englisch. In: *2009 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2009)*. IEEE Computer Society Press, 2009. Cited on page pp. 16, 17, 60, 71.
- [BG06] Antonio Bucchiarone and Stefania Gnesi. “A Survey on Services Composition Languages and Models”. In: *Ws-MaTe Workshop*, 2006. Cited on page p. 16.
- [Dec06] Gero Decker. “Process Choreographies in Service-oriented Environments”. MA thesis. Hasso Platner Institut, Universität Potsdam, Germany, 2006. Cited on page pp. 34, 39.

- [Dec08] Gero Decker. "Choreografiemodellierung: Eine Übersicht". In: *Informatik-Spektrum* 31.2 (2008), pp. 161–166. ISSN: 0170-6012. DOI: 10.1007/s00287-008-0227-3. Cited on page pp. 6, 16, 20.
- [Dec09a] Gero Decker. "Design and Analysis of Process Choreographies". in submission. PhD thesis. Business Process Technology Group, Hasso Plattner Institute University of Potsdam, Germany, 2009. Cited on page pp. 6, 7, 20, 35, 37, 38, 43, 81, 82, 85, 87, 103, 113.
- [Dec09b] Gero Decker. "Realizability of Interaction Models". In: *Proceedings of the 1st Central-European Workshop on Services and their Composition, ZEUS 2009, Stuttgart, Germany, March 2–3, 2009*. Vol. 438. CEUR Workshop Proceedings. CEUR-WS.org, 2009, pp. 55–60. Cited on page pp. 71, 83.
- [DB07] Gero Decker and Alistair P. Barros. "Interaction Modeling Using BPMN". In: *Business Process Management Workshops*. 2007, pp. 208–219. Cited on page pp. 12, 20, 34, 35, 81.
- [DBKL08] Gero Decker, Alistair Barros, Frank Michael Kraft, and Niels Lohmann. "Nondesynchronizable Service Choreographies". In: *Service-Oriented Computing - IC-SOC 2008, Sixth International Conference, Sydney, Australia, December 1–5, 2008, Proceedings*. Lecture Notes in Computer Science. Springer-Verlag, 2008. Cited on page p. 38.
- [DKB08] Gero Decker, Oliver Kopp, and Alistair P. Barros. "An Introduction to Service Choreographies (Servicechoreographien - eine Einführung)". In: *it - Information Technology* 50.2 (2008), pp. 122–127. Cited on page p. 11.
- [DKLW07] Gero Decker, Oliver Kopp, Frank Leymann, and Mathias Weske. "BPEL4Chor: Extending BPEL for Modeling Choreographies". Englisch. In: *Proceedings of the IEEE 2007 International Conference on Web Services (ICWS 2007), Salt Lake City, Utah, USA, July 2007*. Salt Lake City: IEEE Computer Society, 2007, pp. 296–303. Cited on page pp. 6, 12, 18, 34.
- [DKLW09] Gero Decker, Oliver Kopp, Frank Leymann, and Mathias Weske. "Interacting Services: from Specification to Execution". Englisch. In: *Data & Knowledge Engineering* (Apr. 2009). ISSN: 0169-023X. DOI: 10.1016/j.datak.2009.04.003. Cited on page pp. 16, 81.
- [DOZ06] Gero Decker, Hagen Overdick, and Johannes Maria Zaha. "On the Suitability of WS-CDL for Choreography Modeling." In: *EMISA*. Vol. 95. LNI. GI, 2006, pp. 21–33. ISBN: 978-3-88579-189-8. Cited on page p. 34.
- [DW07] Gero Decker and Mathias Weske. "Local Enforceability in Interaction Petri Nets". In: *BPM*. Vol. 4714. Lecture Notes in Computer Science. Springer, 2007, pp. 305–319. DOI: 10.1007/978-3-540-75183-0_22. Cited on page pp. 16, 34, 38, 71.
- [DW08] Gero Decker and Mathias Weske. "Interaction-centric Modeling of Process Choreographies". unpublished. 2008. Cited on page p. 82.

-
- [DD04] Remco Dijkman and Marlon Dumas. “Service-oriented Design: A Multi-viewpoint Approach”. In: *Int. J. Cooperative Inf. Syst* (Dec. 2004), pp. 337–378. Cited on page p. 39.
- [FBS04] Xiang Fu, Tevfik Bultan, and Jianwen Su. “Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services”. In: *TCS: Theoretical Computer Science* 328 (2004). Cited on page p. 38.
- [KKL07] Rania Khalaf, Dimka Karastoyanova, and Frank Leymann. “Pluggable Framework for Enabling the Execution of Extended BPEL Behavior”. English. In: *Proceedings of the 3rd International Workshop on Engineering Service-Oriented Application (WESOA2007)*. 2007. Cited on page pp. 59, 90.
- [KL06] Rania Khalaf and Frank Leymann. “Role-based Decomposition of Business Processes using BPEL”. In: *International Conference on Web Services (ICWS 2006)*. 2006, pp. 770–780. DOI: 10.1109/ICWS.2006.56. Cited on page pp. 33, 91, 102.
- [KL07] Rania Khalaf and Frank Leymann. *Coordination Protocols for Split BPEL Loops and Scopes*. English. Technical Report Computer Science 2007/01. University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2007, p. 30. Cited on page pp. 33, 90.
- [Kop08a] Oliver Kopp. *Cross-Partner-Spheres*. unpublished. 2008. Cited on page pp. 6, 41.
- [KL08] Oliver Kopp and Frank Leymann. “Choreography Design Using WS-BPEL”. English. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*. Vol. 31. 3. IEEE Computer Society Press, 2008, pp. 31–34. Cited on page pp. 18, 34, 71.
- [KL09a] Oliver Kopp and Frank Leymann. “Do We Need Internal Behavior in Choreography Models”. In: *Proceedings of the 1st Central-European Workshop on Services and their Composition, ZEUS 2009, Stuttgart, Germany, March 2–3, 2009*. Vol. 438. CEUR Workshop Proceedings. CEUR-WS.org, 2009, pp. 68–73. Cited on page p. 48.
- [KL09b] Oliver Kopp and Niels Lohmann, eds. *Proceedings of the 1st Central-European Workshop on Services and their Composition, ZEUS 2009, Stuttgart, Germany, March 2–3, 2009*. Vol. 438. CEUR Workshop Proceedings. CEUR-WS.org, 2009. Cited on page p. 48.
- [KWL09] Oliver Kopp, Matthias Wieland, and Frank Leymann. “Towards Choreography Transactions”. In: *Proceedings of the 1st Central-European Workshop on Services and their Composition, ZEUS 2009, Stuttgart, Germany, March 2–3, 2009*. Vol. 438. CEUR Workshop Proceedings. CEUR-WS.org, 2009, pp. 49–54. Cited on page pp. 13, 33, 56.
- [Kop+08a] Oliver Kopp et al. “A Model-Driven Approach to Implementing Coordination Protocols in BPEL”. In: *1st International Workshop on Model-Driven Engineering for Business Process Management (MDE4BPM)*. 2008. Cited on page pp. 97, 98, 108.

- [Kop+08b] Oliver Kopp et al. *A Model-Driven Approach to Implementing Coordination Protocols in BPEL*. Englisch. Technischer Bericht Informatik 2008/02. Universität Stuttgart, Institut für Architektur von Anwendungssystemen: Universität Stuttgart Fakultät Informatik, Elektrotechnik und Informationstechnik Germany, 2008, p. 23. Cited on page pp. 97, 98, 108.
- [Ley05] Frank Leymann. *Tools4BPEL: Schwerpunkt B: Komposition von Prozessen*. Presented in Berlin at Tools4BPEL kick-off meeting. 2005. Cited on page pp. 13, 33, 56.
- [Ley06] Frank Leymann. "Choreography for the Grid: towards fitting BPEL to the resource framework: Research Articles". In: *Concurr. Comput. : Pract. Exper.* 18.10 (2006), pp. 1201–1217. ISSN: 1532-0626. DOI: 10.1002/cpe.v18:10. Cited on page p. 16.
- [LKP08] Frank Leymann, Dimka Karastoyanova, and Mike Papazoglou. "BPM Handbook Chapter: Influential BPM Standards: History and Essence". unpublished. 2008. Cited on page pp. 6, 11, 12.
- [LR00] Frank Leymann and Dieter Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall 2000. ISBN: 0-13-021753-0. Cited on page pp. 30, 31, 56, 57.
- [LR08] Frank Leymann and Dieter Roller. "Cross Process Transactions". unpublished. 2008. Cited on page pp. 56, 68.
- [LKLRO7] Niels Lohmann, Oliver Kopp, Frank Leymann, and Wolfgang Reisig. "Analyzing BPEL4Chor: Verification and Participant Synthesis". Englisch. In: *Web Services and Formal Methods, Forth International Workshop, WS-FM 2007 Brisbane, Australia*. Springer-Verlag, 2007, pp. 46–60. DOI: 10.1007/978-3-540-79230-7_4. Cited on page pp. 71, 84.
- [LW09] Niels Lohmann and Karsten Wolf. "Realizability is Controllability". In: *Proceedings of the 1st Central-European Workshop on Services and their Composition, ZEUS 2009, Stuttgart, Germany, March 2–3, 2009*. Vol. 438. CEUR Workshop Proceedings. CEUR-WS.org, 2009, pp. 61–67. Cited on page p. 71.
- [LSBG99] Heiko Ludwig, Ming-Chien Shan, Christoph Bussler, and Paul Grefen. "Cross-organisational Workflow Management and Coordination: WACC'99 Workshop Report". In: *SIGGROUP Bull.* 20.1 (1999), pp. 59–62. DOI: 10.1145/327556.327641. Cited on page pp. 34, 113.
- [Mar03] Axel Martens. "On Compatibility of Web Services". In: *Petri Net Newsletter* 65 (2003), pp. 12–20. Cited on page pp. 38, 71.
- [Mie06] Ralph Mietzner. "Extraction of WS-Business Activity from BPEL 1.1". deutsch. Diplomarbeit. Universität Stuttgart, Fakultät Informatik Elektrotechnik und Informationstechnik Germany, 2006, p. 138. Cited on page pp. 8, 58, 90, 98–102, 108.

-
- [NCS04] Mangala Gowri Nanda, Satish Chandra, and Vivek Sarkar. “Decentralizing Execution of Composite Web Services”. In: *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2004, pp. 170–187. ISBN: 1-58113-831-9. DOI: 10.1145/1028976.1028991. Cited on page p. 33.
- [ODBH06] Chun Ouyang, Marlon Dumas, Stephan Breutel, and Arthur. H. M. ter Hofstede. “Translating Standard Process Models to BPEL”. In: *Proceedings 18th International Conference on Advanced Information Systems Engineering (CAiSE)*. Vol. 4001. Lecture Notes in Computer Science. Springer Verlag, 2006. DOI: 10.1007/11767138_28. Cited on page p. 18.
- [Pfi07] Kerstin Pfitzner. “Choreography Configuration for BPMN”. Englisch. Diplomarbeit. Universität Stuttgart Fakultät Informatik Elektrotechnik und Informationstechnik Germany, 2007, p. 127. Cited on page p. 75.
- [PDKL07] Kerstin Pfitzner, Gero Decker, Oliver Kopp, and Frank Leymann. “Web Service Choreography Configurations for BPMN”. In: *ICSOC Workshops*. Vol. 4907. Lecture Notes in Computer Science. Springer, 2007, pp. 401–412. ISBN: 978-3-540-93850-7. DOI: 10.1007/978-3-540-93851-4. Cited on page pp. 18, 75.
- [PML07] Stefan Pottinger, Ralph Mietzner, and Frank Leymann. “Coordinate BPEL Scopes and Processes by Extending the WS-Business Activity Framework”. In: *OTM Conferences (1)*. Vol. 4803. Lecture Notes in Computer Science. Springer, 2007, pp. 336–352. ISBN: 978-3-540-76846-3. DOI: 10.1007/978-3-540-76848-7_22. Cited on page pp. 58, 90, 100.
- [RKDL08] Peter Reimann, Oliver Kopp, Gero Decker, and Frank Leymann. *Generating WS-BPEL 2.0 Processes from a Grounded BPEL4Chor Choreography*. Englisch. Technischer Bericht Informatik 2008/07. Universität Stuttgart Fakultät Informatik Elektrotechnik und Informationstechnik, Germany, 2008, p. 95. Cited on page p. 112.
- [Sch08a] David Schumm. “Graphische Modellierung von BPEL Prozessen unter Verwendung der BPMN Notation”. Diplomarbeit, Institut für Architektur von Anwendungssystemen, Universität Stuttgart. MA thesis. 2008. Cited on page p. 18.
- [SHB98] Amit Sheth, Yanbo Han, and Christoph Bussler. “A Taxonomy of Adaptive Workflow Management”. In: *CSCW-98 Workshop, Towards Adaptive Workflow Systems*. 1998. Cited on page p. 25.
- [Ste08] Thomas Steinmetz. “Ein Event-Modell für WS-BPEL 2.0 und dessen Realisierung in Apache ODE”. deutsch. summary. Diplomarbeit. Universität Stuttgart Fakultät Informatik Elektrotechnik und Informationstechnik Germany, 2008, p. 119. Cited on page pp. 7, 56, 59, 60, 90–92, 96, 97, 109.
- [Vet06] Thorsten Vetter. “Anpassung und Implementierung verschiedener Transaktionsprotokolle auf WS-Coordination”. Deutsch. Diplomarbeit. Universität Stuttgart Fakultät Informatik Elektrotechnik und Informationstechnik Germany, 2006, p. 117. Cited on page p. 57.

- [Wan09] Yunxiao Wang. “A Generic Scoping Concept for Workflows”. Diplomarbeit. Universität Stuttgart Fakultät Informatik Elektrotechnik und Informationstechnik Germany, 2009. Cited on page pp. 34, 50, 56, 61, 113.
- [Wee+05] Sanjiva Weerawarana et al. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005. ISBN: 0-13148-874-0. Cited on page p. 11.
- [Wes07] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. First. Springer Verlag, 2007. ISBN: 3-54073-521-6. Cited on page pp. 6, 15, 16, 18, 35, 37–39, 53.
- [Whi05] Stephen White. “Using BPMN to Model a BPEL Process”. In: *BPTrends 3.3* (Mar. 2005), pp. 1–18. Cited on page p. 18.
- [Zah+08] Johannes Maria Zaha et al. “Bridging Global and Local Models of Service-Oriented Systems.” In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C 38.3* (Oct. 28, 2008), pp. 302–318. Cited on page pp. 12, 34, 81.

Specifications and Online Ressources

- [AH07] Wil van der Aalst and Arthur H. M. ter Hofstede. *Workflow Patterns*. 2007. URL: <http://www.workflowpatterns.com/>. Cited on page p. 53.
- [Apa08] Apache Software Foundation. *WS-BPEL 2.0 Specification Compliance*. 2008. URL: <http://ode.apache.org/ws-bpel-20-specification-compliance.html>. Cited on page p. 106.
- [Ast08] Astro. *Seminar Slides*. 2008. URL: <http://www.astroproject.org/downloads/SeminarSlides/ws-cdl.ppt>. Cited on page p. 34.
- [BPE07] BPEL. *Web Services Business Process Execution Language (WS-BPEL)—Version 2.0*. 2007. URL: <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html>. Cited on page pp. 11, 16, 50, 98.
- [BPM09] BPMN1.2. *Business Process Modeling Notation (BPMN), V1.2 – OMG Available Specification*. OMG Available Specification. Object Management Group. Jan. 2009. URL: <http://www.omg.org/spec/BPMN/1.2/>. Cited on page pp. 11, 18, 34.
- [BPM08] BPMN2.0. *Business Process Model and Notation (BPMN), V2.0—Draft Proposal VO.5.2*. Sept. 2008. URL: <http://www.omg.org/cgi-bin/doc?bmi/08-09-04>. Cited on page pp. 19, 49, 73–75.
- [BTP04] BTP. *Business Transaction Protocol (BTP) Revision 1.1*. OASIS. 2004. URL: <http://www.oasis-open.org/committees/business-transactions/>. Cited on page p. 57.
- [Ecl08] Eclipse. *BPEL Project*. 2008. URL: <http://www.eclipse.org/bpel/>. Cited on page p. 16.

-
- [Gro08] Alexander Grosskopf. *Event Flavors in BPMN - Get a Taste*. Oct. 2008. URL: <http://www.bpmn.info/?p=52>. Cited on page p. 73.
- [Jav05] Java. *The Java Language Specification, Third Edition*. 2005. URL: http://java.sun.com/docs/books/jls/third_edition/html/interfaces.html. Cited on page p. 34.
- [Ley01] Frank Leymann. “Web Services Flow Language (WSFL 1.0)”. In: (May 2001). IBM Software Group. Cited on page p. 16.
- [Ory08a] Oryx Framework. *BPEL*. 2008. URL: <http://oryx-editor.org/backend/poem/repository/new?stencilset=/stencilsets/bpel/bpel.json>. Cited on page p. 16.
- [Ory08b] Oryx Framework. *BPEL4Chor*. 2008. URL: <http://oryx-editor.org/backend/poem/repository/new?stencilset=/stencilsets/bpmnplus/bpmnplus.json>. Cited on page p. 18.
- [Ory08c] Oryx Framework. *BPMN*. 2008. URL: <http://oryx-editor.org/backend/poem/repository/new?stencilset=/stencilsets/bpmn/bpmn.json>. Cited on page p. 18.
- [Ory08d] Oryx Framework. *iBPMN*. 2008. URL: <http://oryx-editor.org/backend/poem/repository/new?stencilset=/stencilsets/ibpmn/ibpmn.json>. Cited on page p. 20.
- [RDF98] RDF. *Resource Description Framework (RDF) Model and Syntax Specification*. WD-rdf-syntax-19980819. Available at <http://www.w3.org/TR/WD-rdf-syntax/>. 1998. URL: <http://www.w3.org/TR/WD-rdf-syntax/>. Cited on page p. 106.
- [SOA07] SOAP 1.2. 2007. URL: <http://www.w3.org/TR/soap12/>. Cited on page p. 11.
- [Sch09] Torben Schreiter. *BPMN 2.0, Choreographiemodellierung*. 2009. URL: <http://www.signavio.com/de/unternehmen/blog/72-bpmn-20-bei-signavio.html>. Cited on page p. 19.
- [Tha01] Satish Thatte. “XLANG Web Services for Business Process Design”. In: (2001). Microsoft Corporation. Cited on page p. 16.
- [UDD05] UDDI. *Universal Description, Discovery and Integration v3.0.2 (UDDI)*. 2005. URL: <http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm>. Cited on page p. 11.
- [UML05] UML. *Unified Modeling Language (UML), version 2.2*. 2005. URL: <http://www.omg.org/cgi-bin/doc?formal/05-04-01>. Cited on page p. 11.
- [W3C04] W3C. *Glossary: Choreography*. 2004. URL: <http://www.w3.org/2003/glossary/keyword/All/?keywords=choreography>. Cited on page p. 15.
- [WB09] WS-BA. *Web Services Business Activity (WS-BA) Version 1.2*. Feb. 2009. URL: <http://docs.oasis-open.org/ws-tx/wstx-wsba-1.2-spec-os/wstx-wsba-1.2-spec-os.html>. Cited on page p. 21.
- [WC09] WS-C. *Web Services Coordination (WS-Coordination). Version 1.2*. 2009. URL: <http://docs.oasis-open.org/ws-tx/wscoor/2006/06>. Cited on page pp. 11, 20.

- [WC05a] WS-CAF. *WS-Composite Application Framework (WS-CAF)*. OASIS. 2005. URL: <http://www.oasis-open.org/committees/ws-caf/>. Cited on page p. 57.
- [WC02] WS-CDL. *WS-Choreography Definition Language (WS-CDL)*. Web. 2002. URL: http://www.ebpm1.org/ws_-_cdl.htm. Cited on page p. 54.
- [WC05b] WS-CDL. *Web Services Choreography Description Language Version (WS-CDL) 1.0*. Web. W3C. Nov. 2005. URL: <http://www.w3.org/TR/ws-cdl-10/>. Cited on page pp. 34, 48, 49, 53, 66.
- [WSD01] WSDL. *Web Services Description Language (WSDL) 1.1*. 2001. URL: <http://www.w3.org/TR/2001/NOTE-wsd1-20010315>. Cited on page p. 11.
- [XSL99] XSLT. *XSL Transformations (XSLT)*. 1999. URL: <http://www.w3.org/TR/xslt>. Cited on page p. 106.
- [Yen03] Prasad Yendluri. *Web Services Choreography*. 2003. URL: <http://www.webpronews.com/topnews/2003/09/30/web-services-choreography>. Cited on page p. 37.

Other Literature

- [BN97] Philip A. Bernstein and Eric Newcomer. *Principles of Transaction Processing*. San Francisco: Morgan Kaufmann, 1997. ISBN: 1-55860-415-4. Cited on page p. 57.
- [Bis08] Marc Bischof. *Interprozedurale SSA-Form—Das Konzept der statischen Einmalzuweisung*. 2008. Cited on page pp. 6, 54, 55.
- [CB74] Donald D. Chamberlin and Raymond F. Boyce. “SEQUEL: A structured English Query Language”. In: *FIDET '74: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*. New York, NY, USA: ACM Press, 1974, pp. 249–264. DOI: 10.1145/800296.811515. Cited on page p. 12.
- [Che76] Peter P. Chen. “The Entity-Relationship Model - Toward a Unified View of Data”. In: *ACM Trans. Database Syst.* 1.1 (1976), pp. 9–36. Cited on page p. 11.
- [CAKT99] Peter P. Chen, Jacky Akoka, Hannu Kangassalo, and Bernhard Thalheim, eds. *Conceptual Modeling, Current Issues and Future Directions, Selected Papers from the Symposium on Conceptual Modeling, Los Angeles, California, USA, held before ER'97*. Vol. 1565. Lecture Notes in Computer Science. Springer, 1999. ISBN: 3-540-65926-9. Cited on page p. 105.
- [CJ88] Paul Compton and Bob Jansen. “Knowledge in Context: A Strategy for Expert System Maintenance”. In: *Proceedings of the 2nd Australian Joint Artificial Intelligence conference*. Vol. 406. Lecture Notes in Artificial Intelligence. Adelaide: Springer, 1988, pp. 292–306. Cited on page p. 28.
- [Con+92] Reidar Conradi et al. “Design, Use and Implementation of SPELL, a Language for Software Process Modeling and Evolution”. In: *Proc. European Workshop on the Software Process*. Vol. 635. Lecture Notes in Computer Science. Trondheim, Norway: Springer-Verlag, Berlin, 1992, pp. 167–177. Cited on page p. 34.

-
- [CF98] Flaviu Cristian and Christof Fetzer. “The Timed Asynchronous Distributed System Model”. In: *FTCS*. 1998, pp. 140–149. Cited on page p. 63.
- [DOW08] Gero Decker, Hagen Overdick, and Mathias Weske. “Oryx — Sharing Conceptual Models on the Web”. In: *ER '08: Proceedings of the 27th International Conference on Conceptual Modeling*. Barcelona, Spain: Springer-Verlag, 2008, pp. 536–537. ISBN: 978-3-540-87876-6. DOI: 10.1007/978-3-540-87877-3_49. Cited on page p. 105.
- [DE95] Jörg Desel and Javier Esparza. *Free choice Petri nets*. Vol. 40. Cambridge tracts in theoretical computer science. Cambridge: Cambridge University Press, 1995, pp. viii,244. ISBN: 0-52101-945-1. Cited on page p. 85.
- [Fil98] Robert E. Filman. “Achieving Ilities”. In: *OMG-DARPA Workshop on Compositional Software Architectures*. 1998. Cited on page p. 27.
- [GMS87] Hector Garcia-Molina and Kenneth Salem. “Sagas”. In: *Proc. ACM SIGMOD Conf*. San Francisco, CA 1987, p. 249. Cited on page p. 31.
- [Gla92] Robert L. Glass. *Building Quality Software*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992. ISBN: 0-13-086695-4. Cited on page p. 27.
- [Her09] Daniel Herrscher. *IP in der Fahrzeugvernetzung*. Informatik-Kolloquium Universität Stuttgart. May 2009. Cited on page p. 23.
- [HR01] Theo Härder and Erhard Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung, 2. Auflage*. Springer, 2001. ISBN: 3-540-42133-5. Cited on page pp. 12, 57.
- [Mos81] J. Eliot B. Moss. “Nested Transactions: An Approach to Reliable Distributed Computing”. PhD thesis. 260, Machine Intelligence, eds: Meltzer, and Michie, vars. PublishersT, 1981. Cited on page p. 31.
- [Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN: 1-55860-320-4. Cited on page pp. 53, 54, 82.
- [NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999. ISBN: 3-54065-410-0. Cited on page pp. 53, 54.
- [Pet07] Nicolas Peters. “Oryx Stencil Set Specification”. MA thesis. Hasso Plattner Institut, Universität Potsdam, Germany, 2007. Cited on page pp. 105, 106.
- [Sch08b] Uwe Schöning. *Theoretische Informatik - kurz gefasst*. 4th ed. Spektrum, Akad. Verlag, 2008. ISBN: 9-78-382741-8241. Cited on page pp. 72, 98.
- [SHO95] Stanley Sutton, Dennis Heimbigner, and Leon Osterweil. “APPL/A: A Language for Software Process Programming”. In: *ACM Transactions on Software Engineering and Methodology* 4.3 (July 1995), pp. 221–286. Cited on page p. 34.
- [TS02] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Upper Saddle River, NJ: Prentice Hall, 2002. ISBN: 0-13-088893-1. Cited on page p. 50.

[Voa01] Jeffrey Voas. “Quality Time: Composing Software Component “ilities””. In: *IEEE Software* 18.4 (July 2001), pp. 16–17. ISSN: 0740-7459. Cited on page p. 27.

All links were last followed on July 5, 2009.

A Overview of Coordination Collaboration

For the sake of brevity, Section 7.4 on page 92 discusses the collaboration between the coordination actors (i. e. *Generic Controller* (GC) and *Custom Controller* (CC) as well as *Local Coordinator* (LCo) and *Global Coordinator* (GCo)) from a shortened perspective. To provide a detailed view on the overall interaction, this appendix give a short description of the particular message exchanges. Eleven phases are distinguished:

- initialization phase
- completion phase
- encountering fault phase
- blocking mode
- termination mode
- compensation mode
- closing mode
- compensate within fault handler mode
- resuming from fault handler mode
- faulting fault handler mode
- terminating fault handler mode

A.1 Initialization Phase

The initialization phase starts with the *process instantiated* event of the GC. Although, a corresponding CPS-fragment may not be started at this point, the CC requests the creation of a coordination context since the process may raise the *instance terminated* event before the corresponding fragment is enabled. For this purpose, the LCo has to inform the GCo, since child activities are also affected by the terminating instance. Thus, the LCo registers its fragment to the corresponding GCo of the CPS. Thereby, starting activities of the corresponding process are blocked until the LCo has registered successfully to the GCo. In case of the *instance termination* event the LCo immediately sends the *fragment faulted* message to the GCo.

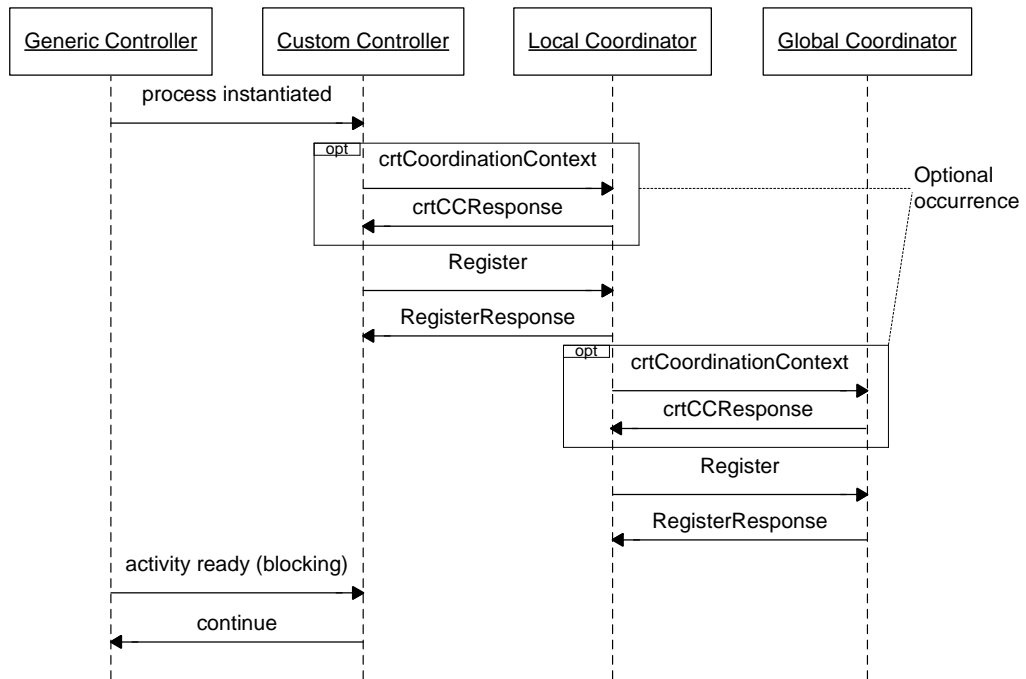


Figure A.1: Initialization phase

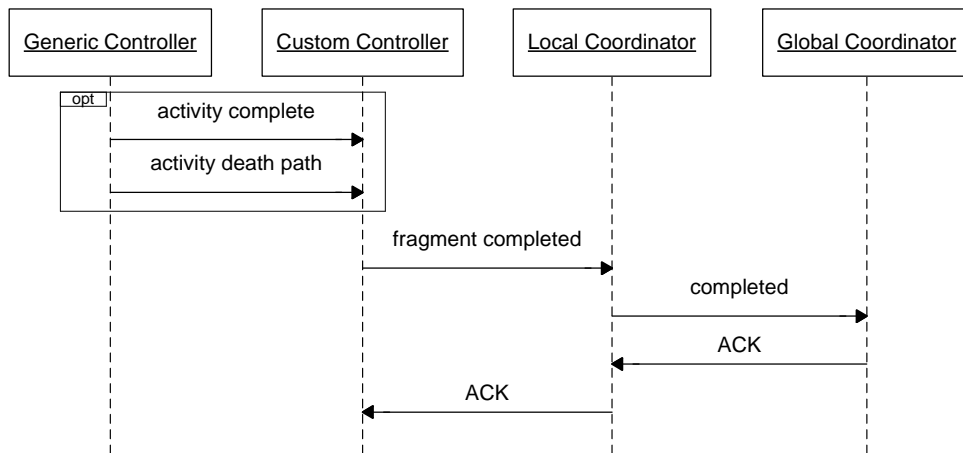


Figure A.2: Completion phase

A.2 Completion Phase

The completion phase for a particular fragment marks the state where all activities of a fragment have completed successfully. This can be achieved either by completing an activity or by triggering *Death Path Elimination*. For a completed fragment LCo and GCo are informed, which has to respond with an acknowledgment.

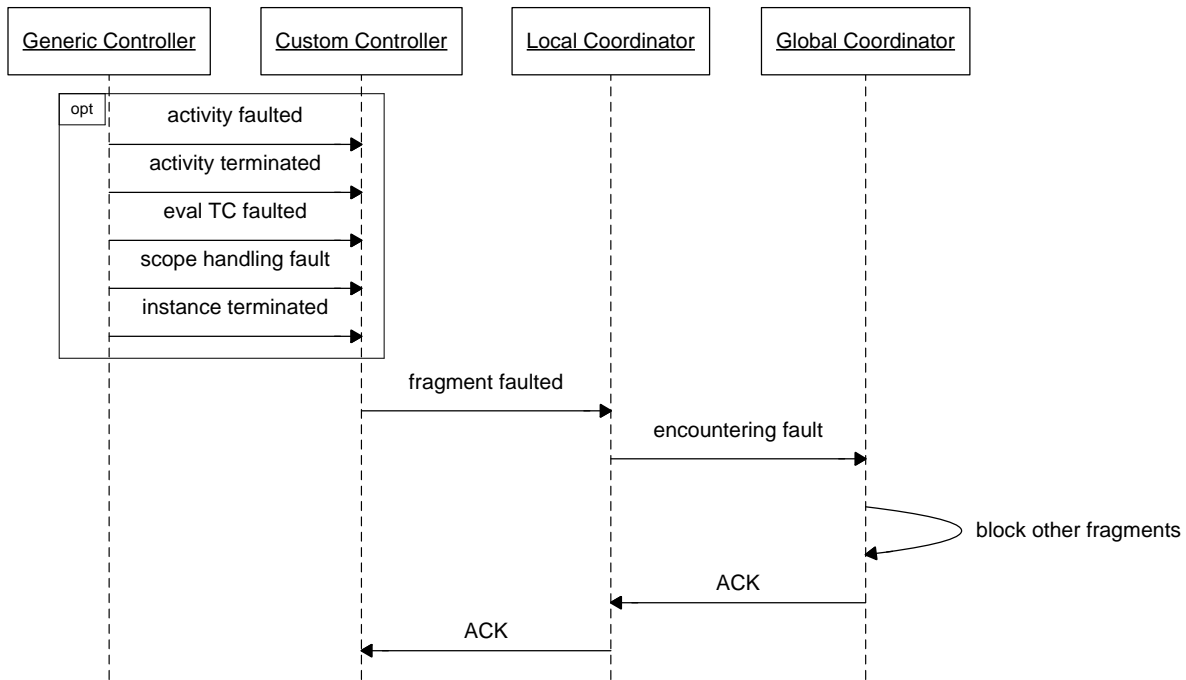


Figure A.3: Encountering fault phase

A.3 Encountering Fault Phase

The encountering fault phase marks the state where at least one activity faults. This is signaled to the CC by a set of events, such as *activity faulted*, *activity terminated*, *evaluation of transition condition faulted* or *instance terminated*. Thereby, LCo and GCo are informed. On encountering fault message, the GCo has to propagate the situation to all CPS-participants to enable their blocking mode where the CC stops releasing blockades immediately. Until fragments are not blocked, they can raise faults on their own. Thus, simultaneous fault handling is supported. Having encountered the resulting fault, GCo and LCo respond with an acknowledgment and start with the particular fault handling messages.

A.4 Blocking Phase

The blocking phase is triggered by the GCo to suspend the fragment execution due a fault in another fragment. For this purpose, the LCo is triggered by the GCo, while the LCo informs the CC to stop releasing blockades. The CC will then stop releasing blockades on its own, so that LCo can take over control. Notice, that for enabling the blocking phase, the GC is not informed, since the CC just stops releasing blockades immediately.

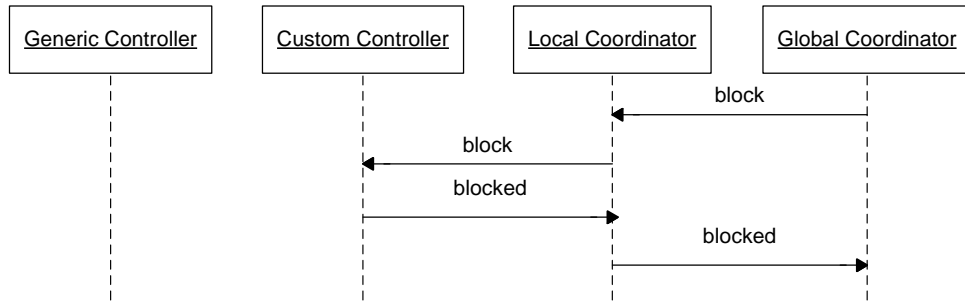


Figure A.4: Blocking phase

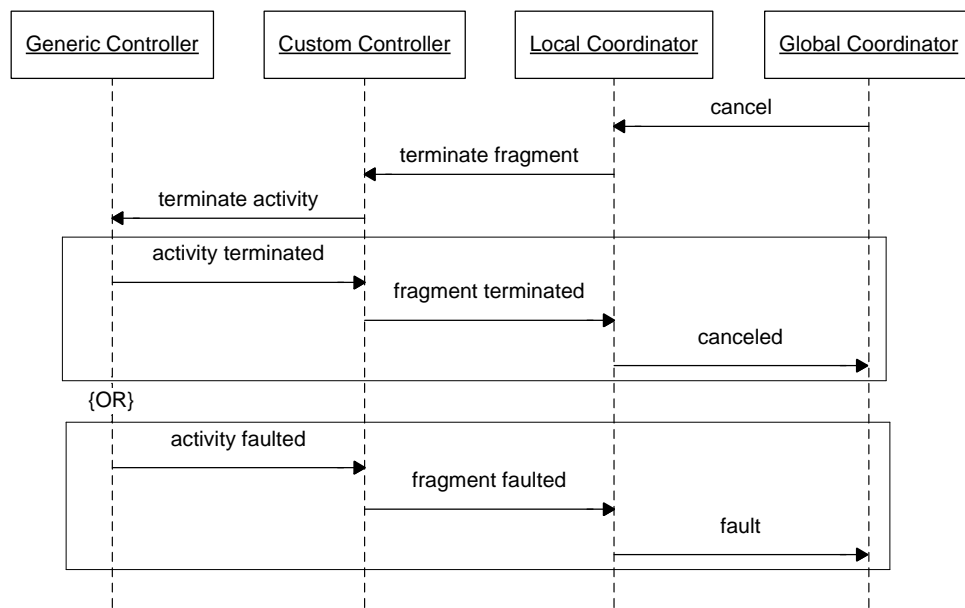


Figure A.5: Termination phase

A.5 Termination Phase

During the fault handling, a possible outcome is the occurrence of a fault where nested activities have to be aborted. For this purpose, the GCo triggers the *cancel* message to terminate certain fragments. This messages results in several *terminate activity* incoming events in the GC. Due to the fact that termination may fail, two possible outcomes are identified: (i) the GC responds with *activity terminated* event indicating successful termination. If all activities of a fragment are terminated successfully, the LCo can trigger *canceled* to the GCo. If the GC responds with (ii) *activity faulted* event, the termination is faulted. This is signaled to LCo and GCo.

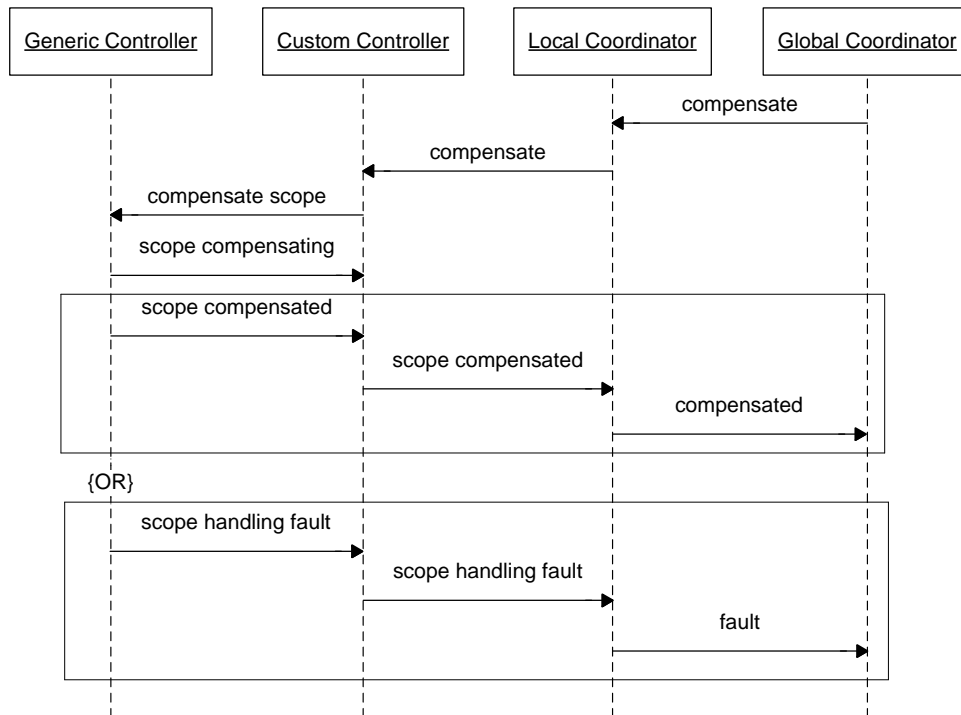


Figure A.6: Compensation phase

A.6 Compensation Phase

Similar to the termination phase, the GCo may request the *compensation* of particular fragments. This results in *compensate scope* incoming events in the GC. In an analogous way, compensation may either be successful or not, resulting in either *compensated* or *fault* message to the GCo.

A.7 Closing Phase

If the complete CPS is finished and no further compensation affecting the particular CPS can be triggered by other scopes, the GCo may request the *exit* of the CPS.

A.8 Compensate within Fault Handler Phase

During fault handling state, it may be possible that some artifacts have to be compensated due to flexible compensation. Hence, the GCo may request *requestSubScopeCompensation*, resulting in the compensation of a scope.

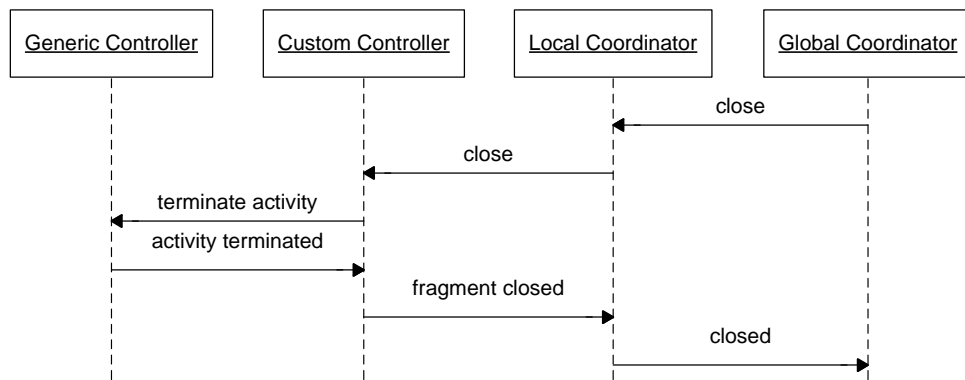


Figure A.7: Closing phase

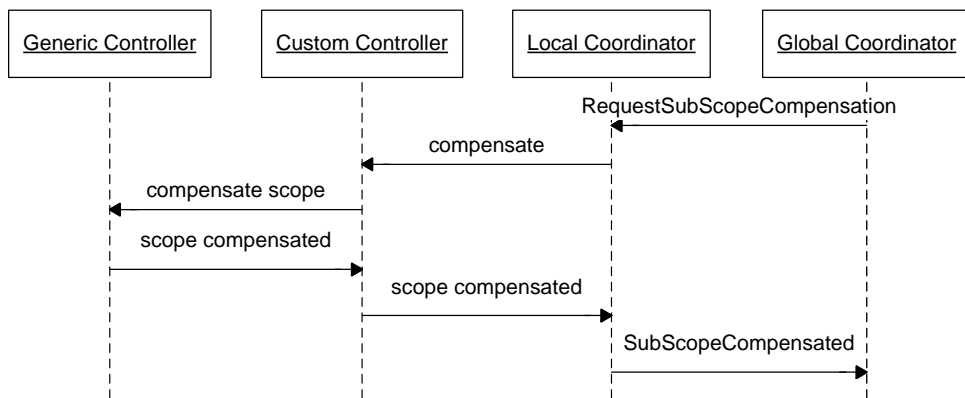


Figure A.8: Compensate within fault handler phase

A.9 Resuming from Fault Handler Phase

In fault handling state, it is possible to resume fragments due to non-cancellation behavior or successful fault handling. Thus, the GCo may trigger *continue* message, leading to *suppress fault* incoming event in the GC. Thereby, blockades are released without propagating the fault.

A.10 Faulting Fault Handler Phase

In contrast to the resumption from fault handling state, the GCo may trigger propagation of a fault due to faulting fault handler. This results in a *continue* incoming event in the GC, leading to a propagation of the fault.

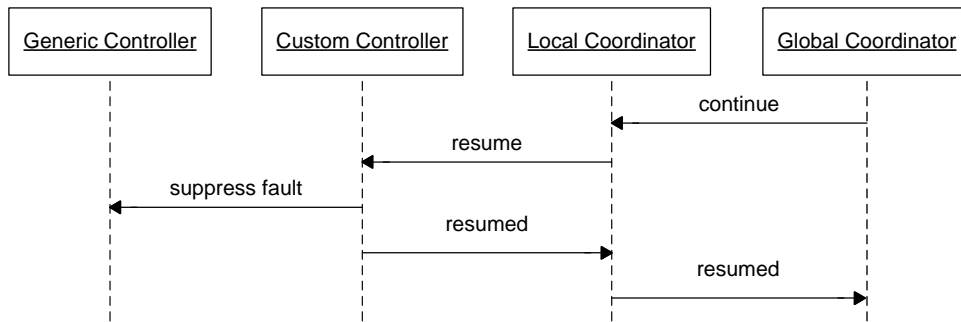


Figure A.9: Resuming phase

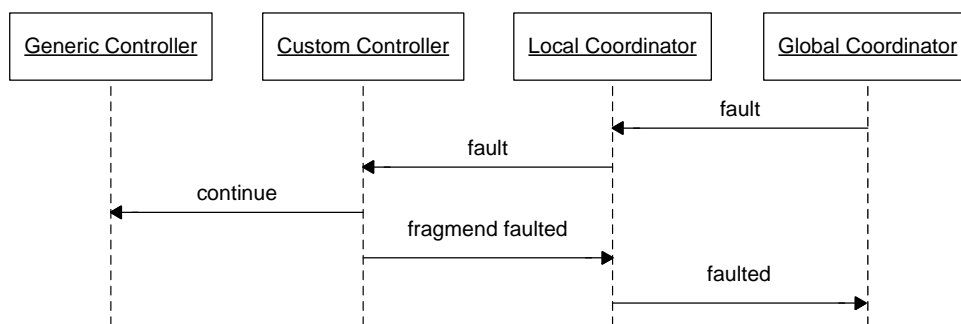


Figure A.10: Faulting phase

A.11 Terminating Fault Handler Phase

If a fault occurs, the BPEL semantic states the termination of all subactivities. Thus, the GCo may request the *termination* of fragments.

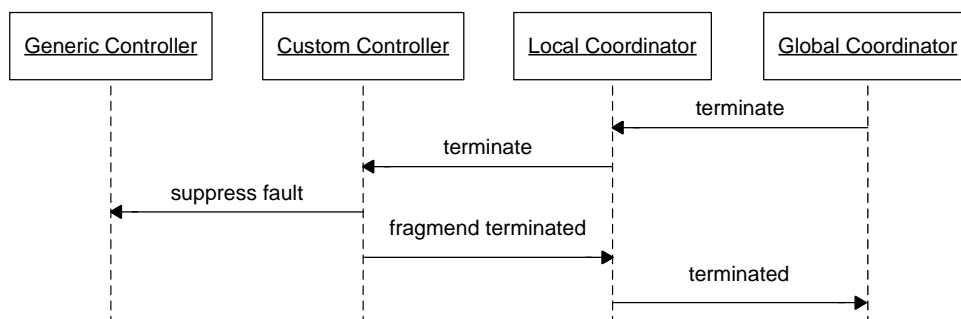


Figure A.11: Termination phase (subactivities)

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.

(Marc Bischof)