Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
70569 Stuttgart

Diplomarbeit Nr. 2444

Extraction of WS-Business Activity from BPEL 1.1

Ralph Mietzner

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. Stefan Pottinger Dipl.-Inf. Thorsten Scheibler
begonnen am:	1. März 2006
beendet am:	13. Juli 2006
CR-Klassifikation:	C.2.4, H.2.4, H.3.5, H.4.1

Contents

1 Introduction **1**
1.1 Structure of this Document 2

2 Basics **3**
2.1 Long-Running Business Transactions 3
2.1.1 Nested Transactions 3
2.1.2 Sagas 5
2.2 Transactional Behavior in BPEL Processes 5
2.2.1 Scopes 6
2.2.2 Fault Handling 8
2.2.3 Compensation 9
2.3 WS-Coordination 9
2.4 WS-Business Activity 10
2.5 ActiveBPEL Engine 11
2.5.1 Architecture 11
2.5.2 Activities 12
2.5.3 ActiveBPEL Process Deployment Descriptors 15
2.6 Summary 16

3 Motivational Example **17**
3.1 Bike Service Example 17
3.2 Summary 19

4 Extraction of WS-Business Activity from BPEL **20**
4.1 Transaction Middleware 20
4.1.1 Coordinators 21
4.2 Actions to Be Controlled by the External Coordinator 22
4.2.1 Events that Trigger Outgoing Scope Life Cycle Messages . . 22
4.2.2 Overview over Incoming Scope Life Cycle Messages 23
4.3 Registration of Scopes for External Coordination 24
4.3.1 WS-C Activities and Participants 25
4.3.2 WS-BA Coordination Types 25
4.3.3 Consequences for the PureBPE 27
4.3.4 Consequences for the BPEL4Tx Coordinator 27
4.3.5 Message Exchange 28

4.4 Mapping of WS-Business Activity Messages to BPEL 30
4.4.1 Shortcomings of the Business Agreement With Participant Completion Protocol 30
4.4.2 The Modified Business Agreement with Participant Completion Protocol (mBAP) 32
4.4.3 Normal Processing Mode 33
4.4.4 A Scope Completed Normally 34
4.4.5 A Scope Faults 35
4.4.6 Termination of Scopes 36
4.4.7 Implicit Fault Handling 38
4.4.8 A Fault Handler Ends 40
4.4.9 Execution of External Activities 43
4.4.10 Compensation 45
4.4.11 Closing of Scopes and Nested Coordinators 48
4.5 Summary 50

5 Prototype Realization 51
5.1 Prototype Architecture Overview 51
5.1.1 Architecture Alternatives 52
5.2 Extraction of Transaction Information from the BPEL Engine . . . 54
5.2.1 Requirements 55
5.2.2 Modified Parts of the ActiveBPEL Engine 55
5.2.3 Transaction Gateway 56
5.2.4 Additional States 64
5.3 Adding JMS to the ActiveBPEL Engine 67
5.3.1 JMS Features for the PureBPE 68
5.3.2 JMS Features for the Normal ActiveBPEL engine 68
5.3.3 JMS URL Syntax 69
5.3.4 New Axis Handler to Send JMS Messages 70
5.3.5 JMS Listeners 72
5.4 Persistence Considerations 74
5.5 BPEL4Tx Coordinator 76
5.5.1 BPEL4Tx Ordering Broker 76
5.5.2 BPEL4Tx WS-BA Default Fault Handler 85
5.5.3 BPEL4Tx WS-BA Participant 86
5.6 WS-C /WS-BA Coordinator 97
5.6.1 WS-C Activation Service Part 97
5.6.2 WS-C Registration Service Part 98
5.6.3 WS-BA Protocol Service 100
5.7 Concurrency Considerations 105
5.7.1 Concurrent Scopes 105
5.7.2 Concurrent External Activities 106
5.8 Summary 106

6 Performance Comparisons **107**
6.1 Test Environment . . . 107
6.1.1 Hardware . . . 107
6.1.2 Test Processes . . . 108
6.2 Data Gathering . . . 110
6.3 Results . . . 110
6.3.1 Result Explanation . . . 110
6.4 Summary . . . 112

7 Summary and Outlook **113**

List of Figures **VIII**

Listings **IX**

List of Tables **X**

Bibliography **XIV**

A A Basic Mapping of Regular Expressions to BPEL **XV**
A.1 Related Work . . . XVI
A.2 Atomic Structures . . . XVI
A.2.1 Concatenation . . . XVI
A.2.2 Alternation . . . XVII
A.2.3 Kleene Star . . . XVII
A.3 Summary . . . XX

B BPEL4Tx Message Visualizer **XXI**

1 Introduction

The business process execution language version 1.1 (BPEL) is the de-facto standard for the modeling of business processes using the web-service stack. [18] Considering the fact that business processes often control long-running business interactions and manipulate business-critical data in databases and other business-critical applications, a fault-handling paradigm that can cope with long-running business-critical business transactions is needed.
BPEL employs a compensation based fault-handling model. This compensation based model called long-running business transactions (LRT) is used to coordinate the outcome of the overall BPEL process based on the outcome of nested activities inside the process. Using fault and compensation handlers, the LRT feature allows a process to flexibly control the fault handling and reversal of nested activities in the case of a fault.
The notion of LRTs, as described in the BPEL specification, is purely local, meaning that a LRT is used to control the fault and compensation handling inside a single business process instance.
However, one of the fundamental concepts of BPEL is *recursive composition.*[47] Recursive composition means that a BPEL process is built from the composition of web-services and provides a web-service interface that in turn can be used by other BPEL processes to form new processes.
The LRT model used by BPEL does not address the participation of external partners in the BPEL long-running transactions. When the need to compensate a part of a BPEL process instance that contains the invocation of an external partner arises, the actions an external partner service has already performed and committed might need to be reverted. In order to perform such a compensation activity the BPEL process instance must invoke a method upon the external partner that compensates the work the partner has already performed. This compensation method must be known to the BPEL process, and must be explicitly provided by the partner service.

The task of this diploma thesis is to extract the LRT feature from BPEL so that it can be handled by an external coordinator based on WS-Coordination (WS-C) [13] and WS-Business Activity (WS-BA) [11].
Coordinating LRTs externally allows external partners to participate in the LRT that controls the outcome of a BPEL process. This behavior has the advantage that the BPEL process instance can simply use coordination protocol messages to control internal parts of the business process instance as well as external partners. Additionally a BPEL process instance itself can be started in the context of an-

other transaction, allowing its outcome to be controlled by the application that started it.

Based on an open source BPEL engine a prototype is to be implemented. The implementation part of this diploma thesis consists of three main parts.

The first part is the extraction of transaction information from the BPEL engine. Therefore an open source BPEL engine must be modified so that it can send and receive messages that are needed to coordinate BPEL long-running transactions externally in a reliable way. The second part is the implementation of a mapper middleware component that maps the transaction information from the BPEL engine to WS-C and WS-BA compliant messages. The third part is the implementation of a WS-C and WS-BA compliant coordinator that controls the transactions in the BPEL engine from the outside.

1.1 Structure of this Document

Chapter 2 covers the basics needed to understand the following chapters. These basics include an analysis of the transaction specific parts of BPEL, as well as a short discussion of some WS* specifications such as WS-Coordination and WS-Business Activity. The chapter is rounded up by a brief overview over the parts of the open source BPEL engine relevant to the prototype which was realized as a part of this thesis.
Chapter 3 provides a motivational example why it is a good idea to use an external coordinator to coordinate BPEL long-running transactions.
Chapter 4 then deals with the theoretical aspects of what is needed to extract transaction information from BPEL. The consequences that arise from such an extraction for the middleware running a BPEL process is discussed and a coordination protocol that is suitable for the handling of BPEL long-running transactions is suggested. An important aspect of this chapter is also the mapping of events inside a BPEL process life cycle to appropriate coordination protocol messages.
Chapter 5 discusses how the theoretical aspects of Chapter 4 have been implemented in a prototype. It is shown in detail how the open source BPEL engine has been modified to cope with new requirements identified in Chapter 4, and how the external coordinator has been designed and implemented.
Chapter 6 shows the implications of the external coordination of BPEL LRTs on the performance of the execution of BPEL processes.
Finally Chapter 7 draws a conclusion and provides an outlook proposing suggestions on how to further prosecute the matter.
In Appendix A a basic mapping from regular expressions to BPEL is discussed since this mapping is needed for the design of the external coordinator.
In Appendix B the MessageVisualizer, a tool to display transaction specific messages is presented.

2 Basics

This chapter covers some basics that are needed for the understanding of the following chapters of this document. Only the aspects of a theme that are necessary for this diploma thesis are covered, other parts can be found in the literature cited.

2.1 Long-Running Business Transactions

Business processes often run over a long period of time, sometimes even days or weeks. Using an ACID-based fault handling paradigm would cause a few problems that result from the short-lived notion of an ACID transaction. [31] [28]
In a long-running business transaction the isolation property of an ACID transactions causes resources to be locked for a long time, making the concurrent execution of applications that share resources nearly impossible.
The isolation property of an ACID transaction specifies that resources used in a transaction are locked until the transaction successfully commits or is rolled back isolating the resources from other applications. In addition to holding locks for a long time, long-running business transactions often access many resources which increases the deadlock rate.
Considering the atomicity property of ACID transactions, a fault inside the business process instance causes the whole transaction (i.e. the whole business process instance) to be rolled back. In a long-running business process this can mean that the work accumulated during weeks is rolled back and needs to be performed again.
In order to overcome the shortcomings of the ACID concept regarding long-running business transactions, several concepts such as *nested transactions* and *Sagas* have been proposed.

2.1.1 Nested Transactions

Basic ACID transactions are also called “flat” transactions since they lack an internal structure. [30] Atomicity and synchronization in flat transactions is always based on the transaction as a whole. Either the whole transaction commits or it aborts. In transactions that contain many operations it is beneficial to introduce

groups of operations. These groups can be formed for example for groups of operations that can be computed concurrently or that reflect the modular structure of a transaction. [33]
Nested transactions [34] allow the building of a hierarchy of transactions called a *transaction tree*. The root transaction in such a transaction tree is called *top-level transaction*. The top-level transaction contains one or more *sub-transactions* which in turn can contain further sub-transactions. If a transaction contains one or more sub-transactions it is called the *parent* of these sub-transactions. The sub-transactions of a parent are called the *children* of their parent transactions. The root transaction of such a nested transaction still has ACID properties while the sub-transactions give up one or more of the atomicity, consistency, isolation and durability properties.

2.1.1.1 Closed Nested Transactions

In the *closed nested transaction* model the sub-transactions give up the durability (D) property. Sub-transactions no longer commit finally but perform a so-called *local* commit. This means their results become visible for their parent transaction but not for other transactions outside the nested transaction. If and only if the root transaction is ready to commit (i.e. all of its nested sub-transactions have committed locally) the sub-transactions can commit finally and durability of the whole transaction is guaranteed.
If a transaction inside the transaction tree aborts, all its nested sub-transactions are aborted too. Nested transactions allow the parallel processing of sub-transactions on different machines. Additionally if an error occurs in one of the sub-transactions only this sub-transaction (and its nested sub-transactions) have to be aborted. The parent transaction then decides about the error handling. Error handling can for example consist of ignoring the failed sub-transaction, a retry of the sub-transaction (maybe with different parameters) or the abort of the parent transaction.
This property of nested transactions is of particular interest for transactions in business processes, since the failure of a part of the business process does not automatically imply that the whole process is reverted and possibly a lot of work is lost.

2.1.1.2 Open Nested Transactions

Open nested transactions [48] are similar to closed nested transactions but don't guarantee isolation. In closed nested transactions on (local) commit of a sub-transaction, its results are made visible only to its parent transactions. This means, only a moderate speed-up of the whole system is possible that results from the concurrent execution of sub-transactions of the same parent transaction that use different resources. Other transactions outside of the nested transaction have to wait for access to resources, until the whole nested transaction has committed or

aborted. The concept of open nested transactions addresses this "closed" isolation of closed nested transactions by making the results of a sub-transaction visible to the outside as soon as a sub-transaction commits locally.
The advantage of this behavior is that other nested transactions can use a resource that has been committed locally by a sub-transaction of another nested transaction that has not yet finally committed. However, the disadvantage is that sub-scopes lose their isolation property. The result is that other transaction see data that can be dirty. Additionally, when the nested transaction fails, the work performed and committed by sub-transactions must be reverted by transaction specific functionality. This functionality is also called *compensation*.

2.1.2 Sagas

The concept of *Sagas* [26] [25] provides a means to support long-lived transactions (LLTs) and compensation. While open nested transactions generally assume that compensation actions are scheduled manually, sagas allow the definition of a compensation-transaction for each sub-transaction. A saga is a top-level transaction that contains one level of sub-transactions. Similar to open nested transactions resources are freed upon commit of a sub-transaction. The saga concept assumes that only such sub-transactions are used in sagas whose results can be made visible before the end of the whole transaction.
On abort of a running saga the compensation-transactions are executed for all sub-transactions that have completed before the need to abort occurred. The compensation-transactions are executed in reverse order as their sub-transactions have completed.
In the case of a system crash, this sort of *backward-recovery* is not tolerable for long-running business-processes where the work of weeks may be reverted during such a compensation. Therefore sagas can be combined with persistent save points. Sagas with save points provide *forward-recovery*. After each sub-transaction has finished its results are saved to persistent store. In the event of a crash and a following recovery of the system, the system restores the latest persistent save point and can continue work from there.

2.2 Transactional Behavior in BPEL Processes

The business process execution language BPEL offers a way to describe business processes using the web service stack. In order to describe the work to be performed by a process, BPEL uses two different kinds of activities, *basic* and *structured* activities.
Basic activities are atomic units of work such as the assignment of data to a variable via an `assign` activity, the invocation of an external partner via an `invoke` activity or the receipt of requests from an external partner via a `receive` activity.

Structured activities allow a recursive composition of activities. A structured activity can group one or more basic or structured activities into a new activity that in turn can be used by another structured activity to compose a new activity. Examples for structured activities are the `sequence` activities which specifies that all nested activities have to be executed in sequential order or the `flow` activity which allows to specify an arbitrary control flow for its nested activities by specifying `links` that link two activities. Other structured activities are the `while` activity which allows the specification of a while loop, or the `switch` activity which provides conditional behavior [18].

2.2.1 Scopes

A special example of an activity is a `scope` activity. A `scope` activity defines a behavior context for its nested activities.
Scopes closely resemble blocks of code in traditional programming languages such as the `begin-end` construct in Ada or the `try-catch` construct in Java. Scopes allow the definition of variables that are only visible for nested activities of the scope. Similar to a `catch` block in Java, BPEL allows the definition of a fault-handler for a scope that catches faults that occurred in nested activities of the scope.
BPEL processes typically control long-running business interactions. As motivated in Section 2.1, fault handling in BPEL therefore employs a compensation based transaction mechanism called long-running transactions (LRT).
LRTs are based on the concept of open nested transactions and sagas. From a transaction point of view a BPEL process instance can be seen as a nested business-transaction that can contain several sub-transactions. These sub-transactions inside the BPEL process are represented by `scope` activities. Scopes in turn can be parent transactions themselves, containing other sub-transactions, since scopes can be arbitrarily nested. The nesting of scopes results in a "transaction-tree" of arbitrary depth, similar to that of nested transactions.
In BPEL the outcome of the root transaction (i.e. the overall BPEL process instance) is based on the outcome of the nested transactions (i.e. the `scope` activities directly nested inside the `process` element). The outcome of the nested transactions (i.e. `scope` activities) in turn is dependent on the outcome of their nested scopes. Since the transaction model is compensation based a means to compensate sub-transactions as requested in the Saga model is needed. BPEL allows to define a flexible compensation behavior for every scope via a so-called *compensation handler*.

A scope must have one nested activity that in turn can be another structured activity (i.e. a `flow`, `sequence` or another `scope` activity). The process itself is a special case of a scope.
Additionally BPEL specifies implicit scopes. Implicit scopes are for example

`invoke` activities with specified fault and/or compensation handlers. In the rest of the document the term "scope" is used to identify all three types of scopes, explicit scopes, the process and implicit scopes. If a differentiation is needed it is explicitly mentioned.
Additionally, scopes allow the definition of *event handlers* and *correlation sets*. Event handlers allow the concurrent processing of events while the nested activities of a scope execute. In comparison to fault and compensation handlers which are considered error handling constructs, event handlers are considered part of the normal processing of a scope.
Correlation sets are a means to correlate incoming messages for a process with a particular process instance based on data in the incoming message identified by *property aliases*. [18]

2.2.1.1 Life Cycle of a Scope

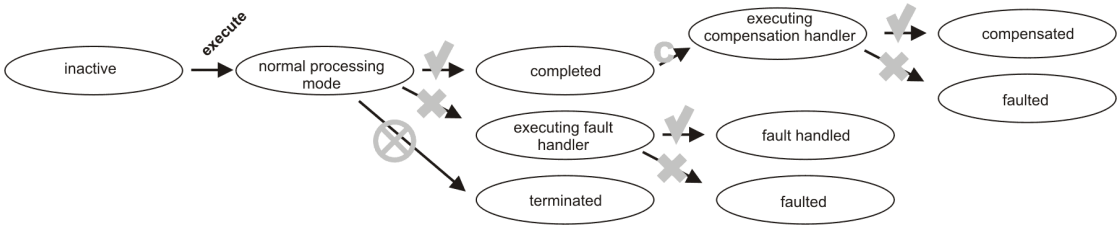

Figure 2.1: Life cycle of a scope.

Figure 2.1 shows the life cycle of a BPEL scope. The execution of a scope is started when its join condition evaluates to true. Then it's nested activity, fault handlers and event handlers are started. In the rest of the document this state will be called *normal processing mode*.
On completion of its nested activity the scope has *completed*. All fault handlers and event handlers are stopped. If an event handler is still executing, the scope only completes after this event handler completes.
Having completed, a snapshot of the scope's variables is recorded. In the case the enclosing scope decides to compensate the scope later, this snapshot is used to perform compensation. A scope can only be compensated by its directly enclosing scope. This also implies that a scope can only compensate its directly enclosed scopes. When a scope needs to compensate its compensation handler is executed. The compensation can either complete normally then the scope is considered *compensated*. A compensation handler can also fault then the scope itself has *faulted*. The fault that occurred in the compensation handler is rethrown to the enclosing scope.
A scope is executing a fault handler after its normal processing mode faulted.
The fault handler for a scope again has two possible outcomes. Either it completes, then the fault has been handled and the control flow continues as if the scope had completed successfully, or the fault handler faults, then the fault that occurred in

the fault handler is rethrown to the enclosing scope. There is, however, an important difference between a scope that completed normally and one that faulted and its fault has been handled by a fault handler. A scope that once faulted can not be compensated.
Another possibility is that a scope in normal processing mode is terminated, i.e. it must stop the processing of all nested activities.

2.2.2 Fault Handling

When a nested activity of a scope faults, a fault handler is executed for this scope. The BPEL specification specifies three types of fault handlers:

- A *normal* fault handler that catches a specified fault. A fault is specified by its name and an optional variable.
- A *catchAll* fault handler that catches all faults that are not caught by one of the above handlers.
- An *implicit default* fault handler. This handler is executed if no normal or catchAll fault handler is specified for a scope.

A scope can have zero or more normal fault handlers plus zero or one catchAll fault handlers.
All three types of fault handlers have some common behavioral aspects. On execution of a fault handler it first terminates all nested activities of the scope the fault occurred in. This is common to all three types of fault handlers. Terminating nested scopes is done by sending a `forcedTermination` fault to these scopes.
Having done so the actual execution of the fault handler begins. The normal fault handler as well as the catchAll fault handler both include one main activity that can be a structured activity that again can contain arbitrary activities. The behavior of these two handlers is therefore specified by the designer of the process.
The behavior of the implicit fault handler, however, cannot be defined by the designer of the process, it is predefined by the BPEL specification.
If a fault handler ends normally, the scope ends and the control flow of the process instance continues as if the scope had completed normally.
A fault that occurs in a fault handler is rethrown to the next enclosing scope of the scope the fault handler belongs to, this ensures that the fault is handled eventually by a fault handler in one of the enclosing scopes or the whole process fails.
If a fault occurs in a scope that is not handled by one of the normal fault handlers and no catchAll fault handler is defined, the implicit default fault handler is installed. Having terminated all nested active activities it compensates all nested scopes of the scope the fault occurred in. After it has compensated all nested scopes it throws the fault to the scope that encloses the scope in which the fault originally occurred. The rethrowing of the fault at the end of an implicit fault handler is the same behavior as if an explicit fault handler had failed.

Fault handlers are installed if a scope enters its normal processing mode. They are uninstalled when the normal processing mode completes or faults. Only one fault handler can be executed for a scope.

2.2.3 Compensation

The BPEL specification specifies two types of compensation handlers:

- A normal compensation handler that is specified by the designer of the process.
- An implicit default compensation handler whose behavior is specified by the BPEL specification. This one is called if the scope is to be compensated and no normal compensation handler is specified for this scope.

A scope can have zero or one normal compensation handlers.
A compensation operates on the data recorded on completion of the scope it belongs to. Compensation handlers are invoked by either one of two means:

- A `compensate` activity in a fault handler or compensation handler of the enclosing scope. `Compensate` activities are allowed only in fault handlers or compensation handlers.
- The implicit default fault handler of the enclosing scope.

`Compensate` activities have an optional `scope` attribute that is used to specify a scope name. The scope with this name is compensated on execution of the `compensate` activity. This behavior is called *named compensate*. Omitting the `scope` attribute means that the compensation handlers of all nested scopes are executed.

2.3 WS-Coordination

Web-Service Coordination (WS-C) provides a framework to coordinate Web-Services. WS-Coordination identifies two main building blocks:

- *Activities* and
- *Participants*

An activity ties together several web-services into a (distributed) application.
A participant is a web-service that contributes to an activity.
All participants of an activity must agree on an outcome of the activity. Therefore a *coordinator* which mediates the agreement on the outcome is needed. Participants and coordinator exchange messages specified in a *coordination protocol* in order to agree on an outcome of the activity.
The framework specified in the WS-Coordination specification [13] provides three

services, an activation service where applications can request the creation of new activity, and an registration service, where applications can register for an activity. The third service is coordination-protocol specific and has to be defined in different specifications. Examples for these specifications are WS-Business Activity [11] and WS-Atomic Transaction [10].
This architecture allows the plug-in of different coordination protocols, such as the compensation based WS-BA protocol for long-running transactions specified in WS-BA or the ACID [30] based one for 2 phase-commit transactions specified in WS-Atomic Transaction.
In order to allow applications to participate in an activity, the activity has first to be created. During creation of an activity a coordination context is created. An activity is created by sending a `CreateCoordinationContext` message to the WS-C activation service. The activation service then returns a `CreateCoordinationContextResponse` message which contains the coordination context for this activity. The coordination context contains a unique identifier for the activity, as well as an endpoint reference where applications that want to participate in the activity can register with a `Register` request. The coordination context for an activity, therefore has to be forwarded to applications that should participate in an activity.
Having received a `RegisterResponse` message from the registration service an application now knows that it has registered with the desired activity and now can send and receive coordination protocol specific messages to and from the coordinator.

2.4 WS-Business Activity

Web-Service Business Activity (WS-BA) builds upon the coordination framework provided by WS-Coordination.
WS-BA is a coordination protocol to coordinate long-running transactions that require a compensation based coordination protocol.
WS-BA offers two different completion protocols, *participant completion* and *coordinator completion*.
Participant completion allows the participants of the coordination to tell the coordinator that they have completed or faulted. Participants need to know when they have completed all necessary work for a business activity.
Coordinator completion is used with participants that rely on a notification from the coordinator to tell them when they have completed all necessary work required for a business activity.
WS-BA specifies two coordination types, *atomic outcome* (`AtomicOutcome`) and *mixed outcome* (`MixedOutcome`). An application has to specify one of the two options on activation of an activity with the WS-C coordinator.
Using atomic outcome, a coordinator for an activity must either direct all partici-

pants to close or all participants to compensate whereas a coordinator for a mixed outcome activity can close or compensate every participant individually.
WS-BA also specifies the messages exchanged between coordinator and participant for one of the two completion protocols during the coordination. These messages are discussed in more detail in Section 4.4.2 where a mapping of BPEL-scopes to WS-BA transactions is discussed.

2.5 ActiveBPEL Engine

The ActiveBPEL engine is a runtime environment for BPEL business processes. Version 1.2 of the ActiveBPEL engine is used to implement the prototype which is part of this thesis. Its code is open source under the GNU General Public License [23].
This section covers in a short form some of the basics of the ActiveBPEL engine needed for the understanding of the prototype implementation. For a more detailed description see the ActiveBPEL architecture overview [2] and [50] or [36].

2.5.1 Architecture

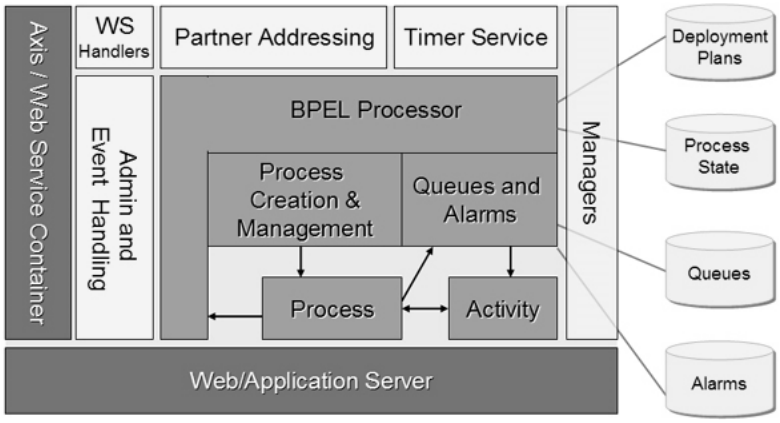

Figure 2.2: ActiveBPEL engine architecture.

Figure 2.2 taken from the ActiveBPEL architecture document [2] shows the main parts of the ActiveBPEL engine.
The Active BPEL engine runs inside an application server or servlet container such as Apache Tomcat. It has been tested with Apache Tomcat 5.x [9]. The Active BPEL engine provides a servlet [43] class, the `AeProcessEngineServlet` that is deployed as a web-application inside the servlet container. It extends the generic `AxisServlet` class which in turn is a subclass of the `GenericServlet` class that implements the `Servlet` interface. The `AeProcessEngineServlet` implements the

`init()` method of the `Servlet` interface that is called upon startup of a servlet [43]. In this method the `AeProcessEngineServlet` starts up the ActiveBPEL engine as well as the Apache Axis web service container. Having been started the engine registers the business processes that have been deployed with the engine with the web service container, making the business process interfaces available to the outside.
The ActiveBPEL engine uses Apache Axis [7] as a web service container. Axis enables the ActiveBPEL engine to send and receive SOAP messages [45] over HTTP. Axis allows the definition of custom handlers that take over the processing of SOAP messages after they have been received and deserialized by the Axis layer. The ActiveBPEL engine registers two handlers `AeDocumentHandler` and `AeRPCHandler` with Axis. These handlers take care of document and RPC style SOAP messages and forward them to the business process engine.

2.5.1.1 Request Processing

Upon receipt of a SOAP message via HTTP through the Axis web service container, Axis calls the `invoke()` method of the handler for SOAP RPC or SOAP document style messages dependent on the nature of the message. The handler classes then invoke the process engine represented by an instance of the class `AeBusinessProcessEngine`, which queues the received message.
Upon dequeuing of a received message the process engine determines using the `QueueManager` class if a `receive` activity or `onMessage` event is waiting for this message. If yes, it hands the message to the responsible business process instance represented by an instance of the `AeBusinessProcess` class. The responsible business process instance is determined via the correlation sets defined for the deployed business processes and the value of the correlation properties inside the received message (for information on correlation sets in BPEL see [18]).
If no `receive` or `onMessage` is waiting for the message and one of the deployed business processes has a `receive` activity or `onMessage` event inside a `pick` activity that correspond to the message and is marked with `createInstance=yes` a new process instance for this process is created and the message is handed over to this process instance.

2.5.2 Activities

Each activity of a BPEL process is represented in the ActiveBPEL engine by a definition object that extends the `AeBaseDef` class. Definition objects are created on deployment, they represent the definition of an activity as specified in the BPEL XML process definition. Implementation objects represent “instances”

of these definitions. On creation of a process instance an implementation object is instantiated for each BPEL object in the process. Activity implementation classes extend the `AeActivityImpl` class and therefore implement the `IAeExecutableBpelObject` interface.
The `IAeExecutableBpelObject` interface provides methods relevant for the life cycle of an activity such as the `execute` method. The `execute()` method is called when the process engine dequeues an activity from the execution queue and wants to execute it.
Upon dequeuing an activity from the execution queue it is in state `QUEUED_BY_PARENT` and its incoming join condition is checked. If its join condition evaluates to true the activity transitions to state `READY_TO_EXECUTE`. The queue manager then calls the `execute()` method of the queued activity, which puts the activity in state `EXECUTING`. Depending on the definition of an activity the work the activity is designed to do is performed in the `execute()` method. Work to do might be to wait for a deadline in the case of a BPEL `wait` activity, or the invocation of a partner in the case of an `invoke` activity.
Only the classes `AeActivityScopeImpl` and `AeActivityInvokeImpl` which represent BPEL scopes and BPEL invokes are described here in more detail since they have an impact on the prototype to be realized in this thesis.

2.5.2.1 Scopes

Scopes are represented in the ActiveBPEL engine by the `AeActivityScopeImpl` class.
The `AeActivityScopeImpl` class has methods to get and set the fault and compensation handlers for the scope. On call of the `execute()` method the `AeActivityScopeImpl` instance sets its child activity and event handlers into the state `readyToExecute` and puts them in the execution queue. When the child activity completes normally, the `childComplete` method of the `AeActivityScopeImpl` instance is called which then terminates its event handlers and fault handlers and creates an `AeCompInfo` object instance containing a snapshot of all scope variables.
The `AeCompInfo` object is needed for a possible compensation of the scope later. The `AeActivityScopeImpl` instance then notifies its `AeBusinessProcess` instance that it has completed which in turn notifies the `AeActivityImpl` subclass instance that represents the parent activity of the scope that the scope has completed.
The BPEL `process` element is represented in the ActiveBPEL engine by the `AeBusinessProcess` class that extends the `AeActivityScopeImpl` class since the process is a special case of a scope.

Fault Handling On faulting of the nested activity of the scope or an event handler the scope's `childCompleteWithFault` method is called. The scope then exe-

cutes its `findOrCreateMatchingFaultHandler` method which puts the matching fault handler for the fault that occurred in the execution queue. If no matching fault handler is found an implicit default fault handler object of the class `AeImplicitFaultHandler` is created and put into the execution queue.
In the case a fault handler finishes it calls the `faultHandlerComplete` method of its `AeActivityScopeImpl` instance which in turn can now signalize its `AeBusinessProcess` instance that it has finished execution. The `AeActivityScopeImpl's normalCompletion` flag is set to `false` indicating that it has completed with a fault.
On faulting of a fault handler the `AeFaultHandler` instance that represents the fault handler calls the `faultHandlerCompleteWithFault` method of the `AeActivityScopeImpl` instance which in turn calls the `objectCompletedWithFault` method of its `AeBusinessProcess` instance to notify it that the scope has faulted.

Compensation Compensation in the active BPEL engine is initiated by instances of the `AeActivityCompensate` class. The `AeActivityCompensate` class represents the BPEL `compensate` activity. The implicit default fault handler as well as the implicit default compensation handler contain one `AeActivityCompensate` class instance as nested activity. When an `AeActivityCompensate` instance's `execute` method is called this method puts the compensation handlers of all directly nested scopes that need to be compensated into the execution queue. In order to obtain the compensation handler it calls the `getCompensationHandler` method of the `AeActivityScopeImpl` instance that represents the nested scope to be compensated. If a compensation handler is specified in the definition of the scope an `AeCompensationHandler` instance is returned. The instance contains child activity instances as defined in the XML of the BPEL process.
In the case when no compensation handler is specified a new instance of the `AeImplicitCompensationHandler` class representing the implicit compensation handler is created and returned. This `AeImplicitCompensationHandler` object contains one child activity which is a `AeCompensateActivityImpl` object.
Compensation handlers operate on the snapshots of a scope created on completion of a scope. Therefore the `AeCompInfo` objects representing the compensation info including a snapshot of its variables on completion time are set for a compensation handler using its `setCompInfo` method. Additionally the callback activity for a compensation handler is set. This is the activity that calls the compensation handler and that the compensation handler has to notify upon completion or faulting. If the compensation handler finishes it calls the `compensationCompleted` method of its callback activity. A faulting compensation handler calls the `compensationCompletedWithFault` method of its callback activity which in turn then marks the compensation or fault handler it belongs to as faulted.

2.5.2.2 Invoke Activities

Invoke activities are represented by an instance of the class `AeActivityInvokeImpl`. If the ActiveBPEL engine detects on deployment that an `<invoke>` activity is an implicit scope (i.e. it has fault and / or compensation handlers) it creates an `AeActivityScopeImpl` instance with an `AeActivityInvokeImpl` instance as child. This is compliant to the BPEL specification which identifies an `<invoke>` activity with defined fault and /or compensation handlers as being equal to a scope with the same handlers and a nested invoke activity without handlers.
If the `execute()` method of an `AeActivityInvokeImpl` instance is called it puts the message needed to invoke the external partner along with the `partnerLink` definition into the invocation queue of the `AeBusinessProcess` instance that contains the invoke. The invocation queue manager then dequeues the invocation requests and hands it over to an instance of the class `AeInvokeHandler` by calling its `handleInvoke` method. The `AeInvokeHandler` then checks the WSDL definition of the partner it has to invoke and builds a Java representation of the invocation SOAP message. This message is then delegated to the Axis layer which serializes the message to XML and sends it with the corresponding (HTTP) handler.

2.5.3 ActiveBPEL Process Deployment Descriptors

The ActiveBPEL engine uses process deployment descriptor files (PDD files) [3] to specify process specific properties. These properties include the specification of endpoint references for the partner links defined in a BPEL process and WSDL files relevant for the process. A complete description of PDD files can be found under [2]. PDD files along with the BPEL process definition file and all necessary WSDL files for partners and the process are packed together into a business process archive (BPR file). These archives are then deployed to the active BPEL engine. Here only the mechanisms to set the endpoint references for partner links are examined since they are relevant for the rest of the document.
The ActiveBPEL engine knows two main possibilities on how to specify the address of a web-service that should be invoked by an `invoke` activity. The first one is to dynamically assign a WS-Addressing endpoint reference [15] to a partner link during the execution of a BPEL process. This is called “dynamic” invocation.
The second one is to set the endpoint reference for a partner link on deployment in the process deployment descriptor file. This so-called “static” invocation allows to define a WS-Addressing endpoint reference for the partner link in the process deployment descriptor. In the version of ActiveBPEL used in the prototype for this thesis the actual address for the external partner is not taken from the `wsa:Address` part of the endpoint reference but from the `soap:address` element of the corresponding binding in the associated WSDL file [4]. This behavior only happens with “static” invocation.
Additionally, “invoker” and “principal” invocation can be specified. Invoker specifies that a message should be sent back to the requester in a synchronous request-

response scenario. The endpoint reference of the requester is taken from the WS-Addressing `ReplyTo` SOAP header of the request message. Principal invocation uses the endpoint reference from a partner definition deployment descriptor based on an authenticated principal.

Not only the endpoint references for external partners must be defined, but also for the services that are provided by a BPEL process. Therefore it is required to specify a service name, for every service provided by a BPEL process, in the process deployment descriptor. This service is deployed to the Axis layer upon deployment of the process and a corresponding WSDL file is created. The service can then be reached via HTTP by its name. If for example the ActiveBPEL engine is installed on `www.example.com` under the directory `active-bpel` the service becomes available under `http://www.example.com/active-bpel/services/serviceName`.

2.6 Summary

Chapter 2 gives an overview over the transaction models that are the basis for the long-running transactions (LRTs) employed in WS-BPEL.
The basic building blocks for these LRTs - `scope` activities have been identified and the life cycle of a scope including fault and compensation handling has been discussed.
Some basics of the ActiveBPEL engine, mainly the internal representation of `scope` and `invoke` activities have been discussed as well as a short overview over the deployment routine of the ActiveBPEL engine.

3 Motivational Example

As motivated in the introduction, the extraction of transaction information from a BPEL process has two main reasons.

- Inclusion of external partners in the BPEL long-running transactions currently internal to a business process.
- Making the long-running transactions inside a BPEL process controllable from the outside and therefore have BPEL processes participate in transactions initiated by other parties.

3.1 Bike Service Example

Figure 3.1 shows a simple BPEL process that is used by "CuBi", a fictional manufacturer of custom bicycles. Upon invocation of the bike service process, it orders a frame for the bike from the frame service as well as other components, such as wheels, gears and brakes needed to assemble the bike, from different component services. Additionally, the timetable of the mechanic who should assemble the bike is updated. All actions are performed concurrently.

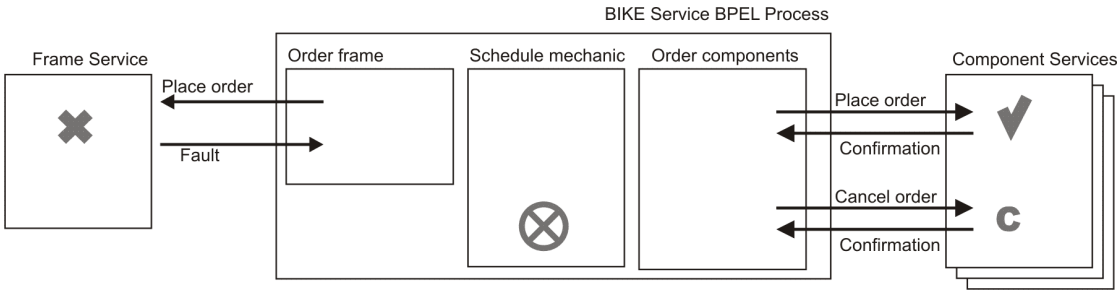

Figure 3.1: Bike service example.

In Figure 3.1 a situation is shown where the components have been ordered successfully. During the invocation of the frame ordering service a fault occurred in this service. The bike service now terminates the scheduling of the assembly mechanic, plus the fault handler is designed to revert the successful order of the components. Therefore it normally must invoke the "compensate" operations of the external partners to revoke the order. This, however, requires that the names

and parameters of the compensate operations of the external partners are known to the invoking bike service.

Using the same example with an external coordinator would lead to the following situation. On start of the bike service process it creates an activity and a coordination context with the external coordinator. Then the process registers as a participant for this activity. Upon starting of sub-scopes (such as the scheduling of the assembly mechanic) these scopes register as participants of the process activity.
When external partners are invoked, they get passed the coordination context, telling them to register with the process activity.
The transaction tree for the situation described above would look as depicted in Figure 3.2 The frame service and component services as well as the work schedul-

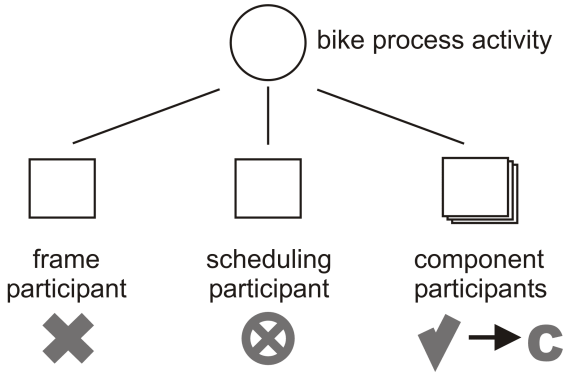

Figure 3.2: Transaction tree of the bike service example.

ing service have registered as participants with the bike process transaction. For the external coordinator it does not make a difference whether participants are external partners or are internal parts of the process.
When the frame service reports a fault, the external coordinator must emulate BPEL specific behavior. It must cancel all active participants (i.e. the scheduling participant), then the BPEL fault handling can begin. As described above the fault handler for the bike service process compensates the completed participants (i.e. the component participants). This is simply done by telling the participants, that have completed, to compensate.
It becomes clear that some sort of protocol is needed that specifies the messages between the participants and the coordinator. Additionally, a means inside a BPEL process is needed to identify parts of the process that should become transaction participants. WS-Coordination provides a coordination framework that allows the creating of transactions and the registration and coordination of participants. WS-Business Activity provides completion protocols that allow the coordination of long-running transactions.
As discussed in Section 2.2 `scope` activities are blocks inside a BPEL process that have transactional behavior. In order to have external partners participate in BPEL long-running transactions scopes have to be coordinated externally too. The

task of this thesis is therefore to extract transaction specific information from a BPEL process and coordinate BPEL long-running transactions using WS-Business Activity.
The need for a common completion protocol is clarified if the example is expanded. The Bike service process instance now determines its suppliers for components dynamically on runtime using a registry like UDDI [37] [47]. Using a completion protocol like WS-Business Activity it can simply use WS-Policy [19] to specify that it searches for external partners that can participate in WS-Business Activity transactions. Having found such partners, the process can be assured that the invocation of these external partners can be canceled or compensated using messages specified in WS-Business Activity. Additionally the BPEL process itself can participate in transactions initiated by external partners. Expanding the example above, Mail-A-Bike an online bike shop allows the configuration and ordering of bikes online. A user might order bikes and bike accessories in a Web-based online shop. Upon placing an order the bike shop starts a new transaction and invokes its external suppliers under this transaction. The bike process from above gets invoked that way too. In the situation discussed above, it would report a fault to the coordinator of the bike shop transaction which then for example can cancel the whole order transaction.

What is needed to have BPEL processes participate in externally initialized transactions and make external partners participate in BPEL long-running transactions is identified in the next chapter.

3.2 Summary

In Chapter 3 a small example illustrating the need for an external coordination of BPEL LRTs is given. The example shows that including external partners in BPEL internal LRTs has benefits. The process designer does no longer need to know the semantics and operation names of transaction specific functions (such as compensation and cancel functions) of external partners but can simply use operations provided by a common completion protocol such as WS-Business Activity.

4 Extraction of WS-Business Activity from BPEL

Having given a motivation in Chapter 3 why it is beneficial to coordinate BPEL long-running transactions externally, this chapter describes the actions to be taken to make transactions in a BPEL process controllable from the outside as proposed in [38] and Appendix C of the BPEL specification [18].

Delegating transaction control to an external coordinator implies that all transaction-related work lies no longer in the responsibility of the BPEL engine but is handed over to the external coordinator through the exchange of messages.

As discussed in Section 2.2 scopes are transaction relevant parts of a BPEL process. In order to manage the local LRTs inside a BPEL process externally, the scope life cycles of a process need to be controlled by the external coordinator. Considering these requirements the external coordinator has to be aware of certain events that occur in the life cycle of a scope, such as the beginning and completion of the normal processing mode of a scope or the faulting of a scope. Additionally the external coordinator must be able to invoke life cycle events, such as the notification that a fault occurred in a nested scope or that a scope has registered with the external coordinator.

4.1 Transaction Middleware

In contrast to BPEL engines that process the BPEL transaction and fault handling internally, a scenario in which the BPEL long-running transactions are controlled externally needs additional components besides the BPEL engine. Figure 4.1 shows the additional components that the external transaction processing requires over the internal transaction processing. In addition to the modified BPEL engine that allows external transaction processing, another component is needed to perform the external transaction processing. From now on the term *external coordinator* is used to identify this component. The modified BPEL engine will be called *PureBPE* from now on. It is a BPEL engine that only copes with the *pure* execution of BPEL processes and delegates functions relevant to the processing of

BPEL LRTs to the external coordinator. Figure 4.1 also shows that messages between the PureBPE and the external coordinator are exchanged to allow external coordination of transaction-relevant parts (i.e. scopes) within a BPEL process. Since these messages are used to control the life cycle of BPEL scopes they are called *life cycle messages* in the further document.
In Figure 4.1, two separate modules are shown that form the external coordinator. The first one is a coordinator which translates life cycle messages from the BPEL engine into messages of the employed external transaction protocol, named *BPEL4Tx coordinator*. The second coordinator is the coordinator for the external transaction protocol that should be used to control the BPEL transactions. In this case WS-C with WS-BA is used. Consequently the second coordinator is named *WS-C/WS-BA coordinator*.

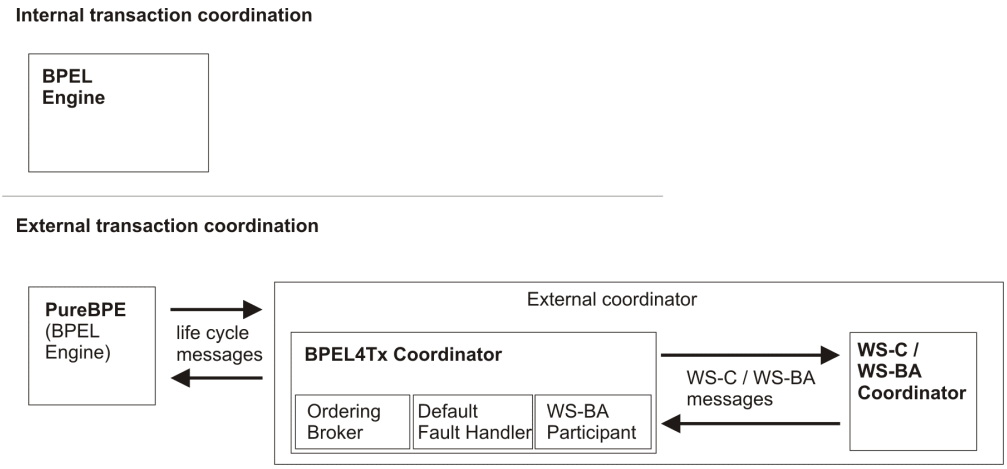

Figure 4.1: Architecture of a BPEL engine with internal / external transaction coordination.

4.1.1 Coordinators

Using two coordinators instead of only one, has the advantage that different coordination protocols can be used to control BPEL LRTs externally. The BPEL engine only sends and receives generic life cycle messages that are translated to the specific transaction protocol by the BPEL4Tx coordinator.
The cost of using two different coordinators instead of only one is the overhead generated by the translation of the messages from the format the PureBPE sends to the BPEL4Tx coordinator to the format the WS-C/WS-BA coordinator understands. Additionally the amount of exchanged messages doubles because every message that is sent directly from the PureBPE to the WS-C/WS-BA coordinator in the one-coordinator architecture, is split into a series of two messages, one from PureBPE to Bpel4TX coordinator and one in translated form from the BPEL4Tx coordinator to the WS-C/WS-BA coordinator.

4.1.1.1 BPEL4Tx Coordinator

The BPEL4Tx coordinator itself is split into three main modules. The *ordering broker* receives the generic messages from the PureBPE and invokes the other modules (*WS-BA Default Fault Handler* or the *WS-BA Participant*) if needed. The two latter modules are transaction protocol specific and can be exchanged for other modules implementing different transaction protocols such as WS-Atomic Transaction [10].

4.2 Actions to Be Controlled by the External Coordinator

In order to control and manage the life cycle of scopes externally, several actions influencing the life cycle of a scope must be performed externally. In Section 2.2.1.1 the events and handlers that influence the life cycle of a scope and therefore its final state are identified. The final state of a scope depends on the outcome of its normal processing mode as well as the outcome of its fault and compensation handlers. Additionally, the outcome depends on the outcome of its enclosing and enclosed scopes. its enclosing scope can request the scope to be compensated or terminated. its enclosed scopes can fault causing the nested activity and therefore the normal processing mode of the scope to fault. In turn the process itself can control the life cycle of its enclosed and enclosing scopes. It can request its enclosed scopes to terminate or compensate and can cause its enclosing scope to fault.
Considering these influences a set of events in a scope's life cycle that are relevant for the outcome of a scope and therefore for the whole process instance can be identified. Additionally a set of life cycle messages to and from the PureBPE is needed that notifies the external coordinator or the PureBPE of a change of a scope's state. Figure 4.2 is based on Figure 2.1 which shows the life cycle of a scope. Figure 4.2 shows at which points in the scope life cycle the external coordinator needs to be notified and in which situations a notification from the external coordinator is needed. In addition to the messages identified in Figure 4.2 some more messages are needed. All life cycle messages and their mapping to WS-Business Activity messages are explained in detail in the next sections.

4.2.1 Events that Trigger Outgoing Scope Life Cycle Messages

Table 4.1 describes the various events in the life cycle of a scope that are of relevance for the outcome of a scope. It also shows the messages that are sent from the PureBPE to the BPEL4Tx coordinator to notify the coordinator of the respective life cycle event inside the PureBPE.

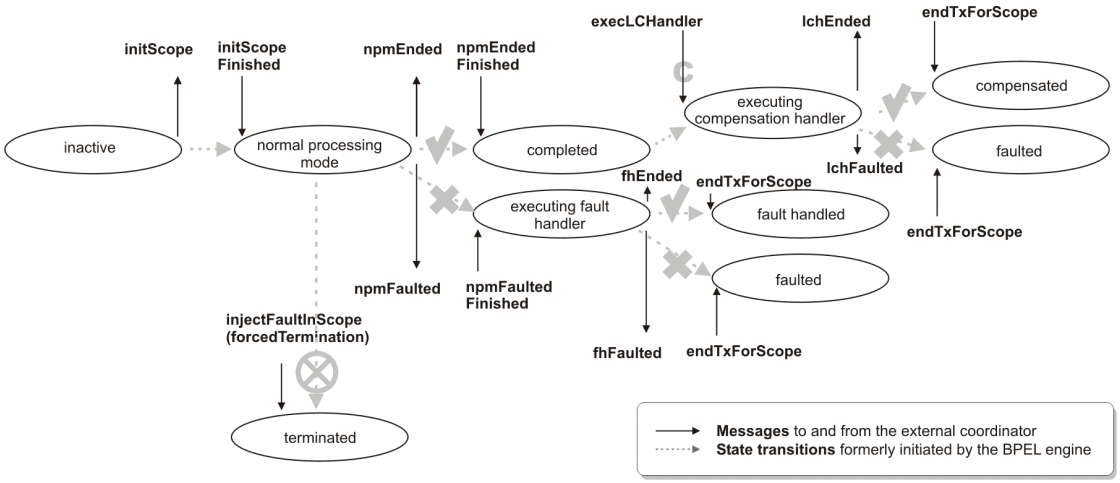

Figure 4.2: Scope life cycle messages.

Event	Description	Message
Initialization	The scope enters the state where it is ready to be executed by the BPEL engine.	initScope
NPM Ended	The scope has finished normal processing mode without throwing a fault.	npmEnded
NPM Faulted	The scope has finished normal processing mode with a fault.	npmFaulted
ExecDefFH	An implicit default fault handler has to be executed on this scope.	execDefFH
FH Ended	A fault handler has finished for this scope without throwing a fault..	fhEnded
FH Faulted	A fault handler has finished for this scope with a fault.	fhFaulted
ExecActivity	An activity that should be handled by the coordinator is ready to execute for the scope. In the case of Bpel 1.1 this is only the `compensate` activity.	execActivity
LCH Ended	A life cycle handler (such as a compensation handler) completed without throwing a fault	lchEnded
LCH Faulted	A life cycle handler (such as a compensation handler) completed with a fault.	lchFaulted

Table 4.1: Outgoing scope life cycle messages.

4.2.2 Overview over Incoming Scope Life Cycle Messages

Table 4.2 shows events in the BPEL4Tx Coordinator that need to be communicated to the PureBPE in order for the PureBPE to be able to continue the processing of the scope.
Tables 4.2 and 4.1 give an overview of the various life cycle events inside the transaction relevant BPEL activities. The following sections describe in more detail

which and why messages are exchanged for which purpose and how the external transaction coordination is performed.

Event	Description	Message
Initialization complete	The scope has been registered with the coordinator.	initScopeFinished
NPM Ended complete	The coordinator has been notified that the scope has finished normal processing without a fault.	npmEndedFinished
NPM Faulted complete	The coordinator has been notified that the scope has finished normal processing with a fault.	npmFaultedFinished
FH Ended complete	The coordinator has been notified that the fault handler for a scope has finished without a fault.	npmEndedFinished
FH Faulted complete	The coordinator has been notified that the fault handler for a scope has finished with a fault.	fhFaultedFinished
ExecLCHandler	The coordinator needs to execute a life cycle handler (e.g. a compensation handler) on the scope.	execLCHandler
InjectFault InScope	The coordinator needs to inject a fault in a scope. This can be necessary if faults need to be thrown upwards in the scope hierarchy, or a scope needs to be terminated.	injectFaultInScope
DisableScope	The coordinator needs to stop the processing of outgoing Links of a scope.	disableScope
EndTxForScope	The transaction information of a scope is no longer needed.	endTxForScope

Table 4.2: Incoming scope life cycle messages.

4.3 Registration of Scopes for External Coordination

In order to coordinate scope life cycles externally, all scopes inside a BPEL process muss first register with the external coordinator. Omitting all activities besides the `process`, `scope` and implicit scopes (i.e. certain `invoke` activities), a BPEL process can be viewed as a tree of scopes.
The basic `process` activity contains zero or more scopes and implicit scopes. An explicit scope can itself contain zero or more other scopes and implicit scopes.

4.3.1 WS-C Activities and Participants

As mentioned in Section 2.3 WS-Coordination knows two main building blocks. Activities and participants. Participants register with an activity. The outcome of the activity is determined by the outcome of the participants. This is similar to BPEL where the final state of a scope is determined by the final state of its nested scopes.
This similarity allows the following mapping of the BPEL-Scope tree to WS-Coordination:

- Each scope which has child scopes is represented by an activity in the sense of WS-C. Implicit scopes are activities too, since they have external children.

- Each scope is represented by a WS-C participant. This participant has to register with the WS-C activity that represents its enclosing scope.

Figure 4.3 illustrates the mapping of the scope tree to a WS-Coordination transaction tree. Scopes with child scopes become WS-C activities. The invoke activity "invoke2_2_1" becomes an activity too, because it is an implicit scope with external children, namely the web-services that are invoked by it. The figure also shows that all scopes that are represented by an activity are also represented by a participant resulting in the need for a mechanism that correlates participants with activities.

4.3.1.1 The Root Activity

In order to allow BPEL processes to participate in transactions that have been initialized by other parties, the process itself registers as a participant with an activity. In cases where a BPEL process does not participate in externally initialized transactions, the external coordinator must explicitly start a *root activity* so that the process can register with this activity..

4.3.2 WS-BA Coordination Types

As discussed in Section 2.4 WS-BA specifies two possible coordination types for a WS-C activity, `AtomicOutcome` and `MixedOutcome`. Using `AtomicOutcome` the coordinator must direct all participants either to close or all participants to compensate. In `MixedOutcome` mode every participant can either be closed or compensated separately.

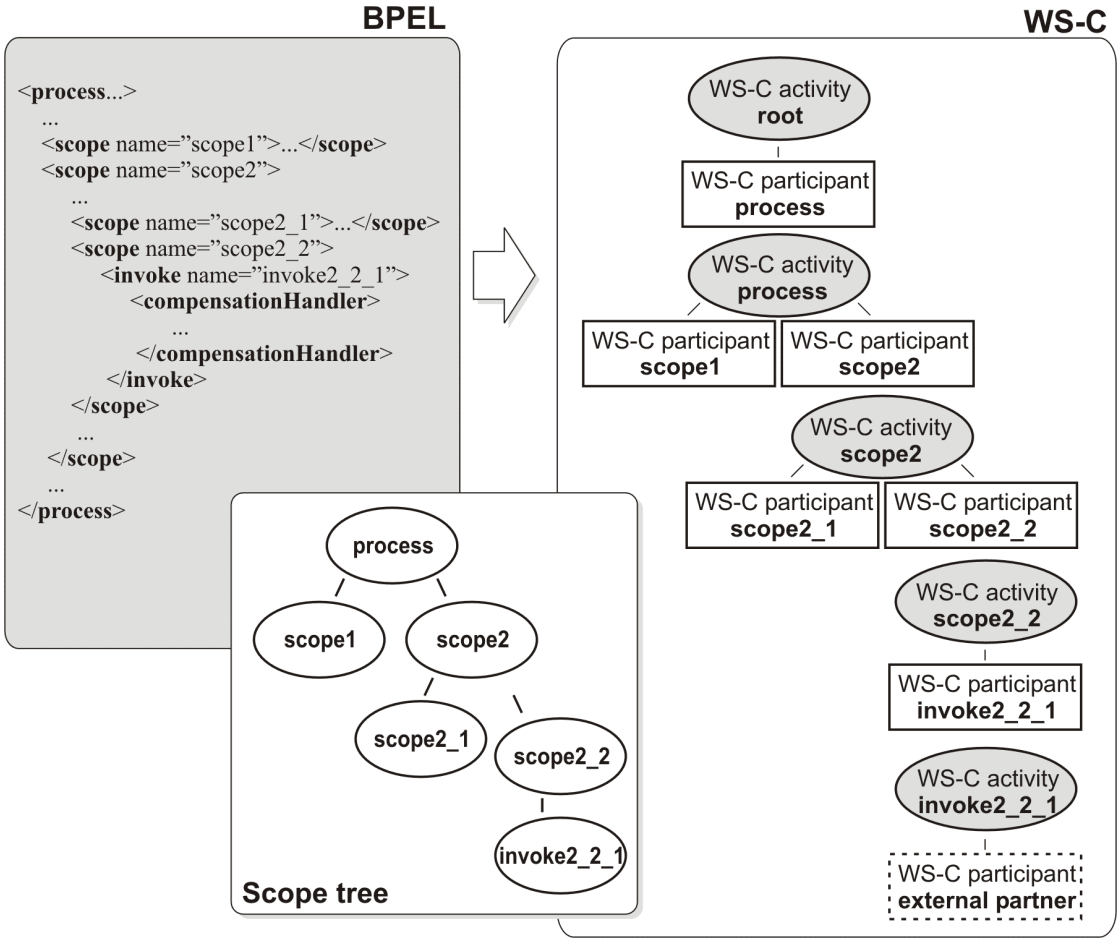

Figure 4.3: Mapping of BPEL scopes to WS-Coordination.

4.3.2.1 Atomic Outcome

The behavior of a scope in case of a fault or compensation is determined by the implicit default compensation and implicit default fault handler respectively in case no compensation handler and no fault handlers are specified. When one of the nested activities of a scope faults, all other nested scopes are terminated if they are still running or compensated if they have already completed. This is the atomic behavior of the implicit fault handler. If a scope has already completed without fault and has no explicit compensation handler, a compensation request for this scope is handled by the implicit default compensation handler. The implicit compensation handler compensates all nested scopes. No specified fault handlers therefore means that all nested scopes either complete or all nested scopes are undone. The behavior of the implicit fault and compensation handlers exactly corresponds to that of the `AtomicOutcome` coordination type specified in WS-Business Activity.

4.3.2.2 MixedOutcome

`MixedOutcome` can be achieved in BPEL by specifying one or more fault handlers and/or a compensation handler. Since fault or compensation handlers can contain `compensate` activities that specify which nested scope to compensate by using their `name` attribute, a scope can complete by only compensating a subset of its directly nested scopes. This is compliant to the `MixedOutcome` mode in WS-BA. If both, the fault and compensation handler are specified to have the behavior of their respective implicit counterparts, explicit fault and compensation handlers provide an atomic outcome mode too. This can be done by specifying handlers that compensate all nested scopes. But since handlers in BPEL can themselves contain arbitrary nested scopes and structured activities, the external coordinator must be able to evaluate the handlers to decide whether they compensate all nested scopes or not. Additionally branches inside fault or compensation handlers can be taken on data determined at runtime making a prediction at the time of the initialization of a scope impossible. Therefore the external coordinator can assume mixed outcome if one of both handlers is specified.

4.3.3 Consequences for the PureBPE

The PureBPE must be able to send an `initScope` message to the BPEL4Tx coordinator. Then it must wait until it receives an `initScopeFinished` message. On receipt of such a message it can install the fault and event handlers of the scope and can execute the immediately nested activity of the scope.

4.3.4 Consequences for the BPEL4Tx Coordinator

On receipt of an `initScope` message the BPEL4Tx coordinator must be able to evaluate the definition of this scope if it contains nested scopes. If yes, the BPEL4Tx coordinator must obtain a coordination context from the WS-C coordinator and register the scope as a new activity with the WS-C activation service. If no fault and compensation handlers are specified for the scope it is registered as a new activity with `AtomicOutcome`. Otherwise it is registered with `MixedOutcome` mode. Additionally, regardless of the presence of child scopes the BPEL4Tx coordinator must register the scope as a WS-C participant using the coordination context of the immediately enclosing scope. In case when no parent context is provided and the scope to be initialized is the process itself the coordinator must register a root context with WS-Coordination with which the process can register. After the registration of the scope the BPEL4Tx coordinator must respond with an `initScopeFinished` message that signalizes the PureBPE that it can continue the processing of the scope.

4.3.5 Message Exchange

Figure 4.4 shows the messages that are exchanged between PureBPE, BPEL4Tx coordinator and WS-Coordination in order to initialize a scope.

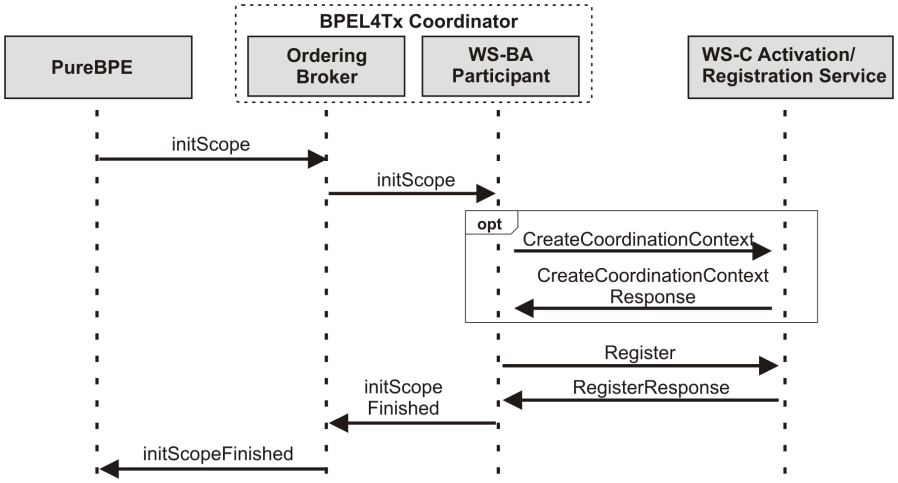

Figure 4.4: Message exchange for initialization of scopes.

4.3.5.1 Message Specifications

The messages exchanged during the initialization of a scope are specified as follows.

InitScope The `initScope` message contains a unique identifier for the scope (the scopeId), the name of the scope, the scopeId of the enclosing scope, the coordination context of the enclosing scope and the BPEL definition for the scope.

- The scopeId is needed in all messages between the PureBPE and BPEL4Tx. The scopeId is unique for every scope and is used to correlate messages with scopes.
- The scope name is needed for named compensates. Named compensation is explained in detail in the Section 2.2.3.
- The scopeId of the enclosing scope is needed for correlation information.
- The coordination context of the parent scope is needed for the registration with WS-Coordination, since this scope has to be registered under the coordination context of its enclosing scope.

- The BPEL definition of the scope is needed for the WS-BA participant module of the BPEL4Tx Coordinator. Instead of sending the whole BPEL definition of the scope, three boolean values indicating the presence of child scopes, compensation handler and fault handlers would be sufficient. However, other coordination protocols than WS-C and WS-BA might need different information, sending the whole BPEL definition is a more generic approach.

InitScopeFinished The `initScopeFinished` message contains the scopeId of the scope that has finished initialization and the coordination context if the scope has child scopes and has therefore requested a coordination context from WS-C.

CreateCoordinationContext The `CreateCoordinationContext` message is specified in the WS-Coordination specification [13]. It contains the following parameters

- The coordination type, an URI that specifies either mixed or atomic outcome. The value of the coordination type can either be the URI `http://schemas.xmlsoap.org/ws/2004/10/wsba/AtomicOutcome` or `http://schemas.xmlsoap.org/ws/2004/10/wsba/MixedOutcome`

- The current context. This is the context of the enclosing scope. The current context is needed by the coordinator if a tree of nested transactions is to be built.

CreateCoordinationContextResponse The WS-C coordinator responds to a `CreateCoordinationContext` request with a `CreateCoordinationContext-Response` message. This message is also specified in WS-C and contains the coordination context for the newly created activity.
The coordination context contains a unique identifier for the context as well an WS-Addressing [15] endpoint reference for the registration service that is to be used by participants that want to register with this activity.

Register The register message contains the completion type which is always `http://schemas.xmlsoap.org/ws/2004/10/wsba/ParticipantCompletion`. Participant completion specifies that the participants tell the coordinator when they have completed or faulted. The message also contains a participant service endpoint reference. This is were completion protocol specific messages should be sent. Additionally the scopeId and scope name are submitted with the message. The scopeId is needed to correlate participants and activities in case a scope is both a participant and an activity and the scope name is used to identify a scope in case of named compensates.

RegisterResponse This is the answer of the WS-C coordinator to a `register` message. It contains an endpoint reference for the WS-BA protocol service which is to be used to send subsequent WS-BA protocol messages to.

4.4 Mapping of WS-Business Activity Messages to BPEL

After a scope has registered as a WS-C participant with a WS-C activity, it is now ready to exchange protocol messages with the external WS-BA based coordinator.

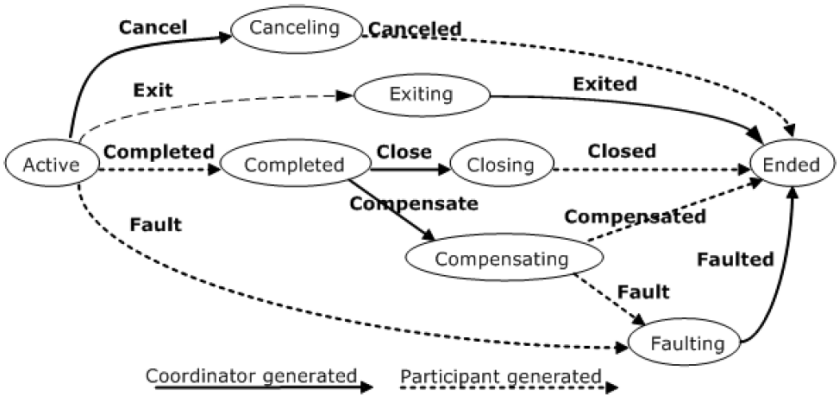

Figure 4.5: States and messages of the WS-BA business agreement with participant completion protocol.

Figure 4.5 shows the states and messages of the WS-BA *business agreement with participant completion* completion protocol. In Appendix C of the BPEL specification [18] it is shown how the *WS-Transaction business agreement* protocol can be used to model BPEL long-running transactions. The WS-Transaction [16] business agreement protocol is the predecessor to the WS-BA business agreement with participant completion protocol. However, the WS-BA business agreement with participant completion (*BAP*) protocol is not sufficient to map all events that occur in the life cycle of a BPEL scope.

4.4.1 Shortcomings of the Business Agreement With Participant Completion Protocol

Regarding the scope life cycle presented in Section 2.2.1.1 and the protocol depicted in Figure 4.5 the normal processing mode of a scope corresponds to the `Active` state of the BAP completion protocol. Having completed its normal processing mode the scope is in state `Completed` of the BAP protocol. This is signalized

through a `Completed` message.
Here the first shortcoming is identified. After signaling normal completion through a `Completed` message, an acknowledgment from the WS-BA coordinator is needed to prevent signal races. Without an acknowledgment a situation could arise where the enclosing scope of the scope that send the `Completed` message faults, but the coordinator has not seen the `Completed` message yet. The WS-BA coordinator tries to terminate the scope that completed instead of compensating it. The acknowledgement message (called `ACK` in [38]) signalizes that the coordinator has taken notice of the completion of a scope and that the enclosing scopes of the completed scope can execute the activities that follow the completed scope in the control flow. Before the `ACK` message is sent the scope is in state `Completing` then it transitions to `Completed`. In state `Completed` a scope can either be closed (if the process completes without a need to compensate the scope) or it can be compensated by its parent scope. Therefore the coordinator either sends a `Close` or `Compensate` message. During compensation initialized by a `Compensate` message a scope is in state `Compensating`. If compensation was successful the coordinator is notified by a `Compensated` message, on fault of the compensation a `Fault` message is sent to the coordinator. Having been compensated normally a scope is in state `Compensated`. Here another shortcoming of the BAP completion protocol is identified. It does not support the flexible compensation initiated by participants that BPEL allows.

4.4.1.1 Flexible Compensation

A BPEL scope can flexibly compensate its directly nested scopes by using `compensate` activities. Therefore a new message from a participant to the coordinator has to be introduced. In [38] this message is called `CompensateSubScopeSuggestion`. This message specifies which participants are to be compensated. Having received such a message the coordinator is in state `CompensatingSubScopeSuggestion` and responds with a `CompensatedSubScopeSuggestion` message if compensation was successful or `CompensatedSubScopeSuggestionWithFault` if a fault occurred during compensation. Having processed a `CompensateSubScopeSuggestion` request the coordinator must be able to receive more such requests since several `compensate` activities can appear in a BPEL fault or compensation handler.
Since `compensate` activities can appear in fault or compensation handlers the coordinator must accept `CompensateSubScopeSuggestion` messages in state `Compensating` and in the state it is when the scope it represents is executing a fault handler.
Now the next shortcoming of the BAP completion protocol regarding the life cycle of BPEL scopes is identified. There is no state that represents a scope that is executing a fault handler.

4.4.1.2 Fault Handling

The BAP completion protocol allows a participant to exit from a transaction through sending an `Exit` message. This is similar to a scope that has faulted but whose fault has been handled by a compensation handler. The coordinator now knows that the participant has completed but cannot be compensated.
Sending a `Faulted` message a participant signalizes that it has faulted. This is similar to a BPEL scope whose fault handler (or compensation handler) has faulted. However, as identified above, a state to which a scope participant can transition before it transitions to an `Exiting` or `Faulting` state is needed. In this state the participant must be able to tell the coordinator to compensate participants. As discussed above this is done through `CompensateSubScopeSuggestion` messages. Therefore another new state, the state `EncounteredFault`, is needed. It represents a scope that is executing a fault handler. The participant leaves this state with either an `Exit` or `Fault` message indicating the successful or unsuccessful outcome of the fault handler.
The behavior of BPEL fault handlers requires all nested active scopes for the scope that the fault occurred in, to be canceled before the fault handler is executed. Canceling of participants in the BAP completion protocol is done by the coordinator by sending a `Cancel` message to a participant. This behavior requires a new message for a participant to tell the coordinator to cancel all nested active participants. In [38] this message is called `EncounteringFault`. Having sent such a message a participant is in state `EncounteringFault`. The coordinator must respond with an `EncounteredFault` message after it has canceled all active participants. Translated to BPEL the `EncounteredFault` message is a "start fault handler" message. The participant is now in state `EncounteredFault`.

4.4.2 The Modified Business Agreement with Participant Completion Protocol (mBAP)

The modified business agreement with participant completion protocol extends the business agreement with participant completion protocol specified in WS-Business Activity [11] in order to meet the requirements demanded by the life cycle of a scope inside a BPEL process. Figure 4.6 shows the messages of WS-BA's business agreement with participant completion protocol (BAP) as well as the additional messages that have been added for the mBAP completion protocol. The need for this messages has been discussed in Subsection 4.4.1.

4.4.2.1 Additional States and Messages

Five new states have been introduced for the mBAP completion protocol.

- Completing

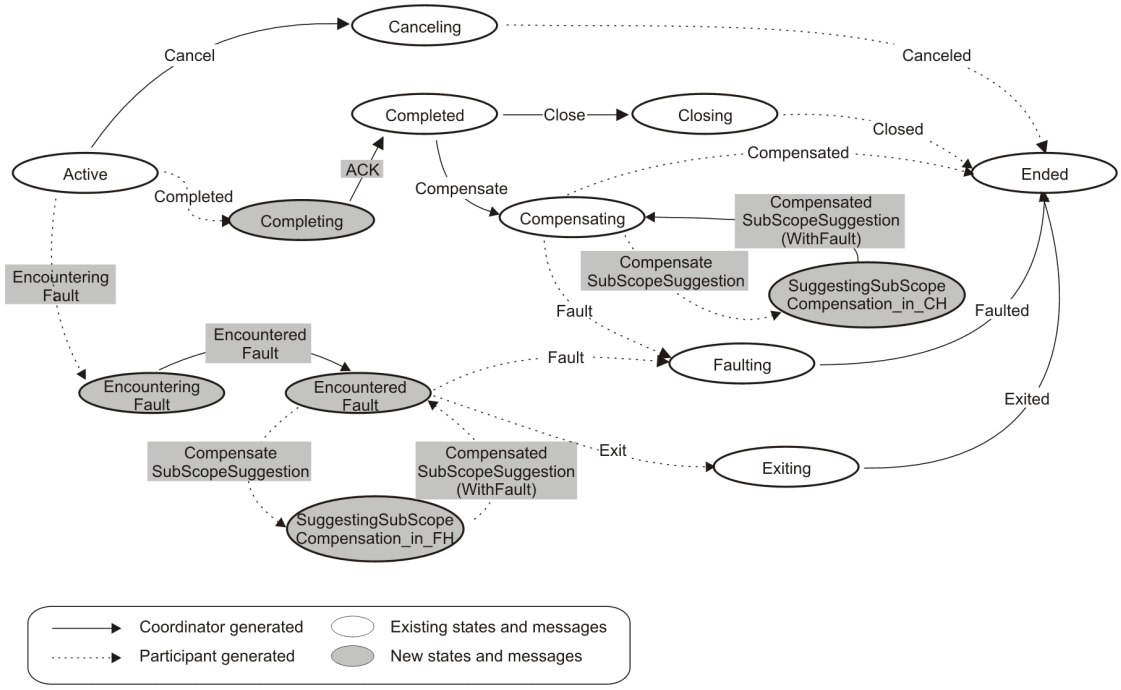

Figure 4.6: The mBAP completion protocol.

- Encountering fault
- Encountered fault
- Compensating sub scope suggestion in a fault handler
- Compensating sub scope suggestion in a compensation handler

Along with these states goes a set of new messages as defined above to trigger these states. A coordinator driving BPEL long-running transactions based on the mBAP completion protocol should be able to understand both the mBAP as well as the WS-BA BAP completion protocol to allow external partners using WS-BA to participate in transactions of a BPEL process instance.
The mapping of BPEL scope life cycle events to the messages of the mBAP completion protocol is discussed below.
Participants wishing to participate in an mBAP-driven coordination must register with the `CoordinationType` parameter of the WS-C `Register` message set to `http://www.iaas.uni-stuttgart.de/iaas/darm/mBAP`.

4.4.3 Normal Processing Mode

After the initialization of a scope as described in Section 4.3 the WS-BA participant representing a scope is in state `Active`. On completion of the scope the WS-BA participant has two choices of messages to send according to the mBAP completion protocol. The two possible messages are `Completed` and `EncounteringFault`.

Translated to BPEL this means that the scope either completed without a fault or a fault occurred during the execution of the scope.

4.4.4 A Scope Completed Normally

The scope completed successfully. A compensation handler is now installed for the scope and the coordinator must be notified that it can now compensate the scope if the need arises.

4.4.4.1 Consequences for the PureBPE

Having completed a scope successfully, the PureBPE must send out an `npmEnded` message and wait for an `npmEndedFinished` message from the coordinator. Then it can install a compensation handler and continue processing the enclosing activity of the scope.

4.4.4.2 Message exchange

Figure 4.7 shows the messages that are exchanged if the normal processing of a scope ends without a fault. After the PureBPE has completed the execution of a scope it sends a `npmEnded` message to the BPEL4Tx coordinator which in turn notifies its WS-BA participant module. This module then sends a WS-BA `Completed` message to the WS-BA coordinator. The WS-BA coordinator now acknowledges the finishing of the normal processing without a fault and responds with an `ACK` message.

All messages only contain one parameter, the scopeId of the scope that completed.

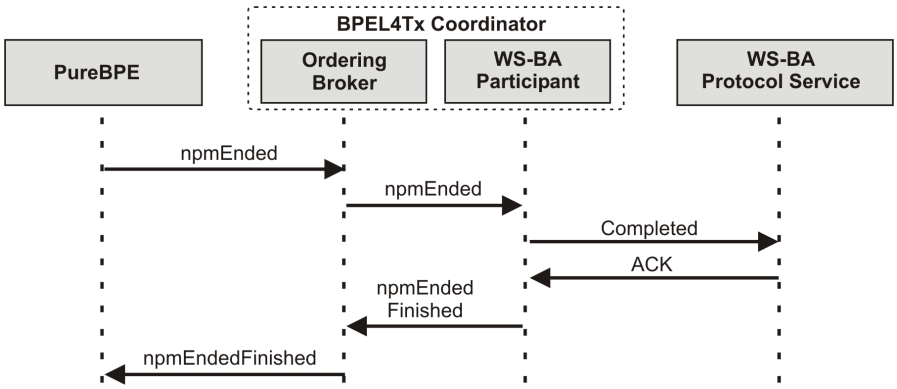

Figure 4.7: Message exchange on completion of a scope.

4.4.5 A Scope Faults

A fault occurred inside a scope. The external coordinator must be notified of the fault that occurred so that it can cancel all nested active scopes of the scope that faulted. Then the corresponding fault handler can be started.

4.4.5.1 Consequences for the PureBPE

Having encountered a fault in a scope the PureBPE must immediately terminate all nested activities that are not scopes. The termination of scopes is the task of the external coordinator. Termination of scopes is done by the external coordinator since there might be external partners that need to be terminated when a scope faults. The PureBPE does not know about these external partners. In order to signalize the coordinator that it can terminate nested scopes the PureBPE sends a `npmFaulted` message and waits for a `npmFaultedFinished` message from the external coordinator. Having received this message the PureBPE knows that all active nested scopes have been terminated and that it can now start its fault handling routine. The `npmFaultedFinished` message therefore is more than an acknowledgment message that all nested scopes have been canceled, it also is a "start fault handling" message.

4.4.5.2 Message Exchange

Figure 4.8 shows the messages that are exchanged if the normal processing of a scope faults. The PureBPE sends out a `npmFaulted` message which signalizes the

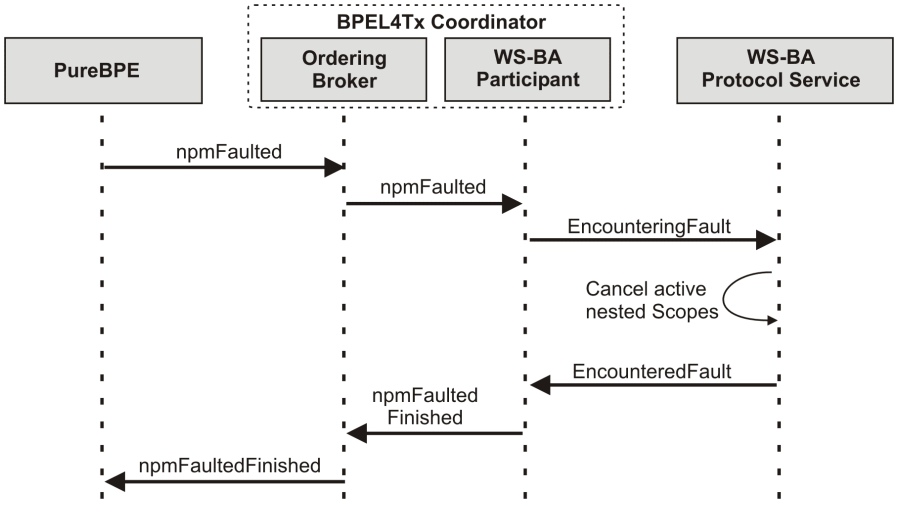

Figure 4.8: Message exchange on faulting of a scope.

BPEL4Tx ordering broker that it must notify the BPEL4Tx WS-BA participant to send an `EncounteringFault` message to the WS-BA coordinator. On receipt

of this message the WS-BA coordinator knows that it has to terminate all active participants of the activity that represents the faulted scope. If no activity in the sense of WS-C represents the faulted scope it does not have nested children and no `Cancel` messages have to be sent. If the faulted scope has nested scopes and is therefore represented by a WS-C activity as well as a WS-C participant, the coordinator for the activity representing the faulted scope sends a `Cancel` message to all participants of that activity that are in the state `Active`. The participant representing the faulted scope is now in state `EncounteringFault`.
After having terminated, those participants respond with a `Canceled` message. What exactly happens if a participant receives a `Cancel` message is explained in detail in Section 4.4.6. After all participants that needed to terminate have responded with a `Cancel` message, the coordinator tells the BPEL4Tx WS-BA participant of the originally faulted scope that all nested scopes are no longer active by sending an `EncounteredFault` message. The participant representing the faulted scope is now in state `EncounteredFault`.
The `npmFaulted` as well as `EncounteringFault` messages must contain all BPEL fault handling specific information of the fault that occurred. This means that the fault name and fault variable name as well as the fault data have to be included. This is necessary for the external coordinator in case it must rethrow the fault to the enclosing scope. Additionally the fault specification might be needed by an external coordinator to determine which fault handler to execute.
All other messages are simply notifications and therefore have only one parameter: The scopeId of the scope that faulted.

4.4.6 Termination of Scopes

The need to terminate a scope occurs in BPEL if a scope or a fault or compensation handler faults and it has nested scopes that are still active (i.e. in normal processing mode). Those nested scopes have to be terminated before a fault handler can execute in their parent scope, or in case of a faulted fault or compensation handler the fault can be rethrown to the parent scope.
According to the BPEL specification those nested scopes should fault with a `forcedTermination` fault. If a fault handler that can handle the `forcedTermination` fault is available for the nested scope it can handle the fault now, otherwise the implicit default fault handler must be executed.
Figure 4.6 is somewhat inconsistent to the description above since it lacks the exchange of fault handling messages in state `Canceling`. Since these messages are already displayed in the branch that leaves the state `Active` with an `EncounteringFault` message, these messages have been removed from Figure 4.6 in order to keep the figure more concise.

4.4.6.1 Consequences for the PureBPE

The PureBPE must be able to accept messages from the external coordinator for scopes that are still active telling it to terminate these scopes. Having received such a message the PureBPE must acknowledge this message and then treat the scope as if it had faulted with any other fault as described in Subsection 4.4.5.1. Additionally, the PureBPE must still be able to terminate nested activities of a scope that are not scopes themselves, since only scopes are terminated by the external coordinator.

4.4.6.2 Message Exchange

Figure 4.9 shows the messages exchanged between the PureBPE and the external coordinators if the WS-BA coordinator wants to terminate a nested scope.

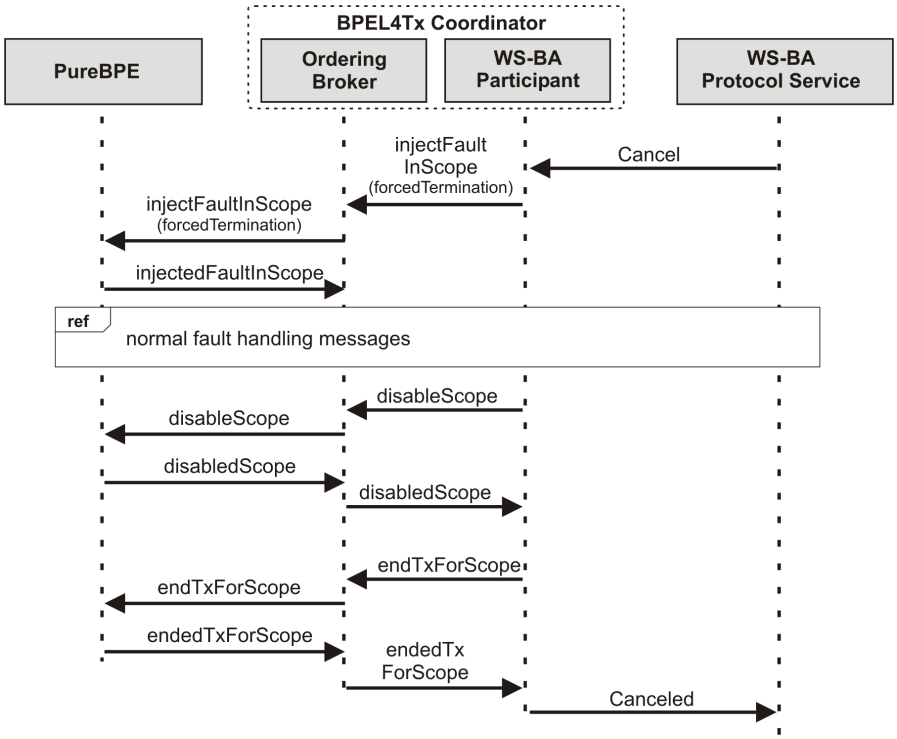

Figure 4.9: Message exchange to terminate a scope.

The WS-BA participant receives a `Cancel` message as specified in the WS-BA specification. It then tells the BPEL4Tx ordering broker to inject a `forcedTermination` fault into the scope that should be terminated. The BPEL4Tx ordering broker must send an `injectFaultInScope` message to the PureBPE telling it to terminate the corresponding scope. The PureBPE throws a `forcedTermination` fault in the scope. Then it responds with an

`injectedFaultInScope` message. Having completed the fault handling (and sent and received the messages necessary for it,) the BPEL4Tx WS-BA participant sends a `disableScope` message to the PureBPE. The `disableScope` message tells the PureBPE to inspect all outgoing links of the scope which is to be terminated and set them to death path. On successful completion of this task the PureBPE responds with a `disabledScope` message.
After the scope has been disabled the BPEL4Tx WS-BA participant tells the BPEL4Tx ordering broker to send an `endTxForScope` message to the PureBPE indicating that it can discard all transaction information for this scope. After doing so the PureBPE responds with an `endedTxForScope` message which triggers the BPEL4Tx coordinator to send a `Canceled` message via its BPEL4Tx WS-BA module to the WS-BA coordinator.

4.4.7 Implicit Fault Handling

After having received a `npmFaultedFinished` message, the PureBPE can execute the fault handler for a scope. Either a fault handler is specified for the fault that occurred, then the PureBPE executes this fault handler or no fault handler is specified for the fault then the implicit fault handler has to be executed. This fault handler is executed by the BPEL4Tx coordinator, so the PureBPE simply stops execution of a scope if it has no defined fault handler and notifies the BPEL4Tx coordinator so that it can handle the fault.
The reason why the implicit fault handler can be executed by the external coordinator is that actually is a request to compensate all nested scopes of the scope the fault occurred in. This is behavior to be executed by the external coordinator anyway, so the entire implicit fault handler is executed by the external coordinator.

A non-implicit fault handler cannot be executed entirely by the external coordinator since it may contain arbitrary BPEL activities (such as assign, scope, sequence, invoke, receive and even fault handler activities). Handling these activities externally would require the external coordinator to be a complete BPEL engine itself and it would require the external coordinator to be aware of variables and their values used inside the BPEL process. Therefore only the information relevant for the external transaction processing is handed over to the external coordinator. Transaction relevant information are `compensate` activities nested in the fault handler since they trigger the compensation of nested scopes which is to be performed by the external coordinator. How `compensate` activities in fault and compensation handlers are treated is described in Section 4.4.9. Additionally the outcome of the whole scope (and therefore of enclosing scopes) is dependent on the outcome of the fault handler, therefore this information is sent to the external coordinator. The handling of the completion of a fault handler is discussed in Section 4.4.8.

In the occasion an implicit fault handler has to be executed the BPEL4Tx ordering broker starts the BPEL4Tx WS-BA default fault handler module that will handle the fault.
This module then tells the corresponding WS-BA participant to notify the WS-BA coordinator to compensate all nested scopes that have completed of the scope on which the default fault handler is executing. The coordinator compensates all nested scopes that are to be compensated. Details on how exactly the compensation of nested scopes works is described in detail in Section 4.4.10.
When all necessary nested scopes are compensated the BPEL4Tx WS-BA participant notifies the BPEL4Tx WS-BA Default fault handler that all nested scopes are compensated.
Despite having possibly completed successfully, the BPEL4Tx WS-BA default fault handler always faults and notifies the BPEL4Tx WS-BA participant of this fault. This is necessary because later on the fault has to be rethrown into the next enclosing scope.
Details on how the situation is handled when a fault handler faults are discussed in Section 4.4.8.

4.4.7.1 Consequences for the PureBPE

After having encountered a fault in a scope and having canceled all nested activities as described in Subsection 4.4.5 the PureBPE can now execute the appropriate fault handler. If none is specified the implicit default fault handler has to be executed. The PureBPE therefore directs the external coordinator to execute this implicit default fault handler. Then it must wait until the external coordinator responded that the external execution of the implicit default fault handler has finished. The PureBPE must not rethrow the fault into the next scope, rethrowing faults is the task of the external coordinator.

4.4.7.2 Message Exchange

Figure 4.10 shows the messages that are exchanged if a fault is handled by the implicit default fault handler.
The PureBPE has to execute a fault handler for a scope that faulted but has no defined handler for that fault (or none at all). It notifies the BPEL4Tx ordering broker by sending an `execDefFH` message, the ordering broker starts the BPEL4Tx WS-BA Default fault handler by sending an `execDefFH` to this module. The BPEL4Tx WS-BA default fault handler then tells the BPEL4Tx WS-BA participant to send a `CompensateSubScopeSuggestion` message to the WS-BA coordinator. The WS-BA coordinator responds with a `CompensatedSubScopeSuggestion` message after it has completed all necessary work.

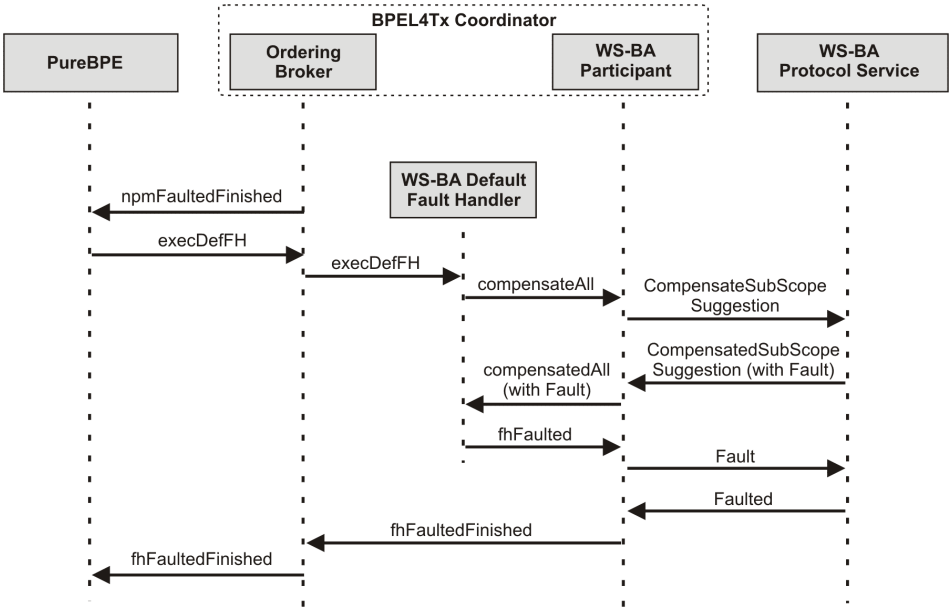

Figure 4.10: Message exchange when a fault is handled implicitly.

The BPEL4Tx WS-BA default fault handler always ends with a `fhFaulted` message to the BPEL4Tx WS-BA participant. The BPEL4Tx WS-BA participant then continues as defined in the next Section 4.4.8. All messages simply contain the scopeId of the scope the fault handling is for since they are all notifications.

4.4.8 A Fault Handler Ends

After a fault handler has finished normally, the scope is no longer relevant for the transaction, because it has ended normal processing and the fault handling has ended. This means that it cannot be compensated because it has not completed normally. Additionally it cannot change its state since it has already finished its normal processing mode. This corresponds to exiting from a transaction in the WS-BA business agreement with participant completion protocol. After the PureBPE has told the external coordinator that this scope is no longer participating in a transaction the PureBPE can discard all transaction information for this scope.

If a fault handler faults, the coordinator first has to terminate all nested scopes in this fault handler that might still be active. Then the coordinator must tell the PureBPE to disable the scope. After disabling the scope for which the faulted fault handler was executed the PureBPE can discard all transaction information. After the transaction information for the fault for which the faulted fault handler was executed, is discarded the fault must be injected into the enclosing scope by the BPEL4Tx coordinator.

4.4.8.1 Consequences for the PureBPE

After receiving the `npmFaultedFinished` message from the BPEL4Tx coordinator the PureBPE executes a fault handler handling the fault that occurred for a scope. If no fault handler is specified an implicit fault handler is executed by the BPEL4Tx coordinator as described in Section 4.4.7. In case of a defined fault handler for a fault that has occurred, the PureBPE can execute the fault handler handling the fault. After the fault handler finished the PureBPE has to notify the BPEL4Tx coordinator that the fault handler has finished.
Then it has to wait for an acknowledgment from the external coordinator and a request to discard all transaction information.

In case a fault handler faults, all active nested activities of the scope must be terminated. The PureBPE therefore must terminate all nested activities that are not scopes and notify the external coordinator. The PureBPE then waits for a message from the external coordinator indicating that it has canceled all active nested scopes. Afterwards the PureBPE must wait for messages from the external coordinator telling it to disable the scope. On receipt of such a message the PureBPE must examine all links that leave the scope and must set them to dead path. After acknowledging the disabling of a scope the PureBPE must wait for a message from the external coordinator telling it to discard all transaction specific information.

4.4.8.2 Message Exchange

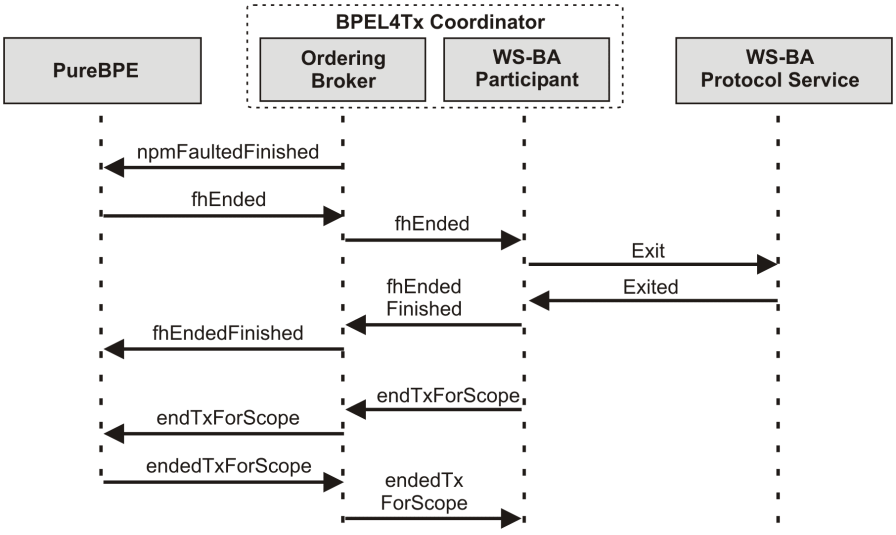

Figure 4.11: Message exchange when a fault handler completes normally.

Figure 4.11 shows the messages that are exchanged if a fault handler for a scope completes normally.

If the fault handler completes normally the PureBPE sends a `fhEnded` message to the BPEL4Tx ordering broker. The ordering broker forwards the message to the BPEL4Tx WS-BA participant which translates it to an WS-BA `Exit` message. After sending the `Exit` message to the WS-BA coordinator the BPEL4Tx WS-BA participant waits for the `Exited` response from the WS-BA coordinator. On receipt of an `Exit` message from a participant the WS-BA coordinator removes the participant from the transaction context and answers upon successful removal with an `Exited` message.
After receiving the `Exited` message, the BPEL4Tx WS-BA participant tells the ordering broker to send a `fhEndedFinished` message to the PureBPE. The PureBPE can now continue normal processing (e.g. of the enclosing scope of the scope that originally faulted). Because scopes that have executed a fault handler cannot be compensated according to the BPEL specification the PureBPE can discard all transaction information. The signal to do so is always given by the external coordinator by sending an `endTxForScope` message which contains a `stateForAudit` parameter to tell the PureBPE what state information is to be saved for a potential audit. Here the value of this parameter is "Exited". After discarding all transaction information the PureBPE responds with an `endedTxForScope` message.

Figure 4.12 shows the message exchange for the situation if a fault handler for a scope faults.
After sending a `fhFaulted` message to the BPEL4Tx ordering broker the PureBPE waits for an `fhFaultedFinished` message. The BPEL4Tx receives the `fhFaulted` message and forwards it to the BPEL4Tx WS-BA participant which translates it into a WS-BA `Fault` message which is sent to the WS-BA coordinator. The coordinator then cancels all nested participants that are in state `Active`. This is not shown in Figure 4.12. A detailed explanation on how participants are canceled is given in Section 4.4.6. The WS-BA signalizes with a `Faulted` message that it has completed handling the actions necessary after receiving a `Fault` message. The WS-BA participant then tells the BPEL4Tx ordering broker module to send an `fhFaultedFinished` message to the PureBPE. In order to disable the scope that has originally faulted the WS-BA participant sends a `disableScope` message to the PureBPE which responds with a `disabledScope` after it has disabled all outgoing links of this scope (i.e. set them to death path). On receipt of the `disabledScope` message the BPEL4Tx WS-BA participant tells the pure BPE to discard all transaction information for the scope and set the state of this scope for a later audit to "Faulted" by sending an `endTxForScope` message with the parameter `stateForAudit` set to "Faulted". The BPEL4Tx WS-BA participant then waits for a response from the PureBPE which sends an `endedTxForScope` message on completion of the deletion of transaction information.
Finally the WS-BA coordinator sends an `injectFaultInScope` message to the PureBPE to signalize that the parent scope should fault. Therefore the `scopeId` parameter of the `injectFaultInScope` message is set to the scopeId of the parent of the scope that originally faulted. The `fault` parameter is set to the fault that occurred during execution of the fault handler.

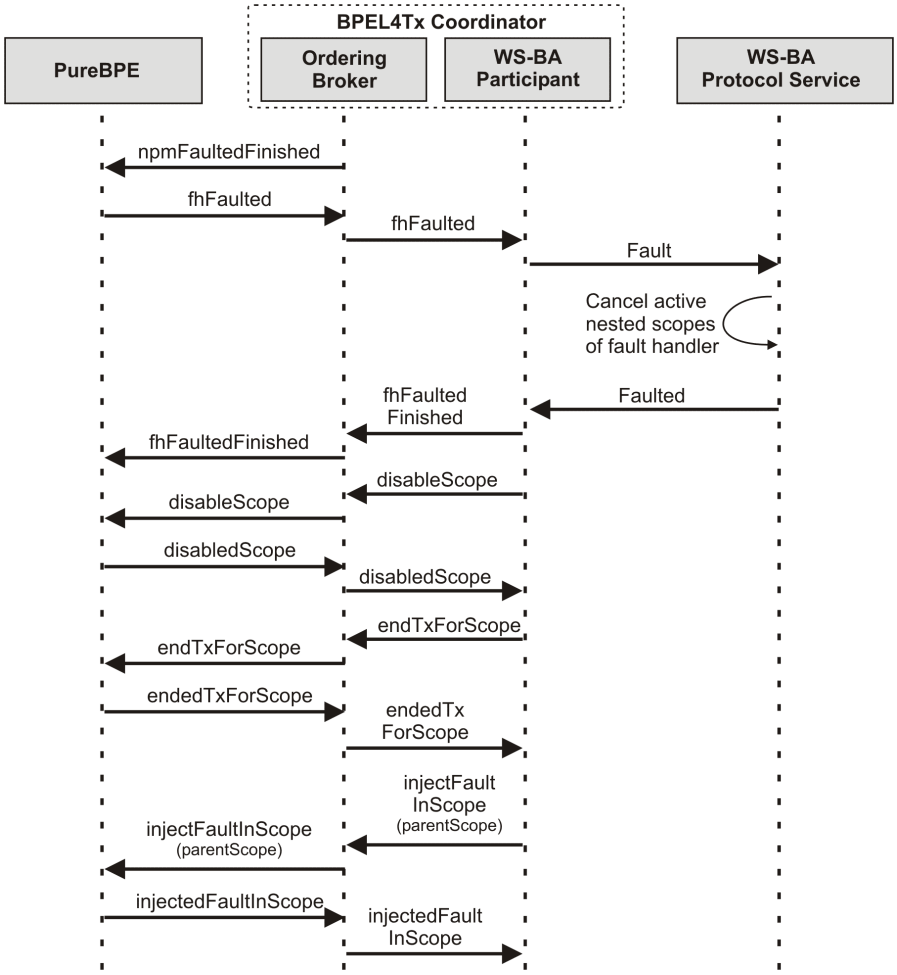

Figure 4.12: Message exchange when a fault handler faults.

The PureBPE injects the fault and responds with an `injectedFaultInScope` message. After that the fault in the parent scope is handled by the PureBPE just like a normal fault and as described in Section 4.4.4.

4.4.9 Execution of External Activities

External activities are activities specified in the BPEL process that should be executed by the external coordinator. In this thesis this is only the BPEL `compensate` activity. When the PureBPE engine is about to execute such an activity it must notify the external coordinator and hand over the execution of the the activity to the external coordinator. Therefore it must include the XML definition of the activity in the message that notifies the external coordinator. After a response that the external coordinator has executed the message, the PureBPE can continue normal processing as if it had executed the external activity internally.
The task of the BPEL4Tx ordering broker in this scenario is to receive the notification that an external activity has to be executed. Then it must evaluate the

activity definition and take the necessary steps to execute the activity.
In case of a `compensate` activity the necessary steps are simple for the BPEL4Tx ordering broker. It simply has to forward the notification to the BPEL4Tx WS-BA participant. The BPEL4Tx WS-BA participant sees that a `compensate` activity needs to be executed. It tells the WS-BA coordinator that it has to compensate one or more participants that represent sub-scopes of the scope in whose handler the `compensate` activity appears.
The WS-BA coordinator then takes the necessary steps described in the Section 4.4.10 to compensate one or more participants. The BPEL4Tx WS-BA participant notifies the PureBPE of the outcome of the execution of the external activity depending on the answer to the compensation request of the WS-BA coordinator. If execution of the external activity has faulted the PureBPE continues as if a normal activity has faulted. On successful execution of the external activity the PureBPE continues as if a normal activity had completed without a fault.

4.4.9.1 Consequences for the PureBPE

The PureBPE must detect activities (here: the compensate activity) that it cannot execute. If such a message is about to be executed by the PureBPE it must notify the external coordinator of such an activity in order to have the external coordinator execute the activity. Then it must wait until the external coordinator responded that the external activity ended or faulted. Upon receipt of the outcome message for the external activity the PureBPE can continue processing the process instance the external activity appeared in, as if a normal activity completed or faulted.

4.4.9.2 Message Exchange

The PureBPE sends an `execActivity` message to signalize that an activity has to be executed by the BPEL4Tx coordinator. The `execActivity` message contains a parameter `activityDef` whose value is the XML representation of the activity from the BPEL XML file. The BPEL4Tx ordering broker receives the message and forwards it to the BPEL4Tx WS-BA participant which in turn sends a `CompensateSubScopeSuggestion` message that contains the `activityDef` to the WS-BA coordinator. The coordinator eventually responds with a `CompensatedSubScopeSuggestion` message as shown in Figure 4.13 or an `CompensatedSubScopeSuggestionWithFault` message as shown in Figure 4.14. Depending on the message received from the WS-BA coordinator the BPEL4Tx WS-BA participant tells the BPEL4Tx ordering broker to send either an `execActivityEnded` or an `execActivityFaulted` message to the PureBPE. Figure 4.13 shows the message exchange for the execution of an external activity that completes normally without faulting. Figure 4.14 shows the message exchange for the execution of an external activity that faults.

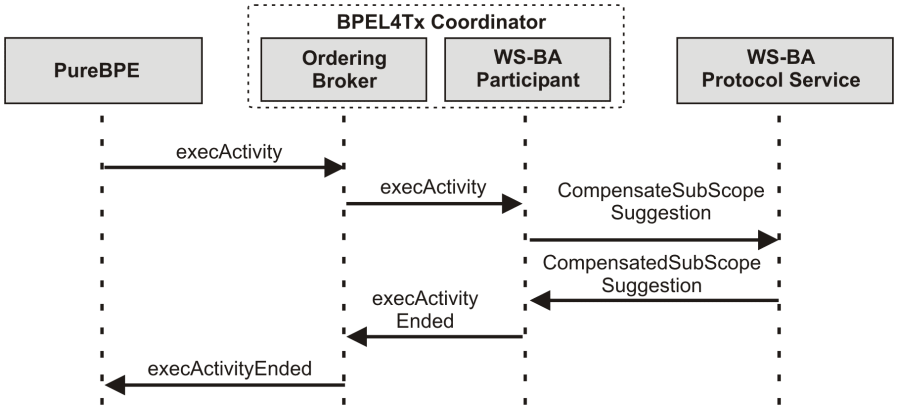

Figure 4.13: Message exchange for an external compensate activity that completes normally.

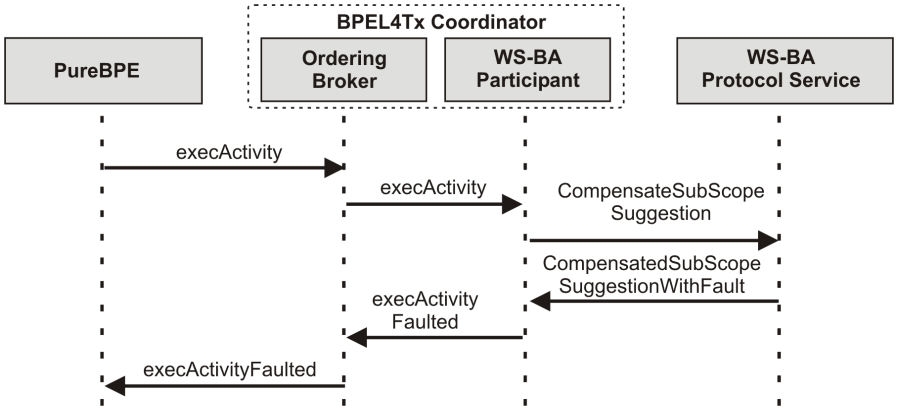

Figure 4.14: Message exchange for an external compensate activity that completes with a fault.

4.4.10 Compensation

Upon receipt of a `CompensateSubScopeSuggestion` message the WS-BA coordinator evaluates the `activityDef` parameter of this message. In this prototype only two possibilities exist. The first possibility is that the activity is a BPEL `<compensate/>` activity without a scope attribute. This means, according to the BPEL specification, that all nested scopes should be compensated. The other possibility is that the activity is a BPEL `<compensate scope="scopeName"/>` activity which has a scope attribute whose value points to one of the nested scopes of the scope in whose fault or compensation handler the activity occurs. The scope attribute specifies that for this `compensate` activity only the nested scope with the name of the scope attribute's value should be compensated.
After the coordinator has evaluated which participants are to be compensated it sends a `Compensate` message to every participant. In order to perform the scope name - participant correlation the WS-BA coordinator must maintain some sort of directory which returns the participant for a scope name. The scope name for a participant has been submitted upon registration of a participant with the

`Register` message.
If the BPEL4Tx WS-BA participant for a scope receives a request to compensate the scope it must check if the scope has a compensation handler or if the implicit default compensation handler has to be used.
For scopes with a defined compensation handler, this defined compensation handler has to be executed by the PureBPE. The BPEL4Tx WS-BA participant must therefore tell the PureBPE to execute the compensation handler for the scope that has to be compensated.
There are two possibilities of the outcome of the compensation handler. Either it completes successfully or it faults. In both cases the BPEL4Tx coordinator needs to be informed of the outcome, it then in turn informs the WS-BA coordinator if the scope has been successfully compensated or not.

Scopes that have no defined compensation handler are compensated implicitly. The implicit compensation handler simply compensates all completed nested scopes in reverse order. This is an activity to be done by the external coordinator anyway, so the external coordinator does not need to notify the PureBPE but executes the implicit fault handler itself. Therefore the BPEL4Tx WS-BA participant tells the coordinator to compensate all completed nested scopes of the scope that should originally be compensated. After all nested scopes are compensated the WS-BA coordinator notifies the WS-BA participant of this fact which in turn tells the WS-BA coordinator that it has compensated the scope. Then the WS-BA participant tells the PureBPE to discard all transaction information.

4.4.10.1 Consequences for the PureBPE

On receipt of a message telling the PureBPE to execute a life cycle handler it must evaluate the message and start the appropriate handler. In the case discussed here this is the compensation handler. The handler is executed using a snapshot of the scope's data recorded upon successful completion of the scope. On faulting or completion of a compensation handler the PureBPE must notify the external coordinator and wait for further messages from the external coordinator. The PureBPE does not have to deal with the execution of implicit default compensation handlers anymore.

4.4.10.2 Snapshot Data for Compensation

As mentioned above, a compensation handler always works on snapshot data recorded on successful completion of a scope's normal processing mode. This data is recorded inside the PureBPE and kept inside it, despite its transaction relevant nature, for reasons of efficiency. Transferring snapshot data to the external coordinator would require all variable data and other snapshot data of a scope to be sent to the external coordinator. Since normal compensation handlers have to

be executed in the PureBPE anyway and implicit compensation handlers do not need snapshot data, the data is kept inside the PureBPE.

4.4.10.3 Message Exchange

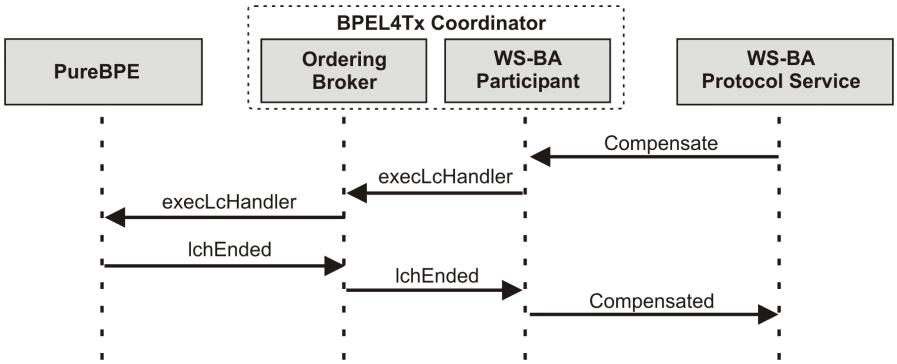

Figure 4.15: Message exchange for compensation with normal completion of the compensation handler.

Figure 4.15 shows the message exchange if a scope needs to compensate and it has a defined compensation handler. This compensation handler ends normally without throwing a fault.
On receipt of a `Compensate` message the BPEL4Tx WS-BA participant evaluates if a compensation handler is defined for the scope it represents or not. In this case one is defined. The BPEL4Tx WS-BA participant now tells the PureBPE engine to start executing the compensation handler by sending an `execLCHandler` message in which the `handler` parameter is set to "compensationHandler". This is the only value supported in this prototype for the `handler` parameter. Other handlers such as termination handlers may be needed if BPEL 2.0 must be supported. The PureBPE receives the message and executes the compensation handler. If the handler completes without a fault the message exchange continues as shown in Figure 4.15, otherwise it continues as shown in Figure 4.16. If a life cycle handler (here the compensation handler) for a scope completes normally the PureBPE notifies the BPEL4Tx coordinator by sending a `lchEnded` message. If the life cycle handler faults a `lchFaulted` message is sent. The WS-BA coordinator is notified with a `Compensated` or `Fault` message respectively. In case of a faulted life cycle handler the WS-BA coordinator must terminate all directly nested scopes of the life cycle handler that are still in normal processing mode (which corresponds to the state `Active` of WS-BA). Then it responds with a `Faulted` message. Afterwards the BPEL4Tx WS-BA participant tells the BPEL4Tx ordering broker to send an `endTxForScope` message to the PureBPE indicating that it can discard all transaction information for the scope. Figure 4.16 shows the message exchange in case a scope needs to compensate and it has a defined compensation handler that faults during its execution.

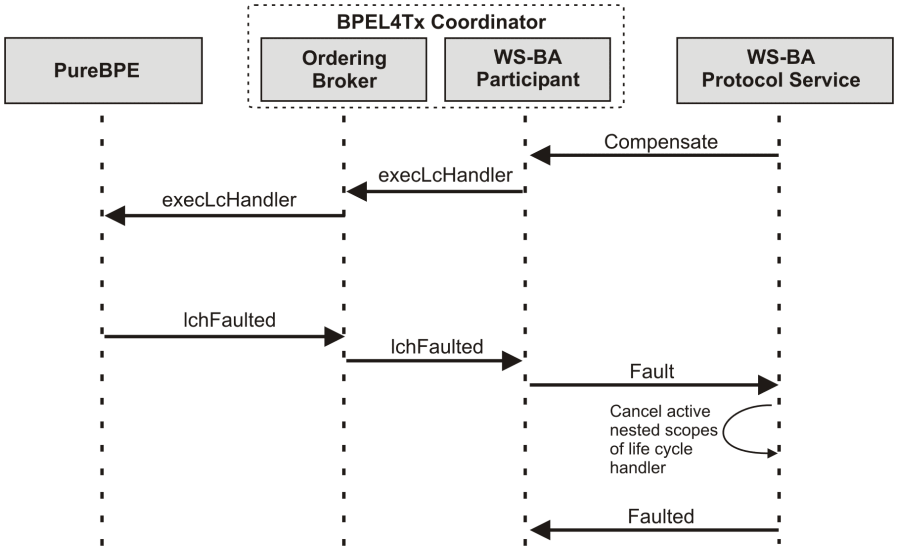

Figure 4.16: Message exchange for compensation with a faulting compensation handler.

Figure 4.17 shows the message exchange if a scope needs to compensate but has no defined compensation handler and the compensation is handled by the implicit compensation handler.
After receiving a `Compensate` message the BPEL4Tx WS-BA participant has discovered that no compensation handler is defined for this scope so it sends a `CompensateSubScopeSuggestion` message to the WS-BA coordinator to advise it to compensate all nested scopes that are in the state `Completed`. After the WS-BA coordinator has compensated all nested scopes it returns with a `CompensatedSubScope` message. This signalizes the WS-BA participant that the implicit compensation handler has finished. To signalize the WS-BA coordinator that the compensation action for this scope has finished the BPEL4Tx WS-BA sends a `Compensated` message to the WS-BA coordinator.

4.4.11 Closing of Scopes and Nested Coordinators

Special treatment inside the WS-BA coordinator is needed if the participant that has completed is a participant of the root activity. If a BPEL process instance is controlled by the coordinator and this process has not been started as a participant of an external transaction, the root activity is the activity that has the BPEL `process` element as a participant.
On completion of a participant of the "root" activity the coordinator needs to check if all other participants have reached a final state too. A final state is either `Completed`, or `Ended`. Depending on the WS-BA coordination type the coordinator has to take the following actions:

- If the coordination type is `MixedOutcome` the WS-BA coordinator must send

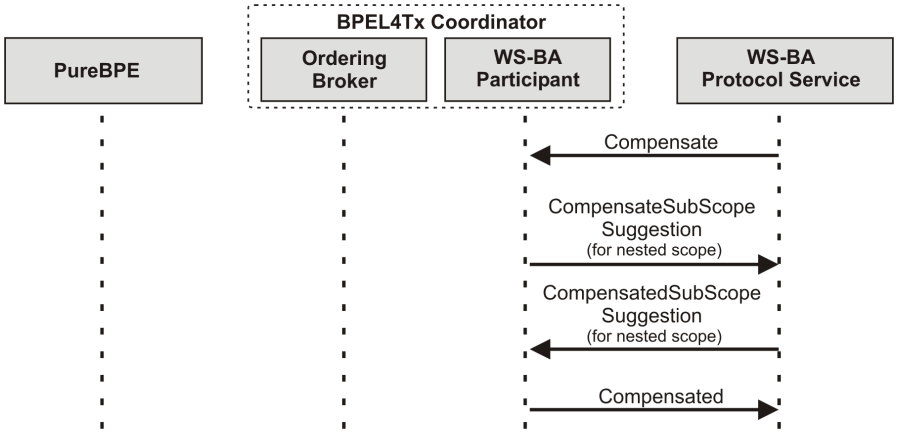

Figure 4.17: Message exchange for compensation with the implicit compensation handler.

a `Close` message to all participants in state `Completed`. If nested coordination contexts have been created by specifying the coordination context of the root activity as `CurrentContext` in the `CreateCoordinationContext` message, the coordinator needs to notify the WS-BA coordinators for those nested activities that they can terminate. Those nested coordinators in turn have to take the actions described in this section. After all scopes that the coordinator sent a `Close` message to, have responded with a `Closed` message and all nested coordinators have ended, the coordinator itself can end.

- In case of an `AtomicOutcome`coordination type, the WS-BA coordinator must check if all participants are in state `Completed`. If yes the coordinator can send a `Close` to all participants and tell the nested coordinators to end. After all scopes and nested coordinators have closed the root coordinator can end.
  If at least one participant is in state `Ended` that means that it has either faulted or compensated then all other participants that are in state `Completed` must now be compensated to achieve atomic outcome. Therefore the WS-BA coordinator must send a `Compensate` message to all those participants and wait until they have responded with a `Compensated` message. Then it can tell all nested coordinators to end. After all nested coordinators have ended the root coordinator can end too.

4.4.11.1 Consequences for the PureBPE

The PureBPE must be able to receive a message telling it that it can discard transaction specific information for a scope. On receipt of such a message the PureBPE must delete all transaction specific information for the particular scope and set the scope to a final state. Then it can forget the scope and must notify the external coordinator that it has finished. If the scope is the process element

itself this particular process instance has ended and the PureBPE can regard the whole process instance as finished.

4.4.11.2 Message Exchange

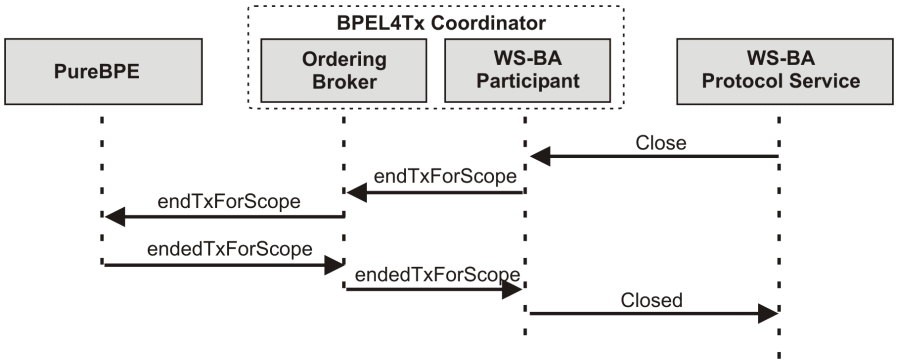

Figure 4.18: Message exchange for closing of scopes.

Figure 4.18 shows the messages exchanged if the WS-BA coordinator decides to close a participant. Upon receipt of a `Close` message the BPEL4Tx WS-BA participant must notify the BPEL4Tx ordering broker that it can tell the PureBPE to discard the transaction information of a scope since it will never be compensated later on. Therefore the BPEL4Tx ordering broker sends an `endTxForScope` message to the PureBPE. If the PureBPE sees a message of this type for a scope identified by its scopeId it discards this scope's transaction information and responds with an `endedTxForScope` message. The BPEL4Tx ordering broker forwards this message to the BPEL4Tx WS-BA participant which in turn notifies the WS-BA coordinator that it has `Closed`.

4.5 Summary

In Chapter 4 a theoretical solution on how to coordinate BPEL LRTs using an external coordinator is presented. The necessary middleware has been identified and messages and their formats that need to be exchanged between the BPEL engine running a process and the external coordinators have been discussed. It has been shown that the business agreement with participant completion protocol from WS-Business Activity is not enough to coordinate BPEL `scope` activity life cycles. Therefore a modified completion protocol, the mBAP protocol has been introduced and a mapping from scope life cycle events to messages of this protocol has been established.

5 Prototype Realization

Chapter 5 deals with the realization of the prototype that is part of this thesis. The prototype demonstrates the theoretical work discussed in Chapter 4.
It is based completely on open source middleware that has been partially modified to comply with the requirements that arise from the problems discussed in Chapter 4.

5.1 Prototype Architecture Overview

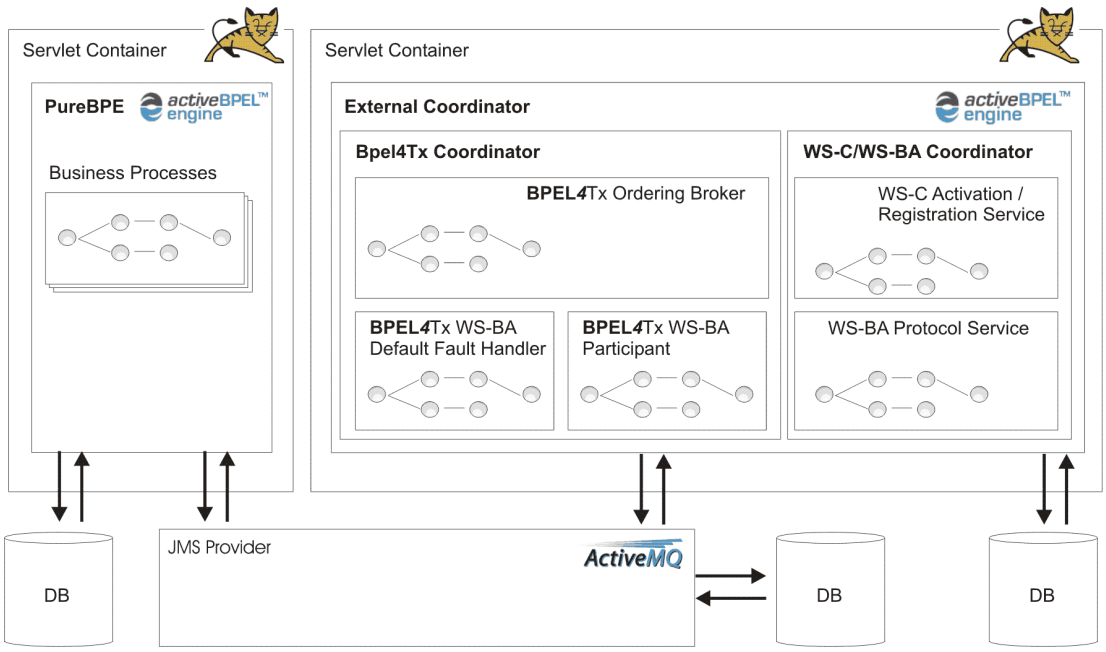

Figure 5.1: Prototype architecture overview.

Figure 5.1 gives an overview of the main parts of the prototype.
Essentially the whole prototype system consists of two Apache Tomcat [9] servlet containers running two modified ActiveBPEL [1] BPEL Business Process engines and an ActiveMQ [6] Java Messaging Service (JMS) provider that allows the sending of messages between these two engines. ActiveMQ and Apache Tomcat are available under the GPL compatible Apache License [8]. Each of these three building blocks, the ActiveBPEL engines as well as the ActiveMQ JMS provider use

a MySQL Database for persistence. MySQL [35] is available under GPL [23] for open source projects. All middleware used in this prototype is open source and can be used free of charge.
Figure 5.1 also demonstrates the two main tasks necessary to build the prototype.

1. The PureBPE. Essentially a modified ActiveBPEL engine that allows external life cycle control of BPEL scopes. Extraction of transaction information from the BPEL engine as well as sending and receiving of life cycle messages from and to the BPEL engine have to be added to the ActiveBPEL engine for this part of the prototype.

2. The external coordinator. The external coordinator can be split in two parts: the BPEL4Tx coordinator and the WS-C/WS-BA coordinator. Figure 5.1 shows the 5 business processes the prototype employs to coordinate BPEL scopes externally.
   The external coordinators are implemented in BPEL to allow the dynamic plugging of new coordinators to the system. In order to plug in a coordinator for a new coordination protocol (for example WS-Atomic Transaction) the deployment of new coordinator process modules would be sufficient.
   The BPEL4Tx coordinator consists of the three BPEL4Tx business processes, BPEL4Tx ordering broker, BPEL4Tx WS-BA default fault handler and BPEL4Tx WS-BA participant. The WS-C activation service / WS-C registration service business process as well as the WS-BA Protocol Service business process form the WS-C/WS-BA coordinator.

5.1.1 Architecture Alternatives

Several architecture alternatives are possible for the implementation of the prototype. Here some of them are discussed shortly and a short motivation is given why the chosen architecture was picked.

5.1.1.1 Different Versions of the ActiveBPEL Engine

Two different versions of the ActiveBPEL engine are needed. One that uses external transaction processing for the actual business relevant business processes to be executed. The other one is needed to execute the transaction relevant coordination processes that control the transaction specific information of the first engine. Internal transaction processing as opposed to external transaction processing is necessary for the second engine because otherwise infinite recursion of transaction controlling would occur. The problem of infinite recursion is visualized in Figure 5.2. The PureBPE sends an `initScope` message which triggers the coordinator for this scope. Upon starting of a new process (which is a special case of a scope) the business process engine that executes the coordinator sends an `initScope` message

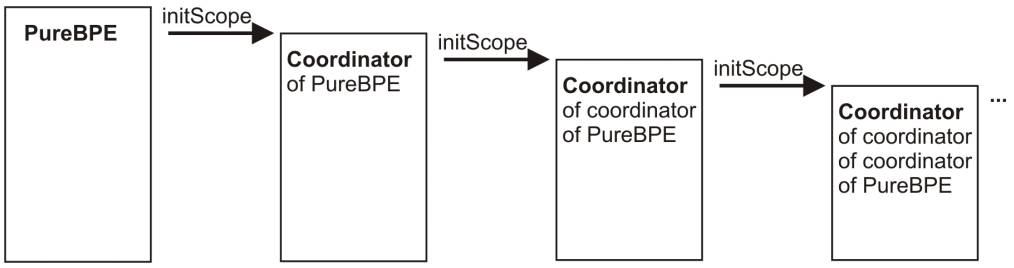

Figure 5.2: Infinite recursion with external transaction processing.

to register with the external coordinator which in turn starts a new process that registers with a coordinator and so on.
In order to prevent an infinite recursion a suitable termination condition is needed. Three possible solutions come to mind:

- Fault handling and compensation is simply turned off for the external coordinator. This is not feasible since the external coordinator needs at least fault handling capabilities that are part of the external coordination.

- In the deployment descriptor for a process the administrator can specify if it uses internal or external coordination. Theoretically this would be a working solution. In this case the external transaction controlling requires fundamental changes in the ActiveBPEL engine application code which is described in the following sections. Having two different transaction control modes in one engine would require a substantial amount of if-then-else statements and/or subclassing which would make the code nearly unmanageable.

- The third alternative is the one used in this prototype. This alternative has the disadvantage that two ActiveBPEL installations are needed one with external and one with internal transaction processing. The advantage of this setup, however, is that the engine that controls the other one still has transaction and fault-handling abilities while the code for the two versions is strictly separated.
  Additionally the use of two separate engines (which would be possible in all three cases) has another advantage. If an external participant for a transaction causes the coordinator to crash, the ActiveBPEL engine which might be running business critical information might still run and possibly valuable data is not affected. Additionally the coordinator can be run in a demilitarized zone of the network to allow external partners to participate in transactions while the sensitive business information can be run behind a firewall.
  Of course one could always go one step further and deploy the BPEL4Tx coordinator and the WS-C / WS-BA coordinator on different machines. This is possible using the prototype presented here, it requires the modification of two WSDL files and two deployment descriptors.

5.1.1.2 WS-Reliable Messaging vs. JMS

A means to reliably send messages between the PureBPE and the external coordinator but also between the single coordinator processes is needed. Considering the fact that the ActiveBPEL engine already can send and receive SOAP messages over HTTP WS-Reliable Messaging [17] would be a straightforward solution.
Since the ActiveBPEL engine does not support WS-Reliable Messaging (WS-RM) out of the box, using WS-RM requires substantial changes in the ActiveBPEL code in order to immediately persist received messages and send and receive the acknowledge messages required by WS-RM.
In a Java environment the Java Messaging Service (JMS) is a convenient way to achieve reliable, asynchronous messaging [29] [44]. In this prototype the open source JMS messaging solution ActiveMQ is used. Since JMS offers an interface that allows to exchange the underlying messaging solution similar to that JDBC allows the exchange of the underlying DBMS it is possible to plug in other messaging solutions.
Using JMS the persistence problem is solved by the JMS provider which keeps messages in non-volatile storage until the transacted consumer signalizes that the business process has received and persisted the message successfully. Additionally guaranteed delivery is easier to achieve using JMS since the message provider keeps the message on the queue or topic until it has been consumed and the consumption has been committed. While the sender (i.e. a BPEL `invoke` activity) can use a "send and forget" approach with JMS it must retry sending messages until it receives an acknowledgment if WS-RM is used.
In an additional step the JMS layer could easily be modified to send messages not by transmitting their SOAP XML representations but by sending Java objects over the JMS transport. This would remove the necessity to serialize and deserialize the messages into XML and therefore improve performance.

5.2 Extraction of Transaction Information from the BPEL Engine

As a BPEL engine the ActiveBPEL engine version 1.2 is used for this prototype. The ActiveBPEL engine is an open source BPEL engine. It has been chosen because source code is available under [1] and other projects at the Institute of Architecture of Application Systems at the University of Stuttgart have been realized using this ActiveBPEL engine [50] [36]. The ActiveBPEL engine is implemented in the Java programming language.

5.2.1 Requirements

The out-of-the-box ActiveBPEL engine controls and maintains its transaction and fault-handling information internally. Based on the events and messages specified in Chapter 4 the ActiveBPEL engine has been modified to send and receive these messages reliably over JMS. Since the external coordinators are implemented in BPEL and the ActiveBPEL engine already provides a means to serialize and deserialize SOAP messages, the message format for the life cycle messages is SOAP.

5.2.2 Modified Parts of the ActiveBPEL Engine

The following parts of the ActiveBPEL engine have been changed in order to comply with the requirements for a PureBPE presented above.

- In the navigator mainly the classes representing scopes, fault and compensation handlers, have been modified. All behavior necessary for transaction processing has been extracted from these classes and has been replaced by calls to methods in a new *JMSTransactionGateway* class that sends and receives the life cycle messages necessary to control scopes from the outside.
- The persistence layer has been modified in order to save additional information for scopes, such as a coordination context and scopeId data.
- The configuration routine of the ActiveBPEL engine has been modified so that the external transaction processing can be configured using the common configuration files for an application inside a servlet container, the `web.xml` file.
- The invoke handler that handles the invocation of external partners has been modified to include WS-Addressing reference properties from a partner link's endpoint reference in the SOAP header of the invocation SOAP message as specified in WS-Addressing [15].
- Additionally the invoke handler includes the coordination context of its enclosing scope in the SOAP header of the outgoing message, if the `invoke` activity that has started the handler is located inside an implicit scope.
  In a production version an inclusion of the coordination context based on WS-policy assertions of the external partner is a more flexible solution. However, this is out of the scope of this thesis.
- The request processing of the ActiveBPEL engine has been modified in a way that it can start business processes under a coordination context. Upon execution of a request that triggers the creation of a new business process instance the engine now checks whether a coordination context is submitted in the SOAP header of the request. If yes, the business process instance registers as a new participant of this context.

The following parts of the ActiveBPEL engine have been modified to accept and send Web-Service invocations via JMS in both, the PureBPE and the BPEL engine running the external coordinators

- The process deployment descriptors have been enhanced to accept JMS destinations to be specified as service endpoint addresses.
- The deployment routine of the ActiveBPEL engine has been modified in a way to automatically start JMS consumers for the JMS destinations specified in the enhanced deployment descriptors.
- The invocation of external partners has been modified in order to enable asynchronous invocation of external partners via SOAP messages sent over JMS.
- A means to handle incoming JMS messages and forward them to the engine has been implemented.

5.2.3 Transaction Gateway

The sending and receiving of JMS scope life cycle Messages has been encapsulated from the code that processes the BPEL objects by the newly created transaction gateway which is implemented using the `JMSTransactionGateway` class. The `JMSTransactionGateway` class acts as a *Messaging Gateway* as discussed in [32]. The `JMSTransactionGateway` class implements the `ITransactionGateway` interface. Using the `ITransactionGateway` interface, allows other transaction gateways than the SOAP over JMS transaction gateway implemented in this prototype to be plugged in.
In the further document the term "transaction gateway" always means the transaction gateway that uses SOAP over JMS messages and is implemented by the `JMSTransactionGateway` class, unless explicitly said otherwise.

Figure 5.3 shows where the transaction gateway is located in the architecture of the PureBPE.

5.2.3.1 Requirements for the JMSTransactionGateway

The JMS transaction gateway must implement the `ITransactionGateway` interface which provides methods for the ActiveBPEL engine to call if certain events in the life cycle of scopes, fault handlers or compensation handlers occur as specified in Chapter 4. Upon invocation of these methods it must assemble the corresponding messages and call the appropriate methods of the Axis stub for the external coordinato (the `BPEL4Tx` stub) so it can send the actual SOAP messages.
Additionally the `ITransactionGateway` interface provides methods that can be called from the outside via messaging in order to enable the BPEL4Tx coordinator to invoke life cycle events on the PureBPE. The `PureBPESkeleton` class gets

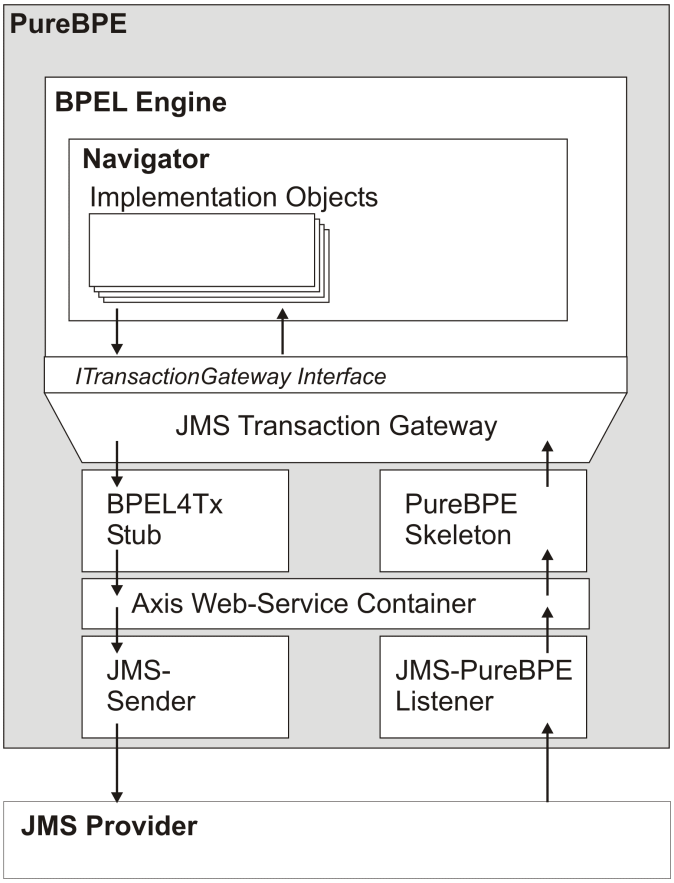

Figure 5.3: Architecture of the PureBPE.

called by the Axis layer on reception of life cycle messages and invokes the necessary methods of the transaction gateway.
Therefore another task of the `JMSTransactionGateway` class is deserializing and interpreting incoming messages and correlating scopeIds in incoming messages to instances of `AeActivityScopeImpl` objects.

5.2.3.2 Architecture of the JMSTransactionGateway

The `JMSTransactionGateway` class is implemented using the singleton pattern [24]. The singleton instance is initialized on startup of the PureBPE ActiveBPEL engine. On startup of the transaction gateway it starts a JMS consumer (the JMS PureBPE-Listener) that waits for messages on the JMS destination on which incoming scope life cycle messages arrive. This destination can be configured in the `web.xml` configuration file for the PureBPE.

5.2.3.3 Member variables

The `JMSTransactionGateway` class gateway contains a set of member variables that allow the correlation of scopeIds that are used in the incoming and outgoing messages and instances of objects used inside the ActiveBPEL engine.

mScopeStatusMap In order to manage the correlation between scopeIds and instances of `AeActivityScopeImpl` objects the `JMSTransactionGateway` class maintains a `java.util.HashMap` that contains `AeActivityScopeImpl` objects that are keyed by their scopeIds. On initialization of a scope its `AeActivityScopeImpl` instance is added to the map. On receipt of an `endTxForScope` message for a scope it is removed from this map.

mScopeCompensationMap If a scope completes its normal processing without a fault, BPEL specifies that compensation information has to be maintained until the scope is compensated or all its parent scopes have completed or faulted.
In the existing ActiveBPEL engine implementation this compensation information is maintained with the enclosing scope of the scope that has completed without a fault. In the modified implementation it is necessary to correlate compensation information with scopeIds, since compensation is controlled externally. On receipt of an `execLCHandler` message that specifies a `compensationHandler` as the handler to be executed, compensation information needs to be retrieved based on the scopeId contained in the `execLCHandler` message. Therefore an instance of the `AeCompInfo` class, representing the compensation information for a scope, is built and added to the `mScopeCompensationMap` after a scope has finished its normal processing mode without a fault.

mScopeActivityMap The `mScopeActivityMap` maintains correlation between activityIds and instances of AeActivityImpl objects. This is necessary if external activities, such as a compensate activity are executed. After finishing this activity, with or without a fault, the transaction gateway must be able to perform a callback on the right activity that it has finished or faulted. In order to find the right activity based on the activityId contained in the respective messages the `mScopeActivityMap` is needed.

5.2.3.4 Methods of the JMS Transaction Gateway

The main methods of the `JMSTransactionGateway` class are described in the following paragraphs.

GenerateScopeID This method creates a unique scopeId for a scope.
The scopeId is a string consisting of three parts, the process instance part, the scope part and the time part. The process instance part of the scopeId is generated by concatenating the String "PID" and the process Id that is generated for every scope by the ActiveBPEL engine.
The scope part of the scopeId is generated by concatenating the String "LID" with the location Id of the scope. The location Id of a scope denotes its Xpath [46] position in the process. A location Id of 5 for a scope denotes that the scope is the 5th activity in the process.
The third part of the scopeId is the time part, it is simply the time in milliseconds at which the scope was initialized concatenated with the string "MS".
These three parts form a unique scopeId that is unique across multiple process instances, due to the process Id. The process Id is even unique across multiple ActiveBPEL engines in a cluster (cluster features are only available for commercial versions of the ActiveBPEL engine). The scope part of the scopeId makes the scopeId unique among all scopes in a process. However, if a scope is inside a `while` activity it might be instantiated several times. It is still at the same position in the XML file therefore its location Id is always the same. But considering the fact that the next iteration of a while scope occurs later than the current iteration, the time part of the scopeId makes the scopeId unique even in case of nested scopes inside a while loop.

Life cycle Methods and Messages

InitScope This method gets called when the PureBPE wants to put an `AeActivityScopeImpl` instance into the execution queue (i.e. wants to execute a scope). Instead of putting the scope directly into the execution queue as it is done in the normal ActiveBPEL implementation, the `initScope` method is called and the `AeActivityScopeImpl` instance of the scope is passed to this method. This behavior is necessary since the PureBPE engine needs to signalize the external coordinator that a scope has initialized and it needs to wait until this scope has registered with the coordinator.
The `initScope` method adds the `AeActivityScopeImpl` instance to the `mScopeStatusMap`. Then it creates an `InitScopeType` object, representing an `initScope` message. The `InitScopeType` takes 5 parameters. The scopeId of the scope which is generated by a call to `generateScopeID`, the scope name which is extracted from the XML-representation of the scope, the coordination contex of the parent scope which is retrieved by calling the `getCoordinationContext` method of the parent scope (if the scope has one), the parent scopeId which is obtained by calling the `getScopeID` method of the parent scope and the scope definition which is the XML-representation of the scope.
The XML-representation of scope objects is saved with the corresponding `AeActivityScopeDef` class instance on deployment of a new process. This behavior has been added to the active BPEL engine.

After invoking the BPEL4Tx stub in order to send the `initScope` message, the PureBPE stops executing this scope until it receives an `initScopeFinished` message for the scope.

InitScopeFinished This method gets called by the Axis PureBPE skeleton if it receives an `initScopeFinished` message. It is the answer to the `initScope` message sent by the PureBPE. On calling of this method the the corresponding `AeActivityScopeImpl` instance from the `mScopeStatusMap` is retrieved based on the scopeId passed in as a parameter. The method then calls the `setCoordinationContext` method of the `AeActivityScopeImpl` instance if a coordination context was returned by the coordinator. This is the case if the scope which has been registered is itself a WS-C activity which means that it has child scopes.
After having set the coordination context, the `initScopeFinished` method puts the `AeActivityScopeImpl` instance into the execution queue. The ActiveBPEL engine can now continue to process the scope normally, i.e. dequeue it and execute it's child activity.

NpmEnded When a scope completed its normal processing mode, i.e. its child activity completed without a fault, the external coordinator needs to be notified of this successful completion. Therefore the behavior of the ActiveBPEL engine is modified. Instead of telling the enclosing activity of the scope that it has completed, the `AeActivityScopeImpl` instance representing the scope, calls the transaction gateway and notifies it that it has completed via the `npmEnded` method. The transaction gateway invokes the BPEL4Tx stub so that it sends a `npmEnded` message containing the scopeId of the scope that has completed.

NpmEndedFinished After this method got called by the Axis PureBPE skeleton, the method signalizes the scope corresponding to the scopeId passed, that it can notify the business activity of its completion. This is done by calling the `objectCompleted` method of the `aeBusinessProcess` instance for the scope. Afterwards the outgoing links of the scope are evaluated.

NpmFaulted This method is similar to `npmEnded` but gets called when a scope completed its normal processing with a fault. The external coordinator must be notified of this fault. A fault handler can only be executed for this scope after the external coordinator has sent an `npmFaultedFinished` message. Therefore the `npmFaulted` method builds a new `FaultType` object that contains the scopeId, a serialized XML representation of the fault as well as the fault name. The message is then sent via the appropriate method in the BPEL4Tx stub. The ActiveBPEL engine stops execution of the scope. In the normal ActiveBPEL engine all nested

activities of the scope are terminated on occurrence of a scope. This is a bit different in the modified ActiveBPEL engine since nested scopes can only be terminated by the external coordinator. Therefore only nested non-scope activities are terminated before the BPEL4Tx stub is invoked.
Child scopes and their nested activities are terminated by the external coordinator which injects `forcedTermination` faults into to these scopes.

NpmFaultedFinished After the external coordinator has been notified of a fault in a scope via the `npmFaulted` method and has finished all necessary processing that is necessary after such a notification, it responds with a `npmFaultedFinished` message that causes the Axis PureBPE skeleton to call the `npmFaultedFinished` method of the transaction gateway. This method notifies the process that the scope has faulted via the `objectCompletedWithFault` method which in turn tells the enclosing activity that the scope has faulted.
After having been notified by the external coordinator that all nested scopes are terminated, the scope can execute its fault handlers. If there is no fault handler defined an implicit default fault handler has to be executed. This implicit fault handler is not executed by the PureBPE itself but by the external coordinator. Therefore an additional method in the transaction gateway is necessary that is called by the modified ActiveBPEL engine. The method `execDefFH`. In case the method `findOrCreateMatchingFaultHandler` returns an explicit fault handler, the corresponding `AeFaultHandler` instance is put in the execution queue.

ExecDefFH This method gets called if the modified ActiveBPEL engine needs to execute a fault handler but none is defined for the scope in the BPEL process. The method calls the `execDefFH` method of the BPEL4Tx stub to send the `execDefFH` message to the external coordinator.

FhEnded After a fault handler has completed without throwing a fault, the corresponding `AeFaultHandler` instance calls the `fhEnded` method of the transaction gateway. This happens only in case the fault handler is not an implicit fault handler. Implicit fault handlers are controlled externally and therefore the coordinator does not need to be notified since it already knows when an implicit fault handler ends.
The `fhEnded` method calls the `fhEnded` method of the BPEL4Tx stub that sends out the `fhEnded` message to the coordinator.

FhEndedFinished After the PureBPE skeleton received a `fhEndedFinished` message it calls the `fhEndedFinished` method of the transaction gateway. This method tells the scope for which the fault handler has been executed that the handler has completed normally. The `AeActivityScopeImpl` class representing the scope can now notify the process that it has completed (albeit with a fault handled by a fault handler).

FhFaulted This method is similar to the method `fhEnded`. It gets called by the `AeFaultHandler` instance that has faulted and submits an XML representation of the fault to the external coordinator.

FhFaultedFinished The external coordinator responds with a `fhFaultedFinished` message if a fault handler has faulted. This includes the faulting of an implicit fault handler. An XML representation of the fault is returned with this message. The PureBPE skeleton then calls the `fhFaultedFinished` method of the transaction gateway. When the `fhFaultedFinished` method is called it deserializes the fault specification from the message and builds an `AeFault` object instance from it. The transaction gateway then notifies the corresponding `AeActivityScopeImpl` instance that the fault handler has faulted and passes the fault that occurred. The scope then notifies the process that it has faulted. Instead of calling the `handleFault` method of the enclosing scope with the fault that occurred the PureBPE stops execution of the scope. Rethrowing of faults is done by the external coordinator by sending `injectFaultInScope` messages to the enclosing scope.

InjectFaultInScope This method is called by the PureBPE skeleton on receipt of an `injectFaultInScope` message. When the `injectFaultInScope` method is called it deserializes the fault specification contained in the message and builds an `AeFault` object instance from it. After that, the `handleFault` method of the `AeActivityScopeImpl` instance representing the scope that should fault, is called and the fault is passed as a parameter. After that method has been called the fault handling is the same as if the child activity of a scope had faulted.

InjectedFaultInScope After having injected the fault into the scope the transaction gateway calls the `injectedFaultInScope` method of the BPEL4Tx stub which sends out the `injectedFaultInScope` message to notify the coordinator of the injection.

ExecActivity If during the processing of a scope an activity occurs that can not be processed by the PureBPE, it calls this method of the transaction gateway. Activities that are executed externally must implement the `IExternalActivity` interface through their abstract `ExternalActivity` superclass. In the scope of this thesis only the `compensate` activity does so.
In further iterations so-called `ExtensionReaders` could be implemented that process BPEL extensions that can then be executed by the external coordinator. On calling of the `execute` method the external activity object instance calls the `execActivity` method of the transaction gateway and then the PureBPE waits for either an `execActivityEnded` message or an `execActivityFaulted` message. Inside the `execActivity` method a unique identifier for the activity is created.

This is done by concatenating the scopeId of the enclosing scope of the activity with the string "AID" and the activity's locationId. This activityId is then passed along with the XML representation of the activity to the Axis stub which sends out an `execActivity` message. The activityId along with the activity implementation object is stored in the `mScopeActivityMap`.
The necessity for the activityId arises from the fact that a more fine-grained identifier below the scopeId is needed to correlate `execActivityEnded` or `execActivityFaulted` messages with Java objects representing the external activity since a scope can contain more than one external activity.

ExecActivityEnded This method gets called by the PureBPE skeleton on receipt of an `execActivityEnded` message. The task of this method is to notify the corresponding scope that an activity that was executed externally has finished without a fault. Therefore the transaction gateway retrieves the appropriate activity implementation object from the `mScopeActivityMap` and calls its `objectCompleted` method. Then the activity is handled as if it were a normal activity that has been executed internally. For example links that leave the activity are updated, before the business process is notified that the activity has completed.

ExecActivityFaulted This method gets called by the PureBPE skeleton on receipt of an `execActivityFaulted` message. Instead of calling the `objectCompleted` method it calls the `objectCompletedWithFault` method passing the fault that has been submitted with the `execActivityFaulted` message. Then the activity is treated like a normal faulted activity.

ExecLCHandler On receipt of an `execLCHandler` message the PureBPE stub calls this method. The method interprets the passed handler variable and executes the handler specified in it for the corresponding scope. In this thesis the only life cycle handler available is the compensation handler. If the handler variable is of the value "compensationHandler" the `AeCompensationHandler` instance for the scope identified by the scopeId is associated with the scope's snapshot data and is put into the execution queue. Then the PureBPE treats the compensation handler as if it was started directly.

LchEnded A life cycle handler has finished without a fault, then it calls this method. This method then notifies the external coordinator of the successful completion of the life cycle handler via the `lchEnded` method of the BPEL4Tx stub.

LchFaulted If it faults, a life cycle handler calls this method of the transaction gateway and passes the fault that occurred as a parameter. The transaction gateway then notifies the external coordinator of the faulting of a life cycle handler.

This is done by calling the appropriate method in the BPEL4Tx stub and passing the serialized representation of the fault.

DisableScope The coordinator might want to disable all links that leave a scope. This is necessary if a scope's fault handler faulted or an implicit fault handler ended on a scope. To achieve this behavior the coordinator sends a `disableScope` message. The transaction gateway's method `disableScope` inspects all outgoing links of the scope and sets them to dead path. After having done so, it calls the transaction gateway's `disabledScope` method.

DisabledScope After all outgoing links of a scope have been disabled this method is called. It invokes the BPEL4Tx stub's `disabledScope` method which sends a `disabledScope` message to the coordinator to signalize the disabling of outgoing links.

EndTxForScope When the coordinator has finished all work it needs to perform on a scope, it sends an `endTxForScope` message. On receipt of such a message the PureBPE skeleton calls the `endTxForScope` method of the transaction gateway. This method inspects if compensation information for the scope is available and if yes discards it since it is no longer needed. Additionally the scope is deleted from the `mScopeStatusMap`, all other entries in maps of the `JMSTransactionGateway` corresponding to the scope that has finished, are deleted too. If the scope that correlates to the scopeId in the scopeId parameter of the `endTxForScope` method is the process, the entire process has finished. Therefore the `endTxForScope` method of the `AeBusinessProcess` class overrides the `endTxForScope` method of the `AeActivityScopeImpl` class. The `AeBusinessProcess` class's `processEnded` method is called which signalizes the engine that the process has finished.

EndedTxForScope This method gets called after a scope has received an `endTxForScope` message and has taken all necessary steps to process it. Via the BPEL4Tx stub's `endedTxForScope` method the coordinator is notified of this event.

5.2.4 Additional States

Using an external coordinator several situations where the PureBPE must wait for answers from the coordinator occur. The problem with asynchronous messaging in situations like this is if one of the participant crashes before the answer arrives. If the external coordinator crashes this does not impose a problem for the PureBPE. It simply waits until the external coordinator has recovered and sends the answer.

This of course implies that the external coordinator is somehow forward recoverable and reliable messaging is used. This is the case in the prototype since JMS with persistence is used and the external coordinator is essentially a BPEL process and the ActiveBPEL engine it runs in is forward recoverable when persistence is turned on.
If the PureBPE crashes and reliable messaging is used the answer message is kept in the messaging system until the PureBPE has recovered. The PureBPE then processes the answer message. In order to process the message correctly the PureBPE needs to know that it has send a message that is answered by the answer message. If not, it is not acceptable to process the answer message.
But how does the PureBPE know if it has sent the message the answer corresponds to or not? One possibility would be to save all the messages which the PureBPE engine has sent and then check if the corresponding request to the answer message has been send. This would need a persistent storage of all outgoing messages, even if many of them are no longer needed.
Since the ActiveBPEL engine already employs the concept of states for the scopes in its normal implementation, the concept to introduce additional states, according to the scope life cycle identified in Section 2.2.1.1, for scopes is quite straightforward.
Since states are automatically persisted by the ActiveBPEL engine, a scope immediately knows in which state it is in after recovery. That means that it also knows which messages it has already send and which it has not send yet.

Figure 5.4 shows the existing states of the ActiveBPEL engine as well as the newly introduced states. The additional states shown in Figure 5.4 are only necessary for two BPEL objects, processes and scopes. Since the `AeBusinessProcess` class that represents a process extends the `AeActivityScopeImpl` class, that represents a scope only the `AeActivityScopeImpl` class has to be modified. The third case where external coordination and therefore the additional states are needed, are implicit scopes. Implicit scopes are `invoke` activities with defined compensation and/or fault handlers.
The ActiveBPEL engine maps these invoke activities to a scope that contains a single invoke activity on deployment. Therefore it is not needed to modify the `AeActivityInvokeImpl` class that represents invoke activities. The external coordination is handled by the enclosing `AeActivityScopeImpl` class.
The following paragraphx explain the new states.

INITIALIZING_SCOPE This state follows `QUEUED_BY_PARENT`. Instead of transitioning directly to state `READY_TO_EXECUTE` [2], scopes states transition to `INITIALIZING_SCOPE` after calling the `initScope` method of the transaction gateway. Having received the `initScopeFinished` answer the scope is set to state `READY_TO_EXECUTE` and then when it's `execute` method is called it is set to `EXECUTING`.

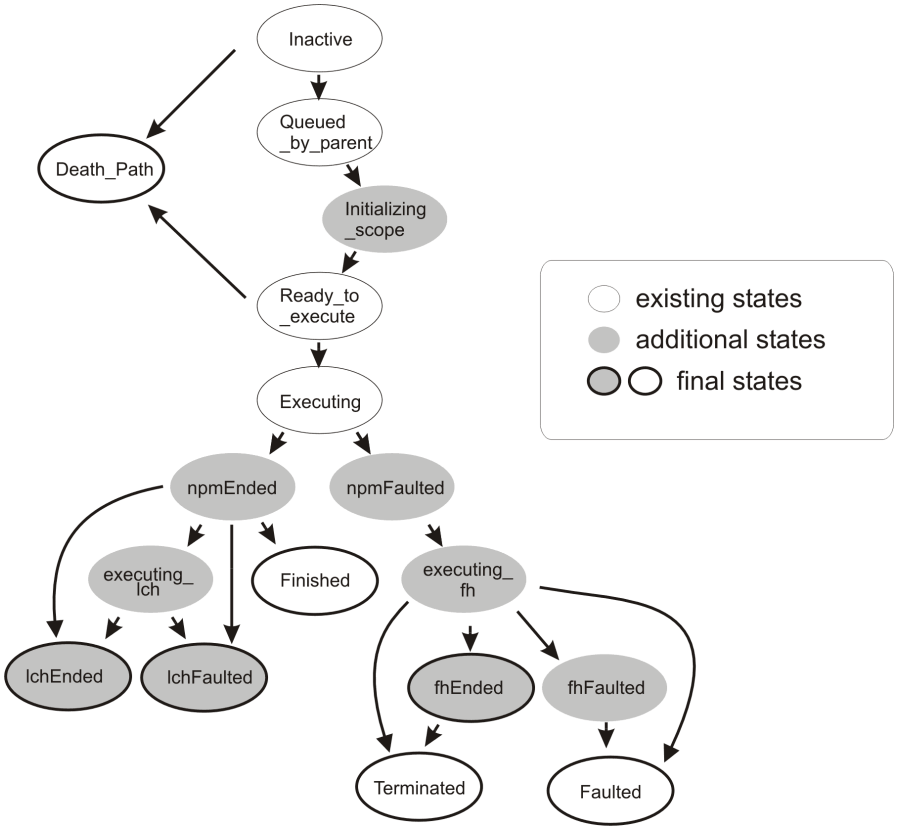

Figure 5.4: New scope states in the ActiveBPEL Engine.

NPM_ENDED A scope is in state `EXECUTING`. After calling the method `npmEnded` of the transaction gateway the scope transitions to the state `NPM_ENDED`. After receiving an `NPMEndedFinished` message the scope is in state `FINISHED` and waits for a close or compensate information.

NPM_FAULTED If a fault occurs in a scope in state `EXECUTING` and the coordinator has been notified via the `npmFaulted` method of the transaction gateway the scope is in state `NPM_FAULTED`.

EXECUTING_FH During the execution of a fault handler the scope is in state `EXECUTING_FH`. The scope's state transitions to `FH_Ended` or `FH_Faulted` when the fault handler is an explicit fault handler and completes or faults. In case the fault handler is an implicit default fault handler the next message to arrive at the PureBPE is the `fhFaultedFinished` message. Then the state transitions directly to `Faulted` or `Terminated` if the fault was a `forcedTermination` fault.

FH_ENDED If a fault handler of a scope terminates without a fault the scope transitions to the state `FH_ENDED`. It stays in that state until it sees

a `fhEndedFinished` message. The scope's state then changes to `Faulted` or `Terminated` if the fault it handled was a `forcedTermination` fault.

FH_FAULTED If a fault occurs in a fault handler of a scope, the scope the fault handler belongs to is set to the state `FH_Faulted`. Until the method `fhFaultedFinished` of the transaction gateway has been called the fault remains in this state. The `fhFaultedFinished` method changes the state of the scope to `FAULTED`.

EXECUTING_LCH A scope is in state `EXECUTING_LCH` during the time it is executing an explicit life cycle handler (such as a compensation handler). In case a scope has a life cycle handler that is completely executed by the external coordinator (such as an implicit compensation handler) the scope's state does not transition to this state but changes directly from `NPM_ENDED` to `LCH_ENDED` or `LCH_FAULTED`.

LCH_ENDED A scope is in this state if a life cycle handler (for example a compensation handler) has successfully completed on a scope.

LCH_FAULTED A scope is executing a life cycle handler. If this life cycle handler fails, the corresponding scope is set to state `LCH_FAULTED`.

5.3 Adding JMS to the ActiveBPEL Engine

Controlling BPEL long-running transactions externally requires the sending and receiving of scope life cycle messages as described in Chapter 4. As motivated in Section 5.1.1.2 JMS is used for the exchange of these messages. In the prototype for this thesis two different parts need JMS support.
Firstly the PureBPE needs to send and receive scope life cycle messages. Secondly, the external controllers are implemented in BPEL and run in a normal ActiveBPEL engine. This engine controls its transaction information internally so it does not need to send and receive JMS messages in order to support the external controlling of transaction processing. However, it must receive JMS messages from and send JMS messages to the PureBPE. Therefore the normal active BPEL engine needs JMS capabilities too.
While the PureBPE needs to be able to send and receive JMS messages independent of the normal message handling for BPEL processes, the ActiveBPEL engine running the coordinator needs a means to handle JMS messages via the normal SOAP message handlers because the coordinator is implemented as a BPEL process. Therefore it must be possible for the coordinator BPEL engine to send JMS messages via BPEL `invoke` activities and receive JMS messages that are used

to trigger `receive` activities and `onMessage` events in the BPEL-process instances that represent the coordinator.

5.3.1 JMS Features for the PureBPE

The JMS features for the PureBPE have to be separated from the normal message handling of the PureBPE. This is necessary since normal messages are interpreted as messages which trigger `receive` activities and `onMessage` events in business processes while scope life cycle messages trigger life cycle events in scopes.
Figure 5.3 shows the architecture overview. The transaction gateway discussed in the previous section calls the BPEL4Tx stub which in turn invokes the AXIS layer that builds the SOAP XML message and hands it over to the JMS handler. The JMS handler then wraps the SOAP XML message into a JMS text message and hands it over to the JMS provider which puts it onto the required JMS destination. On the other "end" of the destination the `JMSPureBPEListener` acts as a *polling consumer* [32] which receives messages from its associated queue, and hands them over to the Axis layer.
Polling consumer means that only after one message has been successfully consumed the next message is taken from the destination. This behavior prevents a throttling of the PureBPE if too many messages receive to be handled in parallel by the engine.
The Axis layer deserializes the SOAP XML messages and invokes the corresponding methods in the PureBPE skeleton. The PureBPE skeleton's methods then call the methods of the transaction gateway.

5.3.2 JMS Features for the Normal ActiveBPEL engine

As discussed above the ActiveBPEL engine which runs the coordinator processes needs to be able to receive JMS messages and forward them to the right process instance. Furthermore, a means is needed to send JMS messages from a BPEL process.
The following steps are needed to meet the requirements:

- Establishing of a protocol that allows to specify that messages sent through a BPEL `invoke` activity are sent as SOAP over JMS instead of SOAP over HTTP.

- Implementation of a new Axis handler that forwards requests that have been marked to be sent via JMS to the JMS provider which in turn forwards it to the right JMS destination.

- Implementation of a JMS listener which listens to a JMS destination that can be associated with a service. Messages arriving on this destination should be forwarded to the handlers of the BPEL engine which forward the messages to the right business process instance.

5.3.3 JMS URL Syntax

In order to send SOAP messages over JMS instead of HTTP a mechanism is needed to specify that a message should be sent via JMS.
Using HTTP, the address of the receiver of the message is specified as follows: `http://localhost:8080/active-bpel/services/BPEL4TxService`. This URL specifies the `BPEL4TxService` on machine `localhost` at port `8080` under `active-bpel/services`.
Consequently, if a service needs to be called via JMS a similar means is needed that specifies the following properties:

- The "transport protocol" must be JMS. (JMS itself is not a transport protocol it can use several transport protocols such as HTTP itself but this is transparent to the JMS client).
- To which JMS destination the message should be sent.
- If the JMS destination is a queue or a topic.
- Which JMS provider is to be used.
- The address of the JMS provider.

As a syntax for this URL the syntax proposed in [20] is used. As proposed in [20], a JMS URL is built after the schema shown in Figure 5.5:

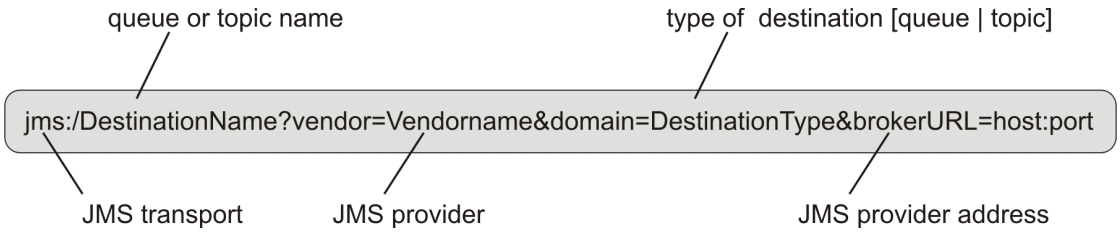

Figure 5.5: JMS-URL Syntax

The `jms:/` part of the URL signalizes that JMS should be used as a transport protocol.
The `DestinationName` path specifies the JMS destination on which the message should be put. Whereas the `domain` property in the query string denotes if the destination is a queue or a topic. Subsequently it can be of the value `queue` or `topic`.
The JMS provider to be used can be specified using the `vendorname` property in

the query string.
The `providerURL` properties denotes the address of the JMS provider specified in the `vendorname` property. The following example shows a WSDL port definition which states that the port with the name "BPEL4TxEnginePLinkTypeService" is reachable over JMS via the `BPEL4TxQueue` queue which is provided by an ActiveMQ JMS provider that can be reached on port 61616 on `localhost`.

```
<wsdl:port name='BPEL4TxEnginePLinkTypeService'
           binding='tns:BPEL4TxEngineSoapBinding'>
  <wsdlsoap:address
    location='jms:/BPEL4TxQueue?vendor=activemq&amp;
              domain=queue&amp;vendorname=ActiveMQ&amp;
              brokerURL=localhost:61616'/>
</wsdl:port>
```

Listing 5.1: WSDL port definition with a JMS URL.

If WS-Addressing is used to specify the endpoint reference the value of the `wsa:Address` element is simply set to the JMS URL as in the following example.

```
<wsa:EndpointReference xmlns:s='someNamespace'>
  <wsa:Address>jms:/restOfTheJMSURL</wsa:Address>
  <wsa:ServiceName PortName='portName'>
   ServiceName
  </wsa:ServiceName>
  ...
</wsa:EndpointReference>
```

Listing 5.2: WS-Addressing endpoint reference with a JMS URL.

5.3.4 New Axis Handler to Send JMS Messages

The previous section showed a means to tell the ActiveBPEL engine to send messages via JMS. In order to actually send SOAP messages over JMS a few more modifications to the ActiveBPEL engine are necessary. These modifications follow the proposal of using Axis with message oriented middleware and JMS made in [42].

5.3.4.1 Invoke Handling in the ActiveBPEL engine

On execution of an `AeActivityInvoke` instance, that represents a BPEL `invoke` activity internally, the ActiveBPEL engine uses an instance of the class `AeInvokeHandler` to built the message that invokes the partner link specified in

the `invoke` activity and hands it over to the Axis layer. The Axis layer then serializes the message to an XML SOAP message and sends it to the specified address. The class `AeInvokeHandler` has been modified so that it can interpret the address URL and recognizes if the URL starts with `jms:/`. If this is the case and the vendor name is set to `ActiveMQ`, the `AeInvokeHandler` sets the `Transport`-property for the Axis call to `JMSAMQTransport`. This `Transport`-property tells the Axis layer that it must use the `JMSAMQTransport` to send this message. In the deployment descriptor for the Axis layer the `JMSAMQTransport` is defined to use the class `JMSSender` to handle messages that use this transport.

5.3.4.2 The JMSSender Axis-Handler

The class `JMSSender` extends the `org.apache.axis.handlers.BasicHandler` class. That means the Axis layer calls the `invoke` method of `JMSSender` if it receives a call whose `Transport`-Property has been set to `JMSAMQTransport`, since the class `JMSSender` is specified as the handler class for `JMSAMQTransport`. The class `JMSSender` establishes a connection to the provider specified in the JMS URL in its `invoke` method. It then registers as a message producer for the specified destinations. After building a JMS text message which contains the SOAP message as the text part, the message is put on the destination.
In order to be able to forward the message to the right service later, the JMS-String-Property `AxisService` is set to the service name specified in the `wsa:EndpointReference` for the partner link of the invoke activity.
Additionally the `AeInvokeHandler` has been extended to include the reference properties defined in the `wsa:ReferenceProperties` in the SOAP header of the message to be sent to the external partner. The inclusion is done as specified in the WS-Addressing specification: The child elements of `wsa:ReferenceProperties` are copied literally to the SOAP header of the message.

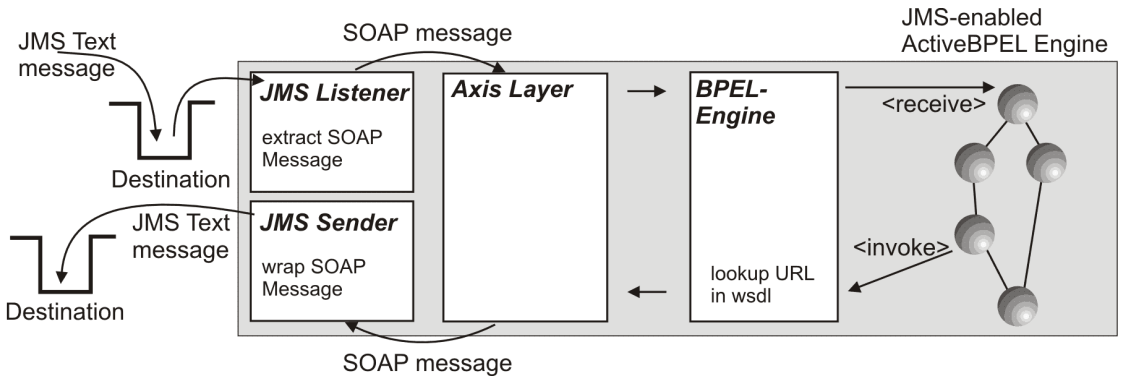

Figure 5.6: Architecture of the JMS-enabled ActiveBPEL engine.

5.3.5 JMS Listeners

The class `JMSListener` is the counterpart of the class `JMS-Sender` described in Section 5.3.4.2.
A `JMSListener` instance waits for messages that arrive on a JMS destination. The destination can be a queue or a topic. The destination is specified in the deployment descriptor (PDD file) for a process. For every service a process provides, a JMS URL, as discussed in Section 5.3.3, can be specified. Therefore the ActiveBPEL engine deployment descriptor's `myRole` element was modified. It now accepts an `address` attribute that allows the specification of a URL under which the service should be reachable. On deployment of a BPEL process (and its associated PDD file) an Axis web service deployment descriptor (WSDD file) is created by the ActiveBPEL engine. The class `AeWSDDService` reads a PDD file for a process and uses the `AeWSDDBuilder` class to build a WSDD file from the PDD file. Both classes have been modified so that they can read the `address` attribute from a `myRole` element inside a PDD file. The value of the `address` attribute (i.e. the JMS URL) is then added to the WSDD file as an `endpointURL` element. According to the Axis specification [7] the `endpointURL` element denotes the URL on which the service that contains it should be reached after deployment.
Figure 5.7 shows the deployment process of a PDD file that contains an address attribute for one of its partner links. After the WSDD file has been created from the PDD file it is then deployed to the Axis web service container. It contains the specification of all the services that a process wants to expose to the outside via the Axis layer. This file is deployed via the `AeAxisWebServicesDeployer` class. This class has been modified for this prototype in the following way: If it detects that a WSDD file contains one or more `endpointURL` elements it evaluates the values of these elements and creates a `JMSListener` thread for every service that contains an `endpointURL` element with a JMS URL value. Every `JMSListener` thread that is created gets the JMS URL for its service as a parameter in its constructor.
Having created all necessary `JMSListener` threads the WSDD file is deployed to Axis. After successful deployment, the `JMSListener` threads are started. On startup of a `JMSListener` thread the `JMSListener` instance connects to the destination specified in the JMS URL and waits for messages on this destination.
The `JMSListener` class is implemented using the *polling consumer* and *transactional client* patterns discussed in [32].

After the JMS listener has received a message from the queue it unwraps the message by taking out the text part of the JMS text message. This text part is the SOAP message that gets handed over to the Axis layer.
Before the Axis layer can be invoked with the message a Axis new message context has to be initialized for the message. The message context also specifies the service that should be invoked with the message. This service is set to the service submitted by the `AxisService` string property of the JMS Text message. After setting the message context the message and its context are handed over to the

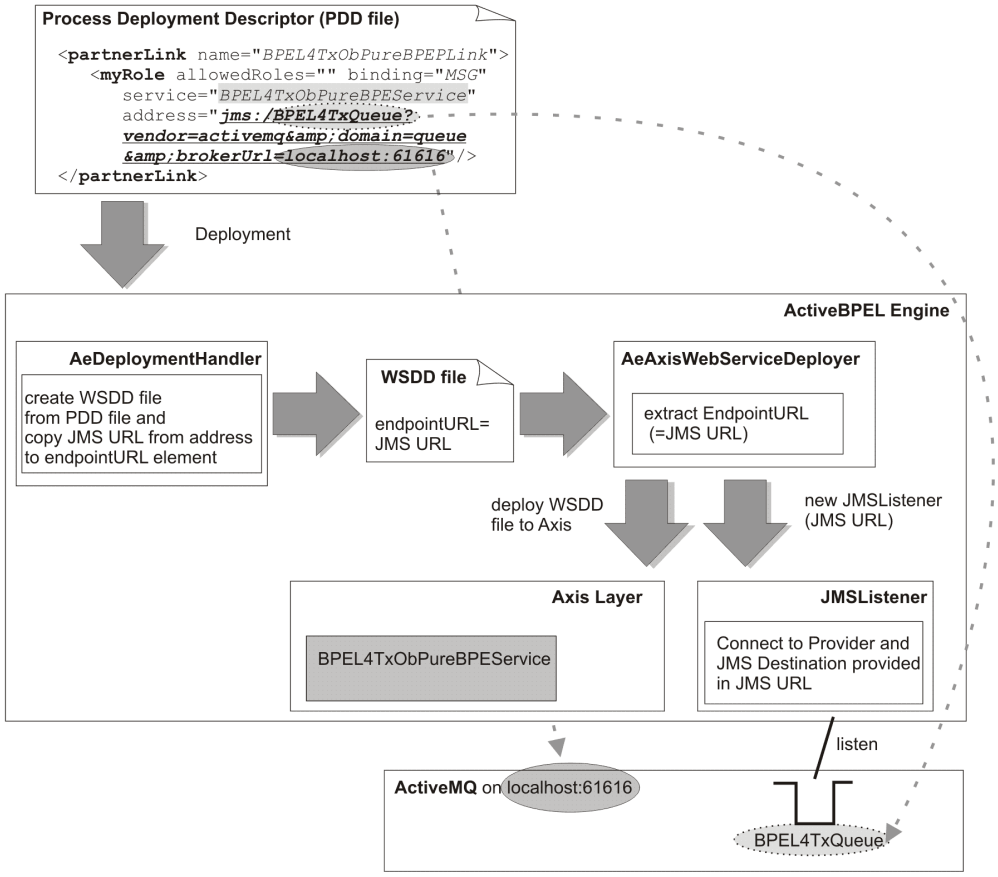

Figure 5.7: Deployment of a BPEL service reachable via JMS.

Axis layer. The Axis layer treats them as if they were received by HTTP. This means that from now on the normal SOAP message handling algorithm of the ActiveBPEL engine takes over and ensures that correlation properties are evaluated and the message is sent to the right process instance and triggers the right `receive` or `onMessage` activity of the process.

In order to ensure that messages are only taken from the JMS destination by the JMS listener if they can be handled by the Axis layer the `JMSListener` class implements the transactional client pattern. This means that the JMS session established between JMS listener and JMS provider is transacted. The session is only committed if the invocation of the Axis handler is successful. If the Axis layer returns a fault because it cannot handle the message the session is rolled back and the message remains on the queue. The transacted sessions ensure that messages do not get lost if they cannot be handled by the ActiveBPEL engine (maybe because the appropriate service is in the process of being redeployed) and can be redelivered later when the ActiveBPEL engine might be ready to receive them.

For every JMS URL only one `JMSListener` instance is started. This is necessary since different services could specify the same JMS Destination. Since all messages are handed over to the Axis layer after receipt by the `JMSListener` instance

anyway, one listener per destination is enough. This design prevents that multiple `JMSListener` instances consume messages from the same destination which would be a problem with JMS topic destinations. In case that several JMS listeners would listen on the same topic and a message would be sent to this topic every listener would deliver the message to the ActiveBPEL engine separately causing the same message to be received multiple times.

5.4 Persistence Considerations

In addition to faults that can occur in a BPEL engine with internal transaction processing, several faults, mostly caused by network idiosyncrasies have been introduced with the externalization of the transaction processing.
When the external coordinator is or becomes unavailable (for example if it crashes or the network link between the PureBPE and the coordinator faults), the messages sent from the PureBPE to that coordinator must not disappear but must be delivered if the external coordinator becomes available again. The same behavior is needed if the PureBPE crashes.
In the prototype JMS is used to exchange messages between the PureBPE and the external coordinator. This means that the sender (the PureBPE or the external coordinator) can use a "send and forget" strategy. They hand over the message to the JMS provider in a transacted session. The first thing the JMS provider does, is to persist the message. Once the send method returns the sender can be assured that the JMS provider will take care of the message and will deliver it eventually (*guaranteed-delivery*) even if the receiver is not available at the point of the sending of the message.
As soon as the receiver (i.e. the external coordinator or the PureBPE) becomes available again and has initialized it checks if messages are waiting on the JMS destination it listens to and processes them.
A persistent configuration for both ActiveBPEL engines, the PureBPE as well as the engine running the coordinator processes is needed to ensure that after an engine has recovered the process instances are in the state they were in before the crash. The ActiveBPEL engine offers such a persistent configuration where after each state change of a BPEL object the process instance is saved to a database.
Several additions have to be made to this persistence algorithm to cope with the problems external transaction processing causes. First of all it is needed to save additional information that has been generated by the external coordinator. Therefore for every scope its scopeId is saved with the scope. Additionally a scope's coordination context is saved to persistent storage. This is necessary if child scopes want to register with that coordination context after the restart of the PureBPE.
On restart, the PureBPE might have incoming life cycle messages for a scope waiting on the JMS destination it listens to. On receipt of such a message the transaction gateway extracts the scopeId and checks the `mScopeStatusMap` if the

`AeActivityScopeImpl` object belonging to that scopeId is in the map. If not (which is the case after a crash of the PureBPE) it restores the process instance that contains the scopeId from the database and continues the processing.
When a process instance is restored, all maps of the transaction gateway are filled with the necessary Java objects and the corresponding scopeIds so that the process instance and the transaction gateway are in the same state as they were before the PureBPE shut down.

5.5 BPEL4Tx Coordinator

The implementation of the BPEL4Tx Coordinator consists of three modules. The ordering broker which receives the life cycle messages from the PureBPE and coordinates the other modules, the WS-BA participant which translates the messages from the ordering broker into WS-Coordination and WS-Business Activity messages and the WS-BA default fault handler which emulates the behavior of the BPEL implicit default fault handler.
All three modules are in fact business processes implemented in BPEL. For every scope that is started inside the PureBPE a corresponding BPEL4Tx ordering broker process instance is started. This is done via an instance creating `receive` activity that receives the `initScope` message from the PureBPE.

5.5.1 BPEL4Tx Ordering Broker

The BPEL4Tx ordering broker is responsible for the orchestration of the BPEL4Tx coordinator. Figure 5.8 shows the role of the BPEL4Tx ordering broker. It receives the PureBPE's life cycle messages and forwards the messages to a BPEL4Tx WS-BA participant or a WS-BA default fault handler when necessary. In the other direction it receives messages from the modules it has started and forwards them to the PureBPE.

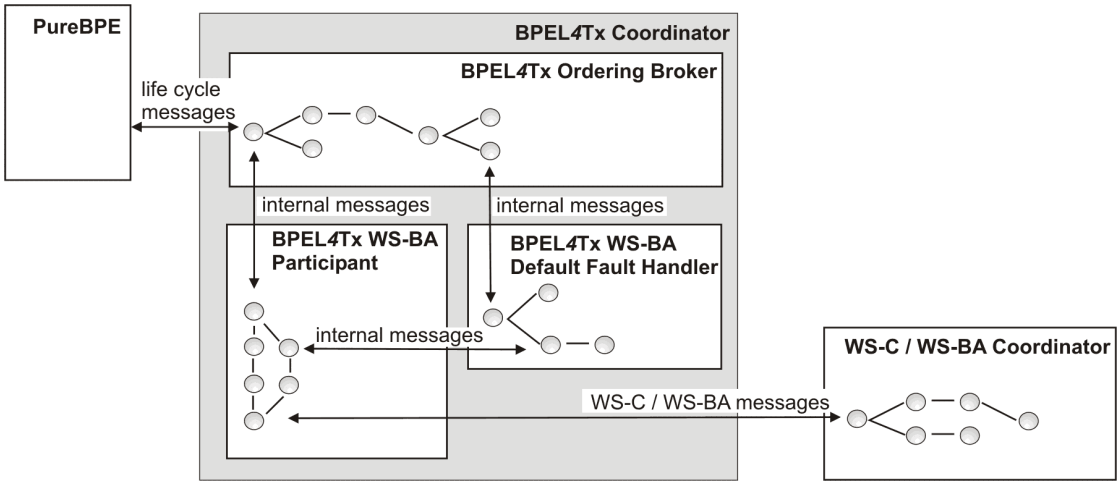

Figure 5.8: Role of the BPEL4Tx ordering broker.

5.5.1.1 Interface

The BPEL4Tx ordering broker exposes three ports to the outside. The `BPEL4TxObPureBPEPort` port which is to be called by the PureBPE, the `BPEL4TxObInternalCommunicationPort` port providing operations and messages

to be called by the other BPEL4Tx modules and the `BPEL4TxOb2ObPort` providing an interface for two BPEL4Tx ordering brokers to exchange information. All three ports are reachable via JMS.

5.5.1.2 Correlation

The BPEL4Tx ordering broker process has one correlation set with one property, the scopeId. All messages to and from the BPEL4Tx ordering broker must contain a scopeId parameter which points to the scope the ordering broker process instance represents. The correlation set is started with the `initScope` message which also starts the process instance.

5.5.1.3 BPEL4Tx Ordering Broker Control Flow

As explained above, the ordering broker's main task is to dispatch messages received from the PureBPE to the BPEL4Tx modules and vice versa. Therefore the structure of its control flow closely resembles the life cycle of a BPEL scope. The control-flow of the ordering broker is explained step by step in the following paragraphs using the notation of state diagrams and mapping of state diagrams (and therefore regular expressions) to BPEL discussed in Appendix A. The sequence of messages is determined by the life cycle of a BPEL scope shown in Figures 2.1 and 4.2.

Normal processing mode of a scope Figure 5.9 shows the control flow of the normal processing mode part of the ordering broker. The states and messages are explained below.

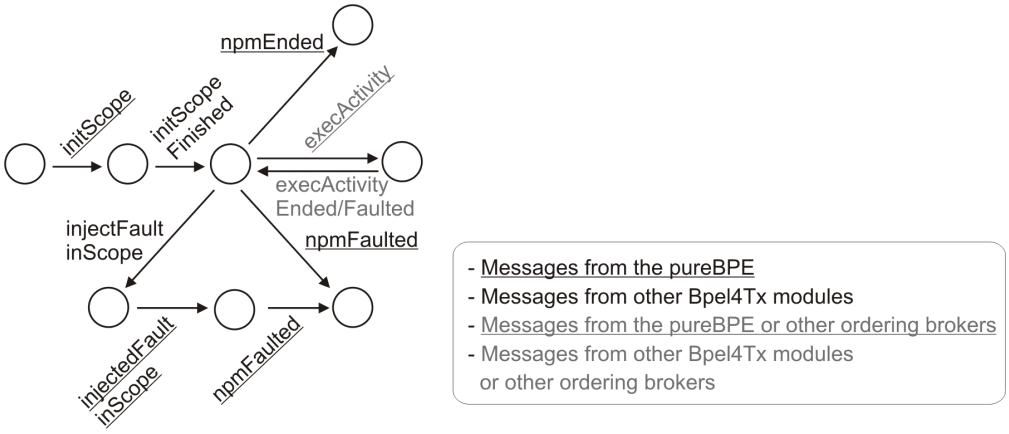

Figure 5.9: State diagram for the normal processing mode.

Initialization A new BPEL4Tx ordering broker process instance is created on receipt of an `initScope` message from the PureBPE.
This means that for every scope in a business process instance that is executed by the PureBPE a new BPEL4Tx ordering broker process instance is started.
Having created a new instance the ordering broker copies and forwards the `initScope` message to the BPEL4Tx WS-BA participant process which in turn starts a new instance on receipt of this message.
The ordering broker process instance then waits for a confirmation of the WS-BA participant that it has registered the scope with the WS-BA coordinator. The `initScopeFinished` message from the WS-BA participant indicating that it has completed initialization is then forwarded to the PureBPE.

Completion of a Scope After signalizing the PureBPE that it has completed initialization of the scope the scope is in normal processing mode (npm) and the BPEL4Tx ordering broker enters the `npmLoop`. The `npmLoop` is a `while` activity that has a nested `pick` activity with `onMessage` events waiting for a `npmEnded` or `npmFaulted` message from the PureBPE indicating the outcome of the normal processing of a scope. Additionally an `injectFaultInScope` or an `execActivity` can be received. The `npmLoop` `while` activity is needed since a sequence of zero or more `execActivity` messages can be received in normal processing mode.

Injecting Faults into a Scope On receipt of an `injectFaultInScope` message from the WS-BA participant the ordering broker copies that message and invokes the `injectFaultInScope` message of the PureBPE. The `injectFaultInScope` message contains a string with the `faultName` and an XML-representation of the fault data. The PureBPE responds with an `injectedFaultInScope` message which is copied and forwarded to the BPEL4Tx WS-BA participant by the ordering broker. After having injected a fault into a scope, the scope can only fault, so the next message to be received by the ordering broker is a `npmFaulted` message. The control flow now can continue where it would continue if it just had seen a `npmFaulted` message after initialization. This is done by assigning a `false` value to the `npmLoop's` loop condition and setting the end condition of the `npmLoop` `while` activity to `npmFaulted` thus leaving the `while` activity and follow the link associated with the `npmFaulted` value.
It is sufficient to accept `injectFaultInScope` messages only when a scope is in normal processing mode (i.e. has not sent a `npmFaulted` or `npmEnded` message) since the BPEL specification says that fault handlers are uninstalled after a scope has completed (i.e. sent a `npmEnded` message), and that a scope can only see one fault at a time (i.e it has already faulted after sending a `npmFaulted` message).

Execution of External Activities in Normal Processing Mode In this prototype only the `compensate` activity is such an external activity. The BPEL specification states that `compensate` activities are only allowed inside fault and compensation

handlers. So why is it needed to execute an external `compensate` activity in normal processing mode? The process shown in Figure 5.10 has a fault handler that has nested scopes. One of these scopes has a nested `compensate` activity. If this activity is executed by the PureBPE it sends an `execActivity` message to the ordering broker for the scope in which the `compensate` activity is nested. This scope is still in normal processing mode but needs to accept the `execActivity` message.
If a `compensate` activity is executed during the normal processing of a scope (that must be nested in a fault or compensation handler), the semantics of this `compensate` activity is the following:
Compensate one or all nested scopes (dependent on the definition of the `compensate` activity) of that scope that executes the fault or compensation handler the `compensate` activity is in.
In Figure 5.10 this would mean that the `compensate` activity in the `NestedScopeInFH` would compensate the nested scopes of the `OuterScope` which executes the fault handler the `compensate` activity is in, i.e. it would compensate the `NestedScope`. The architecture of the BPEL4Tx ordering broker is that for

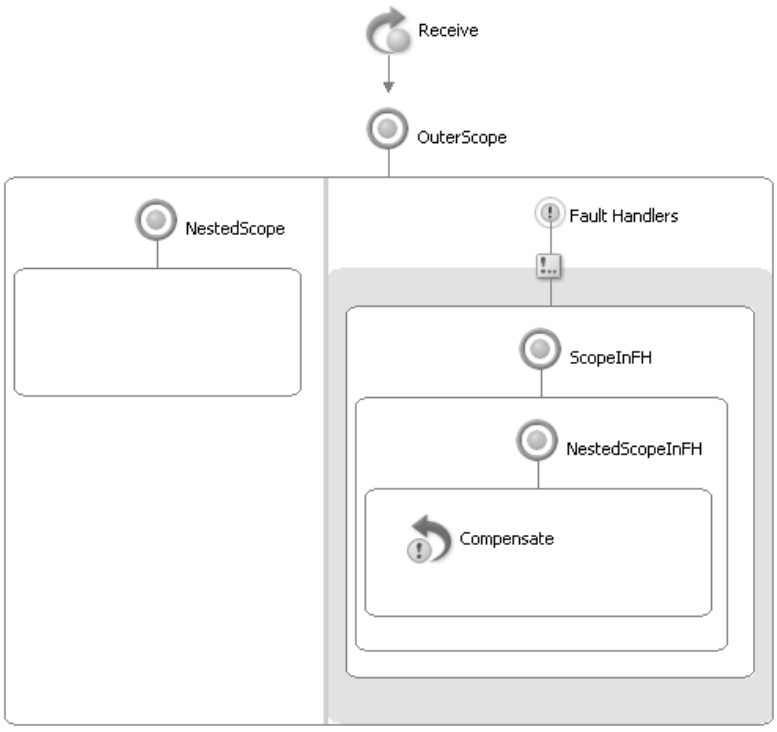

Figure 5.10: BPEL process with handler with nested scopes.

every scope a new ordering broker process instance is started. In the situation of `compensate` activities in nested scopes in fault or compensation handlers this imposes a problem. The PureBPE sends the `execActivity` message to that scope the `compensate` activity is in. If this is a nested scope inside a fault or compensation handler it cannot simply compensate its nested scopes but needs to tell the

scope that contains the handler to compensate its nested scopes. A means to walk up the scope hierarchy is therefore needed.
The "hierarchy climbing" can be done in three places:

1. In the PureBPE. The PureBPE detects that an external activity (more specific a `compensate` activity needs to be executed in a scope that is in normal processing mode). It then walks up the scope hierarchy and sends the `execActivity` to the scope that executes the fault or compensation handler. This would assume that the PureBPE knows that if it encounters a `compensate` activity that it has to walk up the hierarchy. Originally the purpose of the externally executed activities is, that the PureBPE does not know the semantics of those activities, so this solution would render the original purpose of the external activity construct useless.

2. In the WS-BA coordinator. This would require that the WS-BA coordinator has BPEL-specific knowledge which is not the purpose of a generic coordinator.

3. The scope hierarchy is climbed in the BPEL4Tx external coordinator. This is the best solution since it permits the PureBPE to remain clueless about the semantics of external activities and does not require the WS-BA coordinator to have BPEL-specific knowledge.

Considering the discussion above, the BPEL4Tx ordering broker must be able to tell its parent ordering broker (i.e. the one that manages its parent scope) to execute an external activity. This is done via the `execActivity` operation of the `BPEL4TxOb2ObPort`.
On receipt of an `execActivity` message from the PureBPE in normal processing mode the ordering broker copies the `execActivity` message and sends it to its parent ordering broker. This is done by specifying the parentScopeId (contained in the `initScope` message) as the scopeId for the `execActivity` message and setting the `childScopeId` element of this message to the scopeId of the message that the ordering broker manages. If the parent scope is also in normal processing mode its ordering broker forwards the message again to its parent ordering broker. If it is executing a fault handler or compensation handler it handles the message and responds to its child ordering broker (identified by the `childScopeId` element of the `execActivity` message) with either an `execActivityEnded` or `execActivityFaulted` message to signalize the outcome of the external activity.
Having received an `execActivityEnded` or `execActivityFaulted` message, the control flow returns to the `npmLoop`, the process instance now waits for further `execActivity` messages or an `injectFaultInScope`, `npmEnded` or `npmFaulted` message.

NPMEnded On receipt of a `npmEnded` message the BPEL4Tx ordering broker process signalizes the corresponding WS-BA participant that a scope has fin-

ished its normal processing mode without throwing a fault by forwarding the `npmEnded` message. It then waits for the WS-BA participant to answer with a `npmEndedFinished` message. On receipt of a `npmEndedFinished` message, the BPEL4Tx ordering broker knows that the WS-BA participant also knows that the scope has completed its normal processing mode successfully. The `npmLoop's` loop condition is set to false and the end condition of the `npmLoop` is set to `npmEnded`. The control flow now leaves the `npmLoop`.

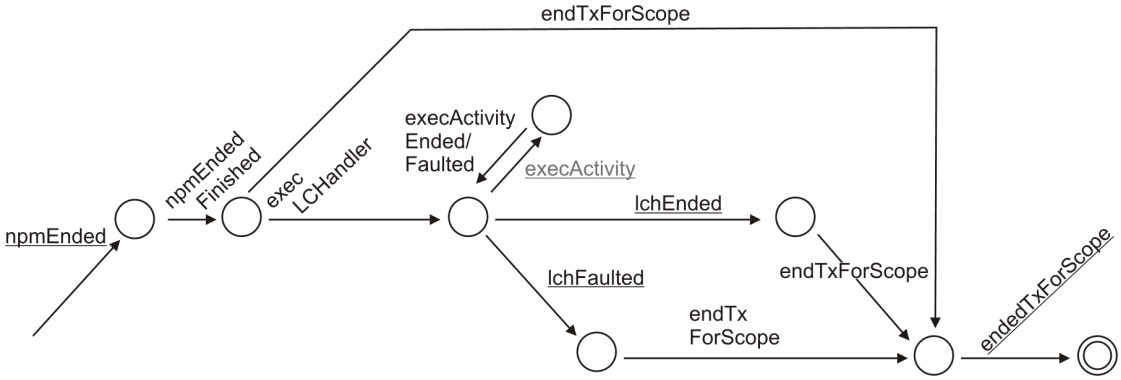

Figure 5.11: BPEL4Tx ordering broker control flow after npmEnded.

Figure 5.11 shows the branch to be taken by the control flow of the ordering broker after a scope finished normal completion without a fault. This branch is taken if the `npmLoop's` end condition is set to `npmEnded`. This is the case if zero or more `execActivity` messages and one `npmEnded` message have been received.
After completing normally, a BPEL scope waits to be either closed or compensated. Therefore the ordering broker waits for an `endTxForScope` message from the WS-BA participant which it can forward to the PureBPE signalizing it to close the scope. The PureBPE responds with an `endedTxForScope` message which is forwarded to the WS-BA participant. After sending the `endedTxForScope` message to the WS-BA participant the BPEL4Tx ordering broker process instance closes.
Instead of an `endTxForScope` message from the WS-BA participant the ordering broker can also receive an `execLCHandler` message indicating that a compensation handler should be executed on this scope, i.e it should be compensated. In order to notify the PureBPE of this fact the ordering broker forwards the `execLCHandler` message. Then the control flow of the ordering broker instance enters the `lchLoop` `while` activity. Inside the `while` activity a `pick` activity with four `onMessage` events waits for a `lchEnded`, `lchFaulted` or an `execActivity` message from the PureBPE or an `execActivity` message from a child ordering broker. The `lchLoop` is necessary since inside a compensation handler zero or more external (`compensation`) activities might occur.

On completion of a life cycle handler either a `lchEnded` message indicating successful completion of the handler or a `lchFaulted` message indicating faulted com-

pletion of the handler must be received by the BPEL4Tx ordering broker.
Having received a `lchEnded` or `lchFaulted` message the `lchLoop` is terminated and its end condition set to `lchEnded` or `lchFaulted` respectively. On receipt of an `execActivity` message from either the PureBPE or a child ordering broker this message is forwarded to the WS-BA participant which returns either an `execActivityEnded` or `execActivityFaulted` message which in turn is forwarded to the PureBPE or the child coordinator. The control flow returns to the beginning of the `npmLoop while` activity.

NPMFaulted Receiving a `npmFaulted` message the BPEL4Tx ordering broker knows that the corresponding scope has thrown a fault. The `npmLoop` then terminates with an end condition of `npmFaulted`. The control flow takes the link from the `npmLoop` while activity which requires the `npmLoop's` end condition to be `npmFaulted`. It then forwards the `npmFaulted` message to the WS-BA participant which in turn responds with a `npmFaultedFinished` message signalizing that it has taken all necessary steps. This message is forwarded to the PureBPE by the ordering broker.
Having completed the `InvokeNPMFaulted` BPEL `invoke` activity, the ordering broker process instance executes a `pick` activity with five `onMessage` events that accept the following messages:

- `FHEnded` from the PureBPE - indicating that a fault handler has ended normally.
- `FHFaulted` from the PureBPE - indicating that a fault handler has faulted.
- `ExecActivity` from the PureBPE - indicating that an external activity must be executed.
- `ExecActivity` from a child ordering broker - indicating that an external activity must be executed that affects child scopes of the scope represented by this ordering broker.
- `ExecDefFH` from the PureBPE -indicating that the implicit default fault handler must be executed.

Implicit Fault Handling A faulted scope is handled by an implicit default fault handler if no fault handler for the fault that occurred, is specified. In such a case, the ordering broker receives an `execDefFH` message. It forwards the `execDefFH` message from the PureBPE to the BPEL4Tx WS-BA default fault handler process for which a new instance instance is created on reception of the message. After the implicit default fault handler has completed (with or without fault) it responds with a `fhFaultedFinished` message. It does not reply with a `fhFaulted` message because the ordering broker already knows that implicit default fault handlers always fault so a `fhFaulted` message is not necessary. For an explanation why

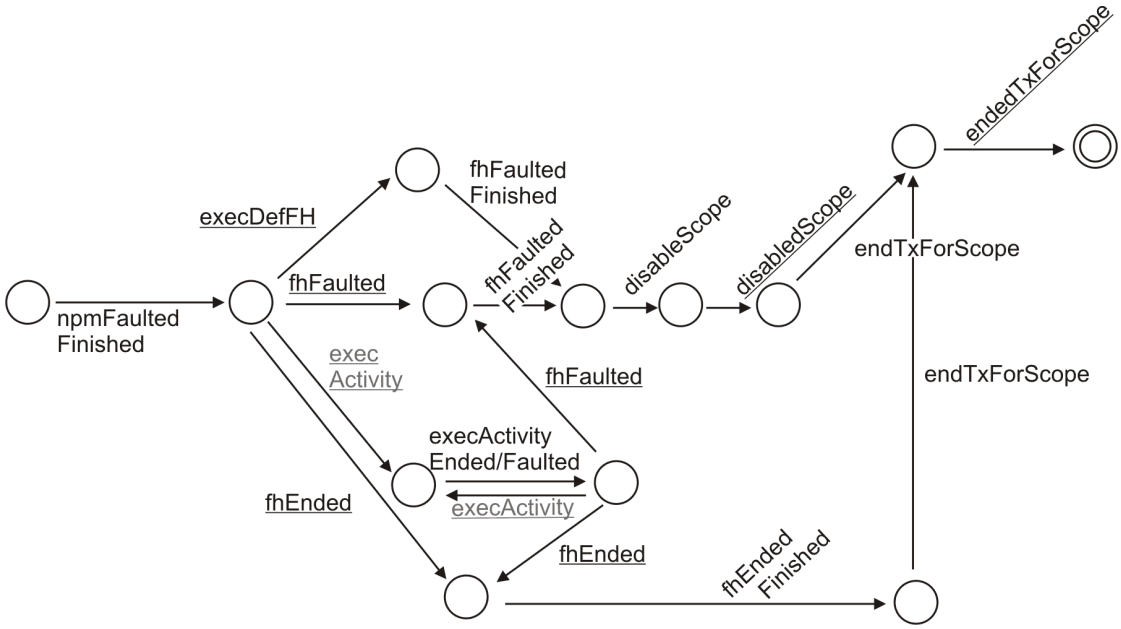

Figure 5.12: BPEL4Tx ordering broker control flow after npmFaulted.

implicit fault handlers always fault see Section 4.4.7.

Normal Fault Handling If a fault handler for a fault is specified, the PureBPE executes this fault handler. The next message the ordering broker receives in this case is one of the three following messages `fhFaulted`, `fhEnded`, or `execActivity`. An `execActivity` message indicates that an external activity (such as a `compensate` activity) has to be executed by the coordinator. Since zero or more `execActivity` messages can be received after the ordering broker has sent the `npmFaultedFinished` message, the `FaultedLoop while` activity is needed. The `FaultedLoop` is entered after the first `execActivity` message has been received. It cannot be entered directly after the `npmFaultedFinished` message since after an `execActivity` message was received no `ExecDefFH` message is permitted. The semantics are that after an external activity which is part of a specified fault handler has been executed an implicit default fault handler cannot be executed on the same scope. The `FaultedLoop` works as following:
If a normal fault handler is executed, zero or more `execActivity` messages can occur until the fault handler has finished. Therefore the `FaultLoop` loops receiving and processing `execActivity` messages until it sees one of the `fhFaulted` or `fhEnded` messages.
Having received a `fhEnded` message the `FaultedLoop` is terminated with the end condition of `fhEnded`. In case of a `fhFaulted` message the loop is terminated with the end condition of `fhFaulted`. After receiving an `execActivity` message the ordering broker forwards it to the WS-BA participant and waits for an `execActivityEnded` or `execActivityFaulted` message indicating the outcome of

the executed Activity. This message is then forwarded to the PureBPE or the child ordering broker dependent on the fact from which the `execActivity` message was received. Having completed the `FaultedLoop` with either a `fhFaulted` or `fhEnded` end condition the control flow takes the appropriate branch depending on the end condition and then the ordering broker waits for a `fhFaultedFinished` or `fhEndedFinished` message which it forwards to the PureBPE. In case of a `fhFaultedFinished` message, the ordering broker waits for another message from the WS-BA participant, the `disableScope` message which it forwards to the PureBPE. Then the coordinator waits for a `disabledScope` message which it forwards to the WS-BA participant.
Now the two branches implicit and normal fault handling can be handled together again. In both cases the ordering broker waits for an `endTxForScope` message from the WS-BA participant which is forwarded to the PureBPE. The PureBPE responds with an `endedTxForScope` message. This message is also forwarded to the WS-BA participant. After having sent the `endedTxForScope` message to the participant the ordering broker process instance ends since no more messages for this scope are to be expected.

Figure 5.13 shows the paths the control flow can take through the BPEL4Tx ordering broker. The state diagram only shows the messages the BPEL4Tx ordering broker receives. `Assign`, `Invoke` and other BPEL activities that are not necessary for the understanding of the structure of the ordering broker process are omitted for clarity of the illustration.

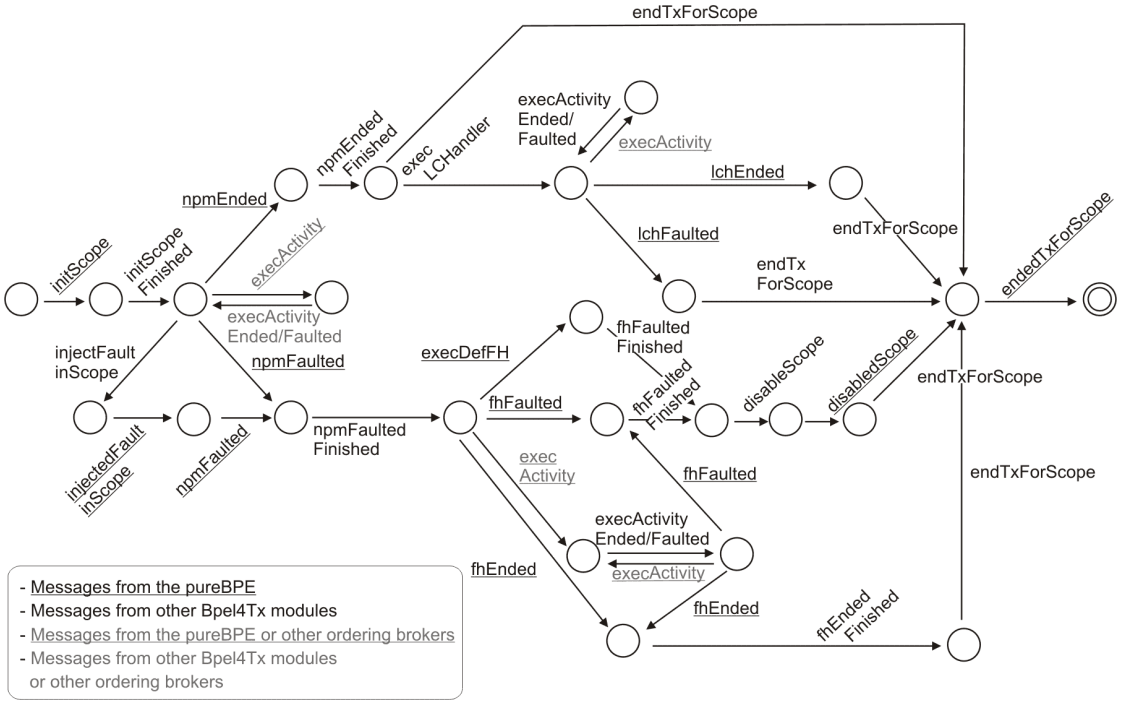

Figure 5.13: Visualization of the BPEL4Tx coordinator BPEL process.

5.5.2 BPEL4Tx WS-BA Default Fault Handler

The BPEL4Tx WS-BA default fault handler behaves as specified in the BPEL specification for the implicit default fault handler. It is a separate module from the WS-BA participant. This design allows to specify a different fault handling behavior without modifying the BPEL4Tx WS-BA participant.

5.5.2.1 Interface

The BPEL4Tx WS-BA default fault handler exposes the `Bpel4TXWSBADefaultFH-Port` to the outside which can be reached via JMS.

5.5.2.2 Correlation

The BPEL4Tx WS-BA default fault handler process has one correlation set, the `scopeId correlation set` which correlates messages based on their `scopeId` property. All messages to the default fault handler must contain the scopeId of the scope the handler belongs to, in order to correlate messages and the right default fault handler process instance.

5.5.2.3 BPEL4Tx WS-BA Default Fault Handler Control Flow

A new default fault handler process instance is started by the BPEL engine upon receipt of an `execDefFH` message on the `Bpel4TXWSBADefaultFHPort`. The `execDefFH` message contains the scopeId of the scope requesting its fault to be handled by the implicit default fault handler. After the `ReceiveExecDefFH` receive activity has created a new instance of the default fault handler process it builds a `compensateAll` message and invokes the WS-BA participant's `compensateAll` operation with it. Afterwards the process waits for the `compensatedAll` or a `compensatedAllWithFault` answer message from the WS-BA participant. On receipt of one of these messages the fault handler knows that compensation has finished or faulted.
According to the BPEL specification implicit fault handlers must rethrow the fault they handled, therefore the default fault handler's behavior is the same if it faults or not, in either case it must rethrow a fault by invoking the `fhFaulted` operation of the WS-BA participant. In case of successful completion of the compensation the fault passed to the WS-BA participant is the fault received with the `execDefFH` message. In case of a faulted compensation, the fault included in the `compensatedAllWithFault` message is copied to the `fhFaulted` message. After sending the `fhFaulted` message the default fault handler has completed and the process instance ends.

5.5.3 BPEL4Tx WS-BA Participant

The BPEL4Tx WS-BA participant is the module of the BPEL4Tx coordinator that communicates with the WS-C / WS-BA coordinator. When other transaction protocols than WS-BA (for example WS-Atomic Transaction) are used, this module and the WS-BA default fault handler can simply be replaced with a WS-AT participant and a WS-AT default fault handler for example.

5.5.3.1 Interface

The BPEL4Tx WS-BA participant exposes three WSDL ports to the outside.
The `BPEL4TxTaParticipantPort` contains all operations needed for internal communication of the BPEL4Tx coordinator. These operations get called by the other BPEL4Tx modules such as the BPEL4Tx ordering broker and the BPEL4Tx WS-BA default fault handler.
The `ActivationRequesterPort` is defined in the WS-Coordination WSDL file [12] and contains all operations that are needed for a WS-C compliant participant that wants to use the WS-C activation service.
The `RegistrationRequesterPort` is also defined in the WS-C WSDL file and contains all operations needed by a WS-C registration service to respond to WS-C registration requests.
The `BusinessAgreementWithParticipantCompletionParticipantAddPort` is based on the `BusinessAgreementWithParticipantCompletionParticipantPort` defined in the WS-Business Activity WSDL file [14]. It specifies all operations needed to make the BPEL4Tx WS-BA participant compliant to the modified business agreement with participant (mBAP) completion protocol specified in Section 4.4.2.
All three ports have SOAP bindings that specify JMS URLs as specified in Section 5.3.3 so that the operations can be reached via SOAP messages sent over JMS.

5.5.3.2 Correlation

Correlation in the WS-BA participant is done by using two correlation sets.
The `parentScope` correlation set correlates messages with a scopeId that belongs to the parent scope of the scope that is handled by this WS-BA participant. This is only needed for correlation purposes if an `injectFaultInScope` message is sent to the parent scope. The `injectedFaultInScope` message response contains the scopeId of the parent scope but needs to be correlated to the scope that initiated the `injectFaultInScope` message. Therefore this special correlation set is necessary.
The `scopeId` correlation set is used for all other messages. It correlates messages, based on a contained scopeId that identifies the scope uniquely that is handled by a BPEL4Tx WS-BA participant.

WS-Addressing Based Correlation WS-Coordination specifies how WS-Addressing [15] is used to exchange endpoint references during the activation and registration process.

Figure 5.15 is taken from the WS-Coordination specification [13] and shows the

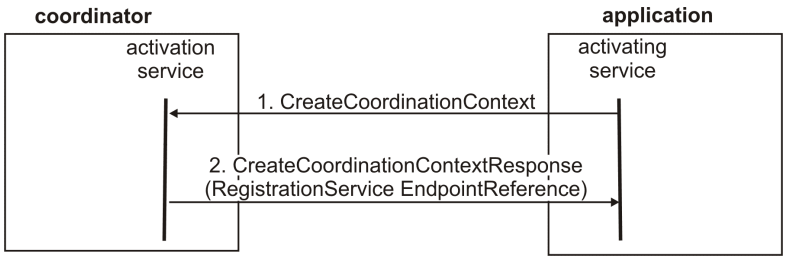

Figure 5.14: Exchange of registration service endpoint reference information.

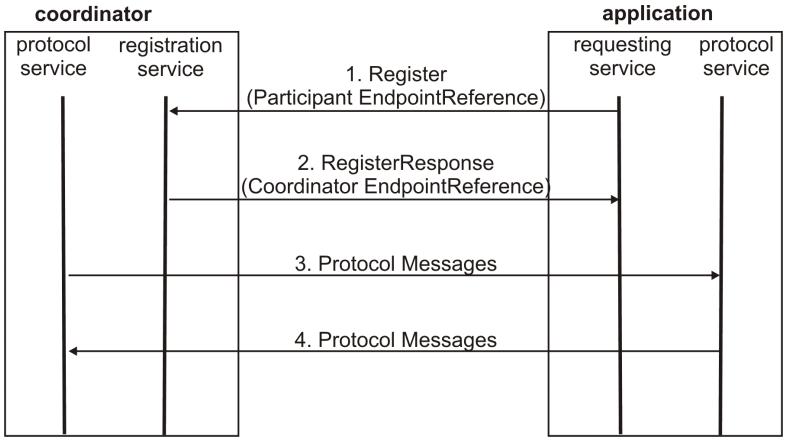

Figure 5.15: Exchange of endpoint reference information during registration of a participant.

exchange of WS-Addressing `EndpointReference` information between a participant and a coordinator. If a participant wants to register with a coordinator it sends a `Register` message to the `RegistrationService` endpoint reference specified in the coordination context. Figure 5.14 shows how the `RegistrationService` endpoint reference is submitted with the `CreateCoordinationContextResponse` message during the activation of a new coordination context.

The `Register` message contains the `ParticipantService` endpoint reference for the protocol service of the WS-BA participant, in this prototype the Address, the ServiceName and PortName for the WS-BA participant's `BusinessAgreementWithParticipantCompletionParticipantAddPort`.

The Role of the ScopeId for Correlation . Upon activation of a new activity the WS-BA participant that creates the new activity must submit its scopeId with the `CreateCoordinationContext` message. If it does not submit a scopeId

the activation service creates one. The `RegistrationService` endpoint reference returned with the `CreateCoordinationContextResponse` message contains a `ReferenceProperty` with the scopeId. Since participants that want to register for this activity must always use the `RegistrationService` endpoint reference, it is made sure that the scopeId can be used to correlate `Register` messages to the right WS-Coordination registration service process instance.
Upon registration, the participant that registers must also submit its scopeId with the `Register` message. This is necessary since participants representing scopes with children are both in the sense of WS-C, participants and activities. The scopeId is used to connect the participant role with the activity role. In case an external participant does not submit a scopeId, one is created by the registration service. The `Participant EndpointReference` submitted with the `Register` message by a WS-BA participant contains the scopeId as a `ReferenceProperty` (if the participant is an external partner it might include other reference properties) ensuring that all protocol messages send to the WS-BA participant by the WS-BA coordinator contain the right scopeId. Vice versa it works the same, the scopeId is included in the `Coordinator EndpointReference` as a `ReferenceProperty` so it is submitted in the SOAP header by the participant every time a protocol message is sent to the coordinator.

Limitations of WS-Addressing Based Correlation and the ActiveBPEL Engine In contrast to other BPEL engines the ActiveBPEL engine does not allow SOAP header based correlation. This means that for example the scopeId contained in the SOAP header for messages received from the WS-BA coordinator is not correlated to the right WS-BA participant process instance. Modifying the ActiveBPEL engine so that it is able to perform correlation based on SOAP headers is out of the scope for this diploma thesis. Additionally a mechanism to make SOAP headers available as message parts in the BPEL process would be needed to allow the BPEL correlation mechanism to be used.
The following workaround is used in this prototype to emulate WS-A based correlation:
Every message exchanged between the WS-BA participant and its coordinator has an additional `referenceProperties` part which contains the reference properties specified for the endpoint reference the message is sent to. The ActiveBPEL engine then performs normal BPEL correlation based on the scopeId contained in this message part.
So instead of having the reference properties in the SOAP header they are in the SOAP body as an additional message part. This is not compliant to the WS-C and WS-A specifications but it is a workaround to emulate the behavior, based on the limitations imposed by the ActiveBPEL engine.

5.5.3.3 BPEL4Tx WS-BA Participant Control Flow

The BPEL4Tx WS-BA participant control flow consists of three parts: the WS-C activation part, the WS-C registration part and the WS-Business Activity (WS-BA) part.
The WS-C activation part deals with the activation of a scope as a WS-C activity and the creation of a new coordination context for it. This part is not executed every time since not every scope needs a coordination context.
The WS-C registration part is responsible for the registration of a scope with its parent scope's coordination context.
The WS-BA part is finally responsible for sending and receiving the messages specified in the modified participant completion protocol in Section 4.4.2. It handles the communication with the WS-BA coordinator.
The structure of the WS-BA part resembles that of the BPEL4Tx ordering broker. Its main task is to transform the messages from the ordering broker and the WS-BA default fault handler to messages understood by the WS-BA coordinator and vice versa. A new BPEL4Tx WS-BA participant process instance is created on receipt of an `initScope` message from the BPEL4Tx ordering broker.
Figure 5.16 shows the paths the control flow can take through the parts of BPEL4Tx WS-BA participant.

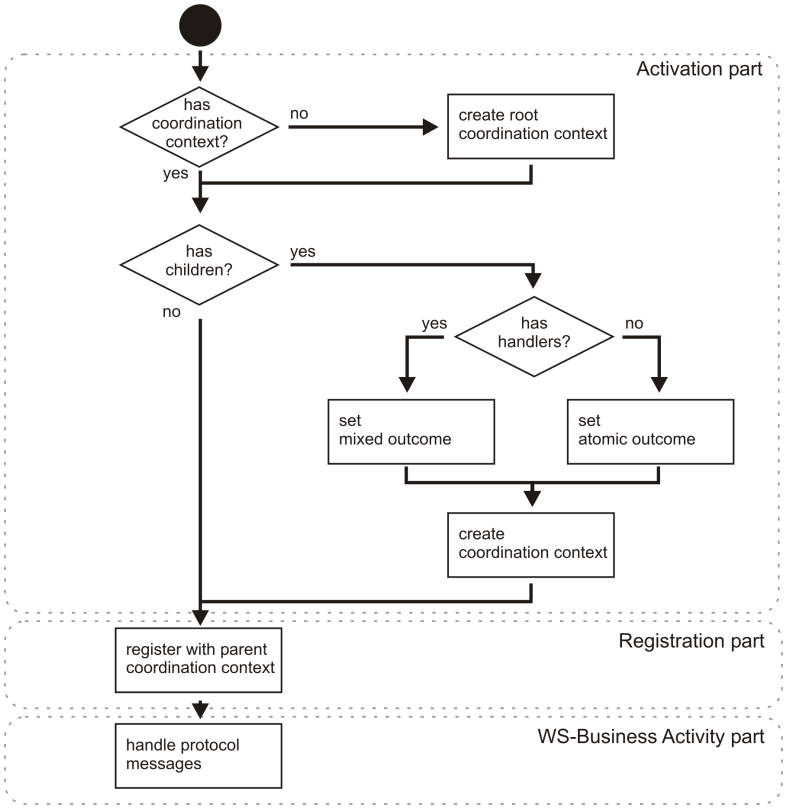

Figure 5.16: Paths through the parts of the BPEL4Tx WS-BA participant.

Creating a Root Coordination Context The first evaluation is done to check if the scope that should be represented by this WS-BA participant process instance, already has a coordination context where it can register. The only situation where this is not the case is if the scope is the process itself and this process has not been started under a coordination context. The process then does not have a parent scope with whose coordination context it can register. Therefore a root coordination context as described in Section 4.3 must be created. In order to accomplish this, a `CreateCoordinationContext` message is sent to the WS-C coordinator which returns a `CreateCoordinationContextResponse` message that contains all necessary transaction information.

Creating a Coordination Context for the Scope The next evaluation to be done by the BPEL4Tx WS-BA participant is to decide whether an WS-C coordination context has to be created or not. This is done based on the `scopeDef` parameter in the `initScope` message which contains the XML-representation of the BPEL scope. Based on the criteria defined in Section 4.3 a scope needs to register as a WS-C activity if the Xpath expression presented in Listing 5.3 evaluates to true. In order to understand the Xpath expression it is necessary to understand how an `initScope` message is built.
The `initScope` message contains a part `parameters` which contains an `initScope` element that is of the type `types:initScopeType`. The `initScopeType` inherits from `types:identificationType` which is the basis for all messages that need a `scopeId` element. The `types:identificationType` contains one `xs:string` element with the name `scopeId`. The interesting element of the `initScopeType` is the `scopeDef` element which can contain either a BPEL scope, invoke or process definition as specified in the schema document for BPEL 1.1 [18]. Since the schema document with the namespace of `http://schemas.xmlsoap.org/ws/2003/03/business-process/` is not correct with regard to the BPEL specification [22] [5], input and output message validation against schemas of the ActiveBPEL engine which runs the coordinator processes must be turned off in order to accept `initScope` messages.

Listing 5.3 shows the four parts needed to determine if a scope has children. If one of the parts evaluates to true the scope has children and must activate a new coordination context for those children to register with.
The first part evaluates if the `scopeDef` element contains a scope child element at any level below the scope the `scopeDef` is for.
The second, third and fourth part deal with implicit scopes, i.e. invokes. If the scope to be evaluated has a nested invoke that either contains a `catch` child element (i.e. a fault handler), a `catchAll` child element (i.e. a default fault handler) or a `compensationHandler` child element (i.e. a compensation handler) it is an implicit scope. An explanation why this is the case can be found in Section 4.3.

```
bpws:getVariableData('initScopeRequestBPEL4Tx','parameters',
  'count(/types:initScope/types:scopeDef/*//bpws:scope)')>0
or
bpws:getVariableData('initScopeRequestBPEL4Tx', 'parameters',
  'count(/types:initScope/types:scopeDef/*
        //bpws:invoke/bpws:catch)')>0
or
bpws:getVariableData('initScopeRequestBPEL4Tx', 'parameters',
  'count(/types:initScope/types:scopeDef/*
        //bpws:invoke/bpws:catchAll)')>0
or
bpws:getVariableData('initScopeRequestBPEL4Tx', 'parameters',
  'count(/types:initScope/types:scopeDef/*
        //bpws:invoke/bpws:compensationHandler)')>0
```

Listing 5.3: Xpath expression to determine whether a scope has children.

Determining WS-BA Outcome Mode Section 4.3 specifies that either atomic or mixed outcome mode must be specified upon creation of a new coordination context based on the defined compensation and fault handlers for a scope.
The Xpath expression shown in Listing 5.4 determines if a scope has defined handlers.

```
bpws:getVariableData('initScopeRequestBPEL4Tx', 'parameters',
  'count(/types:initScope/types:scopeDef/*
        /bpws:faultHandlers)')>0
or
bpws:getVariableData('initScopeRequestBPEL4Tx', 'parameters',
  'count(/types:initScope/types:scopeDef/*
        /bpws:compensationHandler)')>0
```

Listing 5.4: Xpath expression to determine outcome mode.

If one of the two disjunctive parts evaluates to true the outcome mode is mixed outcome, because a fault or compensation handler might not compensate all child scopes. The Xpath expression returns false if no handler is defined, this means that either all child scopes complete or all child scopes are compensated which is atomic outcome behavior.
Having determined the outcome mode, a scope can create a sub-context with its parent context by sending a `CreateCoordinationContext` message to the WS-C activation service. In order to signal the WS-C activation service that this new coordination context is a sub-context of the context of the parent scope the coordination context of the parent scope received with the `initScope` message is copied into the `CurrentContext` element of the `CreateCoordinationContext` message.
After sending the message the BPEL4Tx WS-BA process instance waits for the `CreateCoordinationContextResponse` message from the WS-C Activation Service.

Register the Scope After the activation part has finished (if applicable), the WS-BA participant process instance must register the scope with the WS-C registration service. First of all the process instance assigns the endpoint reference obtained from the `parentContext/RegistrationService` element of the `initScope` message to the partner link which provides the `registerOperation`. The next step is to invoke the `registerOperation` of the WS-C coordinator. Successful registration is signalized to the WS-BA participant by invoking its `registerResponse` operation. After the corresponding `receive` activity has executed, the initialization of a scope is finished.
The process instance can now built the `initScopeFinished` message by setting the scopeId to that received in the `initScope` message and by copying the coordination context received during the activation of the scope (if applicable) to the optional `context` element of the `initScopeFinished` message.
After invoking the BPEL4Tx ordering broker's `initScopeFinished` operation the WS-BA participant waits for messages indicating the outcome of the normal processing mode of the initialized scope.

Handle WS-BA Protocol Messages Having initialized, the scope is now in state `Active`. In this state the following possibilities arise for the WS-BA participant:

- The normal processing mode of the scope ends successfully, indicated by an invocation of the `npmEnded` operation by the BPEL4Tx ordering broker.

- The normal processing mode of the scope faults. This event is indicated by an invocation of the `npmFaulted` operation by the BPEL4Tx ordering broker. The fault's XML-Representation and fault name is supplied with the `npmFaulted` message.

- The parent scope encounters a fault and the WS-BA coordinator decides to cancel this scope. This is indicated by the invocation of the `Cancel` Operation.

In contrast to the BPEL4Tx ordering broker, the BPEL4Tx WS-BA participant does not need to handle `execActivity` messages while being in state `Active` (the state `Active` from WS-BA translates into the normal processing mode discussed in scope life cycle description). Since the ordering broker takes care of the forwarding of the `execActivity` message to the scope's ordering broker that is currently executing a fault or compensation handler, `execActivity` messages are only allowed to appear in the WS-BA participant process on execution of a fault or compensation handler. This behavior might need to be modified if other external activities besides `compensate` activities are permitted in the future, since these activities might occur in normal processing mode.

Normal Processing Mode Ends On receipt of an `npmEnded` message the following activities are executed: First the `completed` operation of the WS-BA coordinator is invoked then the process instance waits for an invocation of its `Ack` operation indicating that the coordinator has taken notice of the completed scope. When the `Ack` operation has been executed the ordering broker is notified of this event by invoking its `npmEndedFinished` operation. Now the scope is in state `Completed`.
It can leave this state by either receiving a `Close` message or a `Compensate` message.
In the case of a `Close` message the WS-BA participant sends an `endTxForScope` message with the `stateForAudit` element set to `Closed` to the ordering broker which then responds with an `endedTxForScope` message. This message is translated to a `Closed` message that is sent to the WS-BA coordinator. After sending this message the WS-BA participant process instance terminates.
On receipt of a `Compensate` message in state `Completed` the WS-BA participant has two possibilities. Either the scope it represents has a defined compensation handler then this must be executed by the PureBPE, or the implicit compensation handler must be executed by the WS-BA participant. In order to decide which handler to execute the WS-BA participant inspects the XML scope definition enclosed in the `initScope` message.

```
bpws:getVariableData(
 'initScopeRequestBPEL4Tx','parameters',
 'count(/types:initScope/types:scopeDef/*
        /bpws:compensationHandler)')
```

Listing 5.5: Xpath expression to determine whether a scope has a compensation handler.

The Xpath expression shown in Listing 5.5 returns the amount of defined compensation handlers (0 or 1). Based on the result of the expression one of the two following paths can be taken.

Explicit compensation. If a compensation handler is defined the PureBPE must execute it. In order to have the PureBPE execute the compensation handler, the WS-BA participant sends an `execLCHandler` with the `handler` element set to `compensationHandler` to the ordering broker and waits for a `lchEnded` or `lchFaulted` answer from the ordering broker. Before receiving one of those two messages zero or many `execActivity` messages can be received. Because this requires a loop, the WS-BA participant process executes a `pick` activity with three `onMessage` events inside a `while` activity. This loop is called the `lchLoop`.
On receipt of a `lchEnded` answer from the ordering broker the WS-BA participant terminates the `lchLoop` and sets its end condition to `lchEnded`. After the `lchLoop` has been left by the control flow and its end condition is set to `lchEnded` the WS-BA participant invokes the `Compensated` operation of the WS-BA coordinator.

On receipt of a `lchFaulted` answer message inside the `lchLoop` the WS-BA participant ends the `lchLoop` and sets its end condition to `lchFaulted`. Having left the `lchLoop` with an end condition of `lchFaulted` the control flow moves to the `invoke` activity which invokes the `Fault` operation of the WS-BA coordinator indicating that the compensation handler faulted. The coordinator responds with a `Faulted` message to indicate that it has noticed and processed the `Fault` message. If an `execActivity` message is received inside the `lchLoop` the WS-BA participant checks if this is a `compensate` activity by examining the activity's `activityDef` activity definition. If yes, it sends a `CompensateSubScopeSuggestion` message to the WS-BA coordinator which in turn responds with a `compensatedSubScopeSuggestion` or `compensatedSubScopeSuggestionWithFault` message. Having received a `compensatedSubScopeSuggestion` message from the WS-BA coordinator indicating successful compensation of its nested scopes or a `compensatedSubScopeWithFault` message indicating a fault during compensation, the WS-BA participant sends an `execActivityEnded` or `execActivityFaulted` message to the ordering broker, then the control flow returns to the beginning of the `lchLoop` waiting for `lchEnded`, `lchFaulted` or more `execActivity` messages.

Implicit compensation. In case no compensation handler is defined, the WS-BA participant takes care of the compensation of all sub-scopes. Therefore it invokes the `CompensateSubScopeSuggestion` operation of the WS-BA coordinator. The `activityDef` element of the `CompensateSubScopeSuggestion` message is set to `<compensate>` indicating that all nested scopes must be compensated. Details on how the `activityDef` element is evaluated can be found under Section 5.6.3. The coordinator responds with a `CompensatedSubScopeSuggestion` or a `CompensatedSubScopeSuggestionWithFault` message which in turn triggers the WS-BA participant to invoke the `Compensated` or `Fault` operation of the coordinator.
Having sent a `Compensated` message or received a `Faulted` message the participant can now invoke the ordering broker's `endTxForScope` method with the `stateForAudit` element either set to `Compensated` or `Faulted`. On receipt of the `endedTxForScope` message the WS-BA process instance terminates.

Normal processing mode faulted On receipt of a `npmFaulted` message in state `Active`, the WS-BA participant sends an `EncounteringFault` message to the WS-BA coordinator. On receipt of the `EncounteredFault` answer the WS-BA participant forwards this message as a `npmFaultedFinished` message to the PureBPE. The scope, represented by this WS-BA participant is now in state `EncounteredFault`. The next message which can be received now is one of the following:

- `compensateAll` - indicating that a default fault handler wishes to compensate all active nested scopes.
- `execActivity` - indicating that an activity occurred in a fault handler in the PureBPE that needs to be handled externally.

- `fhEnded` - the fault handler ended.
- `fhFaulted` - the fault handler faulted.

Implicit fault handling In case a `compensateAll` message is received, the participant sends a `CompensateSubScopeSuggestion` message with an `activityDef` element with the value `<compensate>` indicating the necessary compensation of all sub-scopes to the coordinator, that responds with a `CompensatedSubScopeSuggestion` or a `CompensatedSubScopeSuggestionWith-Fault` message. On receipt of such a message the WS-BA participant invokes the `compensatedAll` or the `compensatedAllWithFault` operation of the WS-BA default fault handler. The next message to wait for is the `fhFaulted` message. The actions to be taken after receipt of such a message are described below.

Explicit fault handling After receipt of an `execActivity` message the WS-BA participant sends an `CompensateSubScopeSuggestion` message to the WS-BA coordinator. The `activityDef` parameter of the `execActivity` message is copied to the `CompensateSubScopeSuggestion` message. Then the control flow enters the `faultedLoop while` activity and waits for further messages. On receipt of the `CompensatedSubScopeSuggestion` or `CompensatedSubScopeSuggestionWith-Fault` answer the WS-BA participant invokes the `execActivityEnded` or `execActivityFaulted` operation of the ordering broker. Then the WS-BA participant waits again for one of the following messages `execActivity`, `fhEnded` or `fhFaulted`. This essentially means that zero to many `execActivitiy` messages can be handled. This is the reason why the `faultedLoop while` activity is needed.
On receipt of a `fhEnded` message the WS-BA participant must tell the coordinator that it now exits from the transaction, this is done by invoking the `Exit` operation of the coordinator. Then the WS-BA participant waits for an `Exited` confirmation. After the exit has been confirmed the WS-BA participant invokes the ordering broker's `endTxForScope` operation with the `stateForAudit` element set to `Exited`. On receipt of the answer `endedTxForScope` the WS-BA participant process instance ends.

The last possibility of a message to be received in the state `EncounteredFault` is the `fhFaulted` message. The receive activity for this message is followed by an invoke of the `Fault` operation of the WS-BA coordinator.
Then the Ws-BA participant waits for a `Faulted` confirmation which in turn triggers the invocation of the `fhFaultedFinished` operation of the ordering broker. Having sent the `fhFaultedFinished` message the WS-BA participant invokes the `disableScope` operation of the ordering broker and waits for its `disabledScope` confirmation. On receipt of this confirmation the `endTxForScope` operation of the ordering broker is invoked, with the `stateForAudit` element of

the `endTxForScope` message set to `Faulted`. Then the WS-Ba participant waits for the `endedTxForScope` confirmation.

Rethrowing of faults into the enclosing scope Having received an `endedTxForScope` confirmation for a scope the WS-BA participant must decide if it rethrows a fault or not.
Faults must be rethrown into the enclosing scope if a fault handler faulted on a scope and the fault that occurred is not a `forcedTermination` fault. In the case of a `forcedTermination` fault the enclosing scope has already seen a fault so no rethrow is needed. Therefore the WS-BA participant evaluates the fault submitted with the `npmFaulted` message whether it is a `forcedTermination` fault or not. If no, the process instance ends here. If yes, and the fault handler faulted the `endedTxForScope` message triggers the WS-BA participant to built an `injectFaultInScope` message with the `fault` and `faultName` elements set to the fault received with the `fhFaulted` message. The `injectFaultInScope` message is send to the parent scope's ordering broker. Therefore the `scopeId` element of the message is set to that of the parent scope. Additionally the `parentScope` correlation set is initialized with the scopeId of the parent scope in order to correlate the `injectedFaultInScope` message from the parent scope's ordering broker. On receipt of this message the WS-BA process instance ends.

Cancel On receipt of a `Cancel` message from the WS-BA coordinator the WS-BA participant builds an `injectFaultInScope` message and assigns its `fault` element a `forcedTermination` fault. This message is sent to the ordering broker by invoking its `injectFaultInScope` operation with the said message.
Then the WS-BA participant waits for an `injectedFaultInScope` message indicating successful injection of the scope. After receipt of this message the participant must wait. The next message to receive is an `npmFaulted` message. How this is handled is described in paragraph 5.5.3.3. After the fault handling has completed the WS-BA participant sends a `Canceled` message to the WS-BA coordinator and the process instance ends.

5.6 WS-C /WS-BA Coordinator

The Implementation of the WS-C / WS-BA Coordinator consists of two BPEL processes. One, providing the functionality of the WS-C coordinator and one providing the functionality of the WS-BA coordinator. The WS-C coordinator process handles messages for the creation, activation and registration of transaction contexts and participants as specified in the WS-Coordination specification. It consists of two parts: the WS-C activation service part which is responsible for the creating of coordination contexts for WS-C activities and the WS-C registration service part with which participants for activities can register.

The WS-BA coordinator process handles messages from WS-BA participants indicating the outcome of transaction participants. It evaluates the messages and performs the necessary activities to comply with the WS-BA specification.

5.6.1 WS-C Activation Service Part

The responsibility of the WS-C activation service part is to create coordination contexts for new WS-C activities.

5.6.1.1 Interface

The WS-C activation service provides the `ActivationCoordinatorPort` defined in the WS-Coordination WSDL file [12] to the outside. It also provides a SOAP binding to reach the port via SOAP messages over JMS.

5.6.1.2 Correlation

The WS-C activation service part does not use a correlation set. The only message it can receive always triggers the creation of a new process instance, so no correlation is needed.

5.6.1.3 Control Flow

Upon receipt of a `CreateCoordinationContext` message the BPEL engine creates a new instance of the WS-C coordinator process. Then the endpoint reference for the partner link of the `InvokeCreateCoordinationContextResponse invoke` activity is set. This is done by copying the endpoint reference of the `CreateCoordinationContext` message's `endpointReference` part's `ReplyTo` element to the partner link.

Afterwards the `CreateCoordinationContextResponse` message is built. Depending on a specified `CurrentContext` parameter in the `CreateCoordinationContext` message, this new context is a nested context of the one specified in the `CurrentContext` parameter. The `RegistrationService` endpoint reference of the `CreateCoordinationContextReponse` is later used by the WS-BA participant to register a participant with the activity (see also Subsection 5.5.3.2. Therefore the scopeId included in the `CreateCoordinationContext` message is added as a `ReferenceProperty` to the endpoint reference. This scopeId is necessary for the registration service part of the WS-C coordinator process to correlate register messages with the right activity. If no scopeId is submitted with the `CreateCoordinationContext` message, a new one is created. The `CreateCoordinationContextResponse` message also contains a unique identifier for the newly created transaction. This identifier is assembled by concatenating the prefix `http://iaas.uni-stuttgart.de/rm/coordinationContext/` with the current processID. Using the current processID ensures that the ActiveBPEL engine takes care of the uniqueness of this part of the Identifier and therefore the uniqueness of the whole identifier is guaranteed. The processID of a process is unique for a process inside a ActiveBPEL engine (or ActiveBPEL engine cluster if the commercial versions that support clustering are used). The activation service part of the WS-C coordinator process ends upon the the sending of the `CreateCoordinationContextResponse` message to the requester.

5.6.2 WS-C Registration Service Part

The registration service part of the WS-C coordinator process is responsible for the management of participants for a WS-C activity. It accepts registration of participants through WS-C `Register` messages and starts a WS-BA protocol service instance for a participant upon registration. The WS-C registration service part is not an extra process in the coordinator BPEL engine. It rather is a part of the WS-C coordinator process.

5.6.2.1 Interface

The WS-C registration service part of the WS-C service process exposes the `RegistrationCoordinatorPort`, defined in the WS-C WSDL file [12] which is reachable over the `RegistrationCoordinatorPortTypeSoapBinding` via JMS.

5.6.2.2 Correlation

The WS-C registration service part uses the `scopeId` correlation set to correlate `Register` messages with the right process instance. The `scopeId` correlation set uses the scopeId contained in in the reference properties of the

`RegistrationService` element of the `CreateCoordinationContextResponse` message for the activity the register message is for.

5.6.2.3 Control Flow and Architecture Considerations

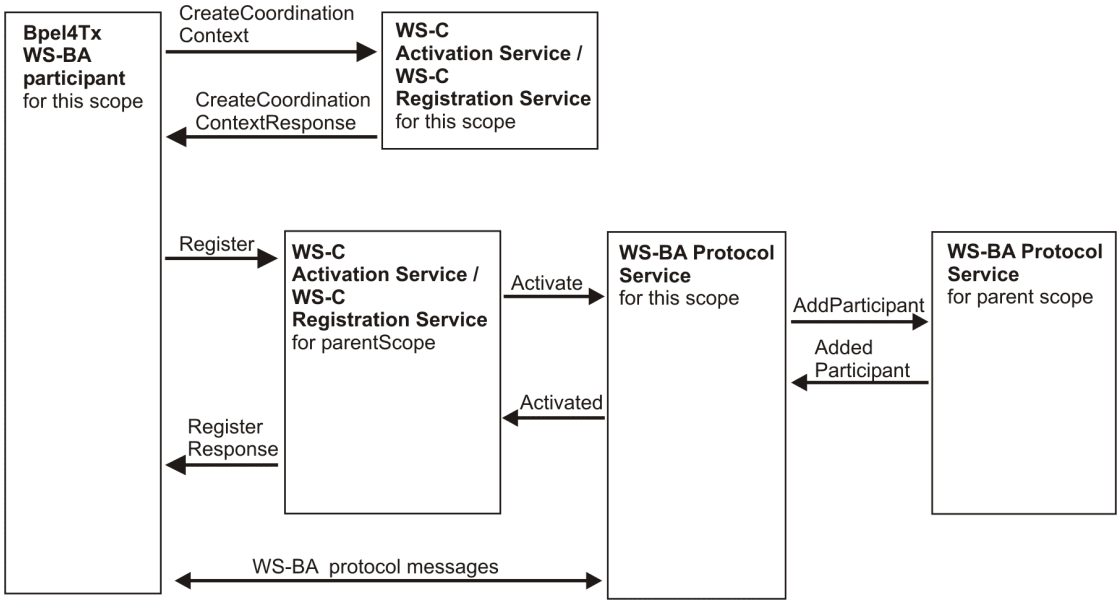

Figure 5.17: Architecture of the WS-C / WS-BA registration part.

Figure 5.17 shows the building blocks used in the prototype to perform a registration of participants according to WS-C and initialize WS-BA protocol services.
After a participant has sent a `Register` message to the registration service of its parent activity (which represents its parent scope). The registration service activates a new WS-BA protocol service. The new WS-BA protocol service which represents the participant that is currently registering now, sends an `AddParticipant` message to its parent WS-BA protocol service that represents the activity the participant is registering for but also its parent scope. This protocol service adds the scopeId and scope name of the new participant to its participant list. The parent WS-BA protocol service needs knowledge about its child protocol services in case of canceling or compensating of scopes. In these cases the parent protocol service needs to advise its child protocol service process instances to send a `Cancel` or `Compensate` message to their children.
Having added the new participant to its list the parent WS-BA protocol service responds with an `AddedParticipant` message.

From a BPEL point of view the WS-C registration part of the WS-C coordinator process is a `while` activity (the `registrationLoop`) with a nested `pick` activity waiting for `Register` and `CloseCoordinator` messages. On receipt of a `Register` message the WS-C registration service starts a new WS-BA protocol service process instance by sending out an `Activate` message. Having received

an `Activated` answer the WS-C registration service sends an `RegisterResponse` message to the WS-BA participant indicating successful registration. Then the control flow returns to the beginning of the `registrationLoop`.
On receipt of a `CloseCoordinator` the WS-C process instance sets the `loopCondition` of its `registrationLoop` `while` activity to false which causes the process instance to end.

5.6.3 WS-BA Protocol Service

The WS-BA protocol service process is responsible for the coordination of WS-BA participants after they have registered for external coordination with the WS-C registration service.

5.6.3.1 Interface

The WS-BA protocol service part exposes two ports to the outside. The `BusinessAgreementWithParticipantCompletionCoordinatorAddPort` exposes all operations required by the modified WS-BA participant completion protocol discussed in Section 4.4.2 and defined in the `BusinessAgreementWithParticipant-CompletionCoordinatorAddPortType` via a SOAP Binding so that it is reachable via JMS.
Additionally the `WSBACInternalCoordinatorPort` offers the operations defined in the `WSBACInternalCoordinatorPortType`. These operations are used for internal communication between different process instances of the WS-BA protocol service. This Port can also be reached via JMS as specified in the corresponding SOAP binding.

5.6.3.2 Correlation

The WS-BA protocol service process uses one correlation set. The `scopeId` correlation set correlates messages based on the scopeId contained in the reference properties part of a message. This is used to emulate WS-A behavior as discussed in Section 5.5.3.2. This correlation set is used for all messages needed in the modified participant completion protocol and all messages used for internal communication between coordinators.

5.6.3.3 Design Considerations

The WS-BA specification states that messages sent to the coordinator that do not correspond to the current state of a participant must be answered by an `Invalid State` fault message. In a BPEL process this can be done by using one of the following two design alternatives:

1. A scope is put around every `receive` or `pick` activity which has event handlers that handle all messages that are not handled by the `receive` activity or the `onMessage` events of the `pick` activity. The event handlers are responsible for throwing the fault.

2. An event loop is used. The process essentially consists of one big while loop with a nested `pick` activity. This `pick` activity contains `onMessage` events for all messages accepted by the coordinator. Inside the `onMessage` events the first thing to be done is to check if a message corresponds to the current state of the participant. If yes the message can be processed, if not a fault is thrown.

The second alternative has been taken in the prototype for the following reasons: Firstly a variable holding the state of the protocol service for a participant is needed anyway since it is necessary to respond to `GetStatus` messages with the current state.
Second the coordinator needs to be compatible to the business agreement with participant completion protocol defined in WS-BA in order to allow external participants that only speak this protocol to participate in transactions coordinated by the coordinator which implements the modified business agreement with participant completion protocol (mBAP) 4.4.2. This means that participants for example can send `Fault` messages before having sent an `EncounteringFault` message. The different behavior of participants depending on their completion protocol can be easily implemented with the 2nd alternative since it only requires to check upon receipt of a message if it complies to the current state and the used completion protocol.

5.6.3.4 Control Flow

The control flow of the WS-BA protocol service process starts with an `Activation` message as depicted in Figure 5.17. Having been started via an `Activation` message the process instance sends an `AddParticipant` message to the parent WS-BA protocol service process if the participant to be represented has a parent activity. Whether an `AddParticipant` message is to be sent is determined by the `isRoot` parameter of the `Activate` message. If it is true there is no parent activity and no `AddParticipant` message has to be sent. This is the case if the WS-BA coordinator is representing the "root activity" (see Section 4.3. Otherwise the `isRoot` parameter is false and the `AddParticipant` message has to be sent. On receipt of the `AddedParticipant` answer the WS-BA protocol service responds with an `Activated` message to signalize that it has been activated. It is now in state `Active`. Considering the design discussed above, the BPEL structure of the WS-BA protocol service is fairly simple. It is a while loop (the `MessageEventLoop`) with a nested pick. This pick has `onMessage` events for the following messages:

- `AddParticipant`

- `Completed`
- `Encountering fault`
- `CompensateSubScopeSuggestion`
- `Exit`
- `Fault`
- `CancelParticipant`
- `CompensateParticipant`
- `ReportStatusChange`
- `EndCoordinator`

Additionally the following message is accepted in an event handler:

- `GetStatus`. Upon receipt of this message the WS-BA coordinator builds a `Status` message with the current state of the coordinator and invokes the `Status` operation of the invoker.

Upon receipt of a `AddParticipant` message the coordinator retrieves the scopeId and the scope name from the message and adds them as a new `ScopeElement` to the `ScopeIDArray` which stores the ids and names of all participants as well as the state they are currently in. After having added the participant to the `ScopeIDArray` the process instance responds with an `AddedParticipant` message. Having sent the `AddedParticipant` message indicating successful addition of the nested participant the control flow returns to the `MessageEventLoop`.

After a `Completed` message has been received, it is first checked if the participant is in state `Active`. If yes the coordinator sets the state variable for this participant to `Completed`. Then it sends a `ReportStatusChange` message with the scopeId and the `Completed` state to its parent coordinator. Having received a `ReportedStatusChange` confirmation, the coordinator responds with a `Ack` message to the participant indicating that it has noticed the completion of the scope.

On receipt of an `Encountering fault` message the coordinator first checks if the participant it coordinates is in state `Active`. If yes, all nested scopes in state active have to be canceled. But before the canceling of nested scopes the parent coordinator must know that this participant is in status `Encountering fault`, so the appropriate `ReportStatusChange` message is sent.
In order to cancel all active nested scopes the coordinator iterates over the `ScopeIDArray` and checks if a scope is in state `Active`. If it has found such a scope it sends a `CancelParticipant` message to the WS-BA protocol service instance representing the nested scope. Having received a `CanceledParticipant` response from the scope the coordinator then continues to cancel the other active nested scopes.

Having completed the canceling of the active nested scopes the state of the participant is changed to `EncounteredFault` and the control flow returns to the beginning of the `MessageEventLoop`.

A `CompensateSubScopeSuggestion` message can occur in two states: `Compensating` or `EncounteredFault`. If such a message is received and the coordinator is in one of those states three steps are taken. First of all a `ReportStatusChange` message is issued to the parent WS-BA protocol service instance notifying it that the scope is in state `CompensatingSubScopeSuggestion`.
Then the coordinator loops over the completed scopes array which contains all scopes in state `Completed`. In case the `CompensateSubScopeSuggestion` message contains a scope name then only the participant with this message has to be compensated if it has already completed. Otherwise all completed participants have to be compensated. Therefore in a third step a `CompensateParticipant` message is sent to each of the WS-BA protocol service instances belonging to the scopes to be compensated.
Having received the `CompensatedParticipant` answer for a scope, the next participant is compensated. Having compensated all requested participants the control flow returns to the `MessageEventLoop`.

Upon receipt of an `Exit` message the state of the participant is set to `Exited`. The parent coordinator is notified of this state change via an `ReportStatusChange` message. Then an `Exited` message is sent to the participant that sent the `Exit` message.
If a `Fault` message is received, the state of the participant is set to `Faulted`. Then a `CancelParticipant` message is sent to all child coordinators that represent participants that are in state `Active`. These participants are for example directly nested scopes of the fault or compensation handler that faulted. Having received a `CanceledParticipant` response from all WS-BA protocol service instances the `CancelParticipant` message was sent to, the parent coordinator is notified with a `ReportStatusChange` message. Then a `Faulted` response is issued to the participant.

On receipt of a `CancelParticipant` message from a parent WS-BA protocol service instance the WS-BA protocol service instance sends a `Cancel` message to its participant and waits for a `Canceled` message in return. Then it notifies its parent WS-BA protocol service instance of the successful cancellation by invoking its `CanceledParticipant` operation.
In the case of an invocation of the `CompensateParticipant` operation of a WS-BA protocol service instance by its parent WS-BA protocol service instance it sends a `Compensate` message to its participant and waits for either a `Fault` or `Compensated` message to be returned by the participant. Depending on the returned message the `CompensatedParticipant` or `CompensatedParticipantWithFault` operation of the parent coordinator is invoked. Figure 5.18 illustrates the actions and messages to be taken by the WS-BA protocol service instances in the case a

`CompensateSubScopeSuggestion` message is received.

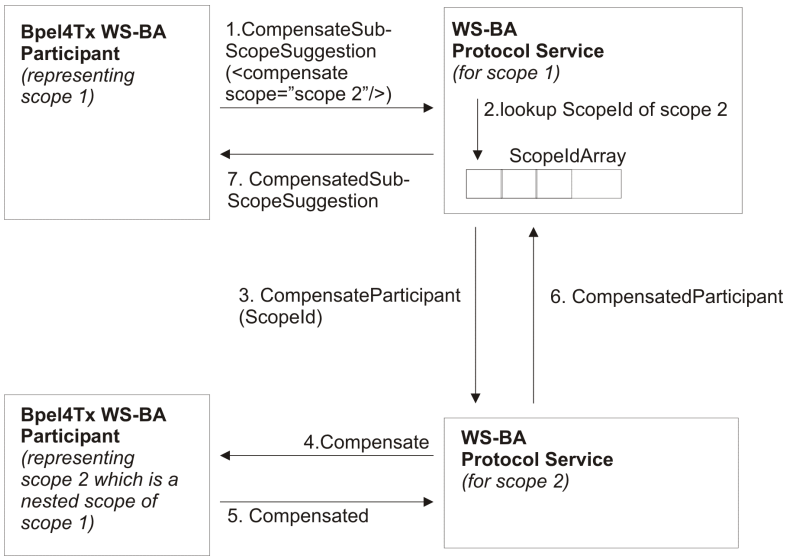

Figure 5.18: Message exchange in case of a CompensateSubScopeSuggestion message.

On receipt of a `ReportStatusChange` message from a child WS-BA protocol service instances the WS-BA protocol service instances sets the state of the participant represented by this child WS-BA protocol service instances to the state contained in the `ReportStatusChange` message. If the new state is `Completed` the array element representing the nested scope is copied to the completed scopes array.
In case the coordinator receiving the `ReportStatusChange` message is the coordinator for a root activity it checks if all nested scopes are in a final state. Final states are:

- Completed
- Compensated
- Exited
- Faulted

If all scopes are in a final state the root coordinator sends an `EndCoordinator` message to all of its child WS-BA protocol service instances. Then the process instance ends.

Upon receipt of an `EndCoordinator` message the coordinator knows that it can finish. Before finishing it needs to close its participant if it is in state `Completed` plus all nested WS-BA coordinators and its WS-C coordinator.
If the WS-BA participant represented by the WS-BA protocol service process instance is in state `Completed` upon receipt of an `EndCoordinator` message, the WS-BA protocol service sends a `Close` message to the participant that responds

with a `Closed` message. Afterwards the WS-BA protocol service loops over the `scopeIdArray` that contains all participants of the activity (i.e. all nested scopes) if there are any, and sends an `EndCoordinator` message to all nested WS-BA coordinators. Before the WS-BA protocol service process instance can finish it needs to close its associated WS-Coordination process. This is done by sending a `CloseCoordinator` message to this process. Having sent the `CloseCoordinator` message the WS-BA protocol service process instance ends.

5.6.3.5 Differences to the WS-C and WS-BA Specifications

The WS-BA coordinator used in this prototype is not fully compliant to the WS-BA specification. As described in Chapter 1 the task of this prototype is to show that external transaction processing for BPEL processes is possible. Therefore the coordinator only implements the functionality needed by the prototype. Additional requirements presented in the WS-BA specifications such as a more specific fault handling, WS-Policy and WS-Security compliance as well as different protocol types are not in the scope of this diploma thesis.

5.7 Concurrency Considerations

The ActiveBPEL engine and therefore also the PureBPE allow the concurrent execution of BPEL activities inside a process instance. Concurrent execution of two activities can be accomplished by placing both activities inside parallel control flows inside a `flow` activity. Additionally BPEL allows the specification of event handlers that are associated with scopes. Activities nested inside event handlers are executed concurrently to activities nested in the scope the event handler belongs to.
Concurrent execution of activities in the PureBPE requires support for concurrency in the external coordinator. Regarding the activities that trigger actions in the external coordinator concurrency issues have to be considered for the `scope` activity and external activities, i.e. the `compensate` activity.

5.7.1 Concurrent Scopes

Since all BPEL4Tx modules start new process instances for every new scope that is initialized in the PureBPE, scopes that are executed concurrently are taken care of by this mechanism. One problem, however, is the concurrent registration of scopes with the WS-C coordinator. Two scopes that are started simultaneously by the PureBPE will register concurrently at the WS-C coordinator. A straightforward

implementation of the registration handling in the WS-C coordinator would use event handlers that can handle register messages concurrently. Serializable scopes could be used to handle the shared variable access of the event handlers. However, the ActiveBPEL engine in the version used here, has a problem with event handlers associated with scopes and serializable scopes. If serializable scopes inside an event handler need to access the message that triggered the event handler they will never acquire the necessary lock for the message. This is the case because the event handler still holds the lock for the message.
Using non-serializable scopes is dangerous since important data could be overwritten in non-serializable access of several event handlers to a variable.
To achieve serializable behavior the following work-around is used. The participant registration handling is performed in a `while` loop. Having processed one `Register` message the control flow returns to the beginning of the while loop and processes the next register message. Since the ActiveBPEL engine allows the queuing of incoming messages, no messages get lost and serializable behavior is achieved.
The same workaround is done in the WS-BA coordinator when concurrent `AddParticipant` and `ReportStatusChange` messages arrive.

5.7.2 Concurrent External Activities

In case of concurrent external activities that appear in the same scope one BPEL4Tx ordering broker and one BPEL4Tx WS-BA participant process instance have to handle several `execActivity` requests concurrently. Again, the same problem identified in Subsection 5.7.1 with the ActiveBPEL engine is to be considered. Therefore the same workaround as explained above is used for the processing of external activities, the `execActivity` handling is put into a `while` activity as explained for the respective processes.

5.8 Summary

Chapter 5 describes the modifications made to the ActiveBPEL engine in order to turn it into a PureBPE as specified in Chapter 4. Additionally the ActiveBPEL engine has been connected to a JMS messaging provider, enabling it to send and receive scope life cycle messages necessary for the external coordination of BPEL long-running transactions.
In a second step, the external coordinators, the BPEL4Tx and the WS-C / WS-BA coordinator have been implemented using BPEL. The architecture of the BPEL processes representing the coordinators is described in detail using the mapping rules from regular expressions to BPEL discussed in Appendix A.

6 Performance Comparisons

Business processes can represent business-critical applications that heavily depend on a fast execution of the business process.
By design, a BPEL engine using external transaction control will be slower than one using internal transaction control. The fact that messages need to be exchanged between the PureBPE and the external coordinator instead of local procedure invocations inside the PureBPE is the most important aspect why an externally controlled BPEL engine is slower than one that controls its transactions internally. Other factors specific to this prototype are the fact that SOAP messages serialized to literal XML are exchanged which require a substantial effort to serialize and deserialize. Additionally, the external coordinators are implemented in BPEL which requires additional middleware to execute these processes. A direct execution of the coordinator functionality by an executable program will possibly be faster. This chapter deals with a performance comparison between a normal ActiveBPEL engine and the prototype setup described in Chapter 5 in order to give an overview on how much external transaction processing affects the performance of the execution of a business process.

6.1 Test Environment

This section describes the hardware used to perform the tests as well as the BPEL processes that were executed. Then the methods how the execution time of a process is determined is described.

6.1.1 Hardware

All tests were executed on one machine with the following specifications:

- 1,60 GHz Intel Pentium M processor.
- 1,25 GB of Ram.

In order to minimize network influence both ActiveBPEL installations for the external transaction processing case as well as the ActiveMQ messaging provider were installed on the same machine.

6.1.2 Test Processes

Two exemplary processes have been chosen to show the influence of external transaction coordination on the performance of the execution of a process: The "loan approval" sample process as an example for a simple process and a more sophisticated process involving the execution of fault and compensation handlers.

6.1.2.1 The Loan Approval Sample Process

This process is taken from the BPEL specification. It does not contain any nested scopes and completes normally, so only the initialization and completion of the process require the invocation of the external coordinator. All external partner calls have been replaced by `assign` activities in order to minimize non-deterministic influences caused by network delays on the duration of the whole process. The `assign` activities simply assign a static value to the output variable of the partner invocation so that the process can continue normally. Figure 6.1

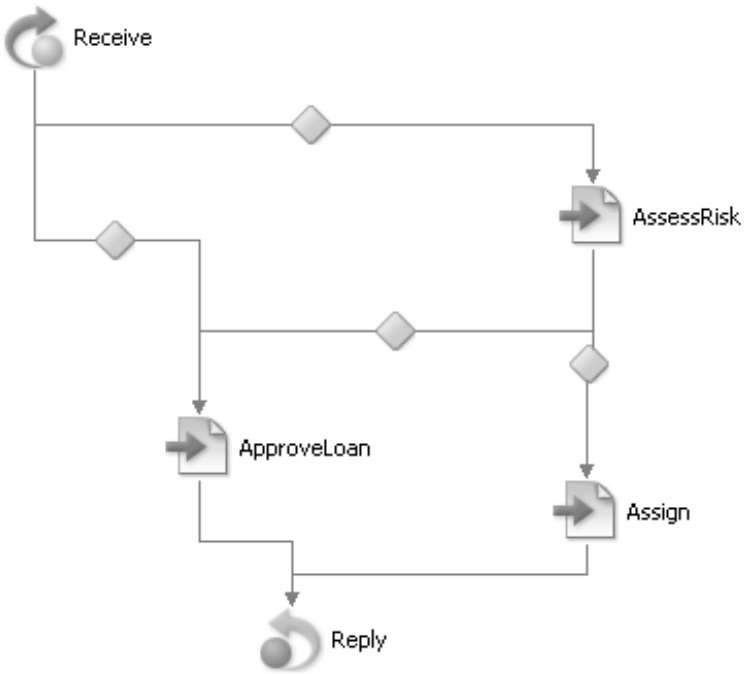

Figure 6.1: Control Flow of the loan approval process.

shows the control flow of the loan approval process. Having received the initial message the flow evaluates the contents and eventually sends a response back.

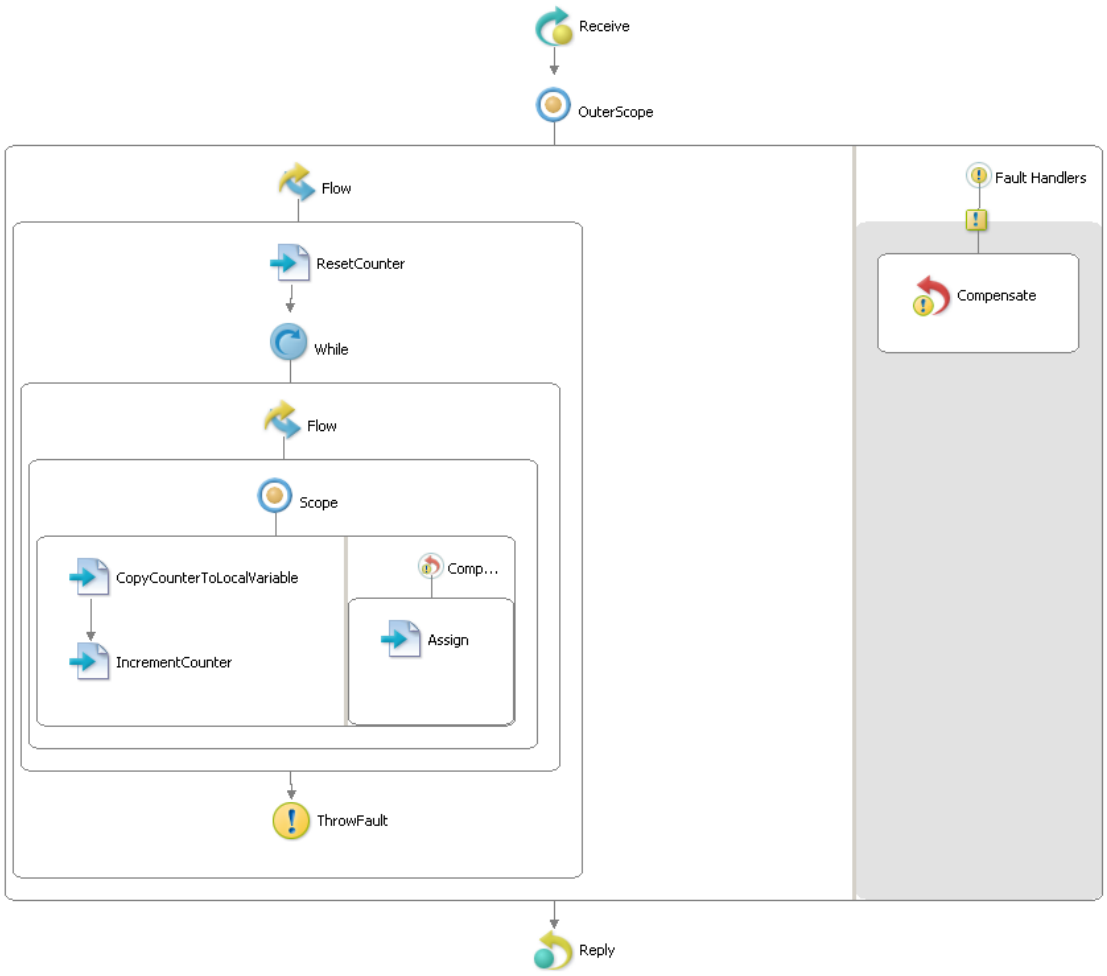

Figure 6.2: Control flow of the process with fault and compensation handlers.

6.1.2.2 The Faulting Process

Figure 6.2 shows the faulting process. The control flow for this process is the following. On receipt of the initial message the `Receive` activity is executed. Then the process starts the "OuterScope". In the "OuterScope" the `integer` counter is set to zero by the `ResetCounter` `assign` activity and the `while` activity is started. The `while` loop executes 10 times. Each time the first `assign` activity copies the current value of the counter to a local variable called counterValue. The second `assign` activity increments the counter by one. Both `assign` activities are nested inside the "ScopeInWhile" `scope` activity.
Having executed the `while` activity 10 times the "'ThrowFault" `throw` activity which throws a fault is executed. The fault is caught by the fault handler of the "OuterScope". This fault handler in turn contains a scope (the "ScopeInFH") which has a nested `compensate` activity. This `compensate` activity compensates all nested scopes of the "OuterScope". This means that the "ScopeInWhile" `scope` has to be compensated 10 times (since it completed 10 times during the while ac-

tivity). Every time the "ScopeInWhile" `scope` gets the signal to compensate its compensation handler is executed and the value of the local variable "counterValue" is copied to the reply message.

6.2 Data Gathering

This section describes how the performance of the business process execution is measured.
The ActiveBPEL engine records the time at which a business process instance is started and the time at which it has reached a final state (i.e. finished or faulted). These values are written to the database. To determine how long the execution of a process has taken the difference between the end and beginning time is calculated. In order to minimize external influences, each process is executed 10 times with either setup and the average of the execution time is taken.

6.3 Results

Table 6.1 shows the results that have been measured during the performance tests.

Process	Internal TA-Processing	External TA-Processing
loan approval	14 ms	8572 ms
faulting process	20 ms	45419 ms

Table 6.1: Performance comparison results.

6.3.1 Result Explanation

The results presented above show a dramatic increase of processing time even for simple processes. In order to show why the increase is so big and to give a starting point for further performance optimizations a breakup of the total processing time into several parts is made below.
The loan approval example is used for further explanation as it is fairly simple so a detailed examination is easily possible. On average the loan approval example takes 8558 ms longer with external transaction processing than with internal transaction processing.
Tweaking the persistence settings of the involved components show where the extra time is coming from.
With persistence turned off in the ActiveBPEL engine running the external coordinators as well as in the JMS messaging provider, the loan approval process takes

390 ms longer with external transaction processing than with internal transaction processing.

6.3.1.1 PureBPE Performance

Message processing takes about 10 ms on average for each of the four messages involved in the loan approval example. Message processing includes serialization/deserialization and reading/writing to and from the queue. Initializing the process in the external coordinator takes 160 ms on average while it takes the external coordinator 190 ms to accomplish the processing of the `npmEnded` message. Adding these numbers amounts to the extra 390 ms needed with the external transaction processing.

6.3.1.2 External Coordinator Performance

In order to understand while it takes the external coordinator 160 ms to initialize the process from the example discussed above, the amount of messages involved in this action has to be considered. During the initialization 12 messages are exchanged. For every message an average processing time of 10 ms has been identified above. Message processing therefore amounts to 120 ms alone. The rest of the time is needed for the execution of the BPEL process instances that constitute the external coordinator.

6.3.1.3 Persistence Influences

Having identified the additional processing time needed by message processing and execution of the coordinator BPEL processes the influence of persistence settings in the external components on the total processing time for the loan approval example is identified here.

- Using transacted JMS Sessions adds 141 ms of processing time.
- Persistent messaging turned on in the ActiveMQ message provider adds 1652 ms of processing time.
- Persistent setup of the ActiveBPEL engine running the external coordinator takes 6375 ms.

These numbers amount to 8572 ms, the average amount the loan approval example needs to complete with external transaction processing.
Considering the influences on the overall performance, performance optimizations must mainly target the middleware used. Persistence in the ActiveMQ message provider and even more in the ActiveBPEL engine is obviously a bottleneck. Additionally serialization and deserialization of messages must be optimized (for example by sending Java object messages instead of literal SOAP-XML messages).

Furthermore the Apache Axis Web-Service container used by the ActiveBPEL engine is Axis version 1.2 while version 1.4 that is faster is already available. A message provider that allows faster queuing of messages is needed too. A lot of message processing could also be avoided by implementing the external coordinators in a "classic" programming language such as Java or C++.
Distributing the prototype over several machines, for example one machine that runs the PureBPE, one that runs the messaging provider, one that runs the external coordinator and a database server, should improve the performance further. However, these optimizations are out of the scope of this thesis but may be the basis for further work.

6.4 Summary

In Chapter 6 the performance differences of internal and external coordination of BPEL long-running transactions are examined using the prototype implemented in the scope of this thesis. The reasons for the substantial decrease in performance caused by external transaction coordination are identified and solutions on how to eliminate these bottlenecks are given.

7 Summary and Outlook

This thesis presents a solution on how to extract long-running transactions from BPEL and coordinate these transactions with an external WS-Business Activity based coordinator. The work presented here is based on discussions in [38] and Appendix C of the BPEL specification [18].
In order to separate LRTs from BPEL three steps have been discussed. Firstly, transaction specific information has to be separated from BPEL. Therefore the transaction relevant elements of a BPEL process, mainly scopes, fault and compensation handlers, have been identified. The term PureBPE has been introduced for a BPEL engine that delegates long-running transactions to an external coordinator. Messages to be exchanged between the PureBPE and the external coordinator have been specified. The term life cycle messages has been introduced for these messages as they coordinate the life cycle of scopes inside a BPEL process. Secondly, the BPEL4Tx coordinator, a piece of transaction specific middleware that translates scope life cycle messages to a coordination protocol, has been specified. The third step is a WS-Business Activity based coordinator which is used to control the life cycles of scopes inside a BPEL process. It is shown that the protocols presented in WS-BA, namely the business agreement with participant completion protocol is not sufficient to fully control BPEL's long-running transactions from the outside.
Therefore a modified business agreement with participant completion protocol (mBAP) has been introduced.

A prototype based on Active Endpoint's ActiveBPEL engine has been implemented to provide a practical example of the theoretical foundations. The ActiveBPEL engine has been modified to be able to send and receive scope life cycle messages that can be evaluated and processed by an external coordinator. Therefore the ActiveBPEL engine was enhanced so that it can connect to a JMS messaging system and send and receive SOAP messages via JMS.
BPEL processes that implement the BPEL4Tx transaction middleware functionality have been implemented. In addition to that a WS-C and a WS-BA coordinator have been implemented in BPEL.

Several points remain open and can be the foundation for further work. The invocation of external partners from BPEL using coordination contexts based on WS-Policy must be specified and implemented. Additionally a correlation mechanism for BPEL that relies on WS-Addressing is needed in order to make the external WS-C and WS-BA based coordinators fully compliant with their respec-

tive specifications.
In order to support BPEL 2.0 in the future, additional life cycle handlers such as a termination handler have to be incorporated in the concept.
In Appendix A a basic mapping from regular expressions to BPEL has been introduced. A more elaborate approach could be the topic of further work in this field.

In order to put the prototype into production, its performance, especially with persistence settings turned on, must be improved. Hints on how this can be done are given in Chapter 6.

List of Figures

2.1 Life cycle of a scope. 7
2.2 ActiveBPEL engine architecture. 11

3.1 Bike service example. 17
3.2 Transaction tree of the bike service example. 18

4.1 Architecture of a BPEL engine with internal / external transaction coordination. 21
4.2 Scope life cycle messages. 23
4.3 Mapping of BPEL scopes to WS-Coordination. 26
4.4 Message exchange for initialization of scopes. 28
4.5 States and messages of the WS-BA business agreement with participant completion protocol. 30
4.6 The mBAP completion protocol. 33
4.7 Message exchange on completion of a scope. 34
4.8 Message exchange on faulting of a scope. 35
4.9 Message exchange to terminate a scope. 37
4.10 Message exchange when a fault is handled implicitly. 40
4.11 Message exchange when a fault handler completes normally. 41
4.12 Message exchange when a fault handler faults. 43
4.13 Message exchange for an external compensate activity that completes normally. 45
4.14 Message exchange for an external compensate activity that completes with a fault. 45
4.15 Message exchange for compensation with normal completion of the compensation handler. 47
4.16 Message exchange for compensation with a faulting compensation handler. 48
4.17 Message exchange for compensation with the implicit compensation handler. 49
4.18 Message exchange for closing of scopes. 50

5.1 Prototype architecture overview. 51
5.2 Infinite recursion with external transaction processing. 53
5.3 Architecture of the PureBPE. 57
5.4 New scope states in the ActiveBPEL Engine. 66

5.5 JMS-URL Syntax 69
5.6 Architecture of the JMS-enabled ActiveBPEL engine. 71
5.7 Deployment of a BPEL service reachable via JMS. 73
5.8 Role of the BPEL4Tx ordering broker. 76
5.9 State diagram for the normal processing mode. 77
5.10 BPEL process with handler with nested scopes. 79
5.11 BPEL4Tx ordering broker control flow after npmEnded. 81
5.12 BPEL4Tx ordering broker control flow after npmFaulted. 83
5.13 Visualization of the BPEL4Tx coordinator BPEL process. 84
5.14 Exchange of registration service endpoint reference information. . . 87
5.15 Exchange of endpoint reference information during registration of a participant. 87
5.16 Paths through the parts of the BPEL4Tx WS-BA participant. . . . 89
5.17 Architecture of the WS-C / WS-BA registration part. 99
5.18 Message exchange in case of a CompensateSubScopeSuggestion message. 104

6.1 Control Flow of the loan approval process. 108
6.2 Control flow of the process with fault and compensation handlers. . 109

A.1 State diagram with a loop. XVII
A.2 State diagram with a loop and different outcomes. XIX

B.1 Message Visualizer screen shot. XXI

Listings

5.1 WSDL port definition with a JMS URL. 70
5.2 WS-Addressing endpoint reference with a JMS URL. 70
5.3 Xpath expression to determine whether a scope has children. 91
5.4 Xpath expression to determine outcome mode. 91
5.5 Xpath expression to determine whether a scope has a compensation handler. 93
A.1 BPEL code for a sequence. XVI
A.2 BPEL code for an alternation. XVII
A.3 BPEL code for a loop construct. XVIII
A.4 BPEL code for a generic loop construct. XIX

List of Tables

4.1 Outgoing scope life cycle messages. 23
4.2 Incoming scope life cycle messages. 24

6.1 Performance comparison results. 110

Bibliography

[1] Active Endpoints. *ActiveBPEL Engine*. http://www.activebpel.org.

[2] Active Endpoints. *ActiveBPEL Engine Architecture*. http://www.activebpel.org/docs/architecture.html.

[3] Active Endpoints. *PDD Format for ActiveBPEL 1.X*. http://www.activebpel.org/docs/pdd1.html.

[4] Active Endpoints. *Active Endpoints - Support Forum : multiple instances of the same service*. http://forums.active-endpoints.com/showthread.php?t=736, June 2005.

[5] Active Endpoints. *Active Endpoints - Support Forum : Which BPEL xsd is the designer using*. http://forums.active-endpoints.com/showthread.php?p=6525, June 2006.

[6] *ActiveMQ*. http://www.activemq.org.

[7] Apache Software Foundation. *Apache Axis Architecture Guide 1.2 version*. http://ws.apache.org/axis/java/architecture-guide.html.

[8] Apache Software Foundation. *Apache License Version 2.0*. http://www.apache.org/licenses/LICENSE-2.0.html.

[9] Apache Software Foundation. *Apache Tomcat*. http://tomcat.apache.org/.

[10] Arjuna, Hitachi, BEA, IBM, IONA and Microsoft. *Web Services Atomic Transaction (WS-Atomic Transaction) 1.0*. http://www.ibm.com/developerworks/library/specification/ws-tx/#atom, August 2005.

[11] Arjuna, Hitachi, BEA, IBM, IONA and Microsoft. *Web Services Business Activity Framework (WS-Business Activity) 1.0*. http://www.ibm.com/developerworks/library/specification/ws-tx/#ba, August 2005.

[12] Arjuna, Hitachi, BEA, IBM, IONA and Microsoft. *Web Services Coordination 1.0 WSDL file*. http://www-128.ibm.com/developerworks/library/specification/ws-tx/wscoor.wsdl, August 2005.

[13] Arjuna, Hitachi, BEA, IBM, IONA and Microsoft. *Web Services Coordination (WS-Coordination) 1.0.* http://www.ibm.com/developerworks/library/specification/ws-tx/#coor, August 2005.

[14] Arjuna, Hitachi, BEA, IBM, IONA and Microsoft. *Web Services Business Activity 1.0 WSDL file.* http://www-128.ibm.com/developerworks/library/specification/ws-tx/wsba.wsdl, August 2005.

[15] BEA, IBM and Microsoft. *Web Services Addressing (WS-Addressing).* http://dev2dev.bea.com/pub/a/2003/03/ws-addressing.html, March 2003.

[16] BEA, IBM and Microsoft. *Web Services Transaction (WS-Transaction).* http://dev2dev.bea.com/pub/a/2004/01/ws-transaction.html, January 2004.

[17] BEA, IBM, Microsoft and TIBCO. *Web Services Reliable Messaging Protocol.* http://www-128.ibm.com/developerworks/library/specification/ws-rm/, Feburary 2005.

[18] BEA, IBM, Microsoft, SAP and Siebel Systems. *Business Process Execution Language for Web Services (BPEL4WS) 1.1.* http://www.ibm.com/developerworks/library/ws-bpel, May 2003.

[19] BEA, IBM, Microsoft, SAP, Sonic and VeriSign. *Web Services Policy Framework (WSPolicy).* http://www-128.ibm.com/developerworks/library/specification/ws-polfram/, March 2006.

[20] Chappel, D., Chun, R., and Meritt, J. *JMS URL Syntax for the Apache Axis Project.* http://marc.theaimsgroup.com/?l=axis-dev&m=105613809524476&q=p3, 2003.

[21] Coleman, J. *Examining BPEL's Compensation Construct.* http://rodin.cs.ncl.ac.uk/Publications/Coleman-ExaminingBPEL.pdf, March 2005.

[22] Corda, Ugo. *WS BPEL issues list. Issue 94: Allow both compensate and other activities in compensation or fault handler.* http://www.oasis-open.org/apps/group_public/download.php/12233/wsbpel_issues_list.html #Issue94.

[23] Free Software Foundation. *GNU GENERAL PUBLIC LICENSE. Version 2.* http://www.gnu.org/licenses/gpl.html, June 1991.

[24] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison Wesley, March 1995.

[25] Garcia-Molina, H., Gawlick, D., Klein, J., Kleissner, K., and Salem, K. Modeling long-running activities as nested sagas. *IEEE Data Eng. Bull. 14*, 1 (1991), 14–18.

[26] Garcia-Molina, H., and Salem, K. *Sagas.* In *SIGMOD Conference* (1987), U. Dayal and I. L. Traiger, Eds., ACM Press, pp. 249–259.

[27] Garett, J. J. *Ajax: A New Approach to Web Applications.* http://www.adaptivepath.com/publications/essays/archives/000385.php.

[28] Gray, J. The transaction concept: Virtues and limitations (invited paper). In *VLDB* (1981), IEEE Computer Society, pp. 144–154.

[29] Hapner, M., Sharma, R., Burridge, R., Fialli, J., and Haase, K. *Java Message Service API tutorial and reference: messaging for the J2EE platform.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[30] Härder, T., and Rahm, E. *Datenbanksysteme: Konzepte und Techniken der Implementierung.* Springer, 1999.

[31] Härder, T., and Reuter, A. Principles of transaction-oriented database recovery. *ACM Comput. Surv. 15*, 4 (1983), 287–317.

[32] Hohpe, G., and Woolf, B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley, 2004.

[33] Leymann, F., and Roller, D. *Production workflow: concepts and techniques.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[34] Moss, J. E. B. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, 1981.

[35] MySQL AB. http://www.mysql.com.

[36] Nitzsche, J. *Entwicklung eines Monitoring-Tools zur Unterstützung von parametrisierten Web Service Flows.* Diplomarbeit: Universität Stuttgart, Institut für Architektur von Anwendungssystemen, March 2006.

[37] Organization for the Advancement of Structured Information Standards (OASIS). *Universal Description, Discovery, and Integration (UDDI).* http://www.uddi.org/whitepapers.html.

[38] Pottinger, S. *Private communication about external WS-BA based transaction processing for BPEL*, April 2006.

[39] Rusinkiewicz, M., and Sheth, A. P. Specification and execution of transactional workflows. In *Modern Database Systems: The Object Model, Interoperability, and Beyond.* 1995, pp. 592–620.

[40] Schmidt, K., and Stahl, C. *A Petri net semantic for BPEL*, 2004.

[41] Schöning, U. *Theoretische Informatik - kurzgefasst, 4. Auflage.* Spektrum Akad. Verl., 2001.

[42] Shah, G. *Axis meets MOM: Reliable Web services with Apache Axis and MOM.* http://www.javaworld.com/javaworld/jw-02-2006/jw-0220-axis.html, February 2006.

[43] SUN Microsystems. *Java Servlet 2.4 Specification.* http://jcp.org/aboutJava/communityprocess/final/jsr154/index.html.

[44] Sun Microsystems. *Java Message Service Specification Version 1.1.* http://java.sun.com/products/jms/docs.html, March 2002.

[45] W3C. *SOAP Version 1.2.* http://www.w3.org/TR/soap12.

[46] W3C. *XML Path Language (XPath) Version 1.0.* http://www.w3.org/TR/xpath.

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

[48] Weikum, G., and Schek, H.-J. Concepts and applications of multilevel transactions and open nested transactions. In *Database Transaction Models for Advanced Applications.* 1992, pp. 515–553.

[49] Wohed, P., van der Aalst, W. M., Dumas, M., and ter Hofstede, A. H. *Pattern Based Analysis of BPEL4WS*, 2002.

[50] Wutke, D. *Erweiterung einer Workflow-Engine zur Unterstützung von parametrisierten Web Service Flows.* Diplomarbeit: Universität Stuttgart, Institut für Architektur von Anwendungssystemen, March 2006.

All links followed on July 11th 2006.

A A Basic Mapping of Regular Expressions to BPEL

This section describes a straightforward way to map a regular expression to the structure of a BPEL process.
The need for this mapping arises if the requirements for the BPEL4Tx ordering broker are examined in an abstract way. From an abstract point of view the BPEL4Tx ordering broker is a business process that receives a sequence of messages from the PureBPE as well as from the BPEL4Tx WS-BA participant. Upon receipt of a message the BPEL4Tx ordering broker changes into a new state depending on the type of message. Upon changing to the new state it sends out messages to the PureBPE or the BPEL4Tx WS-BA participant.
This observation shows that the control flow of the BPEL4Tx ordering broker depends on the messages it receives, a behavior similar to a finite state machine if the sequence of incoming messages is seen as the input and all other activities besides those accepting input messages and structured activities are omitted.
It must even be a deterministic finite state machine or deterministic finite automaton (DFA) considering the fact that the BPEL specification considers "the semantics of a process in which two or more receive actions for the same partner link, portType, operation and correlation set(s) may be simultaneously enabled" as "undefined" [1].
A DFA is a 5-tuple which consists of

- A finite set of states (here: the states of the scope to be handled by the BPEL4Tx ordering broker).
- A finite set called the alphabet (here: All messages can then be received by the BPEL4Tx ordering broker). These messages are identified in Chapter 4.
- A transition function (here: The sequence of messages representing a scope life cycle).
- A start state (here: the state the scope is in before it gets executed by the PureBPE).
- A set of accept states (here: the final state of a scope after its execution has been finished).

[1] BPEL Specification [18] page 56

DFA's accept regular languages that in turn can be represented by regular expressions. [41] Considering the message exchange identified in Chapter 4 a regular expression representing the sequences of messages that need to be accepted by the BPEL4Tx ordering broker can be built.
The resulting regular expression as well as the output and transformation activities of the ordering broker are explained in detail in the following Section 5.5.1.
Here, only a basic general mapping of regular expressions to BPEL activities is given.

A.1 Related Work

An analysis of BPEL regarding common patterns that occur in workflows and their mapping to BPEL has been done in [49]. Several of the atomic structures of a regular expression have been covered in [49].
A petri net semantic for BPEL has been introduced in [40]

A.2 Atomic Structures

Regular expressions are built of a set of atomic structures, concatenation, alternation and the Kleene star. These structures need to be mapped to a BPEL control flow. The following paragraphs propose a mapping for these building blocks. A "pseudo-BPEL" notation is used to simplify the notation.

A.2.1 Concatenation

A sequence of two input symbols *m1* and *m2* where *m1* must appear before *m2* is represented in a regular expression by `m1 m2`. In a BPEL process a sequence of two messages would simply be represented by two `receive` activities inside a `sequence` activity. Alternatively the two `receive` activities can also be nested inside a `flow` activity. Then a `link` with source activity "receive m1" and target activity "receive m2" needs to be modeled to achieve the same semantics as `m1 m2` in the regular expression.

```
<sequence>
   <receive message="m1">
   <receive message="m2">
<sequence>
```

Listing A.1: BPEL code for a sequence.

This structure corresponds to the workflow pattern 1 (WP1) "Sequence" presented in [49].

A.2.2 Alternation

A choice is expressed in a regular expression through a "|" symbol, *m1* or *m2* is written as $m1|m2$. In a BPEL process this can be represented by a `pick` activity with two `onMessage` events waiting for message `m1` and message `m2` respectively as shown in Listing A.2.
This structure somewhat corresponds to the workflow pattern 4 (WP4) "Exclusive Choice" in [49]. The difference, however, is that the solution proposed here only deals with alternate paths that are taken because of the received message, while WP4 in [49] deals with alternate paths taken based on variable data. Therefore in [49] the use of the `switch` activity or links with transition conditions is proposed.

```
<pick>
   <onMessage message="m1">
   <!-- activities to be executed after m1 was received -->
   </onMessage>
   <onMessage message="m2">
   <!-- activities to be executed after m2 was received -->
   </onMessage>
</pick>
```

Listing A.2: BPEL code for an alternation.

Here only the `pick` activity can be used since the further path to be taken by the control flow is determined by the message that is received. The use of a flow with two parallel `receive` activities has another semantic. If one `receive` activity is triggered the other one does not transition to dead path but continues waiting for its message. An exclusive behavior is needed here so the pick activity is used. The `onMessage` event for the received message executes and all other `onMessage` events transition to death path.

A.2.3 Kleene Star

A regular expression $m1m2^*m3$ means that first a symbol *m1* has to be received, then zero or more symbols *m2* then a symbol *m3*. The symbol `*` also called the Kleene star denotes that an input can be received zero or more times.
Illustrated as a state diagram which can be built from a regular expression using common mapping rules the problem is the following: Using the state diagram no-

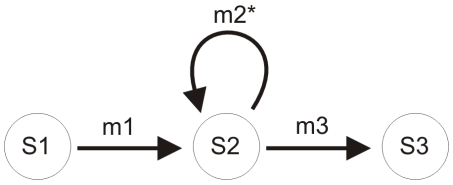

Figure A.1: State diagram with a loop.

tation the problem becomes more obvious. A straightforward solution in BPEL would be three `receive` activities with three links. But BPEL can only describe acyclic graphs. Edges (or links in BPEL terminology) such as the one in Figure A.2.3 going from state `S2` to `S2` are not permitted in BPEL. A structured activity, which permits the execution of a loop for an amount of times not specified at design time of the process, is needed. The BPEL `while` activity can do exactly that. Listing A.3 shows a solution for the problem presented above. Having received message `m1` the control flow enters the while loop and the nested `pick` activity. Now either a `m2` or a `m3` message can be received. On receipt of a `m2` message the `onMessage` event for `m2` is executed (and possible activities following it in the control flow). Then the loop is finished and returns to the `pick` activity. Again the process is waiting for a `m2` or `m3` message. This can be done infinitely.

```
<sequence>
  <receive message="m1">
  <while loopCondition="true">
    <pick>
      <onMessage message="m2">
         <!--activities after m2 was received-->
      </onMessage>
      <onMessage message="m3">
        <sequence>
          <assign false to loopCondition>
        </sequence>
      </onMessage>
    </pick>
  </while>
  <!--activities after m3 was received-->
</sequence>
```

Listing A.3: BPEL code for a loop construct.

If a `m3` message is received in this situation the loop is immediately ended by assigning a `false` boolean value to the `loopCondition`. Having ended the `while` loop activity the control flow now executes the activities that come after `m3`.

The solution presented above is enough for simple cases like the one above where cycles in the state diagram graph either are executed completely or not at all and after the loop ended exactly one path is taken. Regular expressions (and therefore state diagrams) permit alternation after a loop has finished.
The regular expression $(m_2m_3)^*((m_4m_7)|(m_1|(m_2m_6)|m_5))$ which translates into the state diagram shown in Figure A.2.3 shows such a problem.
With an `m4` message the `m2 m3` loop can be exited. Additionally after the $(m_2m_3)^*$ loop has finished regularly (with a m_3 message), either an m_1 or m_4 message must be accepted.
A solution in BPEL similar to that above would be to use a `pick` activity which accepts m_1, m_2 and m_4 messages. On receipt of a m_1 or m_4 message the loop terminates. This is a different semantic than that of the regular expression after

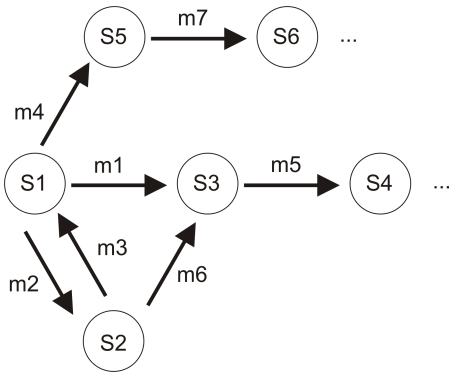

Figure A.2: State diagram with a loop and different outcomes.

receiving a m_4 message the process should be in a different state than after receiving a m_1 message since other activities might follow m_4 leading to a solution where all `receive` activities for subsequent messages of m_1 and m_4 are placed inside the respective `onMessage` event.

```
<flow>
  <while loopCondition="true">
    <pick>
      <onMessage message="m1">
         <assign "S3" to loopEndCondition>
         <assign false to loopCondition>
      </onMessage>
      <onMessage message="m2">
         <pick>
           <onMessage message="m3">
              <!--activities after m3 was received-->
           </onMessage>
           <onMessage message="m4">
              <assign "S3" to loopEndCondition>
              <assign false to loopCondition>
           </onMessage>
         </pick>
      </onMessage>
      <onMessage message="m4">
        <sequence>
          <assign "S5" to loopEndCondition>
          <assign false to loopCondition>
        </sequence>
      </onMessage>
    </pick>
  </while>
  <link from while to receive m6 if loopEndCondition="S3">
  <link from while to receive m7 if loopEndCondition="S5">
  ...
</flow>
```

Listing A.4: BPEL code for a generic loop construct.

However, such a solution prevents the edge from S2 to S3 because it would require a BPEL link from the `onMessage` event which handles m_2 and subsequent messages to the `onMessage` event that handles m_1 and following messages. Since the semantics of a pick activity are that exactly one of the available `onMessage` events executes, links across those events are not a good idea. On execution of one event the others are set to dead path.
What is needed is a more generic loop construct.
The pseudo-BPEL in Listing A.4 presents such a generic loop construct that can cope with the regular expression presented above. The `loopEndCondition` variable saves the outcome of the loop. After the `while` activity has finished the following links ensure that the right paths depending on the paths taken inside the loop, are taken. Therefore links across `onMessage` events are not needed anymore and a loop can end with different results.

A.3 Summary

The solutions for the mapping of regular expressions to BPEL presented above have been developed to map the regular expressions for the BPEL4Tx ordering broker and BPEL4Tx WS-BA participant, which are presented later. They might not represent a total mapping of regular expressions to BPEL, or a proof that this is even possible. This might be the focus of different work. The need for a more elaborate mapping based on formal rules is recognized but is out of the scope for this diploma thesis.

B BPEL4Tx Message Visualizer

The BPEL4Tx Message Visualizer is a tool to analyze the life cycle messages of a scope. The tool generates a graphical representation of life cycle messages and coordination protocol messages similar to the figures used in Chapter 4. Coordinator internal messages are not displayed.
The tool works only if the JMS destinations for the exchange of scope life cycle messages are defined as topics. Then the Message Visualizer listens on these topics and saves the message sequence for each scope.
From a technical point of view the BPEL4Tx Message Visualizer is a servlet that listens on the topics and generates a HTML representation of the message sequence. The servlet is polled periodically by a HTML file using Ajax (Asynchronous JavaScript and XML) [27]. The message sequence is then displayed in the HTML file. Figure B.1 shows a screen shot of the Message Visualizer displaying the "loan approval" sample process's life cycle messages.

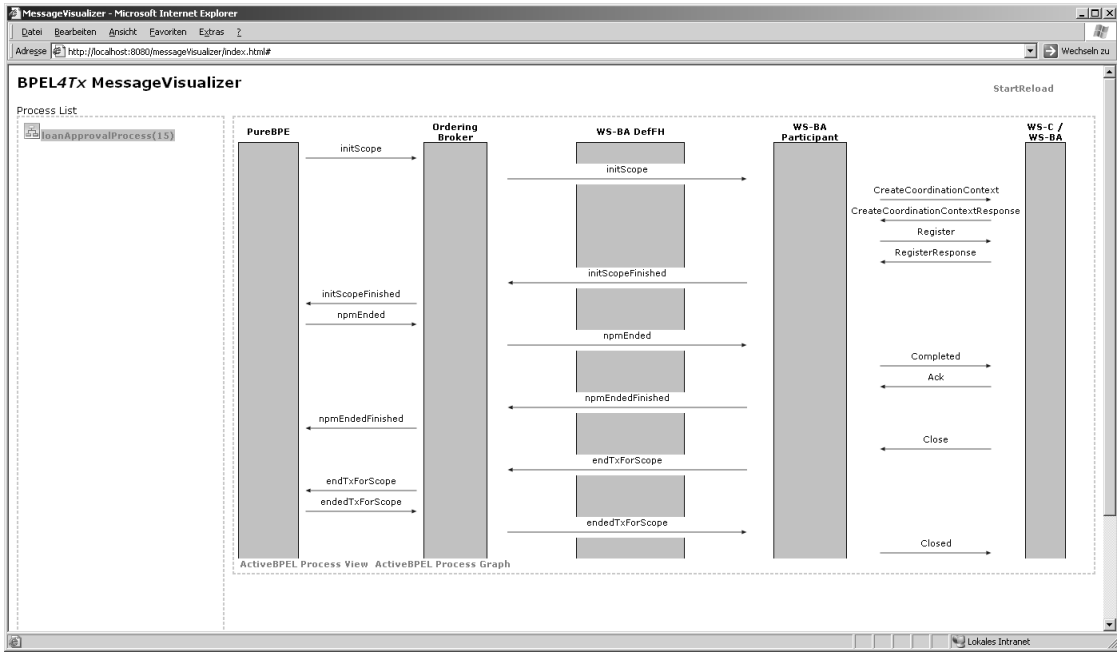

Figure B.1: Message Visualizer screen shot.

Erklärung

Hiermit versichere ich, diese Arbeit
selbständig verfasst und nur die
angegebenen Quellen benutzt zu haben.

(Ralph Mietzner)