Institut für Architektur von Anwendungssystemen
Arbeitsbereich: Workflow Technology
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3006

Unterstützung für externe Transaktionen in Apache ODE

Sebastian Henke

Studiengang:	Informatik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. Oliver Kopp
begonnen am:	4. Januar 2010
beendet am:	6. Juli 2010
CR-Klassifikation:	C.2.4, H.4.1, H.2.4, H.3.5

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

Die *Business Process Execution Language* (BPEL) [OAS07] ist de facto Standard für Workflow-Beschreibungssprachen. BPEL ermöglicht das Erstellen von Geschäftsprozessmodellen, die von einer BPEL-fähigen Workflowmaschine instanziiert und ausgeführt werden können. Dabei aggregiert BPEL Web Services und bietet diese wiederum als Web Services an. Geschäftsprozesse sind typischerweise langlaufend, daher stellt BPEL einen Transaktionsmechanismus vor, der auf langlaufenden Transaktionen basiert. In BPEL werden *Scopes* definiert, die einen Sichtbarkeits- und Gültigkeitsbereich innerhalb eines Prozesses deklarieren. Diese Scopes werden über *Fault-*, *Compensation-* und *Termination-Handler* zu den transaktionalen Aktivitäten eines Prozesses.

WS-Coordination [OAS09c] ist ein standardisiertes Framework für web-service-basierte, kontextorientierte Koordinationsprotokolle, mit dem sich Transaktionen realisieren lassen. Aufbauend auf WS-Coordination existieren die beiden Frameworks *WS-Business Activity* (WS-BA) [OAS09b] und *WS-Atomic Transaction* (WS-AT) [OAS09a]. Die beiden definieren *Coordination Types* und *Coordination Protocols*, die von Web Services verwendet werden können, um Transaktionen zwischeneinander zu realisieren. WS-BA beschreibt Coordination Types und Protocols für langlaufende Transaktionen, während WS-AT Coordination Types und Protocols für ACID-Transaktionen spezifiziert.

Im Mittelpunkt dieser Arbeit steht die Verbindung der transaktionalen Mechanismen von BPEL mit den transaktionalen Mechanismen von WS-BA und WS-AT. Bei Transaktionen handelt es sich um *nicht-funktionale Anforderungen*. Üblicherweise wird versucht, in Anwendungen nicht-funktionale Anforderungen von *funktionalen Anforderungen* – der Geschäftslogik – zu trennen. Dazu wird beim Deployment der Geschäftslogik angegeben, welche nicht-funktionalen Anforderungen für welche Teile der Geschäftslogik gelten. Diese werden dann von der Middleware realisiert.

In dieser Arbeit werden am Beispiel von Apache ODE Verfahren beschrieben, wie eine BPEL-Workflowmaschine um die Fähigkeit zur Unterstüztung von WS-BA- und WS-AT-Transaktionen erweitert werden kann.

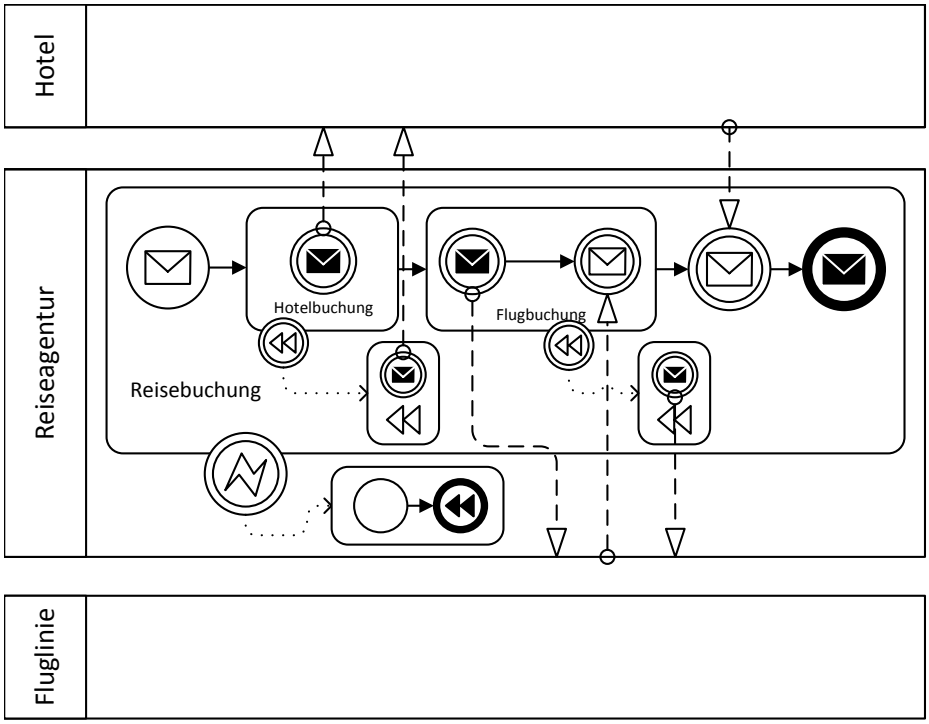

Abbildung 1.1: Einführendes Beispiel

1.1 Beispiel zur Motivation

Abbildung 1.1 zeigt den Buchungsvorgang einer Reiseagentur. Erhält die Reiseagentur einen Reisebuchungsauftrag von einem Kunden, wird ein Prozess gestartet. Dieser Prozess besitzt zwei Aktivitäten: Es werden für den Kunden ein Hotelzimmer und ein Flug gebucht. Dabei kann es zu folgender Situation kommen: Während die Flugbuchung bereits abgeschlossen wurde, schlägt die Buchung des Hotelzimmers fehl. Die Buchung des Flugs muss folglich kompensiert werden. Dies geschieht, indem der Flug storniert wird. Dazu sendet die Reiseagentur eine zweite Nachricht an die Fluggesellschaft.

Diese Kompensierung der Flugbuchung musste explizit definiert werden. Der Prozessmodellierer muss wissen, wo und wie ein Flug bei dieser Fluggesellschaft storniert wird. Zusätzlich ist das Stornieren unter Umständen mit Verwaltungskosten verbunden, die die Reiseagentur in diesem Fall der Fluglinie bezahlen müsste.

An dieser Stelle kann WS-Coordination Abhilfe verschaffen. Die Reiseagentur übermittel der Flugline und dem Hotel bei ihren Anfragen einen Coordination Context. Dieser enthält die Adresse eines Coordinators, der die Koordinierung der *Participants*, dem Hotel und der Fluglinie, für den Prozess übernimmt. Das Hotel und die Fluglinie registrieren sich an diesem

Coordinator. Schlägt nun die Hotelbuchung fehl, teilt das Hotel dies dem Coordinator mit. Der Coordinator informiert daraufhin direkt die Fluggesellschaft über das Scheitern der Transaktion. Dazu muss die Reiseagentur nicht wissen, wo und wie die Fluglinie Stornos entgegennimmt.

Dies ist nur eines vieler Szenarien, die durch Koordination in Geschäftsprozessen realisiert werden können.

1.2 Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen: In diesem Kapitel werden Konzepte und Techniken beschrieben, die dieser Arbeit als Grundlage dienen.

Kapitel 3 – BPEL und WS-Coordination: Im nächsten Schritt erläutert diese Arbeit Grundlagen, die für eine Mapping-Engine benötigt werden. Dazu werden Verfahren zur Extraktion und Manipulation von Prozessmodellen und Instanzen insbesondere in Hinsicht auf Verwendung mit Apache ODE [Apa10d] untersucht und das Pluggable Framework [KKL07] als Basis für ein transaktionales Mapping zwischen WS-Coordination-basierten Protokollen und der Apache ODE gewählt. Es wird ein Vergleich von Coordination Services verglichen und eine formale Basis für *Coordinated Scopes* geschaffen.

Kapitel 4 – Transaktionales Mapping zwischen BPEL und WS-Coordination: In diesem Kapitel werden Mappingverfahren erläutert. Dazu werden die vier Fälle unterschieden, in denen Coordinated Scopes jeweils die Rollen als WS-BA- bzw. WS-AT-Participant und als WS-BA- bzw. WS-AT-Coordinator annehmen. Es wird untersucht welche Möglichkeiten und Einschränkungen bei der Kombination dieser Fälle in Prozessen bestehen.

Kapitel 5 – Realisierung: Eine prototypische Realisierung wird in diesem Kapitel vorgenommen. Dazu wird ein nachrichtenbasierter Entwurf präsentiert und die erforderlichen Entitäten erläutert.

Kapitel 7 – Zusammenfassung und Ausblick Dieses Kapitel fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfpunkte für mögliche zukünftige Arbeiten vor.

2 Grundlagen

In diesem Kapitel werden Konzepte beschrieben, die dieser Arbeit als Grundlage dienen. Ausführliche Erläuterungen finden sich in den Spezifikationen und Dokumentationen, die in den entsprechenden Abschnitten referenziert werden.

2.1 Service Oriented Architecture

Service Oriented Architecture (SOA) ist ein Architekturparadigma, in dem Services die elementaren Bausteine darstellen, aus denen komplexe Systeme konstruiert werden [WCL+05]. Durch SOA werden plattformspezifische und eventuell unkompatible Systeme als plattform- und sprachenunabhängige Services repräsentiert [Melo7]. Services melden sich an einem Service Directory an (Publishing) und können dort von Service Consumern gefunden werden (Service Discovery) [WCL+05]. Dadurch ist lose Kopplung ein wesentliches Merkmal von SOA [Melo7].

2.2 Web Services

Unter Web Services wird ein Bündel von Technologien verstanden [KL04], mit denen sich SOA realisieren lässt. Web Services basieren auf Standards wie HTTP und XML [Melo7]. Services werden durch die Web Service Definition Language (WSDL) [W3Co7a] beschrieben [WCL+05]. WSDL basiert auf XML und beschreibt Services anhand folgender Elemente [W3Co7a]:

- **Type:** Definition eines Datentyps
- **Message:** Definition einer Nachricht unter Verwendung von Types und elementaren XML-Datentypen
- **Operation:** Definition einer abstrakten Operation und den zugehörigen Input- und Output-Messages
- **PortType:** Definition einer Menge von Operationen (vgl. Java Interface)
- **Binding:** Definition, mit welcher Nachrichtenkodierung ein PortType bzw. Operationen aufgerufen werden

- **Port:** Ein Tupel bestehend aus einem Binding, einer Adresse und einem Protokoll
- **Service:** Definition einer Gruppe von Ports, die denselben Service anbieten.

In WSDL wird nicht nur ein Interface (PortType) mit Operationen (Operation) beschrieben, sondern auch, wie (Binding) und wo (Port) auf einen Service zugegriffen werden kann. Für den Nachrichtenaustausch wird zum Beispiel SOAP über das HTTP-Protokoll verwendet [WCL+05], es kann jedoch auch auf andere Protokolle und andere Nachrichtenformate zurückgegriffen werden. Ein großer Vorteil von Web Services ist die gute Unterstützung durch die Industrie [Trö08]. So bieten viele Systeme SOAP-APIs und für viele Sprachen sind bereits Plattformen für Web Services vorhanden.

2.2.1 Weitere relevante WS-Standards und Konzepte

Eine vollständige Übersicht aller Standards findet sich in [WCL+05].

UDDI UDDI (Universal Description and Discovery Interface) [OAS04] ist eine web-services-spezifische Lösung des Discovery-Services in SOA. Durch sie werden die Konzepte *Publish* und *Find* realisiert (siehe Abschnitt 2.1 auf der vorherigen Seite).

WS-Policy [W3C07c] ist ein Framework, um sowohl auf der Seite eines Web Services als auch des Web Service Consumers nicht-funktionale Anforderungen zu modellieren und an XML-Objekte wie Services und Prozesse anzufügen. Policies können wahlweise im XML-Element eingefügt oder als Policy Attachement [W3C07b] von außen angehängt werden.

SOAP ist ein Messaging Framework zum Austausch XML-basierter Nachrichten über vorhandene Protokolle wie z. B. HTTP. Eine SOAP-Nachricht besteht aus maximal einem Header und einem Body. Der Header enthält Endpunktreferenzen und Metadaten wie z. B. Informationen über Verschlüsselung, Signaturen, Kontexte oder Routing. Der Body enthält den Payload wie Objekte, Funktionsaufrufe oder Antworten und den zugehörigen Daten [WCL+05].

Message Exchange Pattern (MEP) beschreiben Nachrichtenmuster, die Partner austauschen, wenn ein bestimmter Zweck erfüllt werden soll. Ein Beispiel für ein MEP ist das Request-Response-Pattern, bei dem ein Partner eine Nachricht mit einer Anfrage formuliert und diese später vom aufgerufenen Partner durch eine weitere Nachricht beantwortet wird [NLL08].

2.3 WS-Coordination

WS-Coordination [OAS09c] gehört zu WS Transactions (WS-Tx) [WVKG08] und ist ein erweiterbares Framework für web-service-basierte Koordination. Ein WS-Coordination Service (Coordinator, Coordination Middleware) besteht aus drei Komponenten:

- Ein *Activation Service* bietet einen Web Service zur Erzeugen einer neuen *Koordinationsaktivität* (siehe Abschnitt 2.3.2). Diese wird durch einen *Coordination Context* repräsentiert (siehe Abschnitt 2.3.1). Bei einer Koordinationsaktivität kann es sich beispielsweise um eine Transaktion handeln [OAS09a].
- Ein *Registration Service* ermöglicht *Participants* die Anmeldung zur Teilnahme an der Koordinationsaktivität (siehe Abschnitt 2.3.3 auf der nächsten Seite).
- *Protocol Services* sind Web Services, die für je einen bestimmten Typ von Koordinationsaktivitäten die notwendigen Schnittstellen für die Coordination Protocols enthalten, die für das Senden und Empfangen von *Coordination Messages* benötigt werden (siehe Abschnitt 2.3.4 auf Seite 17).

2.3.1 Coordination Context

Ein Coordination Context ist ein XML-Element, das verwendet wird, um anderen Services, den *Participants*, alle Informationen mitzuteilen, die sie benötigen, um selbst an der Koordinationsaktivität teilnehmen zu können. Den Participants wird der Coordination Context dazu beispielsweise als Headerfeld einer Application Message übermittelt. Ein Coordination Context enthält mindestens den Identifier, eine Referenz eines Registration Services und einen Coordination Type. Die Koordinationsaktivität selbst wird durch den Identifier repräsentiert. Bei der Referenz eines Registration Services kann es sich beispielsweise um eine Endpunktreferenz aus WS-Addressing [W3C04] handeln. Beim Coordination Type handelt es sich um den Namen einer bestimmten Art von Koordinationsaktivitäten. Coordination Types werden in WS-Coordination nicht definiert, sondern können auf WS Coordination aufbauend beliebig spezifiziert werden. Dies erfolgt beispielsweise in den Spezifikationen von WS-AT [OAS09a] und WS-BA [OAS09b]. Zusätzlich kann ein Coordination Context mit weiteren Elementen erweitert werden, die für bestimmte Coordination Types benötigt werden. Ein Beispiel für einen Coordination Context ist in Listing 2.1 auf der nächsten Seite aufgeführt.

2.3.2 Activation Service

Der Activation Service wird vom *Initiator* benötigt, um eine neue Koordinierungsaktivität für sich erzeugen zu lassen. Dazu empfängt der Activation Service an seinem Endpunkt eine CreateCoordinationContext-Message. Diese Message enthält mindestens einen Coordination

Listing 2.1: Beispiel eines Coordination Contextes

```
<wscoor:CoordinationContext
    xmlns:wscoor="http://docs.oasis-open.org/ws-tx/wscoor/2006/06"
    xmlns:wsa="http://www.w3.org/2005/08/addressing"
    xmlns:x4b="http://iaas.uni-stuttgart.de/da-henkesn/ME/xact4bpel"
    S11:mustUnderstand="true">
  <wscoor:Identifier>
    http://iaas.uni-stuttgart.de/da-henkesn/CScopeInstance/42
  </wscoor:Identifier>
  <wscoor:RegistrationService>
       <wsa:Address>
              http://iaas.uni-stuttgart.de/da-henkesn/ME/RS
       </wsa:Address>
       <wsa:ReferenceProperties>
      <x4b:ScopeReference>
       <x4b:Invoke id="24" />
      </x4b:ScopeReference>
       <wsa:ReferenceProperties>
  </wscoor:RegistrationService>
  <wscoor:CoordinationType>
    http://docs.oasis-open.org/ws-tx/wsat/2006/06
  </wscoor:CoordinationType>
</wsc:CoordinationContext>
```

Type, um dem Service mitzuteilen, was für eine Art von Koordinierungsaktivität erzeugt werden soll. Zusätzlich können abgesehen von beliebigen Elementen, die der Erweiterung dienen, die optionalen Elemente `Expires` und `CurrentContext` mitgegeben werden. `Expires` bestimmt einen Zeitraum, wie lange die Koordinierungsaktivität gültig ist. `CurrentContext` legt fest, ob die zu erzeugende Koordinierungsaktivität einer anderen Koordinierungsaktivität untergeordnet wird (siehe Abschnitt 2.3.5 auf der nächsten Seite). Als Antwort auf die CreateCoordinationContext-Message sendet der Activation Service dem Initiator eine RegisterResponse-Message, die mindestens[1] den Coordination Context für die generierte Koordinierungsaktivität enthält.

2.3.3 Registration Service

Am Registration Service melden sich Participants an. Diese entnehmen den Endpunkt des Registration Service aus dem Coordination Context und registrieren sich, indem sie eine Register-Message an den Registration Service senden. Diese Message enthält mindestens einen Protocol Identifier und einen Endpunkt für den Protocol Service des Participants. Der Protocol Identifier legt dabei das Coordination Protocol fest, nach dem Messages zwischen Participant und Coordinator gesendet werden.

[1] „Mindestens“ bedeutet in Abschnitt 2.3 jeweils, dass die Message Elemente für Extensions enthalten kann.

2.3.4 Protocol Service

Ein Coordination Type kann beliebig viele Coordination Protocols spezifizieren. Coordination Messages, die vom Participant oder Coordinator gesendet werden, werden über den Protocol Service des Coordinators bzw. Participant Protocol Service empfangen (siehe Abschnitte 2.3.6 und 2.3.7 auf der nächsten Seite).

2.3.5 Untergeordnete Koordinierungsaktivitäten – Interposition

Participants können sich nicht nur am Registration Service anmelden, dessen Endpunkt im Coordination Context enthalten ist, sondern können ihren Coordinator frei wählen. Um einen eigenen Coordinator zu nutzen, erzeugen sie an diesem eine untergeordnete Koordinierungsaktivität, indem sie dem Activation Service eine CreateCoordinationContext-Message senden, der im Element `Current Context` den ursprünglich empfangenen Coordination Context enthält. Der eigene Coordinator registriert sich daraufhin am Registration Service, dessen Endpunktreferenz im ursprünglichen Coordination Context enthalten ist. Er übernimmt den Identifier des Coordination Context. Der Participant registriert sich abschließend am Registration Service des eigenen Coordinators mit dem Coordination Context, der ihm bei der Aktivierung zurückgesendet wurde. Auf diese Art und Weise können nicht nur Aktivitäten untergeordnet, sondern auch Coordination Types überbrückt und Coordinators repliziert werden [OASo9c].

In dieser Arbeit wird Interposition als Möglichkeit dargestellt, sich mit mehreren Participants an Koordinierungsaktivitäten zu registrieren, obwohl vom Coordinator nur eine Registrierung für den Participant zugelassen wird (siehe Abschnitte 4.5.2 auf Seite 68 und 4.5.3 auf Seite 71).

2.3.6 WS-Business Activity

WS-Business Activity (WS-AT [OASo9a]) nutzt das WS-Coordination-Framework und spezifiziert Koordinierungsmechanismen nach dem Schema langlaufender Transaktionen [SMo5]. Die Spezifikation enthält die Coordination Types *AtomicOutcome* und *MixedOutcome* und die Coordination Protocols *BusinessAgreementWithParticipantCompletion* (BAPC) und *BusinessAgreementWithCoordinatorCompletion* (BACC). Jeder Coordination Type unterstützt gleichzeitig Participants, die sich für BAPC registrieren und Participants, die sich für BACC registrieren. Abbildung 2.1 auf der nächsten Seite zeigt die Darstellung des BAPC als Coordination Protocol Graph. Im Gegensatz zum BAPC Protocol wird im BACC Protocol die Complete-Message vom Coordinator gesendet. Bei AtomicOutcome müssen sich alle Participants nach Ende der Transaktion im identischen Zustand befinden; entweder im Zustand *Compensated* oder im Zustand *Completed*. Bei MixedOutcome dürfen Participants unabhängig voneinander in den Zuständen *Compensated* oder *Completed* enden.

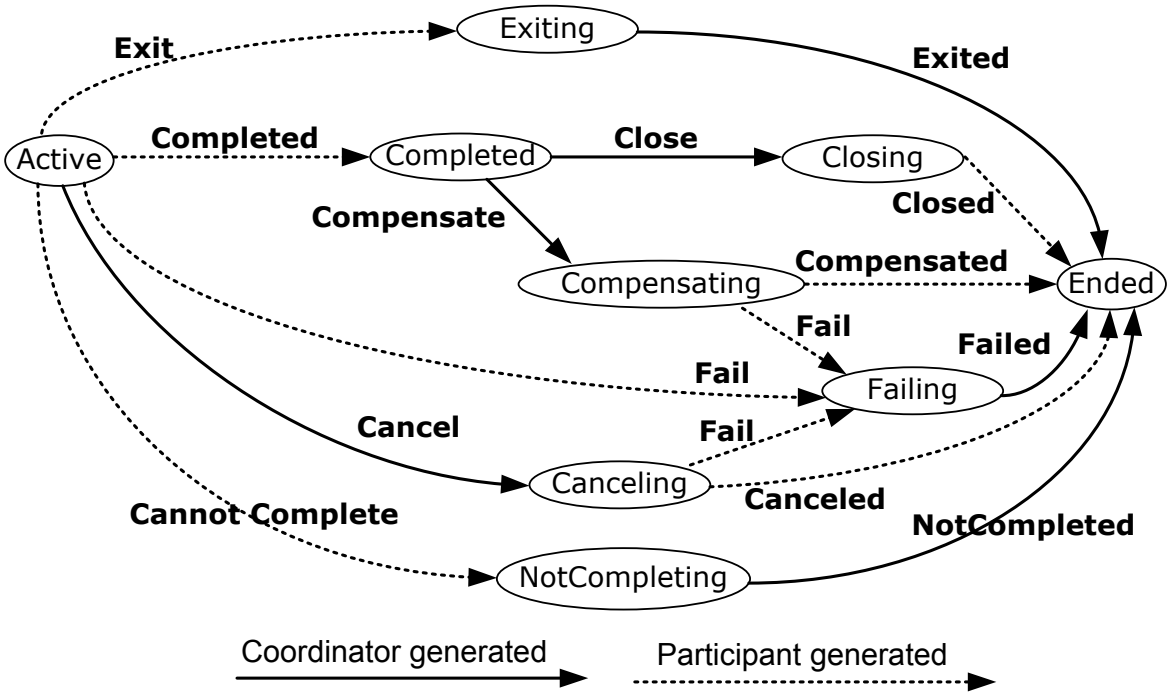

Abbildung 2.1: Coordination Protocol Graph für das BAP Protocol [OAS09b]

Eine ausführlichere Erläuterung kann der Spezifikation [OAS09b] entnommen werden.

2.3.7 WS-Atomic Transaction

WS-Atomic Transaction (WS-AT) setzt ebenfalls auf dem Framework WS Coordination auf. Es definiert Coordination Types und Coordination Protocols für ACID-Transaktionen. Diese werden mit einem Zwei-Phasen-Commit (2PC) abgeschlossen [TS07]. Das 2PC-Protokoll hat dabei zwei Varianten: Für Volatile2PC registrieren sich Participants, die flüchtigen Speicher verwalten, für Durable2PC registrieren sich Participants, die über persistenten Speicher verfügen. Participants können sich für beide Protokolle parallel anmelden. Abbildung 2.2 auf der nächsten Seite stellt das 2PC-Protokoll aus WS-AT als Coordination Protocol Graph dar.

Zusätzlich zu den 2PC-Protokollen wird in WS-AT noch das Completion Protocol definiert. Für dieses Protokoll registiert sich die Anwendung, die für das Einleiten des 2PC verantwortlich ist.

Eine ausführliche Erläuterung kann aus der Spezifikation [OAS09a] entnommen werden.

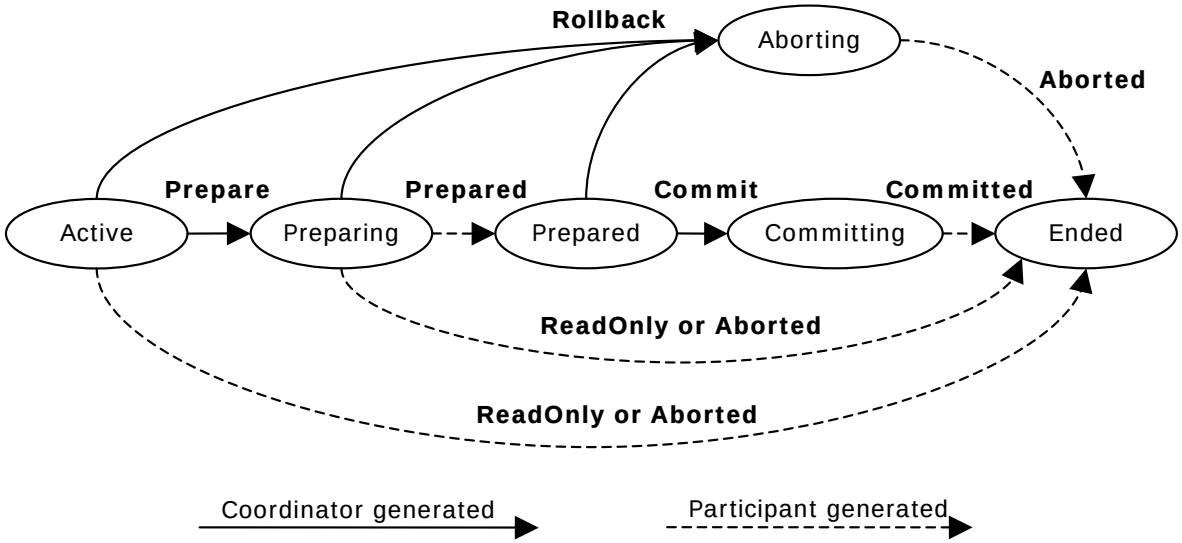

Abbildung 2.2: Coordination Protocol Graph für das 2PC Protocol [OAS09a]

2.4 Business Process Execution Language

Business Process Execution Language (BPEL [OAS07]) ist eine XML-basierte Sprache, um Prozessmodelle zu beschreiben. Durch BPEL werden Web Services durch einen Workflow miteinander verknüpft und als neuer Web Service bereitgestellt. Hierdurch besitzen BPEL-Prozesse alle positiven Merkmale von Web Services und erhalten den hohen Grad an loser Kopplung. Die derzeit aktuelle Version von BPEL ist WS-BPEL 2.0, auf die sich diese Arbeit ausschließlich bezieht.

Prozesse können mit BPEL sowohl als ausführbare Prozesse als auch als sogenannte abstrakte Prozesse beschrieben werden. Abstrakte Prozesse werden beispielsweise verwendet, um die Teile eines Prozesses zu erfassen, von denen ein Geschäftspartner Kenntnis haben muss, ohne zu viel über die eigentliche Implementation, die als Geschäftsgeheimnis gelten kann, zu verraten [WCL+05]. Die Spezifikation von BPEL beschreibt zwei Klassen von Aktivitäten, *Basic Activites* und *Structured Activites*. Während Basic Activities elementare Aktivitäten sind, die keine weiteren Aktivitäten enthalten können (atomic), umschließen Structured Activities weitere Aktivitäten. Dabei legt die Structured Activity fest, wie die enthaltenen Aktivitäten ausgeführt werden sollen. Die Spezifikation von BPEL beschreibt folgende Aktivitäten:

Basic Activities:

- `<receive>`: Empfängt eine Application Message eines Partners. Diese Aktivität kann genutzt werden, um Prozessinstanzen zu erzeugen (`createInstance="yes"`).
- `<reply>`: Sendet eine Antwort an einen Partner, oft auch als Prozessende.

- <invoke>: Sendet einem Web Service eine Application Message oder ruft einen synchronen Web Service auf.
- <assign>: Wird zur Zuweisung von Werten verwendet.
- <throw>: Wirft einen Fault in den umgebenden Scope, der von einem Fault-Handler abgefangen werden kann (siehe Abschnitt 2.4.2 auf der nächsten Seite).
- <exit>: Beendet eine laufende Prozessinstanz sofort, ohne Aufruf von Termination-Handlern.
- <wait>: Wartet eine zu definierende Zeitspanne ab.
- <empty>: Entspricht dem „NULL“ in anderen Programmiersprachen. Wird verwendet, um Fehler abzufangen und zu unterdrücken oder um Synchronisationspunkte in Flows zu setzen.

Structured Activities:

- <sequence>:Führt die umschlossenen Aktivitäten in deren Reihenfolge sequenziell aus.
- <if>: Führt je nach Auswertung einer Bedingung (<condition>) eine Aktivität oder optional eine andere Aktivität aus.
- <while>: Führt eine Aktivität solange aus, wie eine Bedingung erfüllt wird.
- <repeatUntil>: Führt eine Aktivität solange aus, bis eine Bedingung erfüllt wird.
- <forEach>: Ermöglicht das parallele oder sequenzielle Abarbeiten immer der gleichen Aktivität mit verschiedenen Daten.
- <pick>: Führt eine Aktivität aus, sobald ein Event aus einer Liste von Events auftritt. Diese Aktivität kann genutzt werden, um Prozessinstanzen zu erzeugen (createInstance="yes").
- <flow>: Enthält einen Satz von Aktivitäten, die über Transitionen als <source>, <target> und <transitionCondition> verbunden werden. Flows ermöglichen parallele Verarbeitung. Flows beschreiben Workflowmodelle, die als Graphen aus beispielsweise WSFL bekannt sind [KMWL09].
- <scope>: siehe Abschnitt 2.4.1 auf der nächsten Seite.
- <compensate>: siehe Abschnitt 2.4.1 auf der nächsten Seite.
- <compensateScope>: siehe Abschnitt 2.4.1 auf der nächsten Seite.
- <rethrow>: Ein Fault wird von einem Fault-Handler an den umgebenden Scope weiterpropagiert (siehe Abschnitt 2.4.2 auf der nächsten Seite).
- <validate>: Validiert XML-Messages gegen die zugehörigen XML- bzw. WSDL-Definitionen.
- <extensionActivity>: Wird verwendet, um BPEL um neue Aktivitäten zu erweitern.

2.4.1 Transaktionale Konzepte und Faulthandling in BPEL

ACID-Transaktionen führen zu Locking, um die Atomizität zu gewährleisten. Da Business Transactions bzw. Geschäftsprozesse sehr langlebig sein können, ist diese Art der Transaktionssteuerung in geschäftsprozessorientierten Systemen nachteilig [Gra81]. Hier bieten Transaktionskonzepte, die auf Kompensation setzen, einen Vorteil [LR00]. BPEL verwendet das kompensationsbasierte Konzept der *Long Running Transactions* [PML07]. Faults können geworfen und abgefangen werden, um anschließend beispielsweise einzelne Schritte bzw. Aktivitäten erneut zu versuchen oder um einzelne, weniger kostenintensive Teile zu kompensieren, statt komplette Prozessinstanzen bzw. Scopes zurückzusetzen.

Scopes bilden innerhalb von BPEL-Prozessen Ausführungsbereiche mit eigener Lokalität. Sie erlauben das Definieren ausschließlich in diesem Bereich gültiger Variablen, Partnerlinks und CorrelationSets. Zusätzlich können für Scopes FaultHandler, EventHandler, ein CompensationHandler und ein Termination-Handler eingerichtet werden [PML07]. Insbesondere ist die `<process>`-Aktivität, die das Prozessmodell enthält, als Scope zu betrachen. Sie besitzt im Gegensatz zu `<scope>`-Aktivitäten keinen Compensation-Handler.

2.4.2 Fault-Handling

Während des Designs eines Prozesses können beliebig viele `catch`- und `catchAll`-Fault-Handler für jeden Scope und den Prozess definiert werden. Jeder Handler enthält eine Anweisung in Form von genau einer Aktivität, die wiederum eine komplexe Aktivität sein kann. Fault-Handler werden von der Worflow-Engine beim Erreichen des Scopes aktiviert. Tritt ein Fehler während der Verarbeitung der Aktivitäten des Scopes auf, wird ein Fault geworfen. Die noch laufenden Aktivitäten im Scope werden terminiert und das Fault-Handling beginnt. `catch`-Fault-Handler fangen spezifische Faults ab während `catchAll`-Handler alle auftretenden Faults abfangen. Die Aktivität des Fault-Handlers wird ausgeführt. Faults, die nicht vom Designer berücksichtig werden, werden vom impliziten Fault-Handler (Listing 2.2 auf der nächsten Seite) in den übergeordneten Scope weitergeworfen (vgl. Exceptions in Java). Wird der Fault nicht von einem Fault-Handler abgefangen, oder wird er durch `<rethrow>` erneut geworfen, beginnt der überliegende Scope mit dem Fault-Handling. Durch die Verschachtelung von Scopes ergibt sich ein Transaktionsbaum. Die Tiefe entspricht der Anzahl der ineinander verschachtelten Scopes. Die `<process>`-Aktivität entspricht der Wurzel.

2.4.3 Compensation-Handling

Nach erfolgreicher Beendigung der Ausführung eines Scopes wird der Compensation-Handler installiert. Der Compensation-Handler enthält Informationen, wie die Ausführung des Scopes rückgängig (Undo) gemacht werden kann, das heißt, wie ein Zustand erreicht

Listing 2.2: Der implizite Fault-Handler [OAS07]

```
<catchAll>
   <sequence>
      <compensate />
      <rethrow />
   </sequence>
</catchAll>
```

Listing 2.3: Der impliziete Compensation-Handler [OAS07]

```
<compensationHandler>
   <compensate />
</compensationHandler>
```

werden kann, der dem Zustand vor dem Ausführen gleicht oder zumindest ähnelt. Im Falle einer Hotelbuchung z. B. ist Kompensation die Stornierung. Definiert der Designer keinen expliziten Compensation-Handler für den Scope, wird ein impliziter Compensation-Handler installiert, der das Kompensieren aller Compensation-Handler, die innerhalb des Scopes installiert wurden, aufruft. Dies geschieht durch den Start der `<compensate>`-Aktivität (vgl. Listing 2.3). Sie veranlasst das Ausführen der Compensation-Handler der im Scope liegenden und erfolgreich beendeten Aktivitäten in umgekehrter Reihenfolge. Die genaue Reihenfolge für Scopes in Schleifen, parallelen Aktivitäten etc., die von der `<compensate>`-Aktivität kompensiert werden, wird in der BPEL-Spezifikation in [OAS07] vorgeschrieben.

Ein Compensation-Handler kann nur durch den übergeordneten Scope aufgerufen werden und insbesondere nur durch dessen Fault-, Compensation- und Termination-Handler. Dazu existieren in BPEL `<compensate>` und `<compensateScope>`. Während `<compensate>` die Compensation-Handler aller enthaltenen Scopes aufruft, wird mit `<compensateScope>` nur der Compensation-Handler eines einziges Scopes aufgerufen.

2.4.4 Termination-Handling

Der Termination-Handler wird aufgerufen, wenn ein Scope terminiert wird. Wird für einen Scope kein Termination-Handler definiert, wird wiederum ein impliziter Termination-Handler aktiviert (siehe Listing 2.4). Auch er kompensiert alle umschlossenen Scopes, die ohne Fault-Handling durchgeführt wurden.

Listing 2.4: Der impliziete Termination-Handler [OAS07]

```
<terminationHandler>
   <compensate />
</terminationHandler>
```

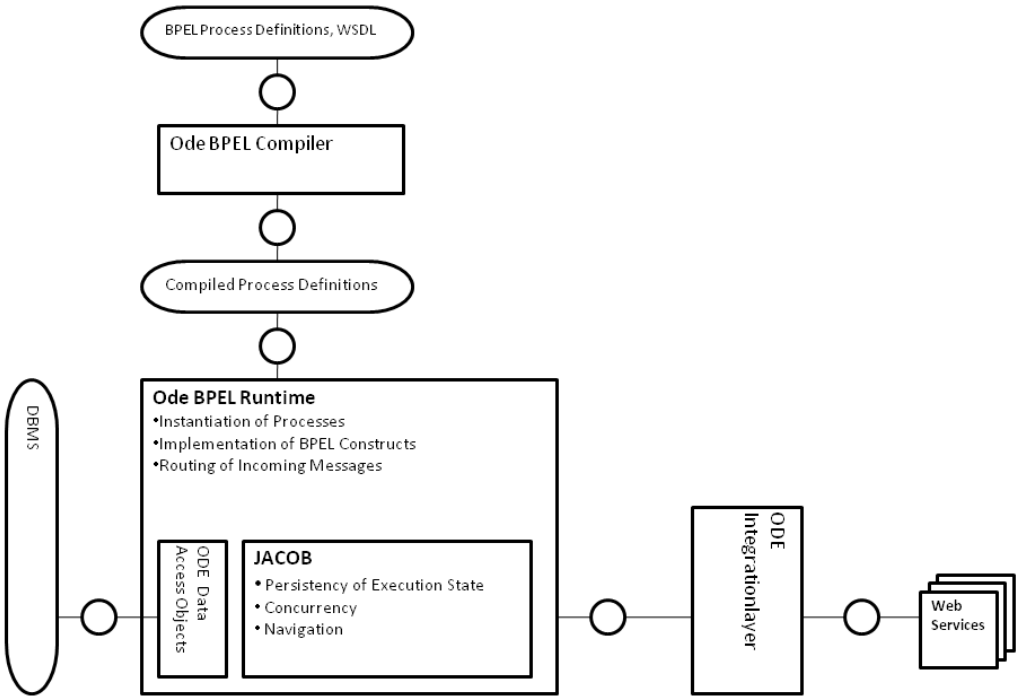

Abbildung 2.3: Architektur von Apache ODE [Apa10c]

2.4.5 Event-Handling

Außer den oben beschriebenen Elementen zur Umsetzung des Transaktionskonzepts, kann ein Scope Event-Handler enthalten, die auf vorher festgelegte Application Messages oder Timeouts reagieren und entsprechende Aktivitäten ausführen.

2.5 Apache ODE

Die Apache Foundation entwickelt als Top-Level-Projekt die Apache Orchestration Director Engine (ODE) [Apa10d], ein Open Source Workflow-Management-System für BPEL-Prozesse. ODE unterstützt die BPEL-Standards 1.1 und 2.0. Die aktuelle stabile Version ist Apache ODE 1.3. Zusätzlich existiert eine Betaversion vom Nachfolger Apache ODE 2.0. ODE kann entweder als JEE-Web Application Archive oder als JBI Service Assembly in einem Servlet- bzw. JBI-Container bereitgestellt werden.

2.5.1 Architektur

Während des Deployments von BPEL-Prozessen werden die BPEL-Prozesse vom ODE-Compiler in das „ODE-Objektmodell“ überführt. Dieses Modell bildet BPEL nicht genau ab sondern entscheidet sich von einem ensprechenden BPEL-Objektmodell in folgenden Punkten [Apa10b]:

- `<receive>`, `<pick>`, `<invoke>` und `<reply>` unterstützen `<fromPart>/<toPart>` nicht.
- Keine Validierung von Variablen (`validate="yes"` wird behandelt wie `validate="no"`), BPEL-Prozesse mit `<validate>`-Aktivität werden nicht kompiliert.
- Die `<compensate>`-Aktivität wird wie `<compensateScope>` behandelt.
- `<pick>` und `<receive>` werden beide wie `<pick>` behandelt, wobei `<pick>` eine `<empty>`-Aktivität als `<onMessage>` zugeordnet bekommt.

Jede Aktivität aus einem BPEL-Prozess wird im ODE-Objektmodell als eigenes Objekt repräsentiert. Durch das Kompilieren können zusätzliche Objekte erzeugt werden. Beispielsweise wird aus dem `<pick>` einer BPEL-Prozessbeschreibung ein `<receive>`-Objekt (heißt in ODE dennoch PICK, verhält sich jedoch wie `<receive>`) mit `<empty>`-Objekt generiert.

Während des Kompilierens werden zusätzlich Namen und Typen aus der WSDL-Beschreibung aufgelöst und weitere Objekte, wie die standardmäßigen Compensation Handler, angelegt. Das Kompilat wird als `cbp`-Datei gespeichert. Alle Informationen, die die BPEL-Engine zur Laufzeit benötigt, sind in dieser Datei enthalten. Die ODE-Runtime kann die kompilierten Prozesse ausführen [Apa10c].

3 BPEL und WS-Coordination

In diesem Kapitel werden sowohl technische als auch theoretische Aspekte beleuchtet, die für einen kombinierten Einsatz von BPEL mit WS-Coordination und den dazugehörigen Protokollen benötigt werden.

In Abbildung 3.1 wird abstrakt dargestellt, wie eine Interaktion von BPEL und WS-Coordination abläuft. Links befindet sich eine BPEL-Engine, rechts das Framework WS-Coordination. Dazwischen befindet sich eine Logik, die für verschiedene Protokolle von WS-Coordination in der Lage ist, Daten und Vorgänge von WS-Coordination auf BPEL zu mappen und umgekehrt. Im Folgenden wird diese Logik *Mapping-Engine* genannt. Der linke Doppelpfeil stellt die Extraktion und Manipulation von Vorgängen in der Workflow-Maschine dar. Der rechte Pfeil steht für den Austausch von Coordination Messages zwischen der Mapping-Engine und den Coordinators und Participants dar.

Zunächst werden drei Möglichkeiten erläutert, wie der Eingriff in die Workflowmaschine zur Extraktion und Manipulation erfolgen kann. Es werden das Pluggable Framework, das Resource Management Framework sowie die ODE Management API vorgestellt und eine Wahl getroffen, welcher der Ansätze sich für eine Realisierung am besten eignet. Im zweiten Abschnitt werden verschiedene Implementierungen des WS-Coordination-Frameworks vorgestellt und verglichen, um schließlich einen Coordinator für den Prototypen auszuwählen. Im dritten Abschnitt wird das Mapping zwischen BPEL und WS-Coordination theoretisch erklärt.

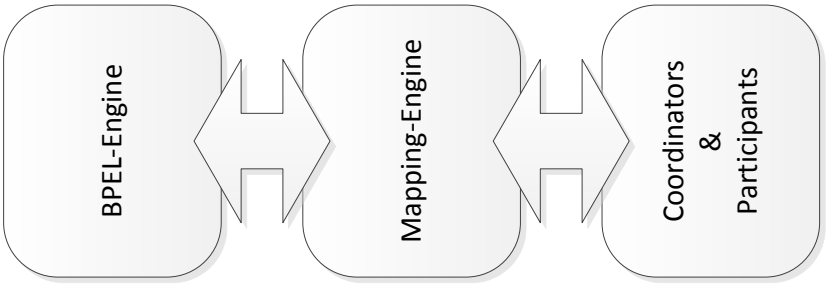

Abbildung 3.1: Interaktion von BPEL und WS-Coordination

3.1 Extraktion und Manipulation von Prozessdaten

Um externe Transaktionen zu ermöglichen, muss die Möglichkeit gegeben sein, in das Workflowsystem, hier Apache ODE, eingreifen zu können. Es müssen Informationen über den aktuellen Zustand von Scopes und Aktivitäten sowohl ausgelesen, als auch deren weiterer Verlauf beeinflusst werden können. Zusätzlich muss auf Variablen und Nachrichten von außen zugegriffen werden können, um Coordination Contexts sichtbar zu machen bzw. anzuhängen. Dazu werden im Folgenden drei Frameworks beschrieben und deren Tauglichkeit für eine Realisierung bewertet. Das BPEL-Eventmodell mit Pluggable Framework wird dabei ausführlicher als die anderen beiden Frameworks behandelt, da dieses schließlich für eine Verwendung herangezogen wird und große Teile dieses Frameworks als Grundlage benötigen. Die anderen beiden Frameworks, das Resource Management Framework sowie die ODE Management API, werden grob umrissen sowie deren Mängel dargelegt.

3.1.1 BPEL-Eventmodell und Pluggable Framework

Thomas Steinmetz präsentiert in seiner Diplomarbeit [Steo8] ein Eventmodell für WS-BPEL 2.0. Dieses Modell basiert auf einem in [KKS+06] vorgestellten Modell für BPEL 1.1. Das Eventmodell definiert eine Menge von Events, die während der Laufzeit der Workflow-Maschine und während des Abarbeitens einer Prozessinstanz auftreten können. Ziel ist es, diese *Outgoing Events* nach außen sichtbar zu machen, um extern die Ausführung eines Prozesses verfolgen zu können. Dies ermöglicht beispielsweise Monitoring [KLN+06]. Zusätzlich soll von außen auch Einfluss auf Prozesse genommen werden können. Dazu gibt es *blockierende* Outgoing Events, die die Ausführung an der auftretenden Stelle im Prozess (ohne Einfluss auf parallel laufende Teilprozesse) unterbrechen. Die BPEL-Engine wartet dann auf ein *Incoming Event* von außen, das die Blockade wieder aufhebt. Zusätzlich existieren weitere Incoming Events, die Einflussnahme ermöglichen. Über sie können beispielsweise Variablen modifiziert, Faults geworfen und Scopes terminiert werden. Durch Einflussnahme auf den Prozessablauf lassen sich Prozesse auch koordinieren [Mieo6] und fragmentieren [KLo6].

Das Eventmodell wird von Steinmetz in mehreren Teilen beschrieben. Jeweils anhand eines Zustandsdiagrammes werden separat Untermodelle für Prozesse, Aktivitäten, Scopes, Schleifen und Links erläutert. Zusätzlich existiert ein Teil mit weiteren Events, die unabhängig von Zustandsdiagrammen auftreten können, sowie ein Teil mit den Incoming Events.

Dieser Abschnitt basiert auf den oben angegebenen Arbeiten und stellt das Eventmodell nicht ausführlich vor, sondern beschreibt von den Untermodellen die Modelle für Prozesse, Aktivitäten und Scopes, da diese Teile relevant für die in Abschnitt 4 auf Seite 59 vorgestellten Mappingverfahren sind. Die Events für Links und Schleifen werden weder erläutert, noch in den Aufzählungen aufgeführt, da sie für diese Arbeit nicht benötigt werden.

Outgoing Events

Wie bereits beschrieben, werden die Events je in einem Zustandsdiagramm dargestellt. Events treten an Transitionen auf. Die Zustandsdiagramme in den Abbildungen 3.2 auf der nächsten Seite, 3.3 auf Seite 29 und 3.5 auf Seite 32 geben stets den Fall wieder, in dem alle Events, die blockieren können, auch tatsächlich blockieren. Im nichtblockierenden Fall entfällt das Warten auf die auflösenden Incoming Events. Events, die blockieren können, werden durch eine (b)-Markierung gekennzeichnet.

Das Prozess-Eventmodell enthält die folgenden Outgoing Events.

- **Process_Deployed:** Ein Prozessmodell wurde deployt.
- **Process_Undeployed:** Ein Prozessmodell wurde undeployt.
- **Process_Instanciated:** Eine neue Prozessinstanz wurde erzeugt.
- **Instance_Running:** Die Ausführung einer Prozessinstanz beginnt.
- **Instance_Suspended:** Eine Prozessinstanz wurde unterbrochen.
- **Instance_Terminated:** Eine Prozessinstanz wurde mit `<exit>` beendet.
- **Instance_Complete:** Eine Prozessinstanz wurde erfolgreich beendet.
- **Instance_Faulted:** In der Prozessinstanz wurde ein aufgetretener Fault nicht abgefangen.

Der Ablauf des Lebenszyklus eines Prozessmodells mit den zugehörigen Events kann aus Abbildung 3.2 auf der nächsten Seite entnommen werden. Genauere Beschreibungen der Lebenszyklen und der Events finden sich in [KKS+06] und in [Ste08].

Das Activity-Eventmodell enthält folgende Outgoing Events:

- **Activity_Ready (b):** Die joinCondition einer Aktivität wird zu `true` ausgewertet.
- **Activity_Executing:** Die Ausführung einer Aktivität wird gestartet.
- **Activity_Executed (b):** Die Ausführung einer Aktivität wurde beendet und wird evtl. blockiert.
- **Activity_Complete:** Eine Aktivität wurde abgeschlossen und es liegen keine Blockaden (mehr) für sie vor.
- **Activity_Dead_Path:** Eine Aktivität liegt im Dead Path und es wurde definiert: „`suppressJoinFailure="yes"`“.
- **Activity_Terminated:** Eine Aktivität wurde terminiert, der Termination-Handler wurde beendet.

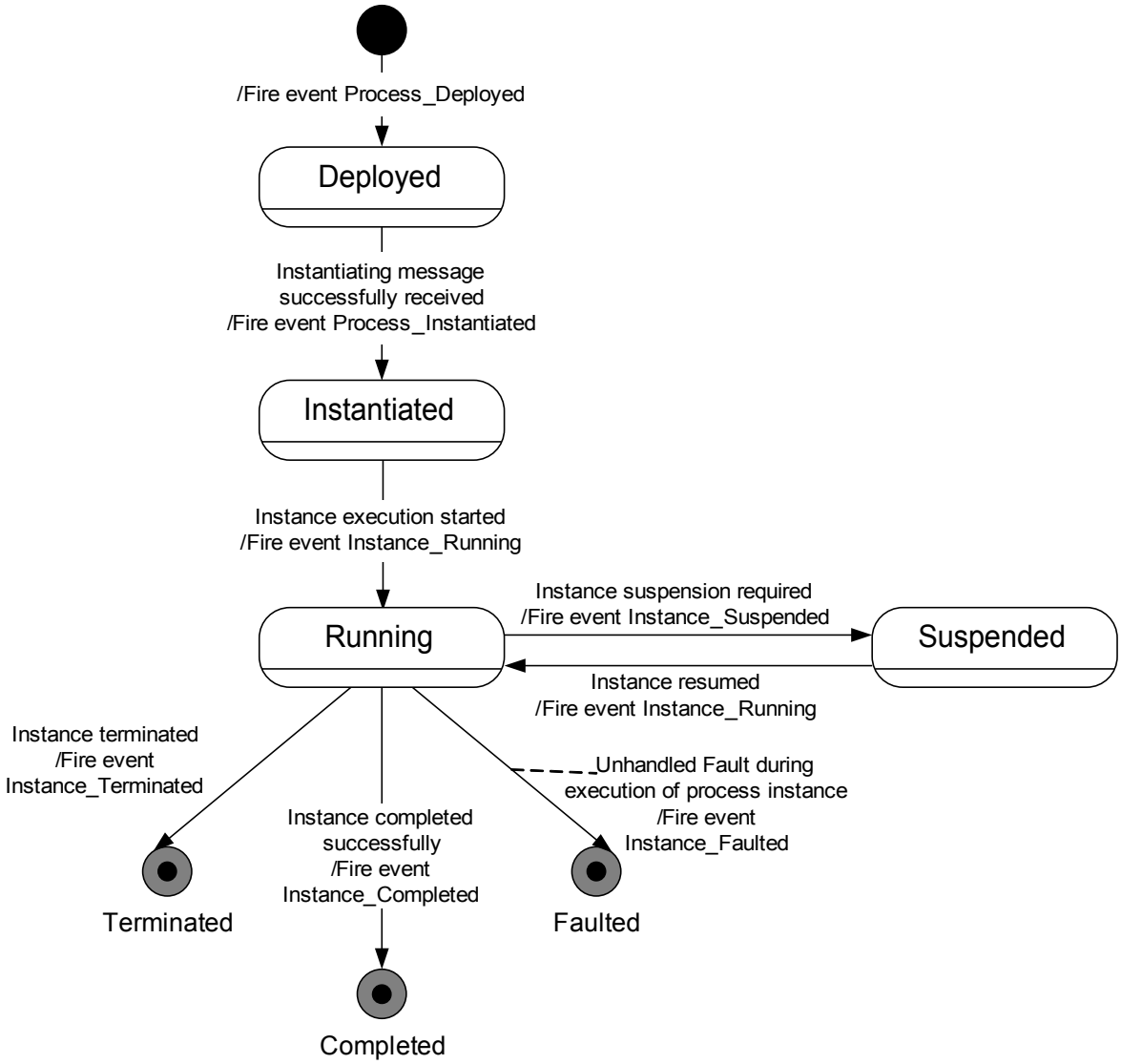

Abbildung 3.2: Zustandsdiagramm für Prozessmodelle und auftretende Events [KKS+06]

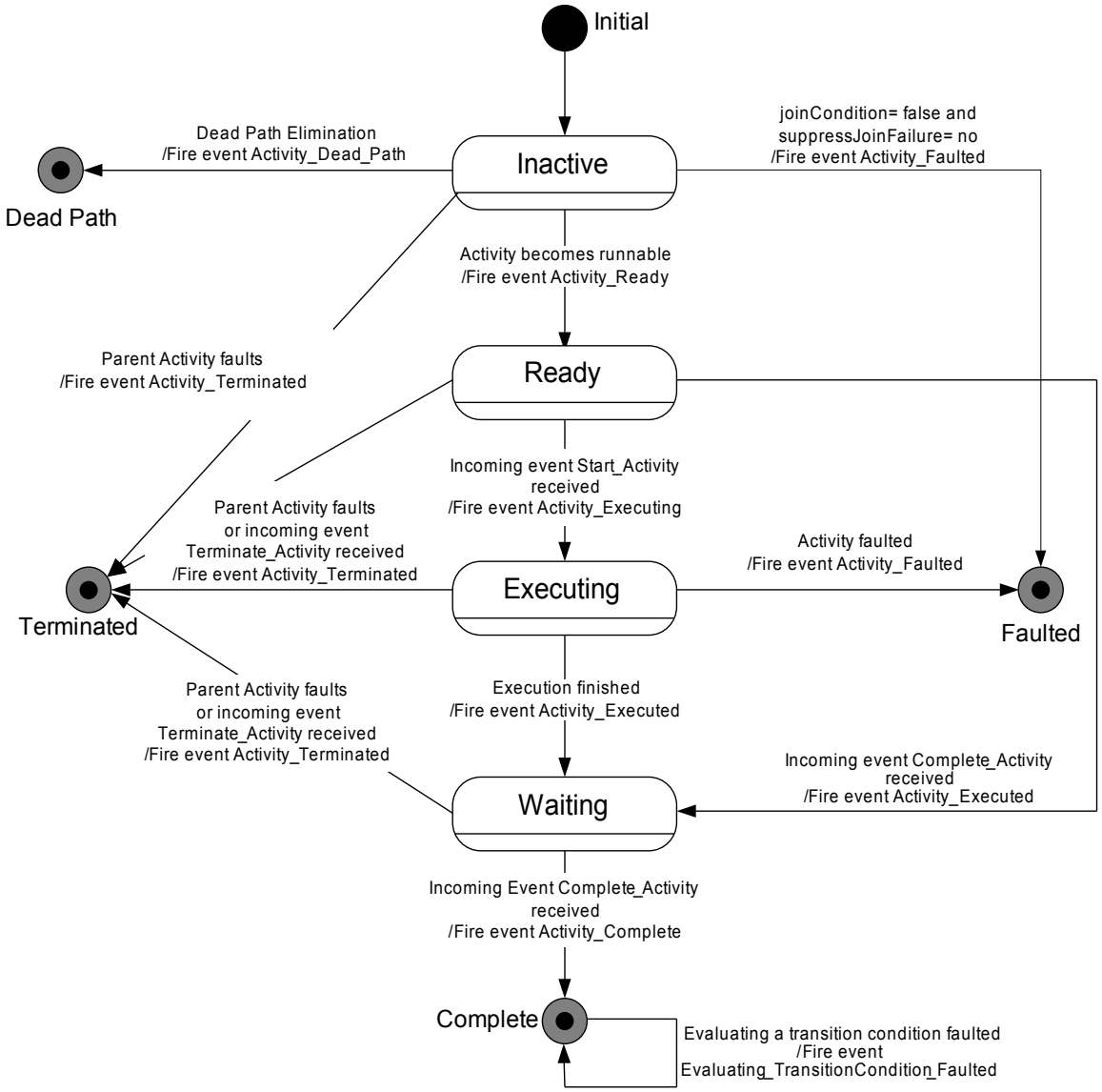

Abbildung 3.3: Zustandsdiagramm für Aktivitäten und auftretende Events [Steo8]

- **Activity_Faulted (b):** Eine Aktivität wurde abgebrochen, da ein nicht abgefangener Fault in ihr aufgetreten ist. Zusätzlich kann dieses Event auftreten, wenn `suppressJoinFailure` auf `no` gesetzt ist, und die joinCondition zu `false` ausgewertet wird.
- **Evaluation_TransitionCondition_Faulted (b):** Die Evaluierung einer Transition Condition ist fehlgeschlagen.

Die beschriebenen Events können am Zustandsdiagramm für Lebenszyklen von Aktivitäten in Abbildung 3.3 auf der vorherigen Seite nachvollzogen werden. Das Eventmodell für Aktivitäten wurde von Steinmetz für BPEL 2.0 in [Ste08] angepasst. Dort können auch genauere Beschreibungen entnommen werden.

Das Scope-Eventmodell enthält folgende Events:

- **Scope_Handling_Fault (b):** Ein impliziter oder expliziter Fault-Handler des Scopes wird aktiviert.
- **Scope_Handling_Event:** Der Event-Handler des Scopes wird aktiviert.
- **Scope_Event_Handling_Ended:** Das Event-Handling eines Events wurde beendet.
- **Scope_Compensating (b):** Der Compensation-Handler des Scopes wird aktiviert.
- **Scope_Compensated:** Der Compensation-Handler des Scopes wurde beendet.
- **Scope_Complete_With_Fault (b):** Ein Fault wurde abgefangen und der Scope wurde ohne Weiterpropagieren des Faults beendet.
- **Scope_Handling_Termination (b):** Der Termination-Handler eines Scopes wird ausgeführt.

Die Events für Scopes können am Zustandsdiagramm für Scopes in Abbildung 3.5 auf Seite 32 nachvollzogen werden. Diese Abbildung enthält nicht die Übergänge zu den Zuständen *Dead Path* und *Faulted*. Diese beiden werden zugunsten einer übersichtlicheren Darstellung nicht in Abbildung 3.5 auf Seite 32 gezeigt, können aber dem Zustandsdiagramm für Aktivitäten aus Abbildung 3.3 auf der vorherigen Seite entnommen werden. Der Zustand *Fault Handling* wird aus dem selben Grund in Abbildung 3.4 auf der nächsten Seite ausgelagert.

Weitere Events sind:

- **Variable_Modification:** Eine BPEL-Variable wurde modifiziert.
- **CorrelationSet_Modification:** Properties eines Correlation-Sets wurden verändert.
- **PartnerLink_Modification:** Mittels `<assign>` wurde einem PartnerLink ein neuer Endpoint zugewiesen.
- **Variable_Read:** Der Inhalt einer Variablen wurde durch das Event *Read_Variable* (s. u.) angefordert.

Diese Events treten unabhängig von den oben beschriebenen Zustandsdiagrammen während der Laufzeit einer Instanz auf. Das Event *Variable_Modification* kann z. B. bei den Aktivitäten `<receive>` und `<assign>` ausgelöst werden.

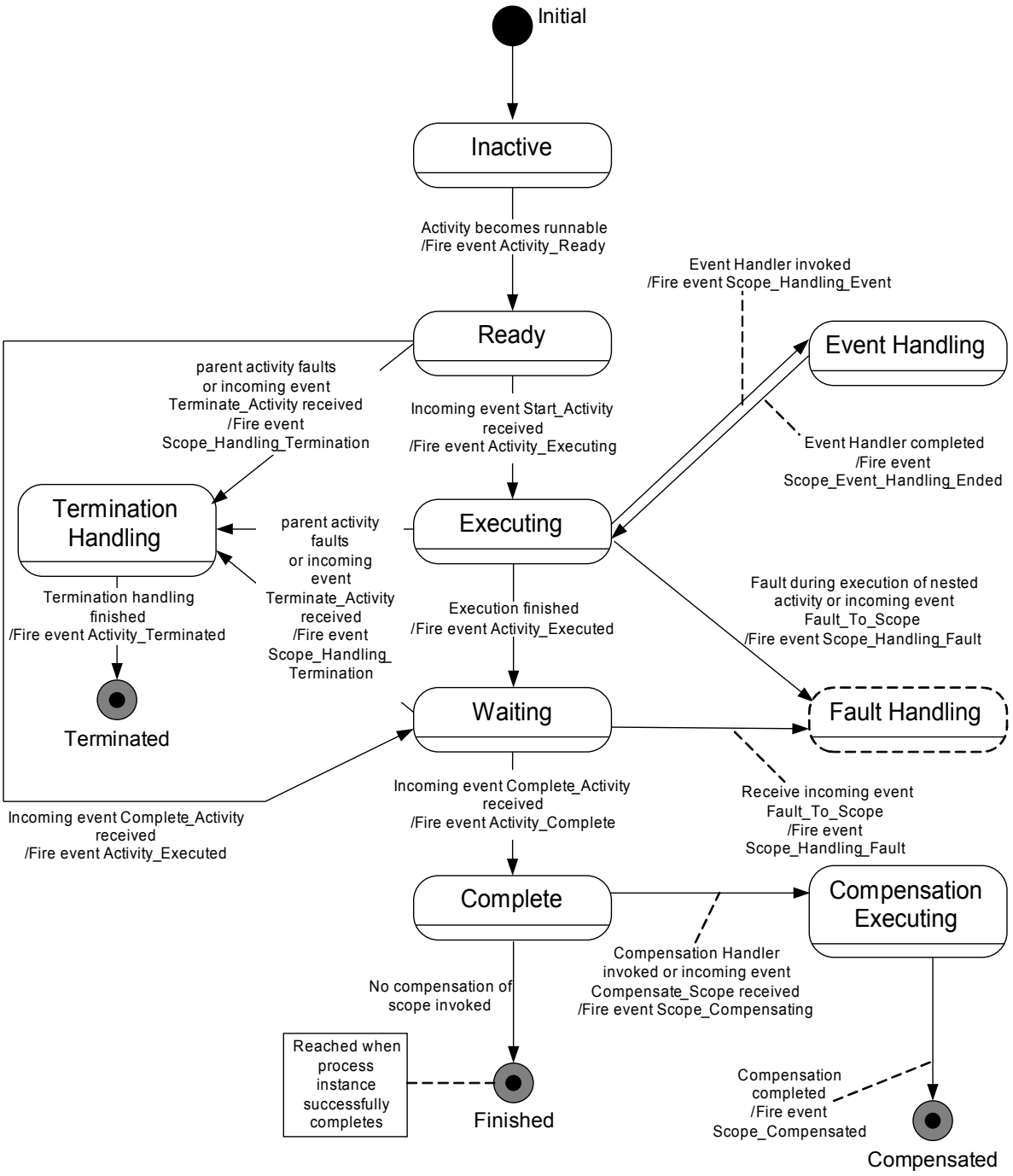

Abbildung 3.4: Zustandsdiagramm für Scopes und auftretende Events [Ste08]

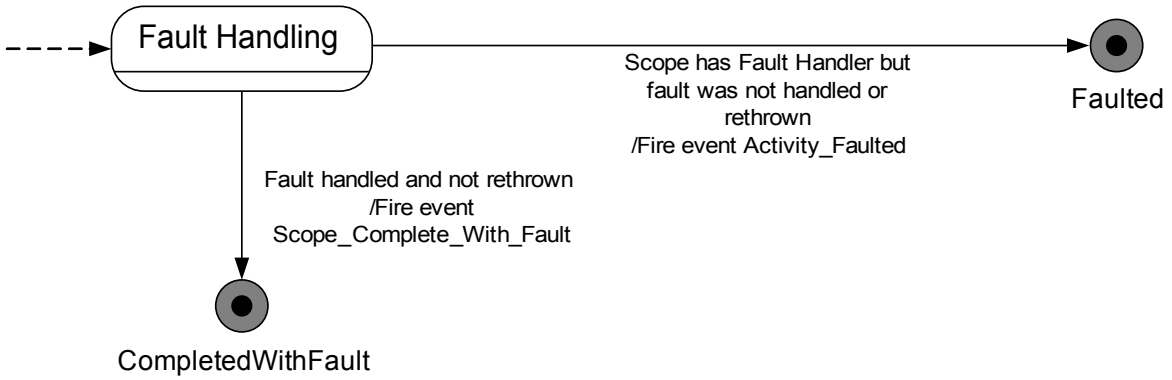

Abbildung 3.5: Fortsetzung des Scope-Zustandsdiagramms ab Zustand *Fault Handling* [Ste08]

Incoming Events

Incoming Events sind Events, die von außen an die Workflow-Maschine gesendet werden, um Einfluss auf ihr Verhalten zu nehmen. Sie dienen z. B. dem Auflösen von Blockaden, dem Modifizieren von Variablen und dem Terminieren bestimmter Aktivitäten oder Scopes.

Folgende Incoming Events mit entsprechender Reaktion der Workflowmaschine werden in [Ste08] bzw. in [KKS+06] definiert[1]:

- **Compensate_Scope:** Ein Scope im Zustand *Complete* wird kompensiert.
- **Fault_To_Scope:** Ein Fault wird in einen Scope im Zustand *Executing* bzw. *Waiting* geworfen.
- **Start_Activity:** Die Blockade des Events *Activity_Ready* wird aufgehoben.
- **Complete_Activity:** Die Blockade des Events *Activity_Executed* wird aufgelöst.
- **Terminate_Activity:** Eine Aktivität wird terminiert.
- **Read_Variable:** Eine Variable wird von außen angefordert. Als Reaktion wird *Variable_Read* geworfen, welches die Variable enthält.
- **Write_Variable:** Die Änderung einer Variable wird von außen angefordert. Als Reaktion wird *Variable_Modification* geworfen.
- **Suppress_Fault:** Ein geworfener Fault wird unterdrückt.
- **Suspend_Instance:** Eine Instanz wird unterbrochen.
- **Resume_Instance:** Eine Instanz wird nach einer Unterbrechnung fortgesetzt.

[1] auch hier ohne Events, die ausschließlich Links und Loops betreffen

- **Continue:** Eine Blockade wird durch dieses Event aufgehoben. Die Workflowmaschine verhält sich dabei so, wie sie es ohne Blockade getan hätte.

Informationen in den Events

Alle Events enthalten Informationen, die das Lokalisieren des genauen Auftrittsortes bzw. Eingriffsortes innerhalb des Prozesses bzw. der Instanz zulassen. Events beinhalten immer den QName des Prozessmodells und die Versionsnummer des Prozesses. Je nach Typ kommen weitere Informationen hinzu. Alle Events, die innerhalb einer Instanz auftreten, werden mit Instanz-ID übermittelt. Aktivitätsbezogene Events besitzen zusätzlich

- die ID des beinhaltenden Scopes
- den XPath-Ausdruck des Scopes
- den XPath-Ausdruck der Aktivität.

Events, die innerhalb von Scopes auftreten, enthalten neben den Informationen von aktivitätsbezogenen Events auch eine Scope-ID. Diese Scope-ID ist notwendig, da ein Scope mehrmals durchlaufen werden kann und nur anhand eines XPath-Ausdrucks nicht mehr zwischen den Durchläufen unterschieden werden könnte.

Die Events *Variable_Modification* und *Variable_Read* übermitteln außerdem den Namen und den Inhalt der (evtl. geänderten) Variablen.

Das Eventmodell in Apache ODE

Steinmetz implementiert sein Eventmodell am Beispiel der Apache ODE 1.1.1. Hier sind folgende Eigenheiten von ODE zu berücksichtigen. ODE unterstützt keinen Termination-Handler, folglich tritt das Event *Scope_Handling_Termination* nie auf [Steo8].

Außerdem verfolgt ODE auch ein verändertes Prozessmodell. Wie sich in Abbildung 3.6 auf der nächsten Seite erkennen lässt, führt ODE die drei zusätzlichen Zustände *Active*, *Disabled* und *Retired* für deployte Prozesse ein. Ein sich im Zustand *Active* befindender Prozess entspricht einem Prozess im ursprünglichen Modell (vgl. Abbildung 3.2 auf Seite 28) im Zustand *Deployed*. Prozesse im Zustand *Retired* dürfen nicht neu instanziiert werden.

Laufende Instanzen werden jedoch fortgeführt. Prozesse im Zustand *Disabled* werden nicht weiter verarbeitet, können aber zu einem späteren Zeitpunkt wieder aktiviert werden [Steo8].

Aus diesem veränderten Modell ergeben sich die folgenden neuen Events:

- **Process_Active:** Ein Prozess befindet sich jetzt im Zustand *Active*.
- **Process_Retired:** Ein Prozess befindet sich jetzt im Zustand *Retired*.

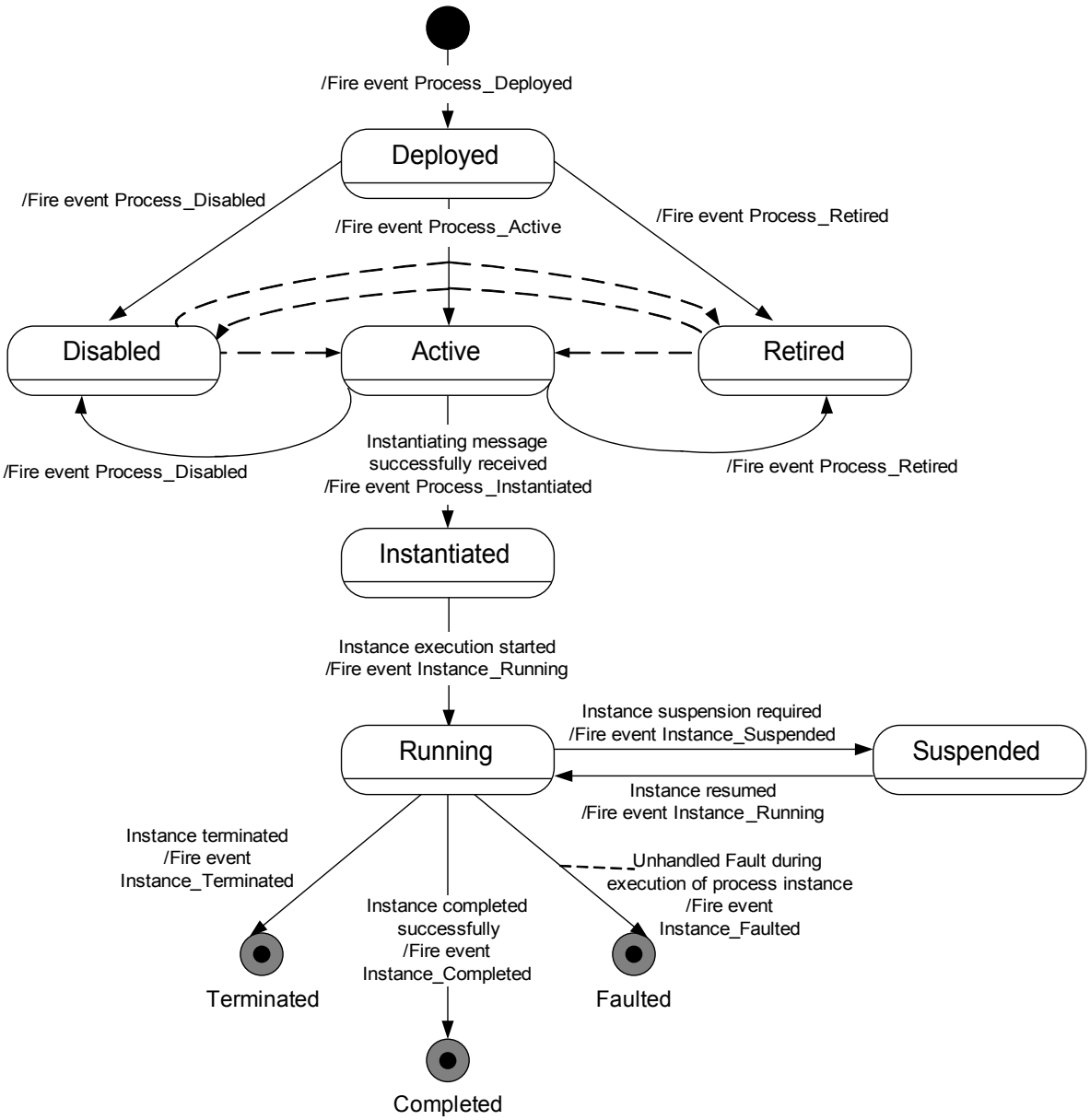

Abbildung 3.6: Modifiziertes Prozessmodell in ODE [Ste08]

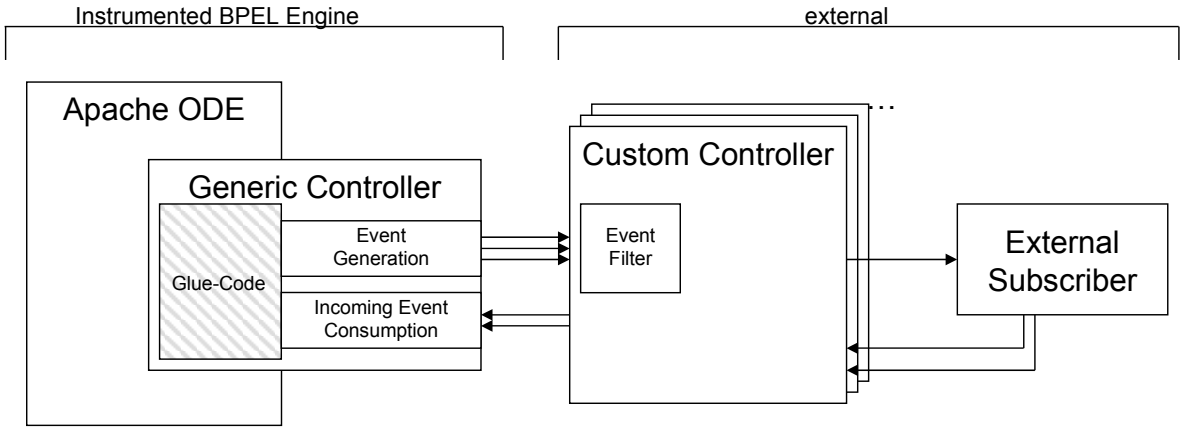

Abbildung 3.7: Pluggable Framework [Steo8]

- **Process_Disabled:** Ein Prozess befindet sich jetzt im Zustand *Disabled*.

Ergänzende Incoming Events, die einen Statuswechsel zwischen diesen zusätzlichen Status anfordern, werden nicht definiert.

Das Pluggable Framework

Um das Eventmodell in Apache ODE 1.1.1 zu realisieren, greift Steinmetz auf das Konzept des in [KKL07] vorgestellten Pluggable Frameworks zurück. Das Framework beschreibt eine Architektur bestehend aus zwei Teilen, dem *Generic Controller* und dem *Custom Controller*, wie in Abbildung 3.7 dargestellt wird.

Der Generic Controller wird dazu über einen Glue-Code in die Workflowmaschine eingehängt. Er ist für das Generieren der Outgoing Events sowie das Empfangen der Incoming Events zuständig und führt die nötigen Eingriffe der Incoming Events in den Prozessen durch. Der Custom Controller empfängt die Outgoing Events und leitet diese an Komponenten weiter, die diese Events verarbeiten können. Incoming Events werden generiert und an den Generic Controller gesendet.

Für seine Implementierung hat sich Steinmetz für eine Realisierung mit *Java Message Service* (JMS) als Nachrichtenplattform entschieden. Custom Controllers können sich an JMS-Queues bzw. JMS-Topics registrieren und tauschen so die Events in Form von Nachrichten mit dem Generic Controller aus.

Es werden drei Kommunikationskanäle definiert:

- JMS-Topic: Outgoing Events, die an alle Custom Controllers gehen, werden an ein Topic publiziert.

- JMS-Queue: Incoming Events von Custom Controllern werden an eine Queue des Generic Controllers geschickt.
- JMS-Temporary-Queue: Outgoing Events werden in diese Queue gestellt, sofern diese nur an einen Custom Controller gerichtet sind.

Ein Custom Controller muss sich beim Generic Controller registrieren, um einzelne Eventtypen als blockierend festzulegen. Ohne Registrierung sind alle Eventtypen nicht-blockierend.

Verwendbarkeit für externe Transaktionen

Das Pluggable Framework liefert grundsätzlich alle notwendigen Events und Informationen aus der Workflowmaschine, um ein transaktionales Mapping zu realisieren (siehe Abschnitt 4 auf Seite 59). Die einzige Erweiterung wird in Abschnitt 4.4 auf Seite 62 erläutert. Dabei wird das Eventmodell für allgemeine Aktivitäten bezüglich `<invoke>`-Aktivitäten so erweitert, dass auf die zu sendende Application Message zugegriffen werden kann. Das Design des Pluggable Frameworks ist unabhängig von Apache ODE. Ein Generic Controller lässt sich auf andere Workflowmaschinen portieren, ohne den Custom Controller ändern zu müssen. Somit kann durch die Verwendung des Pluggable Frameworks für jede Workflowmaschine, die über den Generic Controller verfügt, die Unterstützung von WS-Coordination mit geringem Aufwand realisiert werden.

3.1.2 Resource Management Framework

Das Workflow Reference Model definiert verschiedene Schnittstellen, die ein Workflow-Management-System zur Verfügung stellen soll. Eine davon ist ein Auditing- und Managementschnittstelle (AMS) [Hol95]. Eine einheitliche AMS für BPEL-Systeme gibt es nicht. Typischerweise besitzen BPEL-Systeme eine eigene, sich von anderen AMS unterscheidende API [LLM+08]. Van Lessen et al. beschreiben in [LLM+08] eine einheitliche Schnittstelle, die auf Basis von WS-Standards in BPEL-Systemen realisert werden kann. Ziel ist es, eine gemeinsame Schnittstelle für alle BPEL-Systeme zu entwerfen, sodass BPEL-Systeme beliebig ausgetauscht werden können, ohne Monitoring-Tools an die neuen herstellerspezifischen APIs anpassen zu müssen. Dazu werden Prozesse und Instanzen als durchsuchbare Ressourcen, wie sie im WS-Resource Framework (WSRF [OWT06a]) spezifiziert werden, über einen Web Service publiziert. WSRF bietet dabei den Vorteil notwendige Informationen direkt bereitstellen zu können, ohne spezifische Web Service Consumer dafür erstellen zu müssen. Dieses AMS wird als Resource Management Framework (RMF) bezeichnet.

Resource und WS-Resource werden in der Spezifikation von WS-Resource [OAS06] definiert. Eine Resource hat folgende Eigenschaften:

- Eine Resource ist eindeutig identifizierbar.
- Eine Resource hat beliebig viele Eigenschaften (Properties), die in XML Infoset ausgedrückt werden können.
- Eine Resource kann einen Lebenszyklus besitzen.

Eine WS-Resource wird definiert als Verbindung aus einer Resource und einem Web Service, durch den auf die Resource zugegriffen werden kann. Dieser Web Service hat eine Endpoint Reference, die eindeutig der Resource zugewiesen ist. Er ermöglicht den Zugriff auf die Properties der Resource nach dem Standard WS-Resource Property [OWT06b].

[LLM+08] erwähnt zwei Ansätze zur Realisierung der AMS. Das RMF kann entweder direkt in das BPEL-System eingebaut werden und in dessen Datenmodell eingreifen oder über eine vorhandene herstellerspezifische AMS gestülpt werden. Die zweite Möglichkeit schränkt die Fähigkeiten des RMF in Abhängigkeit der Mächtigkeit der herstellerspezifischen AMS ein. Ebenso ist eine Mischung der beiden Realisierungsansätze denkbar, in der ein Teil der Funktionen auf dem internen Datenmodell aufsetzt und ein Teil auf vorhandene APIs zugreift.

Mapping von Prozessmodellen auf Resourcen

Alle Prozessmodelle werden auf Resourcen gemappt. Die enthaltenen XML-Elemente werden wiederum auf Resourcen gemappt, die in die Prozessmodellresource eingebettet werden, sofern diese Elemente mehrfach auftreten oder Unterelemente besitzen, die mehrfach auftreten. Die übrigen Elemente und Attribute werden auf ResourceProperties gemappt. Ein deployter Prozess wird ebenfalls als Resource dargestellt. Er wird durch den QName des Prozesses identifiziert und erhält als Kindresourcen das Prozessmodell `definition` sowie eine Resource `instances`, die die Instanzen enthält. Instanzen identifizieren sich innerhalb der Resource für alle Instanzen mit deren InstanzId (zum Beispiel `instances/42`). Die Namen der Resourcen innerhalb des Prozessmodells entsprechen jeweils den XPath-Ausdrücken im BPEL-Dokument. Properties erhalten den Namen des Elements bzw. des Attributs im BPEL-Dokument. Somit können alle Elemente im Prozessmodell bzw. in den Instanzen eindeutig identifiziert und adressiert werden.

Elemente wie Aktivitäten, Scopes, usw. in Instanzen enthalten zusätzliche Properties für Status, in denen sie sich gegenwärtig befinden. Die Instanz selbst erhält beispielsweise selbst eine Property `state`, die die Werte `instantiated`, `running`, `suspended`, `terminated`, `faulted` und `complete` annehmen kann, wenn das Eventmodell für BPEL 1.1 [KKS+06] bzw. für BPEL 2.0 [Ste08] und dessen Zustände zugrunde gelegt wird. Elemente wie Variablen, PartnerLinks und CorrelationSets in Instanzen besitzen einen Status, der `uninitialized` oder `initialized` annehmen kann.

Bei zyklischen Aktivitäten wie `<while>` und `<forEach>` wird in Instanzen bei jedem Durchlauf eine Unterresource zur zyklischen Aktivität hinzugefügt, die dem aktuellen Durchlauf der in der zyklische Aktivität liegenden Aktivität entspricht.

Anwendungsgebiet

Im Gegensatz zu herkömmlichen APIs, die beispielsweise über das Getters/Setters-Pattern Zugriff auf Werte erlauben, ermöglicht das RMF ein Durchsuchen bzw. Browsen in den Resourcen. Es erlaubt eine sehr gezielte Adressierung von Werten und kann komplexe Eingriffe vornehmen. [LLM+08] beschreibt zusätzlich ansatzweise das Senden von Events und Setzen von Blockaden.

Verwendbarkeit für externe Transaktionen

Generell bietet das RMF die notwendige Infrastruktur, um alle notwendigen Extraktionen aus Instanzen bzw. Modifikationen in Instanzen vorzunehmen. Da jedoch zum Zeitpunkt der Erstellung dieser Arbeit kein Prototyp vorliegt und eine Implementierung umfangreich ist, wird das RMF nicht als Komponente für die Kommunikation mit der BPEL-Engine verwendet.

3.1.3 ODE Management API

Apache ODE beinhaltet die Auditing- und Managementschnittstelle (AMS [Hol95]) *ODE Management API* [Apa10a]. Sie stellt Web Services zur Verfügung, mit der Daten aus ODE extrahiert werden können und Einfluss auf die Ausführung von Prozessen genommen werden kann. In ODE 2.0-beta-2 werden für diesen Zweck drei Services bzw. PortTypes[2] definiert: `ProcessManagement`, `InstanceManagement` und den `DeploymentService`. Dabei besitzt `ProcessManagement` folgende Operationen:

- `listProcesses`
- `setRetired`
- `listAllProcesses`:
- `setProcessProperty`
- `activate`
- `getExtensibilityElements`

[2]Es handelt sich um PortTypes, diese werden jedoch in ODE als Services bezeichnet.

- `setProcessPropertyNode`
- `getProcessInfoCustom`
- `getProcessInfo`
- `listProcessesCustom`

Mit diesen Operationen können Prozessmodelle aktiviert und zurückgezogen , Informationen über Prozessmodelle ausgelesen und Eigenschaften gesetzt werden (vgl. ODE-Prozessmodell in [Ste08]). `InstanceManagement` besitzt folgende Operationen:

- `getScopeInfoWithActivity`
- `resume`
- `listAllInstancesWithLimit`
- `recoverActivity`
- `getVariableInfo`
- `getScopeInfo`
- `listInstances`
- `listEvents`
- `fault`
- `getInstanceInfo`
- `suspend`
- `delete`
- `listAllInstances`
- `queryInstances`
- `getEventTimeline`
- `terminate`

Diese Operationen dienen der Verwaltung von Instanzen und ermöglichen unter anderem deren Auflisten, Monitoring, Unterbrechen und Weiterausführung. Für Deployment wird ein eigener Service, der `DeploymentService` von ODE angeboten. Er verfügt über die Operationen:

- `listProcesses`
- `deploy`
- `getProcessPackage`
- `undeploy`
- `listDeployedPackages`

Sie ermöglichen neben dem Auflisten von Prozessmodellen deren Deployment und Undeployment.

Verwendbarkeit für externe Transaktionen

Im Gegensatz zum Eventmodell mit Pluggable Framework von [KKS+06] bzw. [Ste08] sendet die ODE Manangement API nicht selbständig Informationen als Nachrichten an potenzielle Konsumenten. Konsumenten müssen Polling betreiben, um Informationen über Zustände aus ODE zu erhalten. Dabei ergibt sich das Problem, dass keine Blockaden gesetzt werden. ODE kann zwischen zwei Pollingaufrufen Schlüsselpositionen überspringen, an denen von der Mapping-Engine in den Ablauf eingegriffen werden muss. Die API bietet zwar die Möglichkeit Instanzen zu unterbrechen, dabei werden Instanzen jedoch komplett unterbrochen, während das Pluggable Framework nur einzelne Transitionen innerhalb der zustandsorientierten Modellen für Aktivitäten blockiert. Mit der API werden parallele Teilprozesse durch das Unterbrechen einer Instanz mit angehalten, auch wenn diese nicht koordiniert werden.

Aufgrund des beschriebenen Verhaltens eignet sich die ODE Management API nicht für die Koordination von Scopes.

3.1.4 Auswahl des Frameworks

Aus den drei vorgestellten Frameworks besitzt nur das Pluggable Framework die Funktionalität, die für externe Transaktionen benötigt wird. Alternativ könnte eine direkte Implementierung vorgenommen werden. Der Aufwand aufgrund der Komplexität und Fehlerträchtigkeit steht aber in keiner Relation zur geringen Modifikation, die im Pluggable Framework notwendig ist. Daher wird für die Realisierung in dieser Arbeit das Pluggable Framework verwendet.

3.2 Implementierungen des WS-Coordination-Frameworks

Für den angestrebte Implementierung in Kapitel 5 auf Seite 109 wird ein Coordinator mit Activation Service, Registration Service und den Protocol Services WS-BA und WS-AT benötigt. Dazu werden vorliegende Implementierungen des Frameworks untersucht und bewertet. Schließlich wird ein Coordinator ausgewählt, der für die spätere Implementierung als Coordination Service zur Verfügung steht (siehe Abschnitt 3.2.6 auf Seite 52).

3.2.1 Bewertungsverfahren

Die Bewertung der Coordinators erfolgt anhand der AHP-Methode [Saa00]. Dazu wird ein Kriterienkatalog aufgestellt, der in Abschnitt 3.2.2 erläutert wird. Es werden Kriterien in zwei Kategorien eingeteilt. Unter Kategorie K1 fallen Kriterien, die für eine spätere Verwendung zwingend notwendig sind. Kategorie K2 enthält weitere Kriterien, die in die Bewertung mit einfließen, die bei schlechter Bewertung eines Coordinators aber nicht zum Ausschluss führen. Jede Kombinaton von Kriterium K_x und Controller C_y erhält eine Zahl aus dem Intervall [0;3]. Sie dient der Bewertung des Coordinators bezüglich des Kriteriums und erhält das Symbol B_{K_x,C_y}.

Kriterien, die eine boolesche Aussage darstellen werden mit drei Punkten bewertet, falls die Aussage des Kriteriums für einen Coordinator wahr ist, null sonst. Jedes Kriterum erhält einen Gewichtungsfaktor G_{K_x}. Die Gesamtbewertung B_{C_y} eines Coordinators berechnet sich nach $B_{C_y} = \sum_{i=1}^{n} B_{K_i,C_y} G_{K_i}$. Der Coordinator, der die höchste Gesamtbewertung erhält, wird für die Implementierung verwendet.

3.2.2 Kriterienkatalog

Eine Übersicht mit Gewichtung und Kategorie über alle Kriterien findet sich in Tabelle 3.1. Ihr kann auch die Zugehörigkeit der Kriterien, zu den Kategorien K1 und K2 entnommen werden. Es werden folgende Kriterien verwendet:

- **Aktualität und Vollständigkeit:** Aktualität und Vollständigkeit eines Coordinators werden anhand der implementierten Standards WS-Coordination, WS-BA und WS-AT bewertet. Für die Verwendung der Standards WS-Coordination 1.2, WS-AT 1.2 und WS-BA 1.2 erhält der Coordinator drei Punkte. Für jede nichtaktuelle oder nicht vorhandene Version eines Standards wird ein halber bzw. ein ein Punkt abgezogen. Aktualität und Vollständigkeit werden für den Coordinator vorausgesetzt. Coordinators die nicht mindestens 2,5 Punkte erreichen, werden nicht eingesetzt. Aktualität und Vollständigkeit sind ein Kriterium der Kategorie K1.
- **Persistenz:** Apache ODE (siehe Abschnitt 2.5 auf Seite 23) und die Mapping-Engine, die in dieser Arbeit implementiert wird (vgl. Abschnitt 5 auf Seite 109), verwenden beide das Persistenzframework JPA, um ihren Datenbestand in einer Datenbank zu verwalten. Um durch einen Coordinator ohne Persistenzfähigkeiten die Persistenz des Gesamtsystems nicht zu verlieren, wird ein Coordinator mit Datenbankunterstützung bevorzugt. Dabei erhält ein Coordinator null Punkte, wenn er die Daten ausschließlich im Hauptspeicher hält, und keine Schnittstellen vorgesehen sind, um Daten persistent zu machen. Einen Punkt erhält er, wenn er selbst keine Datenbankunterstützung besitzt, aber eine Schnittstelle implementiert werden kann, um den Coordinator nachzurüsten. Zwei Punkte erhält ein Coordinator für Datenbankunterstützung und drei Punkte,

Beschreibung	Gewichtung G_{K_x}	Kategorie
Aktualität & Vollständigkeit	0,25	K1
Persistenz	0,25	K2
Erweiterbarkeit	0,25	K1
Skalierbarkeit	0,25	K2
Summe	1	

Tabelle 3.1: Kriterienkatalog mit Kriterium, Gewichtung und Kategorie

wenn er diese über ein datenbankunabhängiges Persistenzframework realisert. Persistenz ist ein Kriterium der Kategorie K2.

- **Erweiterbarkeit:** Die Erweiterbarkeit kann aufgrund der verschiedenartigen Coordinators nur schwer verglichen werden. Daher wird für die Erweiterbarkeit eines Coordinators keine Punkteverteilung vorgenommen, bei der einzelne Punkte aufgrund bestimmter einzelner Eigenschaften vergeben werden. Es wird vielmehr ein Gesamtbild des Coordinators bewertet. Zur Erweiterbarkeit gehören unter anderem das Vorhandensein und die Qualität der Dokumentation, das Vorhandensein und die Qualiät von Kommentaren im Quellcode und die Qualität des Quellcodes. Parizi et al. beschreiben in [PG08] Metriken für BPEL-Prozesse. Es existieren jedoch keine Implementierungen, um diese Metriken auf die BPEL-basierten Coordinators anzuwenden. Daher eignen sich Metriken hier nicht für einen Vergleich. Wie sich eine Bewertung bezüglich der Erweiterbarkeit eines Coordinators zusammensetzt, wird im jeweiligen Abschnitt des Coordinators erläutert. Persistenz ist ein Kriterium der Kategorie K1, da die Erweiterbarkeit für die Implementierung der in Abschnitt 4.6.2 auf Seite 85 erläuterten WS-BA-Erweiterung *Participant Triggered Compensation* benötigt wird.
- **Skalierbarkeit:** Da sowohl Apache ODE als auch die zu entwickelnde Mapping-Engine skalierbar ausgelegt sind, ist dies auch für den Coordinator von Vorteil, um die Gesamtskalierbarkeit des Systems nicht zu verlieren. Als Skalierbarkeit wird dabei in diesem Abschnitt das z. B. in [Fow02] erläuterte Verständnis der *horizontalen Skalierbarkeit* zugrunde gelegt, nach dem eine Anwendung skalierbar ist, wenn zusätzliche Rechner zu der Fähigkeit führen, zusätzliche Anfragen beantworten zu können. Es wird bezüglich der Bewertung lediglich unterschieden, ob ein Coordinator (skalierend) verteilt werden kann (drei Punkte) oder nicht (null Punkte). Die Skalierbarkeit ist ein Kriterium aus der Kategorie K2.

3.2.3 Gewichtung

Tabelle 3.1 enthält den Kriterienkatalog, mit den Kriterien, die im Abschnitt 3.2.2 auf der vorherigen Seite erläutert wurden. Alle Kriterien werden gleich stark gewichtet, erhalten also

den Gewichtungsfaktor 0,25, damit bei der Gesamtbewertung eines Coordinators ebenfalls wieder eine Bewertungsskala von null bis drei entsteht.

3.2.4 Untersuchte Implementierungen

In Abschnitt 3.2.1 auf Seite 41 wurde ein Bewertungsverfahren erläutert. Dessen Kriterienkatalog wurde in Abschnitt 3.2.2 auf Seite 41 zusammengestellt. In diesem Abschnitt werden nun die Coordinators detailliert beschrieben und bewertet.

Bei einer Recherche konnten folgende Coordinators gefunden werden:

- Apache Kandula-1
- Apache Kandula-2
- Implementierung der Diplomarbeit von Vetter
- Implementierung der Diplomarbeit von Müller
- Implementierung der Diplomarbeit von Paluszek
- Implementierung der Diplomarbeit von Mietzner
- JWST
- TracG

Es ist damit zu rechnen, dass weitere Coordinators existieren, die entweder nicht im Internet freigegeben wurden oder über die typischen Mittel, wie Suchmaschinen nicht auffindbar sind.

Apache Kandula-1 und Kandula-2

Bei Apache Kandula [Kan10] handelt es sich um ein Projekt von Apache mit zwei Teilprojekten basierend auf Apache Axis bzw. Apache Axis2. Sie heißen der Axis-Version entsprechend Kandula-1 und Kandula-2. Für die Teilprojekte wird je eine separate Bewertung vorgenommen, da Kriterien existieren, bezüglich derer die beiden Coordinators verschiedene Bewertungen erhalten. Beide Projekte werden nicht weiterentwickelt. Die Projekte befinden sich in verschiedenen Repositorys und verwenden keine gemeinsamen kandula-spezifischen Bibliotheken. Beide Projekte werden unter der *Apache Software Licence Version 2* lizenziert.

Merkmal	Apache Kandula-1
Sprache	Java
Frameworks	Axis
Letzte Aktivität (SVN)	25. Mai 2007
Standards	WS-C 1.0, WS-BA 1.0, WS-AT 1.0
Datenhaltung	HashMap (RAM)
Laufzeitumgebung	Servlet Container
URL	`http://ws.apache.org/kandula/1/index.html`

Tabelle 3.2: Übersicht Apache Kandula-1

Apache Kandula-1

Kandula-1 implementiert die Standards WS-Coordination 1.0, WS-BA 1.0 und WS-AT 1.0 vollständig, sowie Participant Services für die Standards (Aktualität und Vollständigkeit: 1,5). Der letzte SVN-Commit von Kandula-1 stammt vom 25. Mai 2007. Die Projekte werden nicht weiterentwickelt. Der WS-BA-Service von Kandula-1 entstand im Rahmen der Masterarbeit von Erven [EH07].

Das Basispaket von Kandula-1 heißt `org.apache.kandula`. Im Unterpaket `coordinator` befinden sich die Klassen für den Registration und den Activation Service. In den zwei Unterpaketen `coordinator.ba` und `coordinator.at` liegen die Klassen für die Protocol Services WS-BA und WS-AT. Während des Builds des Projektes mit Apache Maven [Mav10] wird das Paket `org.apache.kandula.coordinator.wscoor` aus WSDL-Dateien generiert. Dieses Paket enthält die Stubs der Services. Eine Konfiguration der Ports und Endpoints ist in der Datei `src/conf/kandula.properties` möglich.

Kandula-1 kann durch die Verwendung von Axis auf allen Servlet Containern deployt werden. Es befinden sich wenig Kommentare im Quelltext. Die meisten wurden automatisch generiert. Eine detaillierte Beschreibung der Pakete und ihres Zusammenspiels befindet sich in der Masterarbeit von Erven[EH07]. Auf der Kandula-Homepage befindet sich eine Anleitung für den Build-Prozess, das Deployment und eine kurze Erläuterung zur Verwendung der Participant Services (Erweiterbarkeit: 2). Kontexte werden in Kandula-1 nur im Hauptspeicher verwaltet (Persistenz: 0). Aufgrund fehlender Trennung von Services und Daten ist die Skalierbarkeit dieses Coordinators schlecht (Skalierbarkeit: 0).

Apache Kandula-2

Kandula-2 beinhaltet ebenfalls die Services von WS-Coordination 1.0, WS-BA 1.0, WS-AT 1.0 (Aktualität und Vollständigkeit: 1,5) sowie die Participant Services. Der letzte Beitrag im Subversion-Repository wurde am 19. August 2008 ins Subversion-Repository hochgeladen.

Kriterium	Bewertung
Aktualität & Vollständigkeit	1,5
Persistenz	0
Erweiterbarkeit	2
Skalierbarkeit	0
Gesamtbewertung	**0,875**

Tabelle 3.3: Bewertung Apache Kandula-1

Merkmal	Apache Kandula-2
Sprache	Java
Frameworks	Axis2
Letzte Aktivität (SVN)	19. August 2008
Standards	WS-C 1.0, WS-AT 1.0, WS-BA 1.0
Datenhaltung	HashMap (RAM)
Laufzeitumgebung	Servlet Container
URL	`http://ws.apache.org/kandula/2/index.html`

Tabelle 3.4: Übersicht Apache Kandula-2

Zur Datenverwaltung verwendet Kandula-2 ein Interface `Store` mit den Operationen `put`, `get` und `forget`. Dieses wurde durch eine Klasse `SimpleStore` implementiert, die die Coordination Contexts in einer HashMap im Hauptspeicher vorhält. Daten werden folglich nicht persistent gespeichert. Aus der fehlenden Trennung von Programmlogik und Daten folgt auch eine schlechte Skalierbarkeit (Skalierbarkeit: 0). Das Interface kann jedoch von einer neuen Klasse implementiert werden, die eine Speicherung der Kontexte in einer Datenbank realisiert (Persistenz: 1). Auf diese Weise können Skalierbarkeit und Persistenz hergestellt werden. Kandula-2 verwendet für den Build-Prozess – wie Kandula-1 – Apache Maven.

Das Basispaket heißt wie in Kandula-1 `org.apache.kandula`. Im Unterpaket `coordinator` liegen die Stubs und Implementierungen des Coordination Services. In den zwei Unterpaketen `coordinator.ba` und `coordinator.at` liegen die entsprechenden Klassen für WS-BA und WS-AT. Eine Konfiguration der Ports und Endpoints ist in der Klasse `org.apache.kandula.utility.KandulaConfiguration` möglich. Zusätzlich gibt es in Kandula2 die Unterpakete `context`, mit den Klassen für den Coordination Context und `storage` mit `Store` und `SimpleStore` (siehe oben). Das Paket `org.apache.kandula.participant` enthält eine API für Participants. Diesen fehlt eine vollständige Implementierung von WS-BA. Einzelne Klassen des Pakets enthalten Funktionen, die nicht ausprogrammiert wurden. Kandula-2 ist auf JEE Servlet Containern lauffähig. Der Programmcode von Kandula-2 enthält wenige Kommentare. Die meisten davon wurden automatisch generiert. Weitere Dokumentation ist von Apache nicht verfügbar (Erweiterbarkeit: 1).

Kriterium	Bewertung
Aktualität & Vollständigkeit	1,5
Persistenz	1
Erweiterbarkeit	1
Skalierbarkeit	0
Gesamtbewertung	**0,875**

Tabelle 3.5: Bewertung Apache Kandula-2

Merkmal	Projekt
Sprache	Java
Frameworks	Axis
Letzte Aktivität	Januar 2006
Standards	WS-C 1.0
Datenhaltung	MySQL
Laufzeitumgebung	Servlet Container
Besonderheiten	Keine Protocol Services

Tabelle 3.6: Übersicht der Implementierung der Diplomarbeit von Thorsten Vetter

Implementierung der Diplomarbeit von Thorsten Vetter

Vetter entwickelte im Rahmen seiner Diplomarbeit [Veto6] einen Coordinator. Er basiert auf Apache Axis 1.3 und verwendet eine MySQL-Datenbank zur Speicherung der Daten. Coordination Types erben von der Klasse „CoordinationType“. Dadurch lassen sich schneller als bei beispielsweise Kandula-1 und 2 neue Coordination Types implementieren. Auf die Datenbank wird ohne Verwendung von Transaktionen zugegriffen (Persistenz: 2).

In seiner Arbeit beschreibt Vetter die Implementierung von WS-Coordination, WS-BA und WS-AT. In der vorliegenden Implementierung sind jedoch nur der ActivationService und der RegistrationService aus WS-Coordination enthalten. Sie befinden sich in den Paketen `coordinator.wscoordination.services.activation` bzw. `coordinator.wscoordination.services.registration`. Im Paket `coordinator.wscoordination.utils` in der Klasse Repository befindet sich die Verwaltung der Kontexte inklusive der Anbindung an die Datenbank. In `coordinator.wscoordination.schema` liegen Schemendefinitionen, die – wie die Stubs und Sceletons der Services – auch mit dem Axis-Tool WSDL2Java generiert wurden. In `coordinator.btp` befindet sich desweiteren ein BTP-Transaktionsservice.

Die Repository-Klasse verwendet eine fest einprogrammierte Datei `c:/coordination/mysql.properties`. Um bessere Portabilität zu gewährleisten, sollte die Einbindung der Datenbank so gelöst werden, dass sie weder von der absoluten Position

Kriterium	Bewertung
Aktualität & Vollständigkeit	0
Persistenz	2
Erweiterbarkeit	2
Skalierbarkeit	3
Gesamtbewertung	**1,750**

Tabelle 3.7: Bewertung der Implementierung der Diplomarbeit von Vetter

Merkmal	Projekt
Sprache	BPEL 1.1
Engines	nur ActiveBPEL
Letzte Aktivität	Juni 2006
Standards	WS-C 1.0, WS-BA 1.0, WS-AT 1.0
Kontext-Persistenz	ja, via WfMS

Tabelle 3.8: Übersicht der Implementierung der Diplomarbeit von Thomas Müller

von Dateien noch von festen Datenbank-Parametern abhängt. Mit diesen Veränderungen ist Skalierbarkeit ist durch Nutzung der Datenbank gut (Skalierbarkeit: 3). Dokumentiert wird das Projekt in der Diplomarbeit. Im Projekt befindet sich eine Anleitung (Dokumentation: Erweiterbarkeit: 2).

Implementierung der Diplomarbeit von Thomas Müller

Müller implementiert einen Coordination Service für seine Diplomarbeit [Mülo6]. Er modelliert dazu Prozesse in BPEL 1.1 unter Zuhilfenahme des ActiveBPEL Designers. In den Prozessen wird die ActiveBPEL Extension verwendet, dadurch sind die BPEL-Prozesse nur auf der ActiveBPEL-Engine lauffähig. Implementiert wird jeweils die Version 1.0 der Standards WS-Coordination, WS-BA und WS-AT. Müller beschreibt seine Implementierung der Diplomarbeit in einer Anleitung, die Teil des Projektes ist, als „weder vollständig noch korrekt“. Für WS-BA und WS-AT wurden nicht alle spezifizierten Coordination Messages modelliert, zudem gibt es kein Fault-Handling und keine Unterstützung von Timeouts (Aktualität & Vollständigkeit: 0). Daten werden in Form von Variablen im Prozess selbst gespeichert. Die BPEL-Engine macht diese bei der Ausführung persistent (Persistenz: 3). Die Skalierbarkeit entspricht somit der (typischerweise guten) Skalierbarkeit der Workflowmaschine (Skalierbarkeit: 3).

Die Prozesse sind aufgeteilt in zwei Pakete, je eines für WS-AT bzw. WS-BA. Beide Pakete enthalten einen ActivationService und einen RegistrationService und weitere Prozesse,

Kriterium	Bewertung
Aktualität & Vollständigkeit	0
Persistenz	3
Erweiterbarkeit	2
Skalierbarkeit	3
Gesamtbewertung	**2,000**

Tabelle 3.9: Bewertung der Implementierung der Diplomarbeit von Müller

Merkmal	Projekt
Sprache	BPEL 1.1
Engines	nur ActiveBPEL
Letzte Aktivität	Juli 2006
Standards	WS-C 1.0, WS-BA 1.0, WS-AT 1.0
Kontext-Persistenz	ja, via WfMS

Tabelle 3.10: Übersicht Implementierung der Diplomarbeit von Mietzner

die als Unterprozesse eingebunden werden. Durch die Aufteilung der Prozesse ensteht ein höherer Kommunikationaufwand, der aber eine geringere Komplexität der Prozesse ermöglicht und den Grad der Wiederverwendung steigert. Müller kann auf diese Art zum Beispiel die Prepare-Phase im 2-Phasen-Commit-Protokoll in WS-AT für Volatile2PC und Durable2PC wiederverwenden, da der Ablauf der gleiche ist [Mül06]. Erweitern lassen sich die Prozesse durch Entwickeln eigener Prozesse mit anschließendem Invokieren der vorhandenen Prozesse (Erweiterbarkeit: 2). Eine Dokumentation der Prozesse erfolgt in der Diplomarbeit.

Implementierung der Diplomarbeit von Michael Paluszek

Paluscek verwendet in seiner Diplomarbeit [Pal07] ebenfalls einen Coordinator. Es handelt sich hierbei um den Coordinator von Vetter, der um diplomarbeitspezifische Funktionen erweitert wurde. Hierbei handelt es sich um einen *LoopCoordinatorService* und einen *Scope Coordinator Service*. Da die beiden Services in dieser Arbeit nicht benötigt werden und die Implementierung der Diplomarbeit von Vetter bereits in Abschnitt 3.2.4 auf Seite 46 beschrieben wurde, wird diese Implementierung nicht weiter erläutert und nicht bewertet.

Kriterium	Bewertung
Aktualität & Vollständigkeit	1,5
Persistenz	3
Erweiterbarkeit	1
Skalierbarkeit	3
Gesamtbewertung	**2,125**

Tabelle 3.11: Bewertung der Implementierung der Diplomarbeit von Mietzner

Merkmal	Projekt
Sprache	Java 1.4
Frameworks	Axis, JBoss
Letzte Aktivität	Juni 2004
Standards	WS-C 1.0, WS-BA 1.0
Kontext-Persistenz	Entity Beans

Tabelle 3.12: Übersicht über JWST

Implementierung Ralph Mietzner

Mietzner modelliert wie Müller BPEL-Prozesse als Coordinator für seine Diplomarbeit [Mie06]. Für WS-BA 1.0 und WS-AT 1.0 wurden je zwei BPEL-Prozesse definiert, zum einen WS-Coordination mit Activation- und Registration-Service, zum anderen WS-BA/WS-AT (Aktualität und Vollständigkeit: 1,5). Mietzner modelliert die Prozesse mit Hilfe des ActiveBPEL Designers und verwendet auch die ActiveBPEL-Erweiterung. Wie bei der Implementierung von Müller sind die Prozesse daher nur auf der ActiveBPEL-Engine lauffähig. Ihre Skalierbarkeit gleicht somit der Skalierbarkeit der ActiveBPEL-Engine (Skalierbarkeit: 3). Eine Erweiterung der Prozessmodelle ist theoretisch möglich, jedoch sind die Modelle komplex und führen so bei einer geplanten Erweiterung zu einem hohen Aufwand (Erweiterbarkeit: 1). Dieser Coordinator wird in der Implementierung von [PML07] verwendet.

JWST

Bei JWST [Zam10] handelt es sich um ein studentisches Projekt von Simon Zambrovski. Es basiert auf Axis sowie auf JEE-Komponenten aus JBOSS und beinhaltet alle Komponenten, die in WS-Coordination und WS-BA beschrieben sind. Zusätzlich wurden auch Tools für Monitoring sowie Logging entwickelt. Das Projekt enthält keine Implementierung von WS-AT. Eine einfache Implementierung für Participants sowie Beispiele zu deren Verwendung sind vorhanden. WS-Coordination wurde nach einer Vorversion des

Kriterium	Bewertung
Aktualität & Vollständigkeit	0,5
Persistenz	3
Skalierbarkeit	3
Erweiterbarkeit	1
Gesamtbewertung	**1,875**

Tabelle 3.13: Bewertung von JWST

Merkmal	Projekt
Sprache	Java
Frameworks	Axis2
Letzte Aktivität	Mai 2010
Standards	WS-C 1.2, WS-BA 1.2, WS-AT 1,2
Kontext-Persistenz	via EJB

Tabelle 3.14: Übersicht über TracG

1.0-Standards von 2003 (http://schemas.xmlsoap.org/ws/2003/09/wscoor/wscoor.xsd) implementiert. Die letzten Modifikationen des Projektes stammen von Juni 2004. Zu diesem Zeitpunkt haben die Spezifikationen von WS-Coordination, WS-BA und WS-AT noch nicht in der Version 1.2 existiert (Aktualität und Vollständigkeit: 0,5). Zambrovski implementiert neben den Services, die in den Paketen `org.zambrovski.wst.services.coordination` und `org.zambrovski.wst.services.ba` liegen, auch einen JMX-Monitor, der es ermöglicht den aktuellen Zustand einzelner Kontextinstanzen anzuzeigen. Persistenz wird durch die Verwendung von J2EE-Beans (J2EE 1.4) erreicht (Persistenz: 3; Skalierbarkeit: 3). Dokumentation ist in Form einer Studienarbeit [Zam04] vorhanden. Diese ist veraltet, da nach Abschluss der Studienarbeit am Coordinator weiterentwickelt wurde. Dadurch wird eine Erweiterung erschwert. (Erweiterbarkeit: 1).

TracG

TracG [HRRF10] ist ein Projekt der Universität Hamburg im Arbeitsbereich „Verteilte Systeme und Informationssysteme“ in welchem eine vollständige WS-Coordination-Lösung implementiert wird. Unterstützt werden die Standards WS-Coordination 1.2, WS-BA 1.2 und WS-AT 1.2 mit allen in den Spezifikationen beschriebenen Elementen (Aktualität und Vollständigkeit: 3). Der Coordinator basiert auf Axis2 und wird von Stefan Fink, Michael von Riegen und Martin Hussemann entwickelt.

Kriterium	Bewertung
Aktualität & Vollständigkeit	3
Persistenz	3
Skalierbarkeit	2
Erweiterbarkeit	2
Gesamtbewertung	**2,50**

Tabelle 3.15: Bewertung von TracG

Im Projekt wird im Paket `de.vsis.coordination.coordinator.util.storage` ein Interface `Storage` definiert, welches die Methoden `set`, `get`, `delete` und `cleanOldValues` enthält. Zwei Klassen implementieren dieses Interface. `SimpleStorage` speichert die Kontext-Daten in eine `ConcurrentHashMap`, die die Daten im Hauptspeicher hält. `JDBCStorage` ermöglicht das Einbinden einer MySQL-Datenbank via JDBC. Mit fest einprogrammierter Datenbankunterstützung sind sowohl Skalierbarkeit als auch Persistenz vorhanden. Alternativ können weitere Storage-Klassen von `Storage` implementiert werden. Transaktionen zwischen dem Coordinator und Datenbank werden in `JDBCStorage` nicht eingesetzt (Skalierbarkeit: 3; Persistenz: 2). Die Services für WS-BA und WS-AT befinden sich in den Paketen `de.vsis.coordination.coordinator.ba` und `de.vsis.coordination.coordinator.at`. Kontexte für werden in `de.vsis.coordination.coordinator.context` definiert. Das Projekt wurde als Open-Source-Projekt unter der GNU Lesser General Public License auf Sourceforge freigegeben [HRF10]. TracG wird in [HRR07] vorgestellt. Weitere Dokumentation findet sich in Finks Diplomarbeit [Fin09], in welcher Fink TracG erweitert (Dokumentation: 2).

Schlussfolgerung

Eine Übersicht über das Abschneiden aller Implementierungen findet sich in Tabelle 3.16 auf der nächsten Seite. Abschließend lassen sich folgende Sachverhalte beobachten: fast alle Projekte wurden als Teil einer einen zeitlich begrenzten studentischen und/oder Forschungsarbeit erstellt. Da diese Arbeiten entsprechend zurückliegen, sind die verwendeten Standards veraltet. Nur TracG wird noch aktiv weiterentwickelt und implementiert die Standards WS-Coordination 1.2, WS-BA 1.2 und WS-AT 1.2. An Kandula wurde zwar ebenfalls über einen längeren Zeitraum entwickelt, das Projekt wird jedoch nicht fortgeführt. Die Implementierungen, die als Teil einer studentischen Arbeit entstanden, können als Machbarkeitsstudie angesehen werden. Es muss somit von Fehlern innerhalb der Umsetzung ausgegangen werden. Die Erweiterbarkeit der Projekte ist zwar in allen Projekten gegeben, führt jedoch oft mangels aktueller Standards und aufgrund schlechter Dokumentation zu einem hohen Aufwand.

Das Projekt TracG ist nach dieser Bewertungsmethode und nach dem verwendeten Kriterienkatalog das beste Projekt.

Implementierung	Bewertung
Apache Kandula-1	0,875
Apache Kandula-1	0,875
Implementierung DA[a] Vetter	1,750
Implementierung DA Müller	2,000
Implementierung DA Paluszek	–
Implementierung DA Mietzner	2,125
JWST	1,875
TracG	**2,500**

Tabelle 3.16: Alle Bewertungen: TracG erhält die höchste Bewertung

[a] Diplomarbeit

3.2.5 Generieren von BPEL-Prozessen für WS-Coordination aus CPGs

Eine weitere Möglichkeit Coordinators für vorher definierte Coordination Protocol Graphs (CPG) wie beispielsweise in Abbildung 2.1 auf Seite 18 zu erhalten, stellt Kopp in [KWM+08] vor. Durch die Verwendung eines MDA-Patterns können aus CPGs BPEL-Prozesse generiert werden.

Es liegt zum Zeitpunkt der Arbeit keine Implementierung für WS-BPEL 2.0 vor, so dass diese Möglichkeit nicht näher untersucht wird.

3.2.6 Wahl einer geeigneten Implementierung

Der Coordinator aus dem Projekt TracG erreicht in der Bewertung aus Abschnitt 3.2.4 auf Seite 50 die höchste Bewertung. Da TracG außerdem alle Kriterien aus Kategorie K1 erfüllt, wird dieser Coordinator für die in Abschnitt 5 auf Seite 109 realisierte Implementierung verwendet.

3.3 Coordinated Scopes

Um ein Mapping zwischen BPEL und WS-Coordiation-basierten Transaktionsprotokollen zu definieren, muss zunächst ein Rahmen definiert werden, welche BPEL-Elemente auf welche Weise an Transaktionen teilnehmen können. In [KML08] wird eine abstrakte BPEL-Syntax definiert, die zu diesem Zweck verwendet wird. Die verwendeten Konstrukte werden am Ort der ersten Verwendung eingeführt.

3.3.1 Coordinated Scopes

Definition 3.3.1 (Scopealike)
$\mathcal{A}_{Scopealike} = \mathcal{A}_{Scope} \cup \mathcal{A}_{Invoke}$ (Coordinated Scopes, kurz CScope) seien alle Aktivitäten, die über einen Fault-Handler und einen Compensation-Handler verfügen und im Eventmodell nach [Ste08] Events des Eventmodells für Scopes empfangen und senden können. Insbesondere ist auch die Prozess-Aktivität ein Scopealike, da in [KML08] `process` $\in \mathcal{A}_{Scope}$ definiert wird.

Definition 3.3.2 (Coordination Types)
CT sei die Menge der Coordination Types, die in Spezifikationen definiert werden, die WS-Coordination als Framework benutzen.

Definition 3.3.3 (Coordination Protocols)
CP sei die Menge der Coordination Protocols, die in Spezifikationen definiert werden, die WS-Coordination als Framework benutzen.

Definition 3.3.4 (External Coordination Role)
ECR sei die Menge $\{$`Initiator`, `Participant`$\}$. Eine External Coordination Role (ECR) ecr $\in$ ECR beschreibt die Rolle eines Scopealike als Participant oder Initiator bezüglich eines auf WS-Coordination basierenden Transaktionsmodelles.

Definition 3.3.5 (Internal Coordination Role)
ICR sei die Menge $\{$`new`, `participating`, `independet`$\}$. Eine Internal Coordination Role (ICR) icr $\in$ ICR beschreibt das Verhalten eines Scopealike bezüglich der Zugehörigkeit zu einem möglicherweise koordinierten umschließenden Scopealike. Um die Länge von Formeln zu reduzieren, werden in Formeln bei Bedarf die Anfangsbuchstaben `n`, `p` und `i` als Abkürzung verwendet.

Definition 3.3.6 (Scope Coordination Properties)
Sei $SCP \subset ICR \times ECRs \times CT \cup \{\bot\} \times CP \cup \{\bot\} \cup \mathbb{B}$, dann ist $scp \in SCP$ ein Tupel, das Scope Coordination Properties (SCP) beschreibt und SCP die Menge aller SCP, die in einem Prozessmodell vorkommen. Der boolsche Wert entspricht dabei einem Wert `optional`, der beim Policy Attachment gesetzt werden kann, um damit zu markieren, ob ein Coordinated Scope (Definition 3.3.7 auf der nächsten Seite) durch die Mapping-Engine koordiniert werden muss (vgl. Abschnitt 3.4 auf Seite 57). SCP werden explizit oder implizit gesetzt. Es gelten die Regeln aus Abschnitt 3.3.4 auf Seite 56.

Definition 3.3.7 (Coordinated Scope)
Sei $CSCOPE \subset \mathcal{A}_{Scopealike} \times SCP$, dann ist $cscope \in CSCOPE$ ein um Scope Coordination Properties erweiterter Scopealike und CSCOPE die Menge aller Coordinated Scopes.

Sei $s \in \mathcal{A}_{Scopealike}, irc \in ICR, ecr \in ECR, ct \in CT, cp \in CP$ und $bool \in \mathbb{B}$, so definieren sich die beschriebenen Funktionen auf einem Coordinated Scope wie folgt:

- $\pi_a : CSCOPE \rightarrow \mathcal{A}_{Scopealike} : \pi_a((s,(icr,ecr,ct,cp,bool))) \mapsto s$
- $\pi_{icr} : CSCOPE \rightarrow ICR : \pi_{icr}((s,(icr,ecr,ct,cp,bool))) \mapsto icr$
- $\pi_{ecr} : CSCOPE \rightarrow ECR : \pi_{ecr}((s,(icr,ecr,ct,cp,bool))) \mapsto ecr$
- $\pi_{ct} : CSCOPE \rightarrow CT : \pi_{ct}((s,(icr,ecr,ct,cp,bool))) \mapsto ct$
- $\pi_{cp} : CSCOPE \rightarrow CP : \pi_{cp}((s,(icr,ecr,ct,cp,bool))) \mapsto cp$
- $\pi_{optional} : CSCOPE \rightarrow \mathbb{B} : \pi_{optinal}((s,(icr,ecr,ct,cp,bool))) \mapsto bool.$

Es existiert genau ein SCP pro Scopealike:

$$\forall a \in CSCOPE \wedge \forall b \in CSCOPE(\pi_a(a) = \pi_a(b)) \Rightarrow (a = b)$$

$$\forall a \in \mathcal{A}_{Scopealike} \exists b \in CSCOPE \pi_a(b) = a$$

Der Prozessdesigner kann per WS-Policy explizite SCP an alle Scopealikes angefügen. Die Vorgehensweise für das Anfügen mit WS-Policy wird in 3.4 auf Seite 57 erläutert. Für alle anderen Scopealikes werden nach den in Abschnitt 3.3.4 auf Seite 56 benannten Regeln implizite SCP abgeleitet.

Definition 3.3.8 (Coordinated Process)
`cprocess` $\in CSCOPE$ sei ein Coordinated Process. Für `cprocess` gelte: $\pi_a cprocess =$ `process`. Der Coordinated Process ist ein Sonderfall eines Coordinated Scope.

Im Folgenden ist mit *Scope* stets *Coordinated Scope* gemeint, da jeder Scope zumindest über implizite Scope Coordination Properties verfügt, unabhängig davon, ob diese tatsächlich koordinieren oder koordiniert werden.

3.3.2 Semantik der Internal Coordination Role

Die Werte `new`, `participate` und `independent` erhalten in den SCP eines Scopes folgende Semantik:

- `new`: Für einen Scope wird ein neuer Coordination Context (als Initiator oder Participant) erzeugt.
- `participate`: Der Scope nimmt am Coordination Context seines umgebenden Scopes teil.

- Scopes mit `independent` arbeiten weder mit einem neuen noch mit dem Coordination Context des umgebenden Scopes, sondern verwenden ausschließlich die Koordination von BPEL. Auch weitere eingebettete Scopes können dann nicht den Coordination Context des transitiv umgebenden[3] Scope verwenden.

Es gilt: Für die Prozessaktivität darf als ICR nur `new` oder `independent` definiert werden. Dies liegt daran, dass Prozessaktivitäten keinen umgebenden Scope besitzen, an dessen Transaktion sie teilnehmen könnten.

$$\forall cs \in CSCOPE, \pi_a(cs) = \texttt{process} : \pi_{ecr}(cs) \in \{\texttt{n}, \texttt{i}\}$$

Da `<invoke>`-Aktivitäten keine Möglichkeit des Nachrichtenempfangs bieten ohne vorher eine Nachricht zu senden, können sie in der ICR `new` nur als Initiator auftreten:

$$\forall cs \in CSCOPE, (\pi_a(cs) \in \mathcal{A}_{Invoke}), \wedge(\pi_{icr}(cs) = \texttt{n}) : \pi_{ecr}(cs) = \texttt{initiator}$$

Besitzt ein CScope die ICR `independent`, dürfen die direkt von ihm eingeschlossenen CScopes nur die Rollen `new` und `independant` besitzen.

$$\forall cs \in CSCOPE, \pi_{icr}(cs) = \texttt{i} : (\forall cs_x \in CSCOPE, \pi_a(cs_x) \in \mathit{children}(cs) : \pi_{icr}(cs_x) \in \{\texttt{i}, \texttt{n}\})$$

Besitzt ein Scope die ICR `participating`, sind die Felder Coordination Context, ECR, CT und CP nicht belegt ($\perp$).

3.3.3 Regeln bezüglich Coordination Type und External Coordination Role

Für verschiedene CoordinationTypes und in Abhängigkeit davon, ob ein Scope als Initiator oder Participant bei der Koordination auftritt, ergeben sich weitere Regeln, die vom Prozessmodellierer eingehalten werden müssen. Diese werden für die jeweiligen Coordination Types in Abschnitt 4 auf Seite 59 beschrieben.

Prozessmodelle, die gegen die Regeln verstoßen, führen beim Deployment in der Mapping-Engine zu einem Fehler. Sendet ein Partner des Prozesses eine Application Message mit einem Coordination Context und entspricht dessen Coordination Type nicht dem des Scopes, in dem die Nachricht empfangen wird, so wird von der Mapping-Engine ein Fault in den Scope gesendet (siehe Abschnitte 4.6 auf Seite 80 und 4.8 auf Seite 103). Ebenso wird ein Fault in den Scope geworfen, wenn dieser sich nicht in der ECR `participant` befindet.

[3] transitiv bezüglich der Hierarchierelation aus [KML08]

Definition 3.3.9 (Effective Scope)
Seien $cs \in CSSCOPE$ *und* $cs_{parent} \in CSCOPE$ *zwei beliebige CScopes und gelte* $\pi_a(cs) \in children(cs_{parent})$.

Dann wird die Funktion es, die den Effective Scope von cs zurückliefert, wie folgt definiert:

$$es(cs) = \begin{cases} cs, & \text{falls } \pi_{icr}(cs) = \texttt{new} \\ \bot, & \text{falls } \pi_{icr}(cs) = \texttt{independent} \\ cs_{parent}, & \text{falls } \pi_{icr}(cs) = \texttt{participating} \end{cases}$$

Für die Prozessaktivität gilt:

$$es(\texttt{process}) = \begin{cases} \texttt{process}, & \text{falls } \pi_{icr}(\texttt{process}) = \texttt{new} \\ \bot, & \text{falls } \pi_{icr}(\texttt{process}) = \texttt{independent} \end{cases}$$

Coordination Context

Ein Coordination Context wird für einen Scope zur Laufzeit entweder von der Mapping-Engine oder von einem Coordinator während einer Aktivierung erzeugt. Für alle Aktionen, die während der Lebenszeit eines Scope auftreten, die mit der Koordinierung im Zusammenhang stehen, ist der Coordination Context des Effective Scope zu verwenden.

3.3.4 Implizite SCP und Standardwerte

Für Coordinated Scopes (CScope) müssen keine expliziten Scope Coordination Properties (SCP, siehe Definition 3.3.6 auf Seite 53) definiert werden. Werden für einen CScope keine expliziten SCP deklariert, wird für den CScope die ICR `independent` angenommen. Dies bedeutet, dass er an keinen externen Transaktionen teilnimmt und keine Participants koordiniert. ECR, CT, CP verbleiben dann unbelegt ($\bot$).

Werden für einen CScope explizite SCP definiert, gelten folgende Regeln:

- Wird ICR `new` definiert, so müssen auch CT und ECR definiert werden.
- Wird ICR `participating` definiert, so werden CT, CP und ECR ignoriert.
- Wird ICR `independent` definiert, so werden alle anderen SCP ignoriert.
- Wird `optional` nicht gesetzt, wird es standardmäßig mit `false` belegt.
- Wird ECR `initiator` definiert, wird `optional` ignoriert.

Listing 3.1: XML-Syntax für Scope Coordination Properties

```
<x4b:ScopeCoordinationProperties
    optional="true"|"false"?
    x4b:icr="new"|"participating"|"independent"
    x4b:ecr="participant"|"initiator"?>
  <x4b:CoordinationType>xs:anyURI<x4b:CoordinationType>?
  <x4b:CoordinationProtocol>xs:anyURI<x4b:CoordinationProtocol>?
</x4b:ScopeCoordinationProperties>
```

Listing 3.2: Referenzierung von Scopealikes innerhalb eines Prozessmodells

```
<x4b:process id="QName" expressionLanguage="anyURI"?>
  expression
</x4b:process>
```

3.4 WS-Policy Attachment für Coordinated Scopes

In [W3C07b] wird Policy Attachment spezifiziert. Dabei handelt es sich um ein Konstrukt, das ein Versehen von beliebigen XML-Dokumenten und deren Elementen mit Policies von außen erlaubt. Ein Attachment besteht dabei aus einem `<AppliesTo>`-Element und einer Policy oder PolicyReference. Anhand des `<AppliesTo>`-Elements werden Elemente referenziert, an das die Policy angehängt wird. In [TMW+04] werden Transaction Policies vorgestellt. Diese werden dann in Elemente von WSDL, wie beispielsweise `<Port>` und in Teile von BPEL-Prozessen, wie beispielsweise in `<PartnerLink>` eingefügt oder per Attachment angehängt. Angelehnt an diesen Policies wird der XML-Typ `<x4b:ScopeCoordinationProperties>` in Listing 3.1 definiert. Er enthält alle Felder, die in Abschnitt 3.3 auf Seite 53 definiert wurden.

Um im `<AppliesTo>`-Element aus WS-Policy Attachment [W3C07b] Scopealikes im Prozessmodell zu referenzieren, wird der XML-Typ `<x4b:process>`, wie in Listing 3.2 definiert, verwendet. Dieser Typ referenziert ein Prozessmodell anhand dessen QName und selektiert Scopealikes des Prozessmodells mit einem XPath-Ausdruck.

Ein vollständiges Attachment kann beispielsweise wie in Listing 3.3 auf der nächsten Seite deklariert werden. In diesem Beispiel steht das Attribut `optional` auf `true`, obwohl die Mapping-Engine für den betroffenene Scope selbst koordiniert. Das Attribut wird von der Mapping-Engine ignoriert (siehe Abschnitt 3.3.4 auf der vorherigen Seite).

Listing 3.3: Beispiel eines Policy-Attachments

```
<wsp:PolicyAttachment>
  <wsp:AppliesTo>
       <x4b:process id="ns:p1" xmlns:ns="http://www.example.com/processes"
           expressionLanguage="xpath1.0">
         /process/scope[1]/scope[3]
    </x4b:process>
  </wsp:AppliesTo>
  <wsp:Policy>
    <x4b:ScopeCoordinationProperties
        optional="true"
        icr="new"
        ecr="initiator">
      <x4b:CoordinationType>
        http://docs.oasis-open.org/ws-tx/wsba/2006/06/MixedOutcome
      <x4b:CoordinationType>
    </x4b:ScopeCoordinationProperties>
  </wsp:Policy>
</wsp:PolicyAttachment>
```

4 Transaktionales Mapping zwischen BPEL und WS-Coordination

In Abschnitt 3.1.1 auf Seite 35 wurde gezeigt, wie Informationen in Form von Events aus Prozessen extrahiert werden und Einfluss auf die Ausführung von Prozessen genommen werden kann. In Abschnitt 3.3 auf Seite 53 wurden Coordinated Scopes definiert, die aus einem Scopealike und Scope Coordination Properties (SCP) bestehen. Es wurde aufgezählt, welche Rollen Scopes einnehmen können und welche Einschränkungen für das Modell gelten. Ziel der Arbeit ist, in die BPEL-Engine Unterstützung der Standards WS-BA und WS-AT sowohl als Participant als auch als Initiator zu integrieren. Dazu werden nun Verfahren beschrieben, wie Coordinated Scopes mit Coordinators und Participants für WS-BA und WS-AT interagieren. Zunächst wird erläutert, wie ein Coordination Context in die ausgehenden Application Messages von `<invoke>`-Aktivitäten eingefügt wird. Dazu wird das Eventmodell für `<invoke>`-Aktivitäten erweitert. Danach wird beschrieben, wie Scopes an WS-BA als Initiator (Abschnitt 4.5 auf Seite 64) und Participant (Abschnitt 4.6 auf Seite 80) sowie an WS-AT als Initiator (Abschnitt 4.7 auf Seite 94) und Participant (Abschnitt 4.8 auf Seite 103) teilnehmen können. Alle Mappingverfahren werden dabei als Automaten realisiert, deren Transitionstabellen in den genannten Abschnitten erläutert werden.

4.1 Terminologie

Es folgt eine kurze Zusammenfassung der verwendeten Begriffe mit ihrer Bedeutung, um das Leseverständnis zu erhöhen und Mehrdeutigkeiten aufzuklären, sofern die Begriffe nicht bereits in Abschnitt 3.3 auf Seite 53 definiert wurden.

- **AT-Scope:** Scope oder Prozess in der Internal Coordination Role (ICR) `new`, External Coordination Role (ECR) `initiator`, der seine Participants mittels WS-AT koordiniert und dafür bestimmte Voraussetzungen erfüllt (siehe Abschnitt 4.7.2 auf Seite 94).
- **Coordinator:** Zusammenfassender Begriff für die Web Services Registration Service, Protocol Services und, falls vorhanden, Activation Service, die auf dem selben Datenbestand zusammen agieren (siehe Abschnitt 2.3 auf Seite 15). In der Spezifikation von WS-Coordination [OAS09c] wird hierfür der Bezeichner *Coordination Service* verwendet, wobei ein Coordination Service typischerweise alle drei Services enthält. Dies muss bei der hier verwendeten Bezeichnung *Coordinator* nicht der Fall sein.

- **Coordination Partner:** Participant oder Coordinator
- **Event:** Die Bezeichnung „Event“ wird stets für Events aus dem Eventmodell von Steinmetz (siehe Abschnitt 3.1.1 auf Seite 35) verwendet. Namen von Events werden *kursiv* gedruckt.
- **Externe Koordinierung:** Bezeichnung für Verfahren, bei welchen die Mapping-Engine Participants mithilfe eines Coordinators koordiniert. Participants melden sich am Coordinator an, die Mapping-Engine interagiert mit dem Coordinator über ein Initiator- oder Completion-Protokoll.
- **Interne Koordinierung:** Bezeichnung für Verfahren, bei welchen die Mapping-Engine Participants koordiniert, indem sie sich an der Mapping-Engine registrieren. Die Mapping-Engine sendet die Coordination Messages aus den jeweiligen Coordination-Protokollen.
- **Mapping-Engine:** Zu realisierende Komponente, die Outgoing Events der BPEL-Engine und Coordination Messages interpretiert und entsprechende Coordination Messages bzw. Incoming Events sendet.
- **BPEL-Engine:** BPEL-Workflow-Managementsystem mit Generic Controller aus dem Pluggable Framework (3.1.1 auf Seite 35). Typischerweise besitzen BPEL-Engines keine Unterstützung für das Pluggable Framework. In dieser Arbeit ist es zwingende Voraussetzung, weshalb im Folgenden vom Vorhandensein des Generic Controllers ausgegangen wird.
- **Application Message:** Für Nachrichten, die zwischen Web Services gesendet werden, die Geschäftslogik anbieten (und nicht dem Koordinieren dienen), wird die Bezeichnung *Application Message* verwendet.
- **Message:** Für Nachrichten zwischen Participants und Coordinators wird die Bezeichnung *Message* oder, um noch deutlicher den Unterschied hervorzuheben, *Coordination Message* verwendet. Messages werden serifenfrei gedruckt.
- **ODE:** ODE wird als Kurzform für Apache ODE verwendet.
- **Prozessmodellierer:** Eine Person, die das Prozessmodell modelliert, welches beschrieben wird.

Wie bereits in Abschnitt 3.3 auf Seite 53 erläutert, besitzen alle Scopes entweder implizite oder explizite Scope Coordination Properties. Daher wird in den folgenden Abschnitten nicht zwischen CScope und Scope unterschieden. Falls nicht anders angegeben, beziehen sich „Scope“ immer auf einen Scope zur Laufzeit, während der das Mapping stattfindet.

Listing 4.1: Reference Property für eine <invoke>-Aktivität (zur Laufzeit)

```
<x4b:ScopeReference>
  <x4b:Invoke id="5" />
</x4b:ScopeReference>
```

4.2 Invoke und Weitergabe des Coordination Context

Mit der `<invoke>`-Aktivität werden externe Services aufgerufen und müssen daher, wenn sie aufgerufene Services zur Teilnahme an einer Transaktion auffordern wollen, den Coordination Context mitsenden. Sie gehören, wie in Abschnitt 3.3 auf Seite 53 beschrieben, zu den Scopelikes und tragen dementsprechend eine Internal Coordination Role (ICR). Die Semantik der ICR bezüglich einer `<invoke>`-Aktivität wird wie folgt belegt:

- Beim Ausführen einer `<invoke>`-Aktivität mit $ICR =$ `new` erzeugt die Mapping-Engine einen neuen Coordination Context für die Aktivität und fügt diesen an den Request des aufgerufenen Web Services an.
- Beim Ausführen einer `<invoke>`-Aktivität mit $ICR =$ `participating` fügt die Mapping-Engine den Coordination Context an, der vom Effective Scope (siehe Definition 3.3.9 auf Seite 56) generiert oder empfangen wurde.
- Beim Ausführen einer `<invoke>`-Aktivität mit $ICR =$ `independent` wird kein Coordination Context angefügt.

4.3 Korrelation der Coordination Partners

Für CScope mit ICR `new` und ECR `Initiator` generiert die Mapping-Engine zur Laufzeit einen Coordination Context (siehe Abschnitte 4.5 auf Seite 64 und 4.7 auf Seite 94). Dieser enthält, wie in Abschnitt 2.3 auf Seite 15 beschrieben, eine Endpunktreferenz des Registration Services. Für die Mapping-Engine wird in der Realisierung (siehe Abschnitt 5 auf Seite 109) ein Registration Service implementiert, über den die Registrierung aller Participants abläuft. Alle Participants verwenden für die Registrierung die identische Service Address. Um Participants unterscheiden zu können, werden die in WS-Addressing [W3C04] definierten Reference Properties verwendet. Der Endpunkt des Registration Service im Coordination Context erhält eine Reference Property, wie sie beispielhaft in Listing 4.1 dargestellt wird. `<ScopeReference>` enthält die ID der CScopeinstanz der `<invoke>`-Aktivität, die den Coordination Context in der Application Message versendet.

Ebenso wie bei der Registrierung müssen auch Coordination Messages von Participants (siehe Abschnitte 4.5 auf Seite 64 und 4.7 auf Seite 94) und Coordinators (siehe Abschnitte 4.6 auf Seite 80 und 4.8 auf Seite 103) zugeordnet werden können. Diese werden nicht einer CScopeinstanz, sondern einem Coordination Partner (Entität siehe Abschnitt 5.1) zugeordnet. Die

Zuordnung geschieht wiederum mithilfe einer Reference Property in der Endpunktreferenz des jeweiligen Protocol Services der Mapping-Engine. Handelt es sich bei dem Coordination Partner um einen Coordinatior, so wird bei der Register-Message in der Endpunktreferenz des Participant Protocol Service eine Reference Property `CoordinationPartner` eingefügt. Handelt es sich um einen Participant, so wird diese Reference Property in die Endpunktreferenz des Protocol Service in der RegisterResponse-Message eingefügt.

Die Reference Property hat dabei die Form `<x4b:CoordinationPartner id="x"/>`, wobei `x` die ID des Coordination Partners der Mapping-Engine enthält.

Beim Generieren des Coordination Contextes (in den Abschnitten 4.5 auf Seite 64 und 4.7 auf Seite 94) wird im Coordination Context als Identifier die ID der CScopeinstanz des Effective Scopes gesetzt (siehe Abschnitt 5.1 auf Seite 111).

4.4 Modifiziertes Eventmodell für Invoke-Aktivitäten

Im vorherigen Abschnitt wurde beschrieben, dass eine `<invoke>`-Aktivität einen Coordination Context in den Headers der Application Message mitsenden muss. Das Einfügen soll ebenfalls über Events realisiert werden. Im vorhandenen Eventmodell nach [Steo8] und [KKS+06] besteht keine Möglichkeit, einen Header oder Headerelemente in eine Application Message einzufügen. ODE ab Version 1.3 unterstützt zwar besondere Message Parts in Variablen, die von ODE beim Senden in den Header geschrieben werden, diese müssen aber im Binding bzw. in der Application Message in der WSDL-Datei definiert werden. Außerdem handelt es sich hierbei um eine ODE-spezifische Erweiterung [Apa10f]. Das Pluggable Framework ist als generisches Framework konzipiert worden, wobei ein Generic Controller für verschiedene Workflow-Engines implementiert werden kann. Somit ist die korrekte Funktionsweise für einen Custom Controller nicht garantiert. Insbesondere ODE 1.1.1 verfügt nicht über diese Erweiterung, die das Setzen und Auslesen von Headern unterstützt.

Aus diesem Grund wird an dieser Stelle eine Modifikation des Eventmodells für `<invoke>`-Aktivitäten vorgeschlagen. Abbilung 4.1 auf der nächsten Seite zeigt einen Auschnitt des von [Steo8] erstellten Eventmodells für `<invoke>`-Aktivitäten mit einer Modifikation. Zwischen den Zuständen *Ready* und *Executing* wird ein neuer Zustand *Message Prepared* eingefügt. Der Ablauf der Aktivität ist dabei folgender: Nachdem die Aktivität mit einem Incoming Event *Start_Activity* gestartet wurde, erstellt sie die Application Message, die zum Aufrufen des Web Services verschickt wird. Daraufhin wartet die Aktivität im Zustand *Message Prepared*. Das Outgoing Event *Message_Prepared* zeigt dem Custom Controller dabei das Erreichen des Zustandes an. Jetzt kann der Custom Controller Events an den Generic Controller senden, die das Auslesen und Modifizieren der Application Message anfordern.

Die nachfolgende Liste enthält Incoming Events, die der Generic Controller im Zustand *Message Prepared* empfängt.

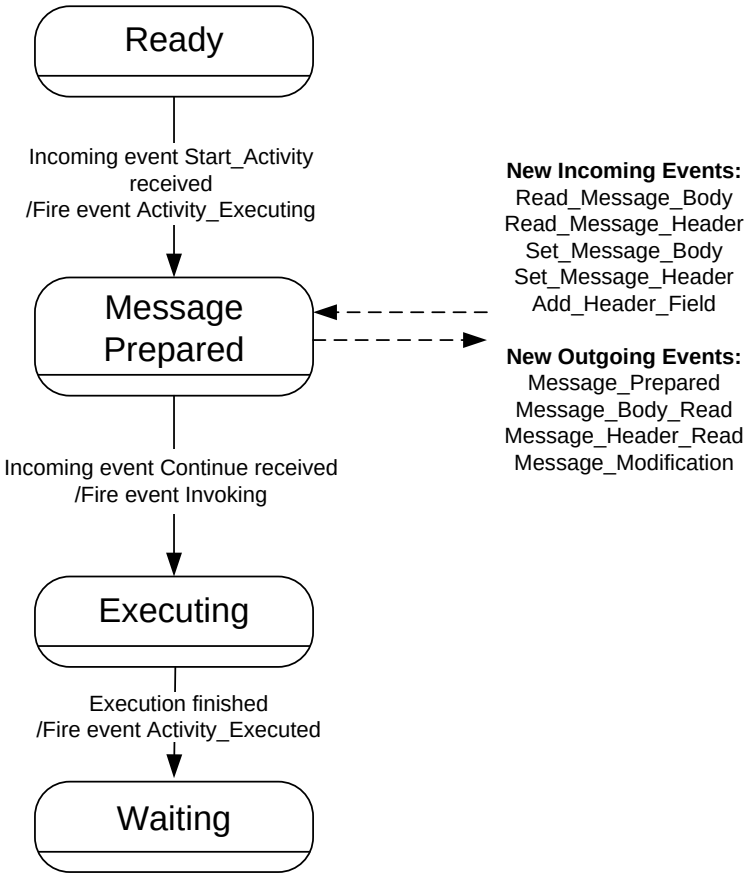

Abbildung 4.1: Ausschnitt mit Modifikationen für das Modell von `<invoke>`-Aktivitäten

- **Read_Message_Body:** Fordert ein *Message_Body_Read*-Event an, das den Inthalt der Application Message enthält.
- **Read_Message_Header:** Fordert ein *Message_Header_Read*-Event an, das den Header der Application Message enthält. Wenn nur die Headers für den Custom Controller notwendig sind, kann mit diesem Event das unnötige Übertragen von (großem) Payload vermieden werden.
- **Set_Message_Body:** Fordert den Generic Controller auf, den Inhalt der Application Message durch einen im Event enthaltenen Inhalt zu ersetzen.
- **Set_Message_Header:** Fordert den Generic Controller auf, den Header durch einen im Event enthaltenen Header zu ersetzen.
- **Add_Header_Field:** Fordert den Generic Controller auf, ein Headerfeld, das im Event enthaltenen ist, an den Header der Application Message anzuhängen.

- **Continue:** Fordert den Generic Controller auf, den Zustand *Message_Prepared* zu verlassen und mit dem Senden der Application Message fortzufahren.

Folgende Outgoing Events werden vom Generic Controller gesendet:

- **Message_Prepared:** Zeigt dem Custom Controller an, dass sich die Aktivität im Zustand *Message Prepared* befindet.
- **Message_Modification:** Informiert den Custom Controller über die erfolgreiche Modifikation, die durch *Set_Message_Body*, *Set_Message_Header* oder *Add_Header_Field* veranlasst wurde.
- **Message_Body_Read:** Übermittelt dem Custom Controller den Inhalt der Application Message als Antwort auf *Read_Message_Body*.
- **Message_Header_Read:** Übermittelt dem Custom Controller den Header der Application Message als Antwort auf *Read_Message_Header*.
- **Invoking:** Bestätigt dem Custom Controller den Erhalt des *Continue*-Events.

Wurden vom Custom Controller alle notwendigen Änderungen und Leseoperationen durchgeführt, weist er den Generic Controller mit einem *Continue*-Event an, das Aufrufen fortzusetzen. Der Generic Controller bestätigt das *Continue*-Event mit einem *Invoking*-Event.

Da es sich bei dem Zustand *Message Prepared* um einen Zustand handelt, der aus der *Executing*-Aktivität extrahiert wurde, gelten für ihn genau die gleichen Transitionen zu anderen Zuständen und die entsprechenden Events wie für den Zustand *Executing*. Folglich kann die `<invoke>`-Aktivität aus dem Zustand *Message Prepared* in die Zustände *Terminated* und *Faulted* übergehen.

4.5 Fall 1: Scope koordiniert aufgerufene Web Services mit WS-BA

Der Initiator einer Transaktion, die mittels eines WS-Coordination-Protokolls abläuft, hat den Vorteil, den für die Koordination verwendeten Coordinator selbst bestimmen zu können. Das bedeutet insbesondere auch, dass er selbst die Rolle des Coordinators übernehmen kann, sofern er die nötigen Schnittstellen besitzt. Wie bereits in Abschnitt 2.4.1 auf Seite 21 beschrieben, verfügt eine BPEL-Engine über ein internes Transaktionskonzept, das wie WS-BA auf langlaufenden Transaktionen basiert. Der Transaktionskontext eines Scopes soll nun per Mapping-Engine als Coordinator nach außen verfügbar gemacht werden, um aufgerufenen Web Services die Teilnahme am internen Scope zu ermöglichen. Der nachfolgende Ansatz beschreibt die Vorgehensweise für WS-BA mit dem Participant Completion Protocol. In Abschnitt 4.5.8 auf Seite 80 befindet sich eine Beschreibung der Modifikationen für Participants, die sich für die Koordination mit dem Coordinator Completion Protocol registrieren.

	Zustand	Outg. Event / Coord.Message	Bed.	Neuer Zustand	Inc. Events/ Coord. Messages
1	--	EVT:Activity_Ready		Executing	EVT:Start_Activity
2	Executing	WSBA:CannotComplete		Fault Handling	EVT:Fault_To_Scope, WSBA:NotCompleting
3	Executing	WSBA:Fail		Fault Handling	EVT:Fault_To_Scope, WSBA:Failed
4	Executing	WSBA:Exit		Fault Handling	EVT:Fault_To_Scope, WSBA:Exited
5	Executing	EVT:Activity_Executed	APC	Complete	EVT:Complete_Activity
6	Executing	EVT:Activity_Executed	NAPC	Waiting	
7	Executing	EVT:Scope_Handling_Fault		Fault Handling	WSBA:Cancel an Participants
8	Executing	EVT:Scope_Handling_Term.		Terminating	WSBA:Cancel an Participants
9	Waiting	WSBA:CannotComplete		Fault Handling	EVT:Fault_To_Scope, WSBA:NotCompleting
10	Waiting	WSBA:Fail		Fault Handling	EVT:Fault_To_Scope, WSBA:Failed
11	Waiting	WSBA:Exit		Fault Handling	EVT:Fault_To_Scope, WSBA:Exited
12	Waiting	WSBA:Completed	APC	Completed	EVT:Complete_Activity
13	Waiting	EVT:Scope_Handling_Fault		Fault Handling	WSBA:Cancel an Participants
14	Waiting	EVT:Scope_Handling_Term.		Terminating	EVT:Continue; WSBA:Cancel an Participants
15	Terminating	WSBA:Fail		Ended	EVT:Fault_To_Scope (Parent); WSBA:Failed
16	Terminating	WSBA:Canceled	APCan	Ended	EVT:Continue
17	Completed	EVT:Scope_Compensating		Compensating	EVT:Continue
18	Compensating	EVT:Scope_Compensated		Ended	
19	Fault Handling	WSBA:Canceled	APCan	Ended	EVT:Continue
20	Fault Handling	WSBA:Fail		Ended	EVT:Fault_To_Scope (Parent); WSBA:Failed

Legende: (N)APC = (Nicht) Alle Participants Completed; APCan = Alle Participants „Canceled“;
Outg./Inc. Event = Outgoing/Incoming Event; Coord. Message = Coordination Message

Tabelle 4.1: Transitionen des Automaten für <scope> und <process> mit ICR `new`

4.5.1 Prozesse und Scopes mit ICR `new`

Das Verhalten, das im Folgenden für Scopes beschrieben wird, gilt gleichermaßen für Prozesse. Für einen Scope mit den oben gegebenen Voraussetzungen wird beim Starten des Scopes (*Activity_Ready*) ein Automat erzeugt. Tabelle 4.1 zeigt die Transitionstabelle des Automaten. Er enthält die Zustände *Executing*, *Waiting*, *Fault Handling*, *Terminating*, *Compensating*, *Completed* und dem Endzustand *Ended*. Mit Ausnahme von *Ended* entstammen diese, von der intuitiven Umbenennung abgesehen, dem Eventmodell für Scopes. Sowohl Outgoing Events als auch Coordination Messages können Transitionen auslösen und führen den Automaten in einen neuen Zustand. Bei der Spalte „Bed.“ handelt es sich um eine Transitionsbedingung, die erfüllt sein muss, um die Transition durchführen zu dürfen. Der Automat hat Zugriff auf eine Liste, die seine Participants beinhaltet. In ihr werden Endpunktreferenz, Coordination Type, Coordination Protocol und der Status der Participants gespeichert. Erhält der Automat eine Registration Message oder Coordination Message, so wird der Zustand des Participants angelegt bzw. aktualisiert, auch wenn keine Transition erfolgt. Diese sind für

eine kompaktere und übersichtlichere Darstellung in der Transitionstabelle nicht enthalten. In der Zustandsübergangstabelle sind nur die Incoming Events, die an die BPEL-Engine gesendet werden, sowie die Coordination Messages, die an die Participants gesendet werden, enthalten. Zugunsten einer übersichtlicheren Tabelle werden weitere Aktionen ausschließlich in der nachfolgenden Liste aufgeführt.

Im Folgenden werden alle Transitionen detailliert erläutert. Die Nummerierung der Liste entspricht dabei der Nummerierung der Transitionen in der Transitionstabelle.

1. Sobald ein Scope vom Zustand *Inactive* in den Zustand *Ready* wechselt, wird mit dem Event *Activity_Ready* signalisiert, dass der Scope von der BPEL-Engine gestartet werden soll. Der Automat exisitiert für diesen Scope zu diesem Zeitpunkt noch nicht und muss generiert werden. Der generierte Automat befindet sich nun im Zustand *Executing*. Die Blockade des *Activity_Ready*-Events wird mittels *Start_Activity* aufgelöst und der Scope ausgeführt. Innerhalb von `<invoke>`-Aktivitäten mit ICR `participating`, die während des Ablaufs des Scopes ausgeführt werden, wird ein Coordination Context im Header der Application Messages mitgeschickt, der den Scope im Feld `Identifier` des Coordination Context referenziert. Die so eingebundenen Participants, rufen einen Registration Service auf, den die Mapping-Engine anbietet.
2. Sendet ein Participant eine CannotComplete-Message, wird von der Mapping-Engine ein *Fault_To_Scope*-Event in den Scope propagiert. Die Message des Participants wird durch NotCompleted bestätigt.
3. Sendet ein Participant eine Fail-Message, wird von der Mapping-Engine ein *Fault_To_Scope*-Event in den Scope propagiert. Die Message des Participants wird durch Failed bestätigt.
4. Sendet ein Participant eine Exit-Message, wird von der Mapping-Engine ein *Fault_To_Scope*-Event in den Scope propagiert. Die Message des Participants wird durch Exited bestätigt.
5. Ist die Ausführung des Scopes beendet und ist während seiner Ausführung kein Fault aufgetreten, sendet die BPEL-Engine das Event *Activity_Executed* an die Mapping-Engine. Haben bereits alle Participants Completed an die Mapping-Engine gesendet, wird mit einem *Complete_Activity* die Blockade im Zustand *Waiting* des Scopes in der BPEL-Engine aufgelöst und der Automat geht in den Zustand *Completed*.
6. Befindet sich im Gegesatz zum vorherigen Punkt noch Participants im Status *Active*, muss der Automat auf deren Completed im Zustand *Waiting* warten.
7. Befindet sich der Automat im Zustand *Executing* und signalisiert die BPEL-Engine der Mapping-Engine mit *Scope_Handling_Fault* einen Fehler, geht der Automat in den Zustand *Fault_Handling* über. Unteraktivitäten des Scopes werden durch den Termination-Handler terminiert oder kompensiert. Da `<invoke>`-Aktivitäten keinen Termination-Handler besitzen, müssen Participants, die sich noch im Zustand *Active* befinden, abgebrochen werden. Dazu wird ihnen eine Cancel-Message gesendet. Participants, die

sich bereits im Zustand *Completed* befinden, werden über das Compensation-Handling ihrer `<invoke>`-Aktivität kompensiert (siehe Abschnitt 4.5.2 auf der nächsten Seite). Die Blockade des Events *Scope_Handling_Fault* bleibt bestehen.

8. Befindet sich der Automat im Zustand *Executing* und sendet die BPEL-Engine der Mapping-Engine ein *Scope_Handling_Termination*-Event, geht der Automat in den Zustand *Terminating* über. Auch hier werden Participants im Zustand *Active* durch Cancel zum Abbruch aufgefordert. Da Apache ODE kein Termination-Handling unterstützt, und das Event *Scope_Handling_Compensation* dementsprechend auch nicht auftritt, wird statt dieser Transition für ODE eine andere Transition eingefügt, die im Zustand *Executing* das Event *Activity_Terminated* für den Scope abfängt. Diese führt den Automaten in den Zustand *Ended* über und sendet allen Participants, die sich noch im Zustand *Active* befinden, eine Cancel-Message.
9. Wie Transition 2, aber aus dem Zustand *Waiting*.
10. Wie Transition 3, aber aus dem Zustand *Waiting*.
11. Wie Transition 4, aber aus dem Zustand *Waiting*.
12. Befindet sich der Automat im Zustand *Waiting*, sendet ein Participant eine Completed-Message und haben daraufhin alle Participants den Status *Completed* erreicht, so kann die Blockade des Scopes im Zustand *Waiting* mittels *Complete_Activity* aufgehoben werden. Automat und Scope befinden sich jetzt im Zustand *Completed*.
13. Befindet sich der Automat im Zustand *Waiting* und empfängt die Mapping-Engine ein *Scope_Handling_Fault*-Event (ausgelöst durch ein vorheriges *Fault_To_Scope*-Event aus den Transitionen 10, 11 und 12), geht der Automat in den
14. Befindet sich der Automat im Zustand *Waiting* und empfängt die Mapping-Engine ein *Scope_Handling_Termination*-Event, geht der Automat in den Zustand *Terminating* über.
15. Befindet sich der Automat im Zustand *Terminating* und sendet ein Participant Fail, so wird ein Fault in den umgebenden Scope geworfen und der Automat geht in den Zustand *Ended* über.
16. Befindet sich der Automat im Zustand *Terminating* und sendet ein Participant Canceled, so wird die Blockade des Scopes mit *Continue* aufgelöst, sofern alle Participants erfolgreich abgebrochen haben.
17. Befindet sich der Automat im Zustand *Completed*, und signalisiert die BPEL-Engine durch *Scope_Compensating* eine Kompensierung des zugehörigen Scopes, geht der Automat in den Zustand *Compensating* über. Die Blockade in der BPEL-Engine wird sofort durch *Continue* aufgelöst. Das Kompensieren der Participants wird mit den Compensation-Handlers der `<invoke>`-Aktivitäten der Participants durchgeführt (siehe Abschnitt 4.5.2 auf der nächsten Seite).
18. Ist die Kompensierung abgeschlossen, sendet die BPEL-Engine ein *Scope_Compensated*-Event. Der Automat geht in den Zustand *Ended* über.

19. Befindet sich der Automat im Zustand *Fault Handling*, besteht im Scope in der BPEL-Engine noch die Blockade des *Scope_Handling_Fault*-Events. Es wurden Cancel-Messages an die Participants gesendet, die sich noch im Zustand *Active* befanden. Im Zustand *Fault Handling* werden deren Antworten nun ausgewertet. Wird von einem Participant eine Canceled-Message empfangen und haben alle anderen Participants ebenfalls eine Canceled-Message gesendet, wird die Blockade mit *Continue* aufgehoben. Je nach weiterem Verlauf des Fault-Handlings kann der Scope am Ende in die Zustände *CompleteWithFault* oder *Faulted* übergehen. Der Automat geht in den Zustand *Ended* über.
20. Befindet sich der Automat wie im vorherigen Punkt im Zustand *Fault Handling*, ein Participant meldet jedoch Fail, so wird wieder ein Fault mit *Fault_To_Scope* in den Scope geworfen. Dadurch endet der Scope auf jeden Fall im Zustand *Faulted*. Der Automat geht in den Zustand *Ended* über.

Zur Erinnerung: Die bestätigenden Coordination Messages Closed der Participants führen zu keinen Transitionen des Automaten, werden aber dennoch gespeichert. Diese können beispielsweise später dazu verwendet werden, ein Monitoring über das erfolgreiche Abschließen zu erstellen oder Informationen über den Status eines Participants zu geben.

Das Eventmodell nach [Steo8] und [KKS+06] definiert keinen Zustand für den Fall der fehlgeschlagenen Kompensation. Ein entsprechender Test mit der Implementierung führte zum Absturz der Prozessinstanz während des Auftretens eines Faults im Compensation-Handler. Es wird daher für diese Arbeit angenommen, dass der Generic Controller dennoch das Event *Scope_Compensated* sendet und der Automat so in den Zustand *Ended* übergeht. Auch im Termination-Handling sind keine verschiedenen Endzustände vorgesehen. Da Apache ODE kein Termination-Handling unterstützt, wird für die Realisierung in Kapitel 5 auf Seite 109 die in Punkt acht beschriebene Vorgehensweise gewählt.

4.5.2 Invoke-Aktivitäten mit ICR `participating`

Scopes und Prozesse mit ICR `new` erhalten erst durch die Verwendung von `<invoke>`-Aktivitäten mit ICR `participating` ihren Existenzgrund. `<invoke>`-Aktivitäten erzeugen die Participants, indem sie den Coordination Context als Headerelement mitsenden. Für ihre Koordinierung wird der Automat mit den Zuständen und Transitionen aus Tabelle 4.2 auf der nächsten Seite verwendet. Endzustände des Automaten sind *NotCompleted* und *Finished*. Da eine `<invoke>`-Aktivität nur einen Web Service aufruft, darf sie auch nur einen Participant besitzten. Um weitere Participants verwenden zu können, muss der Web Service eine eigene untergeordnete Business Activity an einem anderen Coordinator aktivieren (siehe Abschnitt 2.3.5 auf Seite 17). Der Zustand *Terminating* aus Tabelle 4.1 auf Seite 65 wird in den Zustand *Undo* umbenannt, da je nach Fortschritt des Participants entweder kompensiert oder abgebrochen werden muss. Der Automat enthält außerdem die Zustände `Pre Message` und

#	Zustand	Outg. Event / Coord.Message	Bed.	Neuer Zustand	Inc. Events/ Coord. Messages
1	--	EVT:Activity_Ready		Pre Message	EVT:Start_Activity
2	Pre Message	EVT:Message_Prepared		Message Prepared	EVT:Add_Message_Header (CoordContext)
3	Message Prepared	EVT:Activity_Terminated		NotCompleted	
4	Message Prepared	EVT:Scope_Handling_Fault		NotCompleted	
5	Message Prepared	EVT:Message_Modification		Executing	EVT:Continue
6	Executing	EVT:Activity_Terminated	NR	NotCompleted	
7	Executing	EVT:Scope_Handling_Fault	NR	NotCompleted	
8	Executing	EVT:Activity_Terminated	PA	NotCompleted	WSBA:Cancel
9	Executing	EVT:Activity_Terminated	PC	NotCompleted	WSBA:Compensate
10	Executing	EVT:Scope_Handling_Fault	PA	NotCompleted	WSBA:Cancel
11	Executing	EVT:Scope_Handling_Fault	PC	NotCompleted	WSBA:Compensate
12	Executing	EVT:Activity_Executed	PA	Completed	EVT:Complete_Activity
13	Executing	EVT:Activity_Executed	NR	Waiting	
14	Waiting	EVT:Activity_Terminated	NR	NotCompleted	
15	Waiting	EVT:Scope_Handling_Fault	NR	NotCompleted	
16	Waiting	EVT:Activity_Terminated	PA	NotCompleted	WSBA:Cancel
17	Waiting	EVT:Activity_Terminated	PC	NotCompleted	WSBA:Compensate
18	Waiting	EVT:Scope_Handling_Fault	PA	NotCompleted	WSBA:Cancel
19	Waiting	EVT:Scope_Handling_Fault	PC	NotCompleted	WSBA:Compensate
20	Waiting	WSC:Register		Completed	WSC:Register_Response; EVT:Complete_Activity
21	Completed	EVT:Scope_Compensating	PA	Undo	WSBA:Cancel
22	Completed	EVT:Scope_Compensating	PC	Undo	WSBA:Compensate
23	Completed	EVT:Scope_Compensating	PF	NotCompleted	EVT:Continue
24	Undo	WSBA:Compensated		Compensated	EVT:Continue
25	Undo	WSBA:Fail		NotCompleted	Fault_To_Scope (parent)
26	Completed	EVT:Instance_Complete		Finished	WSBA:Close

Legende: NR = Participant nicht registriert. PA = Participant Active; PC = Participant Completed; PF = Participant Failed
Outg./Inc. Event = Outgoing/Incoming Event; Coord. Message = Coordination Message

Tabelle 4.2: Transitionen des Automaten für <invoke> ICR `participating`

`Message Prepared`. Diese werden benötigt, um den Coordination Context in den Header der Application Message für den Aufruf des Web Services einzufügen (siehe 4.4 auf Seite 62). Die nachfolgende Liste erläutert die einzelnen Transitionen des Automaten. Dabei entsprechen die Nummern vor den Listenpunkten den Zeilen in der Transitionstabelle 4.2. Es wird dabei die Tatsache genutzt, dass `<invoke>`-Aktivitäten einen Compensation-Handler besitzen.

1. Die `<invoke>`-Aktivität wird gestartet. Der Automat wird erzeugt und befindet sich im Zustand *Pre Message*. Die Aktivität wird mit *Start_Activity* gestartet.
2. Erhält die Mapping-Engine das *Message_Prepared*-Event, kann die Application Message modifiziert werden. Es wird mittels *Add_Header_Field* ein Coordination Context in

den Header der Application Message eingefügt. Der Coordination Context enthält als Identifier die ID des Automaten, um Messages des Participants dem Automaten zuordnen zu können.

3. Sendet die BPEL-Engine das Event *Activity_Terminated*, wurde die Aktivität abgebrochen. Der Automat geht in den Zustand *NotCompleted*.
4. Sendet die BPEL-Engine das Event *Scope_Handling_Fault*, ist in der Aktivität ein Fault aufgetreten. Der Automat geht in den Zustand *NotCompleted*.
5. Sobald die BPEL-Engine mit einem *Message_Modification* das Anfügen des Headerelements bestätigt, wird die `<invoke>`-Aktivität mit *Continue* zum Verlassen des Status *Message Prepared* aufgefordert. Die Aktivität führt ihren Web-Service-Aufruf durch. Der Participant registriert sich. Die Registrierung kann nur erfolgen, solange sich der Automat im Zustand *Executing* befindet. Eine Registrierung in den anderen Zuständen (*Fault Handling*, *Terminating*) wird abgelehnt. Der Participant hat bei Ablehnung der Registrierung seine Operation abzubrechen.
6. Sendet die BPEL-Engine das Event *Activity_Terminated*, wurde die Aktivität abgebrochen. Hat sich der Participant zu diesem Zeitpunkt noch nicht registriert, geht der Automat in den Zustand *NotCompleted* über.
7. Sendet die BPEL-Engine das Event *Scope_Handling_Fault*, ist in der Aktivität ein Fault aufgetreten. Hat sich der Participant zu diesem Zeitpunkt noch nicht registriert, geht der Automat in den Zustand *NotCompleted* über.
8. Sendet die BPEL-Engine das Event *Activity_Terminated*, wurde die Aktivität abgebrochen. Befindet sich der Participant im Zustand *Active*, wird ihm eine Cancel-Message gesendet. Der Automat geht in den Zustand *Undo* über.
9. Sendet die BPEL-Engine das Event *Activity_Terminated*, wurde die Aktivität abgebrochen. Befindet sich der Participant im Zustand *Completed*, wird ihm eine Compensate-Message gesendet. Der Automat geht in den Zustand *Undo* über.
10. Sendet die BPEL-Engine das Event *Scope_Handling_Fault*, ist in der Aktivität ein Fault aufgetreten. Befindet sich der Participant im Zustand *Active*, wird ihm eine Cancel-Message gesendet. Der Automat geht in den Zustand *NotCompleted* über.
11. Sendet die BPEL-Engine das Event *Scope_Handling_Fault*, ist in der Aktivität ein Fault aufgetreten. Befindet sich der Participant im Zustand *Completed*, wird ihm eine Compensate-Message gesendet. Der Automat geht in den Zustand *NotCompleted* über.
12. Hat die `<invoke>`-Aktivität ihren Aufruf erfolgreich durchgeführt, sendet die BPEL-Engine das Event *Activity_Executed*. Hat sich zu diesem Zeitpunkt der Participant bereits registriert, kann die Aktivität mit einem *Complete_Activity*-Event abgeschlossen werden.
13. Hat sich der Participant im Gegensatz zum vorherigen Punkt noch nicht registriert, muss die Aktivität im Zustand *Waiting* so lange warten, bis er sich registriert hat.

14. Wie Transition 6
15. Wie Transition 7
16. Wie Transition 8
17. Wie Transition 9
18. Wie Transition 10
19. Wie Transition 11
20. Ist die Registrierung erfolgt, wird das *Activity_Executed*-Event an die BPEL-Engine gesendet.
21. Befindet sich die `<invoke>`-Aktivität respektive der Automat im Zustand *Completed* und die BPEL-Engine startet das Kompensieren der `<invoke>`-Aktivität, so kann sich der Participant in den Status *Active*, *Completed* oder *Fail* befinden. Sendet die BPEL-Engine das *Scope_Compensating*-Event, muss dementsprechend der Participant entweder abgebrochen oder kompensiert werden. Wenn er sich im Zustand *Active* befindet, wird ihm eine Cancel-Message gesendet. Die Blockade des *Scope_Compensating*-Event bleibt bestehen. Der Automat wechselt in den Zustand *Undo*
22. Ist der Participant bereits im Zustand *Completed*, wird ihm eine Compensate-Message gesendet. Die Blockade bleibt ebenfalls bestehen. Der Automat wechselt in den Zustand *Undo*.
23. Befindet sich der Participant im Zustand *Failed*, so hat er bereits vorher durch die Fail-Message im überliegenden Scope das Fault Handling ausgelöst. Die Blockade wird mit *Continue* aufgelöst und der Automat wechselt in den Zustand *Ended*.
24. Befindet sich der Automat im Zustand *Undo*, ist in der BPEL-Engine immer noch die Blockade des *Scope_Compensating*-Events gesetzt. Diese wird aufgelöst, sobald der Participant mit der Message Compensated oder Canceled das Kompensieren bzw. Abbrechen abgeschlossen hat. Der Automat wechselt für den ersten Fall in den Zustand *Compensated*.
25. Sendet der Participant stattdessen eine Fail-Message, ist das Kompensieren des Participants fehlgeschlagen. Anstelle des Auflösens der Blockade wird mittels *Fault_To_Scope* ein Fault in den überliegenden Scope gesendet, der das Kompensieren veranlasst hat.
26. Wird die Prozessinstanz der `<invoke>`-Aktivität, deren Ablauf beschrieben wurde, abgeschlossen, sendet die BPEL-Engine der Mapping-Engine das Event *Instance_Completed*. Erst jetzt darf dem Participant vom Automaten die Close-Message geschickt werden.

4.5.3 Invoke-Aktivitäten mit ICR `new`

`<invoke>`-Aktivitäten verfügen über einen Compensation- und Fault-Handler [OAS07]. Im BPEL-Modell ist der Compensation-Handler der `<invoke>`-Aktivität vorgesehen, um Aufrufe

	Zustand	Outgoing Event / Coord.Message	Bed.	Neuer Zustand	Incoming Events/ Coord. Messages
1	--	EVT:Activity_Ready		Pre Message	EVT:Start_Activity
2	Pre Message	EVT:Message_Prepared		Message Prepared	EVT:Add_Message_Header (CoordContext)
3	Message Prepared	EVT:Activity_Terminated		NotCompleted	
4	Message Prepared	EVT:Scope_Handling_Fault		NotCompleted	
5	Message Prepared	EVT:Message_Modification		Executing	EVT:Continue
6	Executing	WSBA:CannotComplete		NotCompleted	EVT:Fault_To_Scope, WSBA:NotCompleting
7	Executing	WSBA:Fail		NotCompleted	EVT:Fault_To_Scope, WSBA:Failed
8	Executing	WSBA:Exit		NotCompleted	EVT:Fault_To_Scope, WSBA:Exited
9	Executing	EVT:Scope_Handling_Fault	NR	NotCompleted	
10	Executing	EVT:Scope_Handling_Fault	PA	NotCompleted	WSBA:Cancel
11	Executing	EVT:Scope_Handling_Fault	PC	NotCompleted	WSBA:Compensate
12	Executing	EVT:Activity_Terminated	NR	NotCompleted	
13	Executing	EVT:Activity_Terminated	PA	NotCompleted	WSBA:Cancel
14	Executing	EVT:Activity_Terminated	PC	NotCompleted	WSBA:Compensate
15	Executing	EVT:Activity_Executed	PC	Completed	EVT:Complete_Activity
16	Executing	EVT:Activity_Executed	PA/NR	Waiting	
17	Waiting	WSBA:CannotComplete		NotCompleted	EVT:Fault_To_Scope, WSBA:NotCompleting
18	Waiting	WSBA:Fail		NotCompleted	EVT:Fault_To_Scope, WSBA:Failed
19	Waiting	WSBA:Exit		NotCompleted	EVT:Fault_To_Scope, WSBA:Exited
20	Waiting	EVT:Scope_Handling_Fault	NR	NotCompleted	
21	Waiting	EVT:Scope_Handling_Fault	PA	NotCompleted	WSBA:Cancel
22	Waiting	EVT:Activity_Terminated	NR	NotCompleted	
23	Waiting	EVT:Activity_Terminated	PA	NotCompleted	WSBA:Cancel
24	Waiting	WSBA:Completed		Completed	EVT:Complete_Activity
25	Completed	EVT:Scope_Compensating		Compensating	WSBA:Compensate
26	Compensating	WSBA:Compensated		Compensated	EVT:Continue
27	Compensating	WSBA:Fail		NotCompleted	EVT:Fault_To_Scope(parent), WSBA:Failed
28	Complete	EVT:Instance_Complete		Finished	WSBA:Close an alle Participants

Legende: PA = Participant Active; NR = Participant nicht Registered; PC = Participants Completed; Coord. Message = Coordination Message

Tabelle 4.3: Transisitonen für <invoke> ICR `new`

von anderen Web Services zu kompensieren. Der implizite Compensation-Handler einer <invoke>-Aktivität entspricht einer <empty>-Aktivität. Daher muss er ohne die Verwendung von WS-BA explizit definiert werden, andernfalls ist für <invoke>-Aktivitäten kein Kompensieren per Compensation-Handler möglich. Durch die Verwendung von WS-BA wird dieser Schritt überflüssig. Daher dürfen <invoke>-Aktivitäten keinen expliziten Compensation-Handler besitzen. Semantisch sind invoke-Aktivitäten mit ICR new also äquivalent zu einem Scope mit ICR new, der ausschließlich eine <invoke>-Aktivität mit ICR participating enthält. Daher ist auch nur ein Participant pro *<invoke>*-Aktivität erlaubt. Wünscht der Participant seinerseits wiederum eine Verwaltung weiterer Participants, muss dieser eine untergeordnete Business Activity an einem Coordinator seiner Wahl aktivieren (vgl. Abschnitt 2.3.5 auf Seite 17).

Für eine <invoke>-Aktivität mit ICR new dürfen weder explizite Fault-Handler noch ein expliziter Compensation-Handler modelliert werden. Diese Funktionalität wird von der Mapping-Engine übernommen. Prozessmodelle, die dies nicht beachten, werden beim Deployment von der Mapping-Engine abgelehnt. Aufgrund der semantischen Äquivalenz der invoke-Aktivitäten mit ICR new zu einem Scope mit ICR new, der ausschließlich eine <invoke>-Aktivität mit ICR participating enthält, ergibt sich der Automat für <invoke>-Aktivitäten mit ICR new aus den anderen beiden. Werden alle Transitionen aus den Tabelle 4.1 auf Seite 65 und 4.2 auf Seite 69 vereint[1] und alle Transitionen, die nie auftreten können, aus der Menge der Transitionen entfernt, entsteht ein Automat, der durch die Zustandstabelle 4.3 auf der vorherigen Seite repräsentiert wird. Die folgende Liste beschreibt wieder alle Transitionen detailliert, wobei die Nummern der Punkte der Liste den Zeilennummern der Transitionen aus Tabelle 4.3 auf der vorherigen Seite entsprechen.

1. Die <invoke>-Aktivität wird gestartet. Der Automat wird erzeugt und befindet sich in Zustand *Pre Message*. Die Aktivität wird mit *Start_Activity* gestartet.
2. Erhält die Mapping-Engine das *Message_Prepared*-Event, kann die Application Message modifiziert werden. Es wird mittels *Add_Header_Element* ein Coordination Context in den Header der Application Message eingefügt. Der Coordination Context enthält als Identifier die ID des Automaten, um Coordination Messages des Participants dem Automaten zuordnen zu können.
3. Sendet die BPEL-Engine das Event *Activity_Terminated*, wurde die Aktivität abgebrochen. Der Automat geht in den Zustand *NotCompleted*.
4. Sendet die BPEL-Engine das Event *Scope_Handling_Fault*, ist in der Aktivität ein Fault aufgetreten. Der Automat geht in den Zustand *NotCompleted*.
5. Sobald die BPEL-Engine mit einem *Message_Modification* das Anfügen des Headerelements bestätigt, wird die <invoke>-Aktivität mit *Continue* zum Verlassen des Status *Message Prepared* aufgefordert. Die Aktivität führt ihren Web-Service-Aufruf durch. Der

[1] Anstelle des Statusnamens „Undo“ muss „Compensating“ verwendet werden.

Participant registriert sich. Findet eine Registrierung nicht statt, werden die nachfolgenden Transitionen genauso durchgeführt. Die Coordination Messages werden in diesem Fall nicht gesendet. Die Registrierung kann nur erfolgen, solange sich der Automat im Zustand *Executing* befindet. Eine Registrierung in den anderen Zuständen (*Fault Handling*, *Terminating*) wird abgelehnt. Der Participant hat bei dieser Verweigerung seine Operation abzubrechen.

6. Sendet der Participant eine CannotComplete-Message, wird von der Mapping-Engine ein *Fault_To_Scope*-Event in den Scope propagiert. Die Message des Participants wird durch NotCompleted bestätigt.
7. Sendet der Participant eine Fail-Message, wird von der Mapping-Engine ein *Fault_To_Scope*-Event in den Scope propagiert. Die Message des Participants wird durch Failed bestätigt.
8. Sendet der Participant eine Exit-Message, wird von der Mapping-Engine ein *Fault_To_Scope*-Event in den Scope propagiert. Die Message des Participants wird durch Exited bestätigt.
9. Sendet die BPEL-Engine das Event *Scope_Handling_Fault*, ist in der Aktivität ein Fault aufgetreten. Hat sich der Participant zu diesem Zeitpunkt noch nicht registriert, geht der Automat in den Zustand *NotCompleted* über.
10. Sendet die BPEL-Engine das Event *Scope_Handling_Fault*, ist in der Aktivität ein Fault aufgetreten. Befindet sich der Participant im Zustand *Active*, wird ihm eine Cancel-Message gesendet. Der Automat geht in den Zustand *NotCompleted* über.
11. Sendet die BPEL-Engine das Event *Scope_Handling_Fault*, ist in der Aktivität ein Fault aufgetreten. Befindet sich der Participant im Zustand *Completed*, wird ihm eine Compensate-Message gesendet. Der Automat geht in den Zustand *NotCompleted* über.
12. Sendet die BPEL-Engine das Event *Activity_Terminated*, wurde die Aktivität abgebrochen. Hat sich der Participant zu diesem Zeitpunkt noch nicht registriert, geht Automat geht in den Zustand *NotCompleted* über.
13. Sendet die BPEL-Engine das Event *Activity_Terminated*, wurde die Aktivität abgebrochen. Befindet sich der Participant im Zustand *Active*, wird ihm eine Cancel-Message gesendet. Der Automat geht in den Zustand *Undo* über.
14. Sendet die BPEL-Engine das Event *Activity_Terminated*, wurde die Aktivität abgebrochen. Befindet sich der Participant im Zustand *Completed*, wird ihm eine Compensate-Message gesendet. Der Automat geht in den Zustand *Undo* über.
15. Wurde die `<invoke>`-Aktivität erfolgreich ausgeführt und hat der Participant bereits *Completed* gesendet – das bedeutet insbesondere auch, dass er sich vorher registriert hat – geht der Automat in den Zustand *Completed* über. Die Blockade der Aktivität im Zustand *Waiting* wird durch *Complete_Activity* aufgelöst.

16. Wurde die `<invoke>`-Aktivität erfolgreich ausgeführt und hat sich der Participant noch nicht registriert oder befindet sich noch im Zustand *Active*, muss die `<invoke>`-Aktivität im Zustand *Waiting* warten. Auch der Automat befindet sich nun im Zustand *Waiting*.
17. Wie Transition 6
18. Wie Transition 7
19. Wie Transition 8
20. Wie Transition 9
21. Wie Transition 10
22. Wie Transition 12
23. Wie Transition 13
24. Befindet sich der Automat und damit auch der Scope im Zustand *Waiting* und erhält die Mapping-Engine die Message Completed vom Participant, kann die `<invoke>`-Aktivität beendet werden. Die Mapping-Engine sendet das Event *Complete_Activity* an die BPEL-Engine.
25. Solange die die `<invoke>`-Aktivität umgebende Prozessinstanz noch nicht beendet ist, kann die BPEL-Engine *Scope_Compensating* an die Mapping-Engine senden. Die Mapping-Engine leitet dieses Event in Form einer Compensate-Message an den Participant weiter.
26. Befindet sich der Automat im Zustand *Compensating* und dementsprechend die `<invoke>`-Aktivität im Zustand *Compensation Executing*, wartet der Participant auf die Bestätigung der Compensate-Message. Sendet der Participant Compensated, wird die Blockade der Aktivität in diesem Zustand mit *Continue* aufgehoben. Der Automat geht in den Zustand *Compensated* über.
27. Befindet sich der Automat im Zustand *Compensating* und der Participant sendet Fail, muss mit *Fault_To_Scope* ein Fault in den umgebenden Scope propagiert werden. Der Automat geht in den Zustand *NotCompleted* über.
28. Mit dem Event *Instance_Completed* signalisiert die BPEL-Engine der Mapping-Engine, dass die Prozessinstanz beendet wurde. Ab diesem Punkt kann die `<invoke>`-Aktivität nicht mehr kompensiert werden. Die Mapping-Engine sendet dem Participant eine Close-Message.

4.5.4 Scope-Aktivitäten mit ICR `participating`

Scopes in der ICR `participating` werden nicht durch die Mapping-Engine, sondern vollständig von der BPEL-Engine koordiniert.

Scopes mit ICR `participating` sind möglich, sollten aber sehr bedacht verwendet werden. Durch ihre Verwendung überlappen sich Transaktionsbereiche von Scopes und Business

Activities. Dadurch werden sie für den Entwickler des Prozessmodells schwer überschaubar und fehlerträchtig. Daher empfiehlt es sich für Scopes ausschließlich die ICR `new` und `independent` zu definieren.

Prozesse mit ICR `participating` exisiteren wie in Abschnitt 3.3 auf Seite 53 festgelegt nicht.

4.5.5 Coordination Types

WS-BA beschreibt die Coordination-Types *MixedOutcome* und *Atomic-Outcome*. Bei Atomic-Outcome hat der Coordinator dafür zu sorgen, dass entweder alle Participants nach Close oder alle nach Compensate gelangen. Bei MixedOutcome dürfen Participants in einem beliebigen der zwei Status enden [OAS09b].

Der Prozessmodellierer kann dem Scope über den in Abschnitt 3.3 auf Seite 53 beschriebenen CoordinationType MixedOutcome oder AtomicOutcome zuordnen. AtomicOutcome entspricht einem Sonderfall von MixedOutcome. Daher darf MixedOutcome immer als CT eingesetzt werden. MixedOutcome kann aber beim aufgerufenen Web Service zum Ablehnen des Requests führen, wenn er keinen Coordination Context mit MixedOutcome akzeptiert. Welches Verhalten tatsächlich implementiert wird, hängt nicht von der Definition im Scope, sondern von den FCT-Handlern[2] ab. Werden die impliziten FCT-Handler verwendet, wird AtomicOutcome realisiert, da das implizite Handling so definiert ist, dass jede erfolgreich ausgeführte Aktivität in umgekehrter Reihenfolge ihrer Ausführung kompensiert wird (siehe auch Kapitel 2 auf Seite 13). Das explizite Definieren von bestimmten Handlern, die beispielsweise auf Compensation verzichten, kann zu *MixedOutcome* führen.

4.5.6 Annahmen und Einschränkungen

In der vorgestellten Vorgehensweise zur Koordination mit WS-BA wird angenommen, dass alle aufgerufenen Participants innerhalb der Laufzeit des koordinierten Scopes (*Executing* und *Waiting*) ihren Beitrag zur Business Activity als Complete melden. Es muss folglich das Message Exchange Pattern (MEP, [NLL08]) innerhalb des Scopes abgeschlossen werden, wenn der Participant erst nach der letzten Application Message des MEP sein Completed sendet.

Diese Eigenschaft gilt auch für die `<invoke>`-Aktivität mit ICR `new`, für die ein eigener Coordination Context erzeugt wird. Dadurch sind für sie nur synchrone Aufrufe (Request/Response) möglich.

[2] Fault-, Compensation-, und Termination-Handler

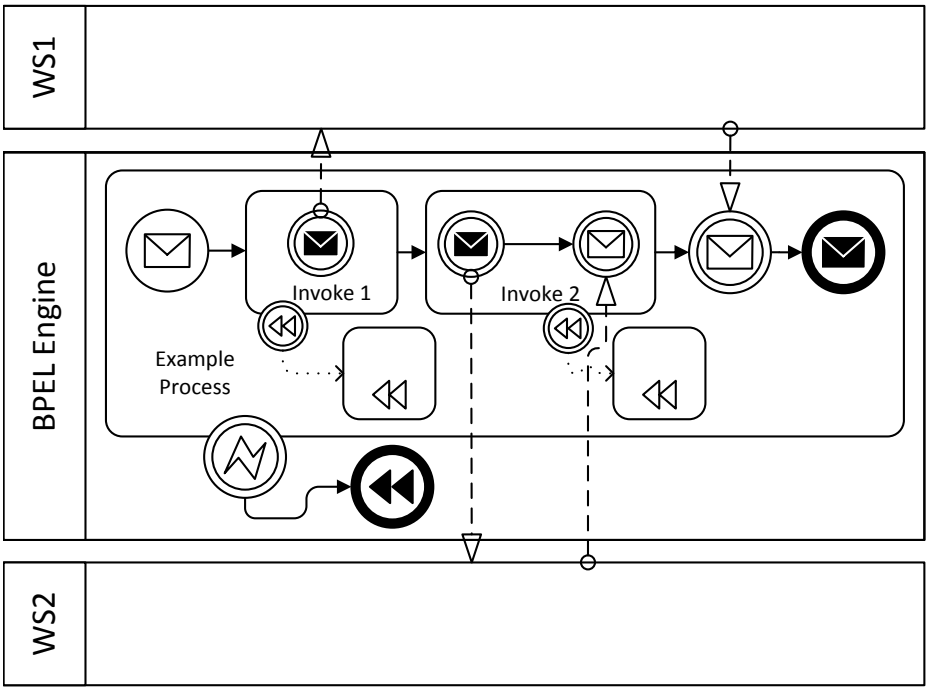

Abbildung 4.2: Prozess, der zwei Web Services aufruft (BPMN)

Für `<invoke>`-Aktivitäten darf kein expliziter Compensation-Handler definiert werden (siehe Abschnitt 4.5.3 auf Seite 71), wenn diese die ICR `participating` oder `new` besitzen. Prozessmodelle, die für solche `<invoke>`-Aktivitäten dennoch explizit Kompensation definieren, werden von der Mapping-Engine beim Deployment abgelehnt.

4.5.7 Beispiel

Abbildung 4.2 zeigt einen Prozess. Der Prozess besteht aus einer Sequenz mit drei Aktivitäten. Es werden zwei Web Services aufgerufen. Web Service 1 wird asynchron aufgerufen. Der Aufruf von Web Service 2 folgt synchron mit einer `<invoke>`-Aktivität, die dem Request-Response-Schema folgt. Die Antwort von Web Service 1 erhält der Scope in der `<receive>`-Aktivität. Für die Darstellung des Prozesses wurde BPMN 1.2 gewählt, wobei die impliziten Compensation-Handler wie in [WDGW08] beschrieben dargestellt werden. Explizite Fault- und Compensation-Handler werden im Prozessmodell nicht definiert. Das Sequenzdiagramm aus Abbildung 4.3 auf der nächsten Seite stellt den Austausch von Coordination Messages, Application Messages und Events während einer Prozessinstanz des Modells dar, wie er im Fall einer erfolgreichen Business Activity (hier AtomicOutcome, alle Participants Completed) auftritt. Es kann nachvollzogen werden, wie der Scope in der BPEL-Engine die einzelnen Status durchläuft.

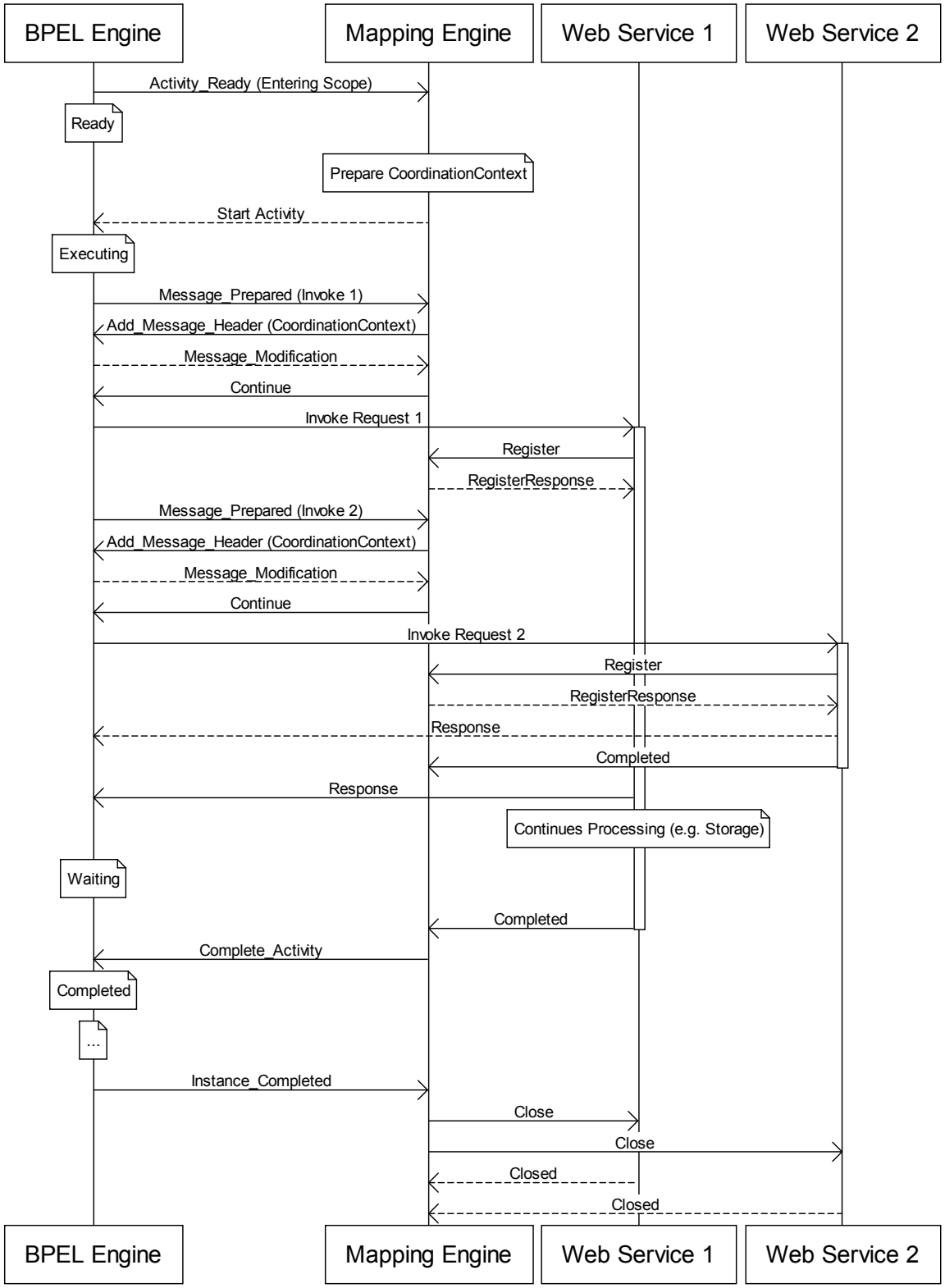

Abbildung 4.3: Mögliches Sequenzdiagramm: Ablauf des Beispiels, Transaktion erfolgreich

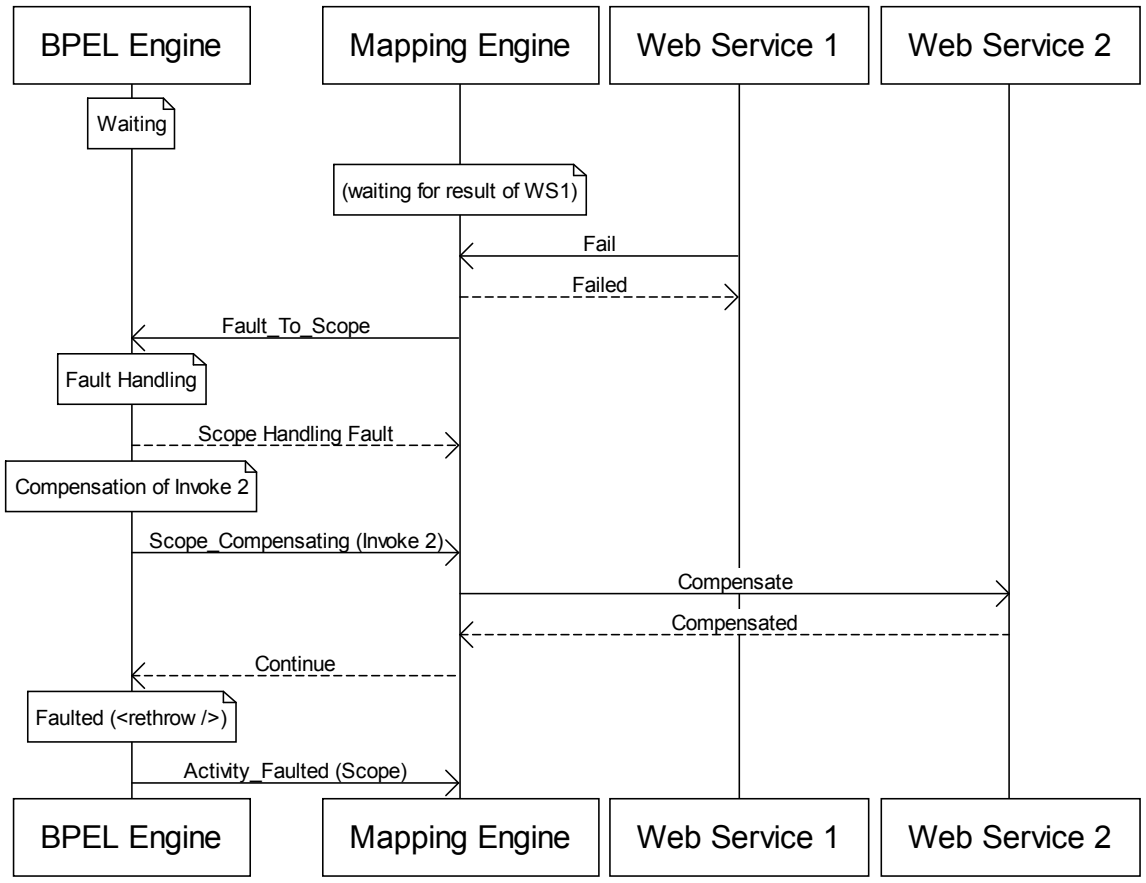

Abbildung 4.4: Ausschnitt im Fehlerfall des Beispiels, Web Service 1 sendet Fail

Das Sequenzdiagramm aus Abbildung 4.4 beschreibt dasselbe Beispiel für den Fall, dass Participant Web Service 1 Fail anstelle von Completed sendet. Bis zum Zustand *Waiting* des BPEL-Prozesses entspricht der Ablauf dem des fehlerfreien Ablaufs. Die Abbildung stellt den alternativen Verlauf ab dieser Stelle dar. Der Fehler im Participant wird von der Mapping-Engine in den Prozess (als Scope) propagiert. Der implizite Fault-Handler des Prozesses übernimmt die Fehlerbehandlung und veranlasst das Kompensieren von Invoke 2. Die Mapping-Engine fängt das Event ab und propagiert eine Compensate-Message an Participant Web Service 2. Ist das Kompensieren abgeschlossen, wirft der Fault-Handler den Fault per `<rethrow>` nochmals und der Prozess bricht mit dem Fault ab.

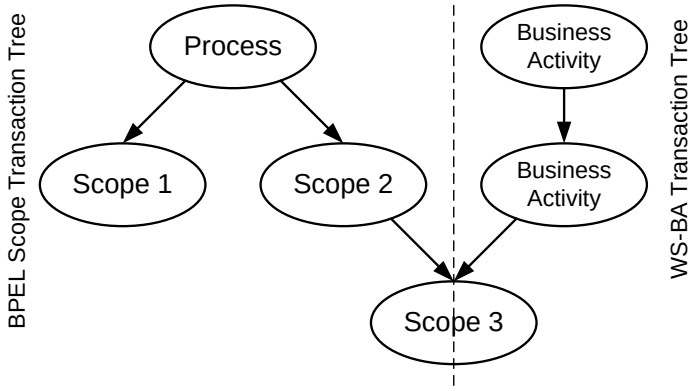

Abbildung 4.5: Zwei überlappende Transaktionsbäume von BPEL und WS-BA

4.5.8 Modifikation für Coordinator Completion

Der oben beschriebene Ansatz zeigt, wie Participants über das *BusinessAgreementWithParticipantCompletion Protocol* (BAPC) koordiniert werden. Als zweites Protokoll beschreibt WS-BA das *BusinessAgreementWithCoordinatorCompletion Protocol* (BACC) [OASo9b], bei dem der Coordinator – in diesem Fall der BPEL-Scope – durch die Mapping-Engine entscheidet, wann ein Participant alle notwendigen Aufrufe erhalten hat.

Participants, die sich mit diesem Protokoll an der Mapping-Engine anmelden, erhalten beim Zustandswechsel des Scopealikes in den Zustand *Waiting* eine Complete-Message. Der Scopealike verbleibt wiederum in diesem Status, bis der Participant sein Completed, Exit, CannotComplete oder Fail zurücksendet. Die weitere Behandlung erfolgt wie bei der Koordinierung mit dem BAPC-Protokoll.

4.6 Fall 2: Scope als WS-Business Activity Participant

Im letzten Abschnitt wurde gezeigt, wie aufgerufene Web Services als Participants einer Business Activity in das transaktionale Konzept des Scopes eingebunden werden können. Ein BPEL-Scope kann ebenso selbst infiziert werden (ECR `Participant`), wenn eine Aktivität eine Application Message empfängt, die einen Coordination Context enthält. Im Abschnitt 3.3 auf Seite 53 wurde dabei ein boolescher Wert in den SCP eines Scopealikes eingeführt, der bestimmt, ob der Scopealike zwingend koordiniert werden muss. Die Schwierigkeit hierbei besteht darin, die beiden Konzepte der externen Business Activity sowie dem internen Scope zusammenzubringen. Die Besonderheit der Teilnahme eines Scopes an einer Business Activity besteht darin, dass der Scope zum Kind zweier Transaktionsbäume wird [KML09]. In Abbildung 4.5 werden zwei mögliche Transaktionsbäume dargestellt. Scope 3 wird dabei

durch das Empfangen eines Coordination Contexts zum Kind beider Bäume. Es entsteht eine Situation verteilter Koordination, wofür BPEL und WS-BA nicht ausgelegt sind [SM05]. Ziel muss sein, die beiden Transaktionsbäume zu einem zusammenzusetzen. Dies ist auf Ebene eines aufgerufenen Scopes nur noch möglich, wenn es sich bei dem aufgerufenen Scope um eine Prozessinstanz handelt. Der Transaktionsbaum muss – sofern wegen verschiedener Zuständigkeitsbereiche machbar – bereits beim Design des gesamten Systems, also auf Ebene der Wurzel, geplant werden.

Da innerhalb eines Scopes bereits Aktivitäten vor dem Empfang des Coordination Context ausgeführt werden können, wird eine Definition benötigt.

Definition 4.6.1 (Infection Receiving Activity)
Die Infection Receiving Activity (IRA) eines laufenden Scopes mit ICR `new` *und ECR* `Participant` *sei die Aktivität innerhalb des Scopealike aber außerhalb der Unterscopealikes, die zuerst eine Application Message empfängt. Dabei wird angenommen, dass die Reihenfolge der Events, die in die Queue eingestellt werden, der Reihenfolge des Empfangs der Application Messages entspricht.*

Wird nach der Infection Receiving Activity innerhalb des Scopealike, aber außerhalb der Unterscopealikes ein weiterer Coordination Context empfangen, wird ein Fault in den Scope geworfen, sofern es sich nicht um den identischen Coordination Context handelt. Scopes, die von außen koordiniert werden sollen, müssen eine Infection Receiving Activity besitzen. Wenn eine Infection Receiving Activity in der eingehenden Application Message keinen Coordination Context erhält, wird ein Fault in den umschließenden Scope geworfen. Im Prozessmodell können durch Parallelität mehrere Aktivitäten als Infection Receiving Activity in Frage kommen. Sie müssen nicht als solche markiert werden, sondern werden durch das Auftreten als erste empfangende Aktivität durch die Mapping-Engine als solche erkannt.

Ein Sonderfall, bei dem ein Scopealike von außen koordiniert werden kann, ist wie oben erläutert gegeben, wenn es sich bei dem Scopealike um einen Prozess handelt. Eine Prozessinstanz wird bei Aufruf des Prozesses als Web Service erzeugt und endet nach einer `<reply>`-Aktivität. In diesem Fall entspricht der Scopealike der Wurzel des BPEL-Transaktionsbaumes und liegt folglich vollständig innerhalb des WS-BA-Transaktionsbaumes. Für `<scope>`-Aktivitäten muss hingegen das Coordinator Completion Protocol modifiziert werden, um als Participant teilnehmen zu können (siehe Abschnitt 4.6.2 auf Seite 85). Daher wird im Folgenden zwischen Prozessen und Scopes als Participant unterschieden.

In dieser Arbeit werden nur die Protokolle Participant Completion Protocol aus WS-Coordination [OAS09c] und das Participant Completion Protocol mit der Modifikation Participant Triggered Compensation für die Koordinierung von CScopes erläutert. Bei beiden Protokollen enthält CP aus den Scope Coordination Properties (SCP) das Participant Completion Protocol. Die Mapping-Engine entscheidet selbständig, welches der Protokolle eingesetzt werden muss. Das Coordinator Completion Protokoll wird nicht implementiert (siehe Restriktionen in Abschnitt 4.6.5 auf Seite 92).

	Zustand	Outg. Event / Coord. Message	Bed.	Neuer Zustand	Inc. Events / Coord. Messages
1	–	EVT:Activitiy_Ready		Executing	EVT:Start_Activity, auf Context warten
2	Executing	EVT:Variable_Modification (IRA)	(*)	Coordinated Executing	WSC:Register
3	Executing	EVT:Variable_Modification (IRA)	(*)	Faulted	EVT:Fault_To_Scope
4	Executing	EVT:Variable_Modification (IRA)	(*)	Faulted	EVT:Fault_To_Scope
5	Executing	EVT:Variable_Modification (IRA)	(*)	Not Coordinated	
6	Coordinated Executing	WSC:Fault (Registration)		Faulted	EVT:Fault_To_Scope
7	Coordinated Executing	EVT:Activity_Faulted		Faulted	WSBA:Fail
8	Coordinated Executing	EVT:Scope_Compl._W_Fault		Completed With Fault	WSBA:CannotComplete
9	Coordinated Executing	EVT:Activity_Terminated		Terminated	WSBA:Exit
10	Coordinated Executing	WSBA:Cancel		Terminating	EVT:Terminate_Activity
11	Terminating	EVT:Activity_Faulted		Faulted	WSBA:Fail
12	Terminating	EVT:Activity_Terminated		Terminated	WSBA:Canceled
13	Coordinated Executing	Activity_Executed		Waiting	WSBA:Completed
14	Waiting	WSBA:Close		Finished	EVT:Complete_Activity; WSBA:Closed
15	Waiting	WSBA:Compensate		Fault Handling	EVT:Fault_To_Scope
16	Fault Handling	EVT:Scope_Compl._W_Fault		Compensated	WSBA:Compensated
17	Fault Handling	EVT:Activity_Faulted		Faulted	WSBA:Fail

(*) Bedingung siehe Erläuterung der Transitionen
Outg./Inc. Event = Outgoing/Incoming Event; Coord. Message = Coordination Message

Tabelle 4.4: Transitionen des Automaten für `<process>` als WS-BA Participant

4.6.1 Prozess als Participant

Prozesse, deren Prozessinstanzen als Participant an Koordination mit WS-BA teilnehmen, können nur die ICR `new` besitzen. Die Infection Receiving Activity entspricht der Aktivität, die als Attribut `createInstance="yes"` trägt. Die Koordinationslogik wird als Automat dargestellt. Die Zustandsmenge des Automaten enthält die Zustände *Executing*, *Coordinated Executing*, *Waiting* , *Fault Handling*, *Terminating*, *Compensating*, sowie die Endzustände *Faulted*, *Not Coordinated*, *Completed With Fault*, *Terminated* und *Finished*. In Tabelle 4.4 sind die Transitionen des Automaten enthalten. Nachfolgend werden die Transitionen einzeln erläutert. Die Nummer der Liste entspricht dabei jeweils der Zeilennummer in der Transitionstabelle, die beschrieben wird. Sofern nicht anders angegeben beziehen sich Events, die für eine Aktivität oder einen Scope gesendet oder empfangen werden, stets auf den Prozess.

1. Eine Prozessinstanz wird gestartet. Die BPEL-Engine signalisiert dies der Mapping-Engine mit dem *Start_Activity*-Event. Die Mapping-Engine generiert einen neuen Automaten für den Prozess. Der Automat startet im Zustand *Executing*.

2. Die BPEL-Engine generiert das Event *Variable_Modification* für die Variable, die durch die Infection Receiving Activity belegt wird. Die Variable enthält einen Coordination Context als Message Part (z. B. impliziten SOAP-Header) vom Coordination Type WS-BA. Die Mapping-Engine verwendet diesen daraufhin für eine Registrierung am Coordinator und meldet sich für das Participant Completion Protocol an. Schlägt die Registrierung fehl, wird per *Fault_To_Scope* ein Fault in den Prozess geworfen und der Automat geht in den Endzustand *Faulted* über. Andernfalls geht der Automat vom Zustand *Executing* in den Zustand *Coordinated Executing* über.
3. Erhält die Infection Receiving Activity einen Coordination Context, der einen anderen Coordination Type als WS-BA enthält, wird per *Fault_To_Scope* ein Fault in den Prozess geworfen und der Automat geht in den Endzustand *Faulted* über. Dies hat zu erfolgen, da der Coordination Context nach Spezifikation [OAS09c] mit dem Attribut `mustUnderstand="yes"` gesendet werden muss. Daher muss abgelehnt werden, selbst wenn der Prozess in den SCP die Koordination als optional deklariert.
4. Erhält die Infection Receiving Activity keinen Coordination Context obwohl dieser in den SCP nicht als obligatorisch (nicht optional) deklariert wurde, wird per *Fault_To_Scope* ein Fault in den Prozess geworfen und der Automat endet im Zustand *Faulted*.
5. Erhält die Infection Receiving Activity keinen Coordination Context und dieser wurde in den SCP optional deklariert, geht der Automat in den Zustand *Not Coordinated* über. Ab diesem Endzustand greift die Mapping-Engine nicht mehr in den Ablauf des Prozesses ein.
6. Wie Transition 2.
7. Befindet sich der Automat im Zustand *Coordinated Executing* und sendet die BPEL-Engine an die Mapping-Engine ein *Activity_Faulted*-Event für den Prozess, wird dieses in Form einer Fail-Message an den Coordinator weiterpropagiert. Der Automat geht in den Zustand *Faulted* über.
8. Befindet sich der Automat im Zustand *Coordinated Executing* und sendet die BPEL-Engine an die Mapping-Engine ein *Scope_Complete_With_Fault*-Event für den Prozess, wird dieses in Form einer CannotComplete-Message an den Coordinator weiterpropagiert. Der Automat geht in den Zustand *Completed With Fault* über.
9. Befindet sich der Automat im Zustand *Coordinated Executing* und sendet die BPEL-Engine an die Mapping-Engine ein *Activity_Terminated*-Event für den Prozess, wird dieses in Form einer Exit-Message an den Coordinator weiterpropagiert. Der Automat geht in den Zustand *Terminated* über. Alternativ kann für diese Transition auch das Event *Instance_Terminated* als Auslöser verwendet werden.
10. Befindet sich der Automat im Zustand *Coordinated Executing* und sendet der Coordinator eine Cancel an die Mapping-Engine, wird von der Mapping-Engine ein *Terminate_Activity*-Event an die BPEL-Engine gesendet. Der Automat geht in den Zustand *Terminating* über.

11. Befindet sich der Prozess in der Terminierung und diese schlägt fehl, so sendet die BPEL-Engine der Mapping-Engine ein *Activity_Faulted*-Event. Der Automat geht vom Zustand *Terminating* in den Endzustand *Faulted* über. Dem Coordinator wird Fail gesendet.
12. Bei erfolgreicher Terminierung sendet die BPEL-Engine ein *Activity_Terminated*-Event. Der Automat geht in den Endzustand *Terminated* über. Dem Coordinator wird eine Canceled-Message gesendet.
13. Hat der Prozess alle Aktivitäten ausgeführt, geht er in den Zustand *Waiting* über. Er sendet ein *Activity_Executed*-Event an den Coordinator. Der Automat geht in den Zustand *Waiting* über. Der Prozess muss in diesem Zustand warten, bis vom Coordinator entweder Close oder Compensate gesendet wird.
14. Sendet er Close, wird der Prozess beendet. Dazu wird ein *Complete_Activity*-Event an die BPEL-Engine gesendet. Dem Coordinator wird dies mit Closed bestätigt.
15. Sendet er Compensate, wird ein per *Fault_To_Scope* ein Fault in den Prozess gesendet und der Automat geht in den Zustand *Fault Handling* über. Der Prozess kompensiert jetzt seine enthaltenen Scopes.
16. Tritt während des Fault-Handling kein Fehler auf, sendet die BPEL-Engine zu Ende des Handling ein *Scope_Complete_With_Fault*-Event an die Mapping-Engine. Diese sendet ein Compensated-Event an den Coordinator und setzt den Automaten in den Endzustand *Compensated*.
17. Tritt während des Fault-Handling ein weitere Fault auf, sendet die BPEL-Engine ein *Activity_Faulted*-Event an die Mapping-Engine. Diese sendet dann eine Fail-Message an den Coordinator. Der Automat endet im Endzustand *Faulted*.

Es ist hervorzuheben, dass bei dem oben vorgestellen Mapping der Scope – anders, als vom Coordinator verlangt – ein Fault behandelt und nicht kompensiert wird. Der Prozessmodellierer muss sich dessen bewusst sein und hat dafür sorgen, dass der Prozess diesen Fault so behandelt, dass der Prozess anschließend als kompensiert gilt. Das Mapping kann nur so realisiert werden, da ein Prozess keinen Compensation-Handler besitzt und weil das Event *Activity_Executed* das letzte blockierende Event der Prozessinstanz darstellt.

Messages, die die Mapping-Engine vom Coordinator erhält und der Bestätigung dienen, ohne eine Transition im Automaten auszulösen (Exited, NotCompleted, Failed), werden nicht an die BPEL-Engine propagiert, aber von der Mapping-Engine gespeichert, um später zu Audit- oder Logging-Zwecken zu Verfügung zu stehen.

Abbildung 4.6 auf der nächsten Seite zeigt einen ähnlichen Prozess wie das Beispiel aus Fall 1. Anstelle der Koordination der aufgerufenen Web Services per WS-BA werden nun explizite Kompensation-Handler definiert. Die Aktivitäten Undo 1 und Undo 2 rufen Operationen der Web Services auf, die die Web Services in einen Zustand versetzen, der als äquivalent zu den Status der Web Services vor dem Ausführen der ersten Aktivität betrachtet

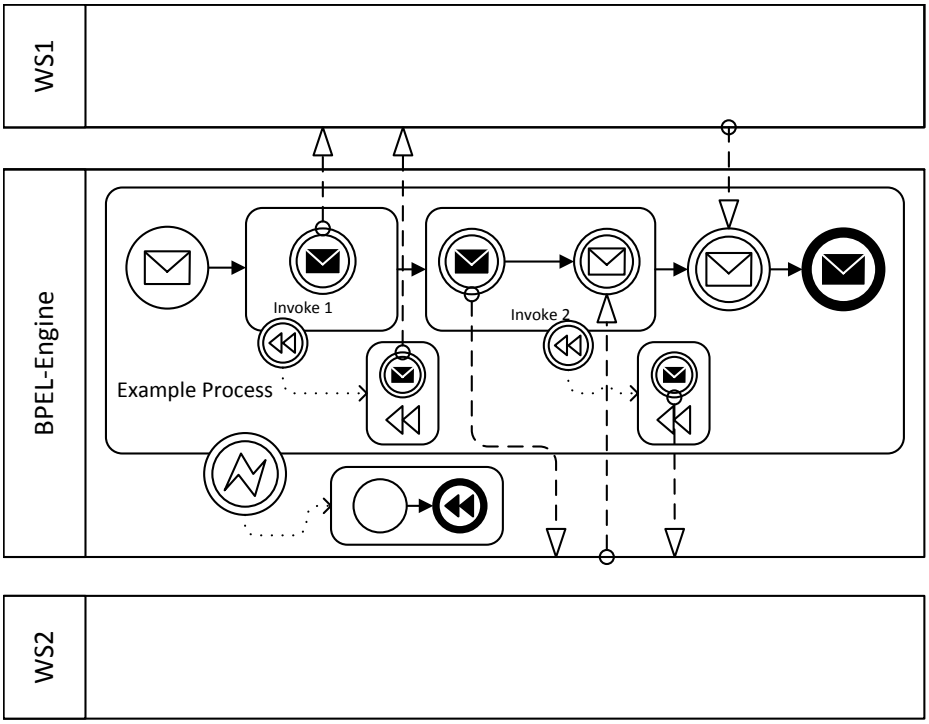

Abbildung 4.6: Beispiel für einen Prozess BPMN

werden kann. Wird der Prozess aufgerufen, empfängt er zusätzlich zum Payload einen Coordination Context. Wird das oben beschriebene Mapping auf das Beispiel angewandt und wird die Business Activity erfolgreich abgeschlossen bzw. kompensiert, ergibt sich das Sequenzdiagramm aus Abbildung 4.7 auf der nächsten Seite. *Invoker* sei dabei der den Prozess aufrufende Web Service.

4.6.2 Participant Completion mit Participant Triggered Compensation

Ein Participant, der an eine Business Activity eine Completed-Message gesendet hat, darf von keiner Transaktion außerhalb der Business Activity zum Kompensieren veranlasst werden, da keine Coordination Message und kein Status definiert wird, der dem Participant erlaubt, sein Kompensieren mitzuteilen. Hat der Participant einmal Completed gemeldet, kann nur noch der Coordinator das Kompensieren verlangen.

Um dennoch dem Participant das selbständige Kompensieren zu ermöglichen, wird in [KML09] ein modifiziertes Coordination Protocol für WS-BA vorgeschlagen. Das Protokoll basiert auf BAPC mit der Modifikation, die in Abbildung 4.8 auf Seite 87 dargestellt wird. Participants bekommen die Möglichkeit, nach der Completed-Message zu kompensieren und

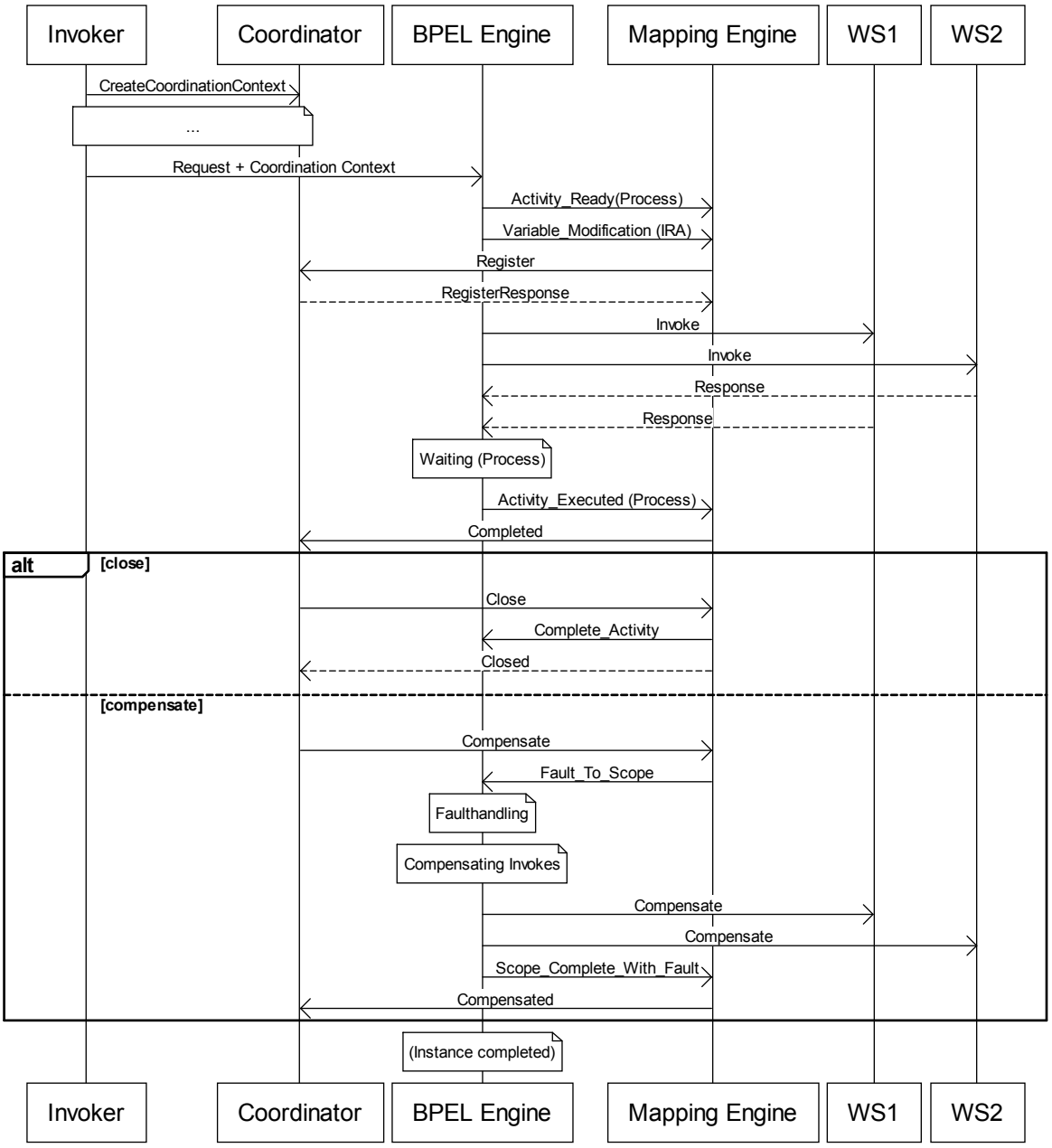

Abbildung 4.7: Sequenzdiagramm des Beispiels mit Close/Compensate

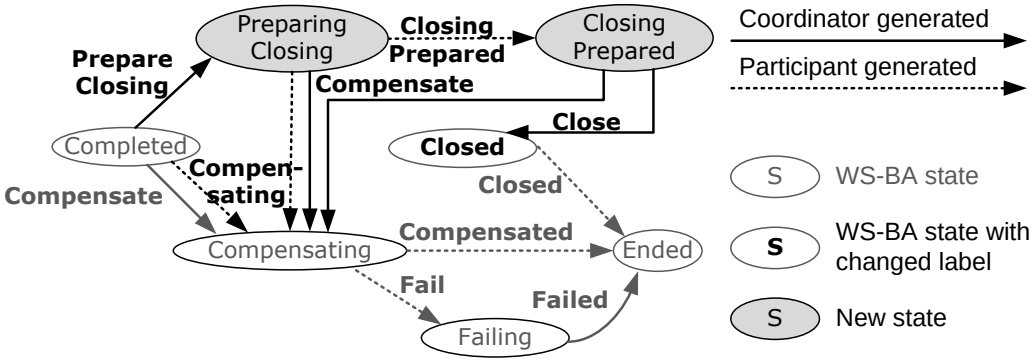

Abbildung 4.8: Modifiziertes Protokoll mit Participant Triggered Compensation [KML09]

dies dem Coordinator mitzuteilen. Die *Participant Triggered Compensation* genannte Modifikation ermöglicht dies, indem eine Voting-Phase vor die Close-Anweisung eingefügt wird. Die Phase besteht aus den beiden neuen Zuständen *Prepare Closing* und *Closing Prepared*.

4.6.3 Scopes als Participants

Wie am Anfang des Abschnitts 4.6 auf Seite 80 beschrieben, liegen BPEL-Scopes und Business Activities in zwei Transaktionsbäumen, die nicht identisch sein müssen. Um BPEL-Scopes dennoch an Business Activities teilnehmen zu lassen, müssen an die Prozessmodellierung zusätzliche Anforderungen definiert werden. Zusätzlich wird die Erweiterung Participant Triggered Compensation für das Participant Completion Protocol verwendet. Der Automat für Scopes als WSBA-Participant aus Tabelle 4.5 auf Seite 91 entspricht dabei im Ablauf bis Ende des Status *Reg.Executing* dem Automaten für Prozesse. Der untere Teil der Transistionen wurden für das BAPC mit Participant Triggered Compensation erweitert. Dazu werden die Zustände *Procwaiting*, *Prepare* und *Prepared* eingeführt. *Procwaiting* bedeutet dabei, dass der Prozess sich bereits im Status *Waiting* befindet, der Coordinator aber noch nicht PrepareClose gesendet hat. *Prepare* bedeutet, dass der Coordinator PrepareClose gesendet hat, aber der Prozess sich noch nicht im Zustand *Waiting* befindet, also noch andere Aktivitäten ausgeführt werden. *Prepared* besagt, dass sich der Prozess im Zustand *Waiting* befindet und vom Coordinator eine Entscheidung in *ClosingPrepared* erwartet wird.

1. Während der Ausführung der umgebenden Prozessinstanz wird der Scope erreicht. Die BPEL-Engine sendet der Mapping-Engine das Event *Activity_Ready*. Die Mapping-Engine erzeugt für den Scope einen Automaten und sendet ein *Start_Activity*-Event an die BPEL-Engine zurück. Der Automat startet im Zustand *Executing* und wartet auf den Eingang eines Coordination Context.

2. Die BPEL-Engine generiert das Event *Variable_Modification* für die Variable, die durch die Infection Receiving Activity belegt wird. Die Variable enthält einen Coordination Context als Message Part vom Coordination Type WS-BA. Die Mapping-Engine verwendet diesen daraufhin für eine Registrierung am Coordinator und meldet sich für das modifizierte Participant Completion Protocol (siehe Abschnitt 4.5.8 auf Seite 80) an. Schlägt die Registrierung fehl, wird per *Fault_To_Scope* ein Fault in den Scope geworfen und der Automat geht in den Endzustand *Faulted* über. Andernfalls geht der Automat vom Zustand *Executing* in den Zustand *Coordinated Executing* über.
3. Erhält die Infection Receiving Activity einen Coordination Context der einen anderen Coordination Type als WS-BA enthält, wird per *Fault_To_Scope* ein Fault in den Scope geworfen und der Automat geht in den Endzustand *Faulted* über. Dies hat zu erfolgen, da der Coordination Context nach Spezifikation [OAS09c] mit dem Attribut `mustUnderstand="yes"` gesendet werden muss. Daher darf der Scope so nicht ausgeführt werden. Der Fault kann im Scope abgefangen und anderweitig behandelt werden. Die Koordination durch die Mapping-Engine ist beendet.
4. Erhält die Infection Receiving Activity keinen Coordination Context, obwohl dieser in den SCP nicht als obligatorisch (nicht optional) deklariert wurde, wird per *Fault_To_Scope* ein Fault in den Scope geworfen und der Automat endet im Zustand *Faulted*. Der Fault kann im Scope abgefangen werden und anderweitig behandelt werden. Die Koordination durch die Mapping-Engine ist beendet.
5. Erhält die Infection Receiving Activity keinen Coordination Context und dieser wurde in den SCP optional deklariert, geht der Automat in den Zustand *Not Coordinated* über. Ab diesem Endzustand greift die Mapping-Engine nicht mehr in den Ablauf des Prozesses ein.
6. Siehe Zeile 2.
7. Befindet sich der Automat im Zustand *Coordinated Executing* und sendet die BPEL-Engine an die Mapping-Engine ein *Activity_Faulted*-Event für den Scope, wird dieses in Form einer Fail-Message an den Coordinator weiterpropagiert. Der Automat geht in den Zustand *Faulted* über.
8. Befindet sich der Automat im Zustand *Coordinated Executing* und sendet die BPEL-Engine an die Mapping-Engine ein *Scope_Complete_With_Fault*-Event für den Scope, wird dieses in Form einer CannotComplete-Message an den Coordinator weiterpropagiert. Der Automat geht in den Zustand *Completed With Fault* über.
9. Befindet sich der Automat im Zustand *Coordinated Executing* und sendet die BPEL-Engine an die Mapping-Engine ein *Activity_Terminated*-Event für den Scope, wird dieses in Form einer Exit-Message an den Coordinator weiterpropagiert. Der Automat geht in den Zustand *Terminated* über. Alternativ kann für diese Transition auch das Event *Instance_Terminated* als Auslöser verwendet werden.
10. Befindet sich der Automat im Zustand *Coordinated Executing* und sendet der Coordinator eine Cancel an die Mapping-Engine, wird von der Mapping-Engine ein *Termi-*

nate_Activity-Event an die BPEL-Engine gesendet. Der Automat geht in den Zustand *Terminating* über und der Scope beginnt mit der Terminierung.

11. Befindet sich der Scope in der Terminierung und diese schlägt fehl, so sendet die BPEL-Engine der Mapping-Engine ein *Activity_Faulted*-Event. Der Automat geht vom Zustand *Terminating* in den Endzustand *Faulted* über. Dem Coordinator wird Fail gesendet.
12. Bei erfolgreicher Terminierung sendet die BPEL-Engine ein *Activity_Terminated*-Event. Der Automat geht in den Endzustand *Terminated* über, dem Coordinator wird eine Canceled-Message gesendet.
13. Hat der Scope alle Aktivitäten ausgeführt, geht er in den Zustand *Waiting* über. Die BPEL-Engine sendet ein *Activity_Executed*-Event an den Coordinator. Der Automat wechselt daraufhin in den Zustand *Completed*. Die Mapping-Engine sendet ein *Complete_Activity*-Event an die BPEL-Engine und löst somit die Blockade des Scopes im Zustand *Waiting* auf. Dieser geht ebenfalls in den Zustand *Completed* über.
14. Befinden sich Scope und folglich auch der Automat im Zustand *Completed*, und fordert der Coordinator mit einer Compensate-Message das Kompensieren des Scopes an, wird ein *Fault_To_Scope*-Event an die BPEL-Engine gesendet. Dieses Event kann nicht an den Scope gesendet werden, da der Scope im Zustand *Completed* kein Fault-Handling mehr ausführen kann. Deshalb muss das Event an einen überliegenden Scope gesendet werden, der sich noch in der Ausführung befindet. Eine Vereinfachung ist das Senden des *Fault_To_Scope*-Events an das `<process>`-Element der laufenden Prozessinstanz. Es muss vom Prozessmodellierer gewährleistet werden, dass durch die modellierten Fault-Handler der betroffene Scope (direkt oder indirekt) kompensiert wird. Der Automat geht in den Zustand *Compensating* über.
15. Das Ende der Kompensierung des Scopes wird durch die BPEL-Engine mit einem *Scope_Compensated*-Event signalisiert. Der Automat befindet sich beim Eintreffen des Events im Zustand *Compensating*. Die Mapping-Engine propagiert das Event an den Coordinator, indem sie eine Compensated-Message an den Coordinator sendet. Der Automat geht in den Endzustand *Compensated* über.
16. Schlägt die Kompensierung des Scopes fehl, sendet die BPEL-Engine ein *Activity_Faulted*-Event an die Mapping-Engine. Diese meldet dem Coordinator den Fehlschlag mit einer Fail-Message.
17. Kann der Scope aufgrund von expliziten Fault- oder Compensation-Handlern, die kein Kompensieren vorsehen, nicht kompensiert werden, empfängt die Mapping-Engine am Ende der Prozessinstanz von der BPEL-Engine das *Instance_Completed*-Event. Die Mapping-Engine sendet eine Fail-Message an den Coordinator und die Automat geht in den Zustand *Faulted* über. Dieser Zustand entspricht nicht dem des Scopes. Der Scope befindet sich weiterhin im Zustand *Completed*.
18. Befinden sich der Scope und folglich auch der Automat im Zustand *Completed* und startet die BPEL-Engine das Kompensieren des Scopes, wird von ihr das

Scope_Compensating-Event an die Mapping-Engine gesendet. Der Automat geht in den Zustand *Compensating* über und die Mapping-Engine sendet die in Abschnitt 4.6.2 auf Seite 85 eingeführte Compensating-Message an den Coordinator. Über eine erfolgreiche Kompensierung wird der Coordinator wieder über die Transitionen 15 und 16 informiert.

19. Befindet sich der Automat im Zustand *Completed*, kann vom Coordinator eine PrepareClosing-Message eintreffen. Dabei kann sich der Prozess noch in der Ausführung befinden und gegebenenfalls Kompensierung des Scopes anstoßen. Daher muss gewartet werden, bis sich die `<process>`-Aktivität im Zustand *Waiting* befindet. Erst dann darf dem Coordinator die ClosingPrepared-Message gesendet werden. Der Automat geht in den Zustand *Prepare* über.
20. Im Zustand *Prepare* des Automaten kann es in der Prozessinstanz weiterhin zu Kompensierung (*Scope_Compensating*) kommen. Dem Coordinator wird dies per Compensating mitgeteilt. Der Automat geht in den Zustand *Compensating* über.
21. Auch der Coordinator kann, während sich der Automat im Zustand *Prepare* befindet, eine Compensate-Message senden. Dabei wird ebenfalls das in Transition 14 beschriebene Verhalten durchgeführt.
22. Befindet sich der Automat im Zustand *Completed* kann die BPEL-Engine der Mapping-Engine ein *Activity_Executed* für die `<process>`-Aktivität gesendet werden. Da in diesem Zustand der Coordinator noch kein PrepareClosing gesendet hat, muss die Prozessinstanz in diesem Zustand auf die Message warten. Der Automat geht in den Zustand *Procwaiting* über.
23. Im Zustand *Procwaiting* kann der Coordinator entweder Compensate oder PrepareClosing senden. Sendet er Compensate, muss der Scope kompensiert werden. Die Mapping-Engine sendet ein an die `<process>`-Aktivität gerichtetes *Fault_To_Scope*-Event an die BPEL-Engine. Der Automat geht in den Zustand *Compensating* über. Das weitere Verfahren entspricht wieder dem der Transitionen 15, 16 und 17.
24. Befindet sich der Automat im Zustand *Procwaiting* und sendet der Coordinator PrepareClosing, kann die Mapping-Engine sofort mit ClosingPrepared antworten. Der Automat geht in den Zustand *Prepared* über.
25. Befindet sich der Automat im Zustand *Prepare* und die `<process>`-Aktivität der Prozessinstanz geht in den Zustand *Waiting* (*Activity_Executed*) über, wird die ClosingPrepared-Message an den Coordinator gesendet. Der Automat geht in den Zustand *Prepared* über und wartet auf die Entscheidung des Coordinators.
26. Befindet sich der Automat im Zustand *Prepared* und entscheidet der Coordinator für Close, kann die Blockade der `<process>`-Aktivität im Zustand *Waiting* aufgelöst werden. Dazu wird der BPEL-Engine das *Complete_Activity*-Event gesendet. Die Entscheidung des Coordinators wird durch Closed bestätigt.

#	Zustand	Outg. Event / Coord. Message	Bed.	Zustand'	Events/Messages/Aktionen
1	--	EVT:Activitiy_Ready		Executing	EVT:Start_Activity; auf Context warten
2	Executing	EVT:Variable_Modification (IRA)	(*)	Coordinated Executing	WSC:Register
3	Executing	EVT:Variable_Modification (IRA)	(*)	Faulted	WSBA:Fault_To_Scope
4	Executing	EVT:Variable_Modification (IRA)	(*)	Faulted	WSBA:Fault_To_Scope
5	Executing	EVT:Variable_Modification (IRA)	(*)	Not Coordinated	
6	Coordinated Executing	WSC:Fault (Registration)		Faulted	EVT:Fault_To_Scope
7	Coordinated Executing	EVT:Activity_Faulted		Faulted	WSBA:Fail
8	Coordinated Executing	EVT:Scope_Compl._W_Fault		CompletedWithFault	WSBA:CannotComplete
9	Coordinated Executing	EVT:Activity_Terminated		Terminated	WSBA:Exit
10	Coordinated Executing	WSBA:Cancel		Terminating	EVT:Terminate_Activity
11	Terminating	EVT:Activity_Faulted		Faulted	WSBA:Fail
12	Terminating	EVT:Activity_Terminated		Terminated	WSBA:Canceled
13	Coordinated Executing	EVT:Activity_Executed		Completed	EVT:Complete_Activity; WSBA:Completed
14	Completed	WSBA:Compensate		Compensating	EVT:Fault_To_Scope (Prozess)
15	Compensating	EVT:Scope_Compensated		Compensated	WSBA:Compensated
16	Compensating	EVT:Activity_Faulted		Faulted	WSBA:Fail
17	Compensating	EVT:Instance_Completed		Faulted	WSBA:Fail
18	Completed	EVT:Scope_Compensating		Compensating	WSBA:Compensating
19	Completed	WSBA:PrepareClosing		Prepare	
20	Prepare	EVT:Scope_Compensating		Compensating	WSBA:Compensating
21	Prepare	WSBA:Compensate		Compensating	EVT:Fault_To_Scope (Prozess); WSBA:Compensating
22	Completed	EVT:Activity_Executed (Prozess)		Procwaiting	
23	Procwaiting	WSBA:Compensate		Compensating	EVT:Fault_To_Scope (Prozess); WSBA:Compensating
24	Procwaiting	WSBA:PrepareClosing		Prepared	WSBA:ClosingPrepared
25	Prepare	EVT:Activity_Executed (Prozess)		Prepared	WSBA:ClosingPrepared
26	Prepared	WSBA:Close		Finished	EVT:Complete_Activity(Proc); WSBA:Closed
27	Prepared	WSBA:Compensate		Compensating	EVT:Fault_To_Scope (Prozess); WSBA:Compensating

Legende: (*) Bedingung siehe Erläuterung der Transtitionen; Outg./Inc. Event = Outgoing/Incoming Event; Coord. Message = Coordination Message; Bed. = Bedingung

Tabelle 4.5: Transitionen des Automaten für <scope> als WS-BA Participant

27. Befindet sich der Automat im Zustand *Prepared* und entscheidet der Coordinator für Compensate, muss der Scope kompensiert werden. Dazu wird das in Transition 23 beschriebene Verhalten durchgeführt.

4.6.4 Weitergabe des Coordination Context

In den vorgestellten Ansätzen zur Koordinierung von Scopes und Prozessen können innerhalb der koordinierten Scopes bzw. innerhalb des Prozesses <invoke>-Aktivitäten auftreten. Um diese Aktivitäten mitzukoordinieren, sind folgende Verfahren denkbar: Eine Weitergabe des von extern erhaltenen Coordination Context stellt dabei keine gute Möglichkeit dar, da

die Participants unabhängig vom eigenen Scope koordiniert würden. Dies kann dazu führen, dass der Prozessmodellier an einer bestimmten Stelle im Prozessmodell davon ausgeht, dass sich ein Participant in einem bestimmten Status befindet. Dieser kann sich jedoch durch die Koordinierung, auf die der Scope bzw. Prozess keinen Einfluss hat, bereits in einem anderen Status befinden. Solche Prozessmodelle sind daher fehlerträchtig. Eine bessere Lösung ist das Einfügen eines neuen Scopes in den Scope bzw. Prozess mit ICR `new` und ECR `Initiator`, der dann weitere externe Partner wiederum mittels WS-BA koordiniert. Auf diese Art und Weise entsprechen die Transaktionsgrenzen der Business Activities den Scopes.

4.6.5 Annahmen und Einschränkungen

In diesem Abschnitt wurde gezeigt, wie Prozesse und Scopes von extern mit WS-BA koordiniert werden können. Dabei kann für koordinierte Scopes nur das Coordination Protocol BAPC mit der Modifikation Participant Triggered Compensation eingesetzt werden, da andernfalls das Kompensieren des Scopes durch Aufruf seines Compensation-Handlers durch keine valide Coordination Message dem Coordinator mitgeteilt werden könnte. Zusätzlich wurde für koordinierte Scopes gefordert, dass im Verlauf der gesamten Prozessinstanz sein Compensation-Handler erreichbar bleiben muss. Folglich darf der Prozessmodellierer nicht alle Fault- und Compensation-Handler beliebig setzen.

Für die `<process>`-Aktivität wurde festgelegt, dass sie sich mit dem Participant Completion Protokoll anmeldet (siehe Abschnitt 4.6.1 auf Seite 82). `<scope>`-Aktivitäten wurden über das modifizierte Completion Protocoll aus Abschnitt 4.5.8 auf Seite 80 koordiniert. Eine Unterstützung für das Coordinator Completion Protokoll wurde nicht erläutert. Für `<scope>`-Aktivitäten muss dieses ebenfalls modifiziert werden. Die Begründung ist mit der aus Abschnitt 4.5.8 auf Seite 80 identisch. Die Modifikation sowie Koordinierung mit dem Coordinator Completion Protokoll sind nicht Teil der Arbeit.

Es darf unter Verwendung der vorgestellten Automaten maximal eine Scopeinstanz oder Prozessscopeinstanz pro Prozessinstanz koordiniert werden.

Beweis 1
Angenommen es werden (i) zwei Scopeinstanzen oder (ii) eine Scopeinstanz und die Prozessscopeinstanz mit WS-BA koordiniert, können die folgenden Szenarien konstruiert werden, die zeigen, wie die Automaten die Business Activities gegenseitig zu Inkonsistenzen führen.

i Die zwei koordinierten Scopeinstanzen seien Scopeinstanz A und Scopeinstanz B und beide Scopeinstanzen befänden sich im Zustand Completed und ihre Automaten in der Mapping-Engine im Zustand Prepared. Dies ist der Fall, wenn sich `<process>` bereits im Zustand Waiting befindet. Nun sendet der Coordinator von Scopeinstanz A Compensate*. Dadurch muss ein Fault in den Prozess gesendet werden. Aufgrund der Forderung, dass alle Compensation-Handler von (mit WS-BA koordinierten) Scopes erreicht werden können, führt dies zur Kompensierung*

beider Scopeinstanzen. Angenommen der Coordinator von Scopeinstanz B sende jetzt Close. Die Mapping-Engine besitzt nun keine Möglichkeit mehr, dem Coordinator mitzuteilen, dass Scopeinstanz B bereits kompensiert wurde. Dabei ist irrelevant, ob eine Scopeinstanz die andere umschließt.

ii Die Prozessscopeinstanz befinde sich im (Eventmodell-)Zustand Waiting. Die Mapping-Engine hat die Completed-Message an den Coordinator gesendet. Die Scopeinstanz befindet sich dementsprechend im Zustand Completed. Die ClosingPrepared-Message für die Scopeinstanz wurde von der Mapping-Engine bereits an den Coordinator gesendet, ihr Automat in der Mapping-Engine befindet sich folglich im Zustand Prepared. Nun sendet der Coordinator der Prozessscopeinstanz die Compensate-Message an die Mapping-Engine. Diese sendet daraufhin einen Fault in die `<process>`-Aktivity, woraufhin der Prozess einschließlich der Scopeinstanz kompensiert wird. Sendet der Coordinator der Scopeinstanz jetzt eine Close-Anweisung, besitzt die Mapping-Engine keine Möglichkeit, dem Coordinator mitzuteilen, dass die Scopeinstanz bereits kompensiert wurde.

■

Auch eine Modifikation der Automaten, die diese Inkonsistenz immer vermeidet, ist nicht möglich. Die Mapping-Engine darf für keine der Scopeinstanzen eine ClosingPrepared-Message senden, solange andere koordinierte Scopeinstanzen über den Coordinator und die Mapping-Engine noch kompensiert werden können. Dies führt dazu, dass in einer Situation mit zwei Scopeinstanzen, deren Automaten sich im Zustand Waiting befinden, für keine von beiden eine ClosingPrepared-Message gesendet werden darf, da die Scopeinstanzen jeweils noch über eine Compensated-Message für die jeweils andere Scopeinstanz kompensiert werden kann. Die Scopeinstanzen sind im Zustand Waiting gefangen.

Eine Möglichkeit, mehrere Scopeinstanzen pro Prozessinstanz mit WS-BA koordinieren zu können liegt darin, die Annahme, dass der Compensation-Handler der Scopeinstanz erreicht werden kann, aufzuweichen. Dann kann die ClosingPrepared-Anweisung bereits erfolgen, sobald der Compensation-Handler der Scopeinstanz nicht mehr erreicht werden kann. Um diesen Zeitpunkt in der Prozessinstanz zu bestimmen, kann das Eventmodell um ein Event *Handler_Dead_Path*-Event erweitert werden, welches für den Compensation-Handler vom Automat abgefangen wird, und daraufhin die ClosingPrepared-Message an den Coordinator sendet. Bei dieser Möglichkeit dürfen dann mehrere per WS-BA koordinierte Scopeinstanzen pro Prozessinstanz auftreten, solange das Kompensieren einer der Scopeinstanzen nie zum Kompensieren der anderen Scopeinstanzen führen kann. Dieser Ansatz wird hier nicht weiter untersucht.

Apache ODE unterstützt keine Termination. Demzufolge tritt das Event *Scope_Handling_Termination* nicht auf. Daher wird in den Transitionen, die durch eine Cancel-Message des Coordinators ausgelöst werden, der Zustand *Terminating* des Automaten übersprungen. Diese Transitionen führen direkt in den Zustand *Terminated* und senden dem Coordinator die Message Fail. Es darf keine *Canceled*-Message erfolgen, da

diese die erfolgreiche Durchführung der Termination angeben würde. Alternativ kann Termination-Handling durch Fault-Handling gelöst werden. In diesem Fall muss in diesen Transitionen die Mapping-Engine einen termination-spezifischen Fault in den Scope werfen, wobei der Prozessmodellierer für diesen `catch`-Anweisungen modellieren kann. Eine Termination, die von der BPEL-Engine ausgelöst wird, kann so nicht behandelt werden. Vom fehlenden Termination-Handler in ODE betroffen sind aus Tabelle 4.4 auf Seite 82 und 4.5 auf Seite 91 jeweils die Transitionen mit der Nummer 10.

4.7 Fall 3: Scope koordiniert aufgerufene Web Services mit WS-AT

Während in den ersten beiden Fällen die WS-BA-Unterstützung von Scopes diskutiert wurde, geht es in den nächsten beiden Fällen um die Unterstützung von WS Atomic Transaction. Es wird beschrieben, inwiefern Scopes als langlaufende Transaktionen überhaupt mit dem Konzept der ACID-Transaktionen kombiniert werden können und welche Einschränkungen und Annahmen dazu nötig sind.

4.7.1 Atomic Scopes

[Apa10e] beschreibt eine BPEL-Erweiterung *Atomic Scopes*, die durch Erweiterung im Ablauf und Design eines isolated Scopes atomares Verhalten erreicht. Statusänderungen und Zuweisungen werden jeweils erst beim Completen eines Atomic Scopes wirksam und sind vor dem Scope-Ende nur innerhalb des Scopes sichtbar. Wenn der Scope vorher abbricht, werden Änderungen verworfen. Aktivitäten, die auf eingehende Events von außen warten (`<wait>` und `<pick>`), sind in atomic Scopes nicht erlaubt, sofern diese keinen Kontrolleinfluss auf den Scope haben. Ebenso sind Event-Handler innerhalb Scopes nicht gestattet. MEPs müssen im Scope vollständig durchgeführt werden. `<receive>`- und `<pick>`-Aktivitäten dürfen nur vorkommen, wenn sie auch als erste auszuführende Aktivität im Scope liegen. Ein atomic Scope befindet sich am Ende in einem der drei Status *Successful Completion*, *Unsuccessful Completion* oder *Scope Rollback*.

4.7.2 AT-Scopes

Da es in der vorliegenden Arbeit hauptsächlich um das Koordinieren der Teilnehmer geht und insbesondere atomic Scopes eine ODE-spezifische Erweiterung darstellen, die eigene Kompensationsregeln durch die BPEL-Engine verfolgt, werden hier schwächere Anforderungen gestellt. Ein Scope, der seine Participants per WS-AT koordiniert, erhält den Namen AT-Scope. Die in diesem Abschnitt beschriebenen Koordinationsverfahren gelten jeweils für Prozesse und Scopes. Es wird für eine bessere Lesbarkeit nur der Begriff „AT-Scope“ verwendet.

- Ein AT-Scope besitzt ICR `new` und ECR `Initiator`. Sein CoordinationType ist in [OASo9b] definiert.
- AT-Scopes müssen *isolated* Scopes sein, um so Atomizität des Scopes zum Prozess hin zu realisieren.
- Ein AT-Scope gilt als abgeschlossen, wenn der Scope erfolgreich beendet wird und alle Participants ebenfalls im Zustand *Commited* enden. Ein späteres Kompensieren des Scopes darf nicht über den impliziten Compensation-Handler stattfinden. Es muss dafür ein expliziter Compensation-Handler definiert werden, der keine `<compensate>` und `<compensateScope>`-Aktivitäten enthalten darf. Der Compensation-Handler darf aber einen weiteren AT-Scope enthalten.
- Innerhalb des AT-Scope dürfen keine Compensation- Termination- und Fault-Handler gesetzt werden. Diese Anforderung muss definiert werden, damit innerhalb des AT-Scopes Faults in den überliegenden Scope weiterpropagiert werden, das Terminieren in Aktivitäten innerhalb des AT-Scopes in seiner Transitivität nicht unterbrochen wird und durch keinen dieser Handler Einfluss auf Aktivitäten außerhalb des AT-Scopes oder außerhalb des Prozesses (durch externe Aufrufe) genommen werden kann.
- Alle Requests und Antworten eines MEPs des Prozess müssen entweder innerhalb oder außerhalb eines AT-Scopes liegen. Sie dürfen AT-Scope-Grenzen nicht überlappen. MEPs, die im AT-Scope beginnen, müssen auch im AT-Scope enden.

4.7.3 Prozesse und Scopes mit ICR `new`

Als Coordinator sind zum Koordinieren von AT-Scopes zwei Konzepte denkbar. Entweder kann wieder die Mapping-Engine die Rolle eines Coordinators übernehmen (interner Coordinator), wie in Fall 1 (Abschnitt 4.5 auf Seite 64) beschrieben, oder ein externer Coordinator kann verwendet werden, um dann mit dem Completion-Protokoll das Committen zu veranlassen. Diese Möglichkeiten werden im Folgenden erläutert.

Ablauf mit internem Coordinator

Transitionstabelle 4.6 auf der nächsten Seite enthält alle Transitionen des Automaten, der die Participants eines Scopes mit WS-AT koordiniert. Der Automat enthält die Zustände *Executing*, *Waiting*, *Prepared* und die zwei Endzustände *Commited* und *Aborted*. Solange sich der AT-Scope im Zustand *Executing* oder *Waiting* befindet, ist der Automat ebenfalls im Zustand *Executing* bzw. *Waiting*. Befindet sich der AT-Scope im Zustand *Completed*, ist der Automat im Zustand *Commited*. Befindet sich der AT-Scope in einem beliebigen anderen Zustand außer *Event-Handling*, ist der Automat im Zustand *Aborted*. Der Zustand *Event-Handling* wird ignoriert, da der Scope nach dem Event-Handling wieder in den Zustand *Executing* zurückkehrt. Die Zustände *Commited* und *Aborted* signalisieren, ob die Atomic

#	Zustand	Outg. Event / Coord. Message	Bedingung	Neuer Zustand	Inc. Events / Coord. Messages
1	--	EVT:Activity_Ready		Executing	EVT:Start_Activity
2	Executing	WSAT:Aborted		Aborted	WSAT:Rollback; EVT:Fault_To_Scope
3	Executing	EVT:Scope_Handling_Fault		Aborted	WSAT:Rollback
4	Executing	EVT:Scope_Handling_Termination		Aborted	WSAT:Rollback; EVT:Continue
5	Executing	EVT:Activity_Executed		Waiting	WSAT:Prepare
6	Waiting	EVT:Scope_Handling_Fault		Aborted	WSAT:Rollback
7	Waiting	EVT:Scope_Handling_Termination		Aborted	WSAT:Rollback; EVT:Continue
8	Waiting	WSAT:Aborted		Aborted	WSAT:Rollback; EVT:Fault_To_Scope
9	Waiting	WSAT:Prepared	APP	Prepared	EVT:Complete_Activity
10	Prepared	EVT:Scope_Handling_Fault		Aborted	WSAT:Rollback
11	Prepared	EVT:Scope_Handling_Termination		Aborted	WSAT:Rollback; EVT:Continue
12	Prepared	EVT:Activity_Complete		Commited	WSAT:Commit

Legende: APP = Alle Partcipants Prepared; Outg./Inc. Event = Outgoing/Incoming Event; Coord. Message = Coordination Message

Tabelle 4.6: Transitionen für `<scope>` und `<process>` als WS-AT Initiator, intern koordiniert

Transaction erfolgreich abgeschlossen oder zurückgesetzt wurde. Die einzelnen Transitionen bewirken dabei Folgendes:

1. Ein AT-Scope steht vor der Ausführung. Er befindet sich im Zustand *Ready* und die BPEL-Engine sendet das Event *Activity_Ready* an die Mapping-Engine. Wie in Fall 1 in Transition 1 der Transitionstabelle 4.1 auf Seite 65 wird ein Automat generiert, der im Zustand *Executing* beginnt und der AT-Scope mit *Start_Activity* gestartet. Der AT-Scope wird nun ausgeführt. Bei den innerhalb des AT-Scopes auftretenden `<invoke>`-Aktivitäten wird der Coordination Context jeweils in den Requests in den Header eingefügt (vgl. 4.5 auf Seite 64). Die so infizierten Partner registrieren sich für die Atomic Transaction. Die Registrierung einzelner Participants wird in den jeweiligen `<invoke>`-Aktivitäten abgeschlossen (siehe Abschnitt 4.7.6 auf Seite 100). Zusätzliche Participants können sich registrieren solange sich der Automat im Zustand *Executing* befindet.
2. Sendet im Zustand *Executing* einer der Participants eine Aborted-Message, wird der BPEL-Engine ein *Fault_To_Scope*-Event gesendet. Die BPEL-Engine löst einen Fault im AT-Scope aus und sendet allen übrigen Participants eine Rollback-Message. Der Automat geht in den Endzustand *Aborted* über.
3. Tritt während der Ausführung des AT-Scopes ein Fehler auf, wird allen Participants eine Rollback-Message gesendet. Ob der Fault abgefangen wird, obliegt dem Prozessmodellierer des Prozesses. Das Fault-Handling hat keinen Einfluss mehr auf die Atomic Transaction. Der Automat geht in den Zustand *Aborted* über.

4. Wird der AT-Scope im Zustand *Executing* terminiert, sendet die BPEL-Engine der Mapping-Engine ein *Scope_Handling_Termination*-Event. Die Mapping-Engine sendet an alle Participants eine Rollback-Message. Der Automat geht in den Zustand *Aborted* über. Da es in Apache ODE keine Termination-Handler gibt, muss bei ODE statt *Scope_Handling_Termination* das *Activity-Faulted*-Event abgefangen werden.
5. Die BPEL-Engine signalisiert der Mapping-Engine mit *Activity_Executed* das Beenden des AT-Scopes. Dieser wartet im Zustand *Waiting*. Allen Participants wird die Prepare-Message geschickt.
6. wie Transition 3
7. wie Transition 4
8. Befindet sich der AT-Scope im Zustand *Waiting* und sendet ein Participant eine Aborted-Message, wird die Atomic Transaktion zurückgesetzt. Dazu sendet die Mapping-Engine allen übrigen Participants eine Rollback-Message und der BPEL-Engine ein *Fault_To_Scope*-Event. Der Automat geht in den Zustand *Aborted* über.
9. Sendet ein Participant eine Prepared-Message und haben alle anderen Participants auch eine Prepared-Message gesendet, muss der AT-Scope beendet werden. Die Mapping-Engine sendet der BPEL-Engine ein *Complete_Activity*-Event. Der Automat geht in den Zustand *Prepared* über.
10. wie Transition 3
11. wie Transition 4
12. Der Automat muss im Zustand *Prepared* auf das Event *Activity_Complete* warten, da sich andernfalls bereits ein *Scope_Handling_Fault*- oder *Scope_Handling_Termination*-Event in der Queue befinden könnte. Daher darf er erst nach dem Empfangen des *Activity_Complete*-Events an alle Participants die Commit-Message senden. Der Automat geht dann in den Endzustand *Commited* über.

Bei AT-Scopes ist eine Zuordnung zwischen `<invoke>`-Aktivität und Participant wie in Fall 1 (WS-BA) in Abschnitt 4.5.2 auf Seite 68 nicht nötig, da im AT-Scope nicht relevant ist, welcher Participant den *Rollback* verlangt, da *Rollback* ohnehin an alle Participants gesendet wird und nicht wie im Fall von WS-BA den `<invoke>`-Aktivitäten das Kompensieren überlassen wird. Auch im erfolgreichen Fall kommt von allen Participants ein *Committed*, wodurch ein Unterscheiden unnötig ist. Aus diesem Grund werden die Participants in der Realisierung in Kapitel 5 auf Seite 109 auch nicht den Entitäten für `<invoke>`-Aktivitäten, der des AT-Scopes zugeordnet. Dies führt zu dem Vorteil, beliebig viele Participants mit AT-Scopes koordinieren zu können. In Fall 1 entsprach die Anzahl der Participants immer genau der Anzahl der aufgerufenen `<invoke>`-Aktivitäten mit ICR `participating`.

Ablauf mit externem Coordinator

Anstelle eines internen Koordinierens der Participants kann auch ein Coordinator verwendet werden. Dies ist bei WS-BA nicht der Fall, da durch einen zusätzlichen Coordinator die Beziehung zwischen `<invoke>`-Aktivitäten und Participants verloren ginge (siehe Abschnitt 4.5.2 auf Seite 68). Bei AT-Scopes ist – wie im vorherigen Abschnitt beschrieben – keine Korrelation zwischen `<invoke>`-Aktivität und Participant nötig. Für AT-Scopes wird der Vollständigkeit wegen im Folgenden ein Verfahren beschrieben, mit welchem Participants über einen Coordinator verwaltet werden können. Dieses Verfahren ist nicht Teil der Realisierung, da es zu keinem Gewinn an Funktionalität führt. Das Verfahren mit interner Koordinierung wird implementiert.

Wird anstelle der Mapping-Engine als Coordinator ein externer Coordinator verwendet, wird in der Transition 1 des AT-Scopes von der Mapping-Engine eine neue Atomic Transaction über den Activation Service gestartet. Die Mapping-Engine meldet sich dabei selbst mit dem Completion-Protokoll aus [OAS09a] an, um das Ende der Atomic Transaction steuern zu können. Der erhaltene Coordination Context wird in den `<invoke>`-Aktivitäten mit ICR `participating` des AT-Scopes an die aufgerufenen Web Services mitgesendet. Somit melden sich die Participants am externen Coordinator an. Die Ausführung des AT-Scopes bleibt identisch. Falls ein Participant abbricht, meldet dieser dem Coordinator Aborted. Der Coordinator leitet der Mapping-Engine ein Aborted weiter, welches von der Mapping-Engine wiederum als Fault in den AT-Scope propagiert wird. Leitet der Coordinator kein Abort ein und befindet sich der AT-Scope im Zustand *Waiting*, sendet die Mapping-Engine eine Commit-Message an den Coordinator und leitet damit den Zwei-Phasen-Commit ein. Der Coordinator sendet nach Eingang der Votes der Participants entweder *Commited* oder Aborted an die Mapping-Engine. Die Mapping-Engine beendet den AT-Scope entsprechend entweder mit *Complete_Activity* oder *Fault_To_Scope*.

In der beschriebenen letzten Situation kann es zu Inkonsistenz kommen: Während der Coordinator die Prepared-Phase durchführt, kann der AT-Scope terminiert werden. Bei interner Koordinierung kann er in diesem Fall anstelle des Commit eine Rollback-Message an alle Participants senden. Da der AT-Scope bei externer Koordinierung an diesem Punkt über das Completion-Protokoll aber keine Möglichkeit mehr besitzt, in den 2PC einzugreifen, kann es zur Inkonsistenz kommen. Die Participants machen ihr Ergebnis dauerhaft, während der AT-Scope abbricht. Um diese Inkonsistenz zu verhindern, existieren zwei Lösungen: Entweder muss ein verändertes Completion-Protokoll – wie zum Beispiel das aus Abbildung 4.9 auf der nächsten Seite – verwendet werden oder der BPEL-Engine wird ein Terminieren im Zustand *Waiting* nicht erlaubt. Da die zweite Möglichkeit in der Realisierung (siehe Kapitel 5 auf Seite 109) mit dem Pluggable Framework nicht durchführbar ist, wird nachfolgend ein modifiziertes Completion-Protokoll vorgestellt.

Während der Initiator im in [OAS09b] eingeführten Completion-Protokoll nur den 2PC mit einer Commit-Message anstößt und nur noch das Ergebnis per Aborted bzw. *Commited*

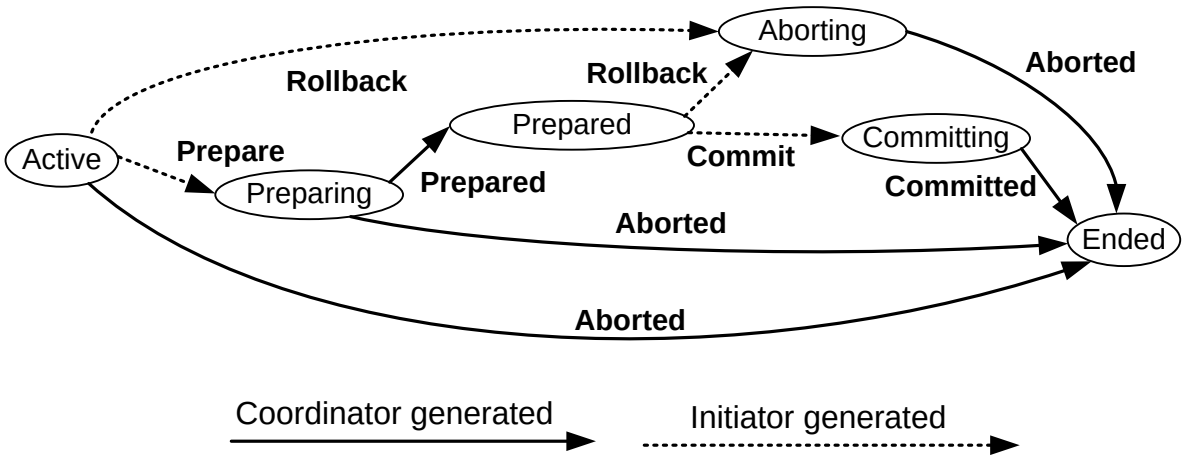

Abbildung 4.9: Modifiziertes Completion-Protokoll für WS-AT

angezeigt bekommt, erhält er im modifizierten Protokoll aus Abbildung 4.9 wie in [TS07] die Möglichkeit, nach der Prepare-Phase aus dem Zustand Prepared für ein Commit oder Rollback zu entscheiden.

Die Zustände des Automaten sind die identischen wie wie bei interner Koordinierung. Die Transitionstabelle des Automaten für externe Koordinierung mit dem modifizierten Completion-Protokoll entspricht der Transitionstabelle 4.6 auf Seite 96 für interne Koordinierung. Anstatt Coordination Messages von und an Participants zu empfangen bzw. zu senden, wird je nur genau eine Message vom Coordinator empfangen bzw. an den Coordinator gesendet. Bei den Messages handelt es sich nicht um Messages aus dem 2PC-Protokoll sondern um die gleichnamigen Messages aus dem modifizierten Completion-Protokoll. Die Bedingung *Alle Participants Prepared* (APP) gilt als immer erfüllt.

4.7.4 Invoke-Aktivitäten mit ICR `participating`

Der Automat für Invoke-Aktivitäten mit ICR `participating` enthält die Zustände *Pre Message* und *Message_Prepared*, *Executing*, *Waiting* sowie die Endzustände *Completed* und *Not-Completed*. Tabelle 4.7 auf der nächsten Seite stellt die Transitionstabelle des Automaten dar. Wie in der ICR `new` werden die Zustände *Pre Message* und *Message_Prepared* dazu verwendet, den Coordination Context in die Application Message für den Web-Service-Aufruf einzubringen. Compensation-Handler dürfen für die per WS-AT koordinierten `<invoke>`-Aktivitäten nicht gesetzt werden. Die Semantik ist bei der `<invoke>`-Aktivität folgende: Der Web Service der `<invoke>`-Aktivität wird aufgerufen. Die Application Message enthält einen Coordination Context. Erst wenn sowohl der Aufruf als auch das Registrieren des Participants

#	Zustand	Outg. Event / Coord. Message	Bed.	Neuer Zustand	Inc. Events / Coord. Messages
1	--	EVT:Activity_Ready		Pre Message	EVT:Start_Activity
2	Pre Message	EVT:Message_Prepared		Message Prepared	EVT:Add_Message_Header (CoordContext)
3	Message Prepared	EVT:Activity_Terminated		NotCompleted	
4	Message Prepared	EVT:Scope_Handling_Fault		NotCompleted	
5	Message Prepared	EVT:Message_Modification		Executing	EVT:Continue
6	Executing	EVT:Activity_Terminated		NotCompleted	
7	Executing	EVT:Scope_Handling_Fault		NotCompleted	
8	Executing	EVT:Activity_Executed	PR	Completed	EVT:Complete_Activity
9	Waiting	EVT:Activity_Terminated		Terminated	
10	Waiting	EVT:Scope_Handling_Fault		NotCompleted	
11	Waiting	WSC:Register	PNR	Completed	EVT:Complete_Activity

Legende: Outg./Inc. Event = Outgoing/Incoming Event; Coord. Message = Coordination Message; P(N)R = Partcipant (nicht) registriert; Bed. = Bedingung

Tabelle 4.7: Transitionen des Automaten für `<invoke>` mit ICR `participating`

abgeschlossen sind, gilt die `<invoke>`-Aktivität als beendet. Da die Participants – wie bereits in Abschnitt 4.7.3 auf Seite 95 aus der Perspektive von AT-Scopes beschrieben – vom Effective Scope der `<invoke>`-Aktivität koordiniert werden, besitzt der Automat keine Transitionen, innerhalb derer Coordination Messages ausgelöst werden. Die Transitionen 1 bis 5 werden verwendet, um den Coordination Context in die Application Message einzufügen (vgl. Abschnitt 4.5.3). Die Transitionen 6 bis 11 dienen dazu einen Status im Automaten zu setzen, der für Monitoring verwendet werden kann und gegebenenfalls, um Blockaden von Outgoing Events aufzulösen (Transitionen 8 und 11).

4.7.5 Scope-Aktivitäten mit ICR `participating`

Für Scope-Aktivitäten mit ICR `participating` muss kein separates Verhalten modelliert werden. Sie werden ausschließlich von der BPEL-Engine gesteuert, wie Scopes mit ICR `participating` aus Fall 1 in Abschnitt 4.5 auf Seite 64.

4.7.6 Invoke-Aktivitäten mit ICR `new`

Der Automat für `<invoke>`-Aktivitäten mit ICR `new` enthält die Zustände und Transitionen aus Tabelle 4.8 auf der nächsten Seite. Die Zustände entsprechen dabei den Zuständen aus der Transitionstabelle 4.7 für `<invoke>`-Aktivitäten mit ICR `participating`. Im Vergleich zum Automaten für `<invoke>`-Aktivitäten mit ICR `participating` muss am Ende der `<invoke>`-Aktivität die Atomic Transaction dauerhaft gemacht werden (commited) werden. Es können sich beliebig viele Participants anmelden, solange sich die Aktivität im Zustand

#	Zustand	Outg. Event / Coord. Message	Bed.	Neuer Zustand	Inc. Events / Coord. Messages
1	--	EVT:Activity_Ready		Pre Message	EVT:Start_Activity
2	Pre Message	EVT:Message_Prepared		Message Prepared	EVT:Add_Message_Header (CoordContext)
3	Message Prepared	EVT:Activity_Terminated		NotCompleted	
4	Message Prepared	EVT:Scope_Handling_Fault		NotCompleted	
5	Message Prepared	EVT:Message_Modification		Executing	EVT:Continue
6	Executing	WSAT:Aborted		Aborted	WSAT:Rollback; EVT:Fault_To_Scope
7	Executing	EVT:Activity_Terminated	PNR	NotCompleted	
8	Executing	EVT:Activity_Terminated	PR	Aborted	WSAT:Rollback
9	Executing	EVT:Scope_Handling_Fault	PNR	NotCompleted	
10	Executing	EVT:Scope_Handling_Fault	PR	Aborted	WSAT:Rollback
11	Executing	EVT:Activity_Executed	PNR	Waiting	
12	Executing	EVT:Activity_Executed	PR	Waiting	WSAT:Prepare
13	Waiting	WSC:Register	PNR	Waiting	WSAT:Prepare
14	Waiting	EVT:Activity_Terminated	PNR	NotCompleted	
15	Waiting	EVT:Activity_Terminated	PR	Aborted	WSAT:Rollback
16	Waiting	EVT:Scope_Handling_Fault	PNR	NotCompleted	
17	Waiting	EVT:Scope_Handling_Fault	PR	Aborted	WSAT:Rollback
18	Waiting	WSAT:Aborted		Aborted	WSAT:Rollback; EVT:Fault_To_Scope
19	Waiting	WSAT:Prepared	APP	Prepared	EVT:Complete_Activity
20	Prepared	EVT:Activity_Terminated		Aborted	
21	Prepared	EVT:Scope_Handling_Fault		Aborted	WSAT:Rollback
22	Prepared	EVT:Activity_Complete		Commited	WSAT:Commit

Legende: Outg./Inc. Event = Outgoing/Incoming Event; Coord. Message = Coordination Message; P(N)R = Partcipant (nicht) registriert; Bed. = Bedingung; APP = Alle Participants in Prepared

Tabelle 4.8: Transitionen des Automaten für `<invoke>` mit ICR `new`

Executing befindet. Da die Mapping-Engine keine Informationen darüber besitzt, wieviele Participants sich für die Atomic Transaction anmelden, wird davon ausgegangen, dass der durch die Application Message aufgerufene Participant dafür sorgt, dass sich alle Participants anmelden, solange sich die `<invoke>`-Aktivität im Zustand *Executing* befindet. Über dieses Verhalten hat der Participant nur im Falle eines synchronen Aufrufs die Kontrolle. Andernfalls kann sich nur ein Participant registrieren, der aber einen eigenen interpositionierten Coordinator (siehe Abschnitt 2.3.5 auf Seite 17) verwenden kann, um andere Participants teilnehmen zu lassen. Für `<invoke>`-Aktivitäten gelten die selben Forderungen wie für AT-Scopes aus Abschnitt 4.7.2 auf Seite 94. Für sie dürfen keine expliziten Fault-Handler modelliert werden. Nachfolgend werden die Transitionen einzeln erläutert. Die Nummerierung der Liste referenziert auf die jeweilige Nummer der Transition in der Tabelle.

1. Transitionen 1 bis 5 werden benötigt, um den Coordination Context in die Application Message einzufügen. Dieses Verfahren wurde in Abschnitt 4.5.3 ausführlich erläutert.
2. Siehe Transition 1.
3. Siehe Transition 1.
4. Siehe Transition 1.
5. Siehe Transition 1.
6. Meldet ein Participant während des Aufrufs Aborted, wird Rollback an alle übrigen Participants gesendet. Der Automat geht in den Zustand *Aborted* über.
7. Wird die `<invoke>`-Aktivität während des Aufrufs abgebrochen (*Activity_Terminated*) und es hat sich noch kein Participant registriert, geht der Automat in den Zustand *NotCompleted* über.
8. Wird die `<invoke>`-Aktivität während des Aufrufs abgebrochen (*Activity_Terminated*) und es haben sich bereits beliebig viele Participants registriert, geht der Automat in den Zustand *Aborted* über. Alle Participants erhalten eine Rollback-Message.
9. Tritt in der `<invoke>`-Aktivität während des Aufrufs ein Fault auf (*Scope_Handling_Fault*) und es hat sich noch kein Participant registriert, geht der Automat in den Zustand *NotCompleted* über.
10. Tritt in der `<invoke>`-Aktivität während des Aufrufs ein Fault auf (*Scope_Handling_Fault*) und es haben sich bereits beliebig viele Participants registriert, geht der Automat in den Zustand *Aborted* über. Alle Participants erhalten eine Rollback-Message. Die Participants werden bereits nach Erhalt des *Scope_Handling_Fault*-Events mit Rollback benachrichtigt, weil die Aktivität durch den erzwungenermaßen vorhandenen impliziten Fault-Handler später auf jedenfall das Event *Activity_Faulted* auslöst. *Scope_Complete_With_Fault* kann nicht ausgelöst werden. Durch das frühe Senden sparen die Participants Rechenzeit.
11. Befindet sich der Automat im Zustand *Executing* und die BPEL-Engine meldet für die `<invoke>`-Aktivität *Activity_Executed*, geht der Automat in den Zustand *Waiting* über. Hat sich noch kein Participant registriert, muss auf die Registrierung gewartet werden.
12. Befindet sich der Automat im Zustand *Executing* und die BPEL-Engine meldet für die `<invoke>`-Aktivität *Activity_Executed*, geht der Automat in den Zustand *Waiting* über. Haben sich bereits Participants registriert, wird diesen eine Prepare-Message gesendet.
13. Befindet sich der Automat in Zustand *Waiting* und die erste Registrierung geht ein, wird diesem Participant sofort eine Prepared-Message gesendet.
14. Gleich wie Transition 7
15. Gleich wie Transition 8
16. Gleich wie Transition 9
17. Gleich wie Transition 10

18. Gleich wie Transition 6
19. Befindet sich der Automat im Zustand *Waiting*, von allen Participants bis auf einen wurde bereits Prepared empfangen und der letzte Participant meldet Prepared, wird die `<invoke>`-Aktivität mit *Complete_Activity* beendet. Der Automat muss im Zustand `Prepared` warten, bis die BPEL-Engine für die `<invoke>`-Aktivität das Event *Activity_Completed* sendet, da erst dann kein Fault und keine Terminierung mehr erfolgen kann.
20. Wurde von der BPEL-Engine ein *Activity_Terminated*-Event für die `<invoke>`-Aktivity abgesendet, bevor diese das *Complete_Activity*-Event empfangen hat, kommt es im Zustand *Prepared* zum Zurücksetzen der Atomic Transaction. Allen Participants wird Rollback gesendet.
21. Wurde von der BPEL-Engine ein *Scope_Handling_Fault*-Event für die `<invoke>`-Aktivity abgesendet, bevor diese das *Complete_Activity*-Event empfangen hat, kommt es im Zustand *Prepared* zum Zurücksetzen der Atomic Transaction. Allen Participants wird Rollback gesendet.
22. Meldet die BPEL-Engine *Activity_Completed* für die `<invoke>`-Aktivität, wird Commit an alle Participants gesendet. Der Automat geht in den Endzustand *Commited* über.

4.7.7 Annahmen und Einschränkungen

Die Automaten der AT-Scopes und `<invoke>`-Aktivitäten besitzen im Gegensatz zu den Coordinated Scopes aus Abschnitt 4.6 auf Seite 80 ausschließlich Transitionen, die durch Events ausgelöst werden, die Zustandsübergänge des jeweiligen AT-Scopes bzw. der `<invoke>`-Aktivität signalisieren. Zudem werden von der Mapping-Engine ausschließlich Events an die BPEL-Engine gesendet, die den jeweiligen AT-Scope bzw. die `<invoke>`-Aktivität adressieren. Daher können beliebig viele AT-Scopes und `<invoke>`-Aktivitäten in einer Prozessinstanz ausgeführt werden. Auch eine parallele Ausführung ist möglich. Wie bereits in Abschnitt 4.7.2 auf Seite 94 begründet, dürfen AT-Scopes keine AT-Scopes umschließen. Aufgrund der Äquivalenz der `<invoke>`-Aktivität mit ICR `participating` im direkt umschließenden AT-Scope mit ICR `new` zu einer `<invoke>`-Aktivität mit ICR `new`, dürfen `<invoke>`-Aktivitäten mit ICR `new` und Coordination Type aus WS-AT, nicht innerhalb eines AT-Scopes liegen.

4.8 Fall 4: Scope als WS-AtomicTransaction Participant

In Fall 2 (4.6 auf Seite 80) wurde bei eingehenden Application Messages mit Coordination Context zwischen Prozess (`createInstance="yes"`) und Scopes unterschieden. Diese Unterscheidung ist bei WS-AT nicht notwendig, da eine AT auf jedenfall nach Ablauf eines Scopes entweder erfolgreich war (commit) oder zurückgesetzt wurde und nicht über die

identische Atomic Transaction kompensiert werden kann. Wie in Fall 3 (vgl. 4.7 auf Seite 94, AT-Scopes) müssen auch hier die AT-Scope-Eigenschaften erfüllt werden. Im Folgenden wird die Vorgehensweise für Prozesse und Scopes mit ICR `new` erläutert, auch wenn nur der Bezeichner „Scope“ verwendet wird. Auf eine ausführliche Darstellung in Tabellenform wird an dieser Stelle verzichtet.

- Ein Scope wird gestartet. Die erste auszuführende Aktivität ist eine `<receive>` bzw. `<pick>`-Aktivität. Diese empfängt eine Application Message mit einem Coordination Context mit einem Coordination Type WS-AT.
- Die BPEL-Engine generiert das Event *Variable_Modification* für die Variable, die durch die `<receive>`/`<pick>`-Aktivität belegt wird. Die Variable enthält den Coordination Context, den die Mapping-Engine daraufhin für eine Registrierung für das in CP (aus den SCP) angegebene Protocol (Volatile 2PC oder Durable 2PC [OAS09a]) am Coordinator verwendet. Die `<receive>`- bzw. `<pick>`-Aktivität befindet sich im Zustand *Waiting*. Ist die Registrierung erfolgt, weist die Mapping-Engine die BPEL-Engine mit einem *Complete_Activity*-Event an, die Aktivität zu beenden.
- Die Prozessinstanz wird weiter ausgeführt.
- Tritt während der Ausführung ein Fault im Scope auf, sendet die BPEL-Engine das Event *Scope_Handling_Fault* an die Mapping-Engine. Die Mapping-Engine sendet daraufhin die Message Aborted an den Coordinator. Wird der Scope terminiert (*Activity_Terminated*-Event), sendet die Mapping-Engine ebenfalls Aborted an den Coordinator.
- Befindet sich der Scope im Zustand *Executing* oder *Waiting* und erhält die Mapping-Engine eine Rollback-Message vom Coordinator, schickt sie der BPEL-Engine ein *Fault_To_Scope*-Event für den Scope.
- Ist der Scope erfolgreich durchlaufen, wartet er im Zustand *Waiting*. Die Mapping-Engine wartet auf eine Prepare-Message vom Coordinator. Sobald diese eingeht, sendet die Mapping-Engine Prepared an den Coordinator. Der Coordinator hat nun die Möglichkeit die AT mit Rollback zurückzusetzen oder kann sie mit Commit erfolgreich abschließen. Rollback wird hierbei von der Mapping-Engine mit *Fault_To_Scope* in die Prozessinstanz propagiert, für Commit sendet sie ein *Complete_Activity*-Event an die Prozessinstanz (`<process>`).

4.8.1 Terminierung und Konsistenzprobleme

Bereits in Fall 3 trat ein Möglichkeit auf, in der BPEL-Scopes terminiert werden konnten und die AT dennoch mit Commit abgeschlossen wurde. Durch eine Erweiterung des Initiator-Protokolls (Abbildung 4.9 auf Seite 99) konnte die Inkonsistenz vermieden werden. In Fall 4 tritt dieses Problem wieder auf. Befindet sich ein Scope im Zustand *Waiting* und hat die Mapping-Engine bereits ein Prepared an den Coordinator gesendet, kann der Scope terminiert werden. Während der Scope bereits terminiert wurde, erhält die Mapping-Engine

				enthaltener CScope von			
				Fall 1 Abs. 4.5 Initiator WS-BA	Fall 2 Abs. 4.6 Participant WS-BA	Fall 3 Abs. 4.7 Initiator WS-AT	Fall 4 Abs. 4.8 Participant WS-AT
umschließender CScope von	Fall 1 Abs. 4.5	Initiator	WS-BA	Ja (1)	Ja (2)	Ja (3)	Ja (4)
	Fall 2 Abs. 4.6	Participant	WS-BA	Ja (5)	Teils (6)	Ja (7)	Ja (8)
	Fall 3 Abs. 4.7	Initiator	WS-AT	Ja (9)	Nein (10)	Nein (11)	Nein (12)
	Fall 4 Abs. 4.8	Participant	WS-AT	Ja (13)	Nein (14)	Nein (15)	Nein (16)

Tabelle 4.9: Kompatibilität koordinierter Scopes bezüglich ihrer Hierarchie

vom Coordinator Commit. Als Participant kann diese Inkonsistenz aber nicht durch das modifizierte Initiator-Protokoll behoben werden, da der Prozess nicht das Initiator Protokoll mit dem Coordinator fahren darf. Die Inkonsistenz kann nur durch eine Änderung in der BPEL-Engine vermieden werden. Es muss dafür für den AT-Scope festgelegt werden, dass er im Zustand *Waiting* nicht mehr terminiert werden darf und keine Faults mehr auftreten dürfen.

4.8.2 Annahmen und Einschränkungen

Es werden keine weiteren Annahmen und Einschränkungen getroffen.

4.9 Hierarchie koordinierter Scopes

Tabelle 4.9 auf der vorherigen Seite beschreibt, welche Coordinated Scopes (CScopes) welche anderen CScopes enthalten dürfen. Links in der Tabelle stehen jeweils die umschließenden, oben die enthaltenen CScopes. Im Folgenden werden die einzelnen Felder erläutert, die dazu jeweils mit einer Zahl versehen wurde. Diese Zahl entspricht der Zahl in der nachfolgenden Liste.

1. CScopes aus Fall 1 dürfen beliebig geschachtelt werden, da sie Scopelogik direkt nach außen als WS-BA Coordinator mappen (vgl. Abschnitt 4.5 auf Seite 64).
2. CScopes aus Fall 1 dürfen maximal einen CScope aus Fall 2 umschließen, da Fall 1 Scopelogik nach außen als WS-BA Coordinator mappt (siehe Abschnitt 4.5 auf Seite 64). Dabei muss berücksichtigt werden, dass Fall 2 nur einmal pro Prozessinstanz auftreten darf (siehe Beweis 1 auf Seite 92).
3. CScopes aus Fall 1 dürfen CScopes aus Fall 3 beliebig umschließen. CScopes aus Fall 3 haben keinen Einfluss auf den umschließenden CScope, da sie keine Transitionen besitzen, die von Outgoing Events ausgelöst werden, die bezüglich der Zustandsübergänge des umschließenden Scopes gesendet werden. Ebenfalls enthält der Automat keine Transitionen, die Einfluss auf Scopes außerhalb des AT-Scopes nehmen.
4. CScopes aus Fall 1 dürfen CScopes aus Fall 4 beliebig umschließen. Die Begründung ist die identische wie in Punkt 3.
5. CScopes aus Fall 2 dürfen CScopes aus Fall 1 umschließen (siehe Abschnitt 4.6 auf Seite 80).
6. CScopes aus Fall 2 dürfen weitere CScopes aus Fall 2 nur enthalten, wenn sichergestellt ist, dass in der Prozessinstanz nur eine CScopeinstanz aus Fall 2 von außen koordiniert wird. Durch `optional="true"` aus den SCP (vgl. Definition 3.3.6 auf Seite 53) werden nicht alle CScopeinstanzen aus Fall 2 von außen koordiniert (siehe Abschnitt 4.6 auf Seite 80).
7. CScopes aus Fall 2 dürfen CScopes aus Fall 3 beliebig umschließen. CScopes aus Fall 3 haben keinen Einfluss auf den umschließenden CScope (siehe Abschnitt 4.5 auf Seite 64).
8. CScopes aus Fall 2 dürfen CScopes aus Fall 4 beliebig umschließen. CScopes aus Fall 4 haben keinen Einfluss auf den umschließenden CScope (siehe Abschnitt 4.5 auf Seite 64).
9. CScopes aus Fall 3 dürfen CScopes aus Fall 1 beinhalten. Die Close-Message, die beim *Instance_Complete*-Event versendet wird (siehe Abschnitt 4.5 auf Seite 64), könnte dann bereits nach dem *Activity_Completed*-Event für die umschließende CScopeinstanz des AT-Scopes gesendet werden, denn nach Abschnitt 4.7.2 auf Seite 94 darf der Compensation-Handler der AT-Scopes keine `<compensate>`- und `<compensateScope>`-Aktivitäten enthalten.

10. CScopes aus Fall 3 dürfen keine weiteren CScopes aus Fall 2 enthalten, da die für Fall 3 definierten AT-Scopes innerhalb des Compensation-Handlers keine `<compensate>`- und `<compensateScope>`-Aktivitäten erlauben (siehe Abschnitt 4.7.2 auf Seite 94). CScopes, die mit WS-BA koordiniert werden (Fall 2), können nach Abschluss des AT-Scopes folglich nicht mehr kompensiert werden. Dies verstößt gegen die Annahmen aus Abschnitt 4.6.5 auf Seite 92.
11. AT-Scopes dürfen keine weiteren AT-Scopes enthalten, siehe Abschnitt 4.7.2 auf Seite 94.
12. AT-Scopes dürfen keine weiteren AT-Scopes enthalten, siehe Abschnitt 4.7.2 auf Seite 94.
13. Siehe 9.
14. Siehe 10.
15. AT-Scopes dürfen keine weiteren AT-Scopes enthalten, siehe Abschnitt 4.7.2 auf Seite 94.
16. AT-Scopes dürfen keine weiteren AT-Scopes enthalten, siehe Abschnitt 4.7.2 auf Seite 94.

5 Realisierung

In Kapitel 4 auf Seite 59 wurden Mapping-Verfahren vorgestellt, die darlegen, wie ein Workflow-Managementsystem mit auf WS-Coordination basierenden Protokollen Participants koordinieren und selbst als Participant koordiniert werden kann. Es wurden dazu Zustandsautomaten beschrieben, die – abhängig von deren Status – verschiedene Aktionen ausführen und Nachrichten entweder als Event an das Workflowsystem oder als Coordination Message an einen Coordinator oder Participant senden. In diesem Kapitel soll nun vorgestellt werden, wie die Mapping-Engine implementiert wird. Es wird geklärt, welche Komponenten dazu benötigt werden, wie diese mit anderen Komponenten interagieren und wie das System sowohl persisitent, als auch konsistent gehalten wird.

Abbildung 5.1 zeigt die Architektur der Mapping-Engine. Sie basiert als message-basiertes System auf der Java Platform Enterprise Edition (JEE) und wird dementsprechend in einem

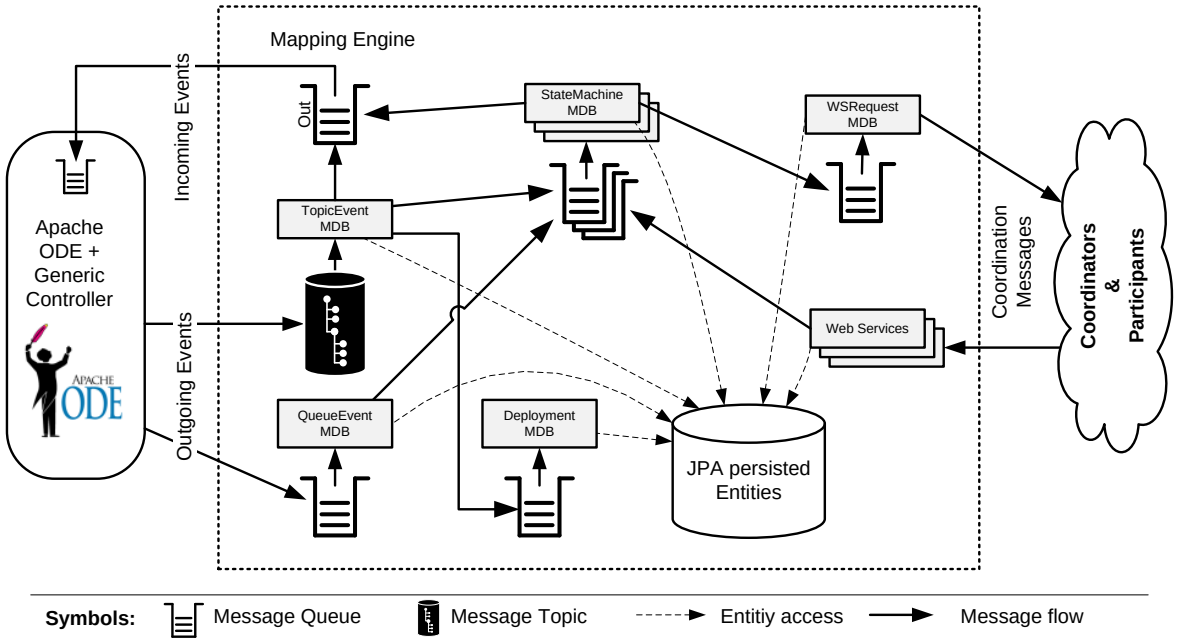

Abbildung 5.1: Architektur der Mapping Engine

JEE-Application-Server ausgeführt. Es wird der Application Server *Glassfish* [Ora10] verwendet. Als Workflow-Managementsystem wird Apache ODE eingesetzt. Die Wolke steht für beliebige Participants und Coordinators, die von der Mapping-Engine koordiniert werden bzw. die Mapping-Engine koordinieren. Für das Testen der Implementierung wird der Coordinator TracG (siehe Abschnitt 3.2.4 auf Seite 50) verwendet.

Die Mapping-Engine beinhaltet folgende Message Driven Beans (MDB):

- **TopicEventMDB:** Sie empfängt die Events, die vom Pluggable Framework auf dem Topic publiziert werden (siehe Abschnitt 3.1.1 auf Seite 35). Sie versieht sie nach dem *Content-Enricher*-Pattern [HW03] mit der ID einer CScopeinstanz und leitet sie nach dem *Message-Router*-Pattern [HW03] an die StateMachineMDB weiter (siehe Abschnitt 5.2 auf Seite 112). Die TopicEventMDB löst außerdem Blockaden auf, die in den Prozessinstanzen beim Versenden von Outgoing Events gesetzt werden und keiner Behandlung durch einen Automaten bedürfen. *Process_Deployed*-Events werden an die DeploymentMDB weitergeleitet (siehe Abschnitt 5.6 auf Seite 117).
- **QueueEventMDB:** Die QueueEventMDB empfängt Nachrichten, die der Generic Controller nur an die Mapping-Engine als Custom Controller sendet (siehe Abschnitt 3.1.1 auf Seite 35). Sie wird ausschließlich für die Registrierung der Mapping-Engine als Custom Controller benötigt. Die genaue Vorgehensweise dazu wird in [Ste08] erläutert.
- **StateMachineMDB:** Hierbei handelt es sich um eine Menge von MDB. Jeweils eine MDB implementiert einen der Automatentypen, wie sie in Abschnitt 4 auf Seite 59 beschrieben wurden (siehe Abschnitt 5.5 auf Seite 117).
- **WSRequestMDB:** Diese MDB wird von den Automaten benutzt, um Coordination Messages an Coordinators und Participants zu senden. Sie übersetzt dabei JMS-Nachrichten in Web Service Messages (siehe Abschnitt 5.4 auf Seite 116).
- **DeploymentMDB:** Sie erhält *Process_Deployed*-Events und generiert aus dem mitgesendeten Prozessmodell mit Hilfe von Policy Attachments CScope- und CScopeInstance-Entitäten, die in der Datenbank abgespeichert werden (siehe Abschnitt 5.6 auf Seite 117).

Im Message Queueing System des Application Servers wird eine Outgoing Queue für den Generic Controller angelegt. In diese werden Nachrichten für den Generic Controller abgelegt, die dem Transaktionsmanagement unterliegen. Zusätzlich wird für jeden PortType aus dem in WS Coordination [OAS09c] definierten Registration Service und den in WS-BA [OAS09b] und WS-AT [OAS09a] definierten Coordination Protokollen eine Klasse erstellt, die die entsprechenden Operationen als JAX-WS Web Service bereitstellt (siehe 5.3 auf Seite 115). Die Coordination Messages aus den Coordination Protokollen werden dabei in JMS-Nachrichten umgewandelt und in die Queue der zuzuordnenden StateMachineMDB gestellt (siehe Abschnitt 5.3 auf Seite 115).

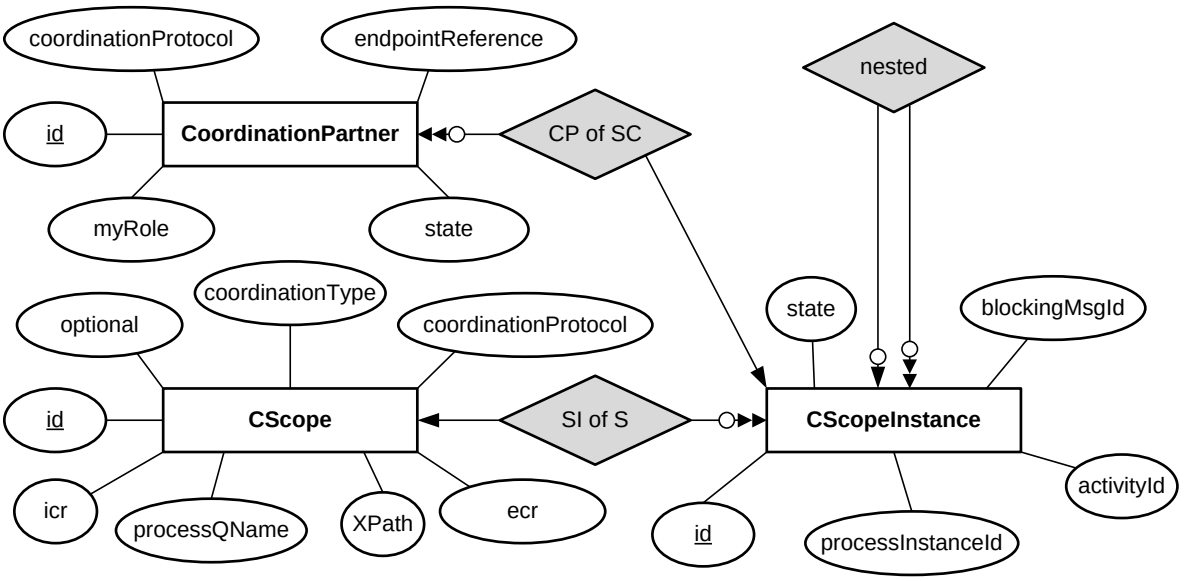

Abbildung 5.2: Entity-Relationship-Modell für die Mapping-Engine

5.1 Datenmodell und Datenverwaltung

Abbildung 5.2 stellt ein Entity-Relationship-Modell (E/R-Modell) dar. Es enthält alle Entitäten, die für die Realisierung der Mapping-Engine benötigt werden. Die Entitäten werden über JPA auf Javaklassen gemappt und verwaltet. Es handelt sich um ein minimales, soweit wie nötig normalisiertes Modell, mit den folgenden Entitäten.

CScope Ein CScope repräsentiert einen statischen Coordinated Scope (CScope). Statisch bedeutet in diesem Zusammenhang, dass es sich nicht um einen CScope zur Laufzeit handelt, der sich in einem Status befindet, sondern um den CScope, wie er im Prozessmodell beschrieben wird. Ein CScope enthält die in Definition 3.3.6 auf Seite 53 eingeführten Scope Coordination Properties (SCP). Eine Entität CScope wird benötigt, um während des Startens einer CScopeinstanz (*Activity_Ready*-Event) den Automatentypen des zu erzeugenden Automaten zu bestimmen. Während des Deployments eines Prozessmodells wird die BPEL-Beschreibung nach CScopes durchsucht und für jeden CScope eine entsprechende CScope-Entität erzeugt (siehe Abschnitt 5.6 auf Seite 117). Das Attribut `XPath` wird benötigt, um bei der Instanziierung eines CScopes der Instanz den passenden CScope und Events für `<process>`-Aktivitäten zuordnen zu können (siehe Abschnitt 5.2 auf der nächsten Seite). Die Zuordnung eines CScopes zu einem Prozessmodell erfolgt über das Attribut `processQName`. Da für Prozessmodelle keine weiteren Daten persistent gehalten werden müssen, wird keine eigene Entität benötigt.

CScopeInstance CScopeInstance repräsentiert einen CScope während der Ausführung (CScopeinstanz). Die Relation SI of S legt dabei fest, von welchem CScope der laufende Scope instanziiert wurde. Diese Relation wird benötigt, da der CScope den Automatentyp für eine CScopeinstanz bestimmt. Über das Attribut state wird der Zustand des Automaten für die CScopeinstanz gespeichert. CScopeinstanzen, die ausschließlich von der BPEL-Engine koordiniert werden, befinden sich immer im Zustand *Not Coordinated*. Da das Pluggable Framework bzw. Apache ODE keine Activity-ID für das `<process>`-Element vergibt, wird als Primärschlüssel das Attribut id verwendet. Das Pluggable Framework verwendet für Incoming Events, die eine Blockade auflösen, das Attribut replyToMsgId. Dieses dient der Korrelation einer Antwort. Da ein Automat in bestimmten Situationen (siehe Kapitel 4 auf Seite 59) auf weitere Coordination Messages warten muss, bevor er eine Blockade auflösen darf und sich währenddessen der Zustand einer Scopeinstanz lediglich in der Datenbank befindet, wird das Attribut blockingMsgId eingeführt. In einer CScopeinstanz kann nur eine Blockade gleichzeitig vorliegen, daher muss hierfür keine separate Entität eingeführt werden. Um Events in Apache ODE korrekt zuordnen zu können, verlangt das Pluggable Framework für Incoming Events, wie beispielsweise *Fault_To_Scope*, im Event ein Attribut `processID`. Dieses Attribut wird für CScopeinstanzen übernommen, wird aber in processInstanceId umbenannt, da die Bezeichnung `processID` den Leser verwirren kann. Die Instanzen eines Prozessmodells als eigene Entität zu speichern ist nicht nötig, da lediglich die ID einer Prozessinstanz für Korrelation benötigt wird und keine weiteren ihrer Daten. In den Automaten in Abschnitt 4 auf Seite 59 sind Transitionen möglich, in welchen als Aktion ein Fault in einen umgebenden Scope gesendet werden muss. Um dessen ID zu erhalten, wird die Relation nested eingeführt. Außerdem erlaubt die nested-Relation das Bestimmen des Effective Scopes aus Definition 3.3.9 auf Seite 56.

CoordinationPartner: Eine CScopeinstanz tauscht während der Abarbeitung Coordination Messages mit Coordinators und Participants aus. CoordinationPartner repräsentiert einen Participant oder Coordinator. Je nach der eigenen Rolle der Mapping-Engine wird dementsprechend das Attribut myRole mit den Werten `coordinator` bzw. `participant` belegt. In Abschnitt 4.7.3 auf Seite 98 wurde ein Verfahren für externe Koordinierung vorgestellt. In diesem Fall könnte, falls implementiert, myRole den Wert `initiator` annehmen. In den in Abschnitt 4 auf Seite 59 vorgestellten Automaten interagiert (Relation CP of SC) eine CScopeinstanz immer mit beliebig vielen Participants oder mit einem Coordinator. Jeder Participant und Coordinator besitzt dabei einen Zustand state, der mit einem der im entsprechenden Coordination Protocol definierten Status identisch ist.

5.2 Zuordnung der BPEL Events

Das Pluggable Framework publiziert Events, die an alle Custom Controller gehen, an ein Topic und sendet Events, die an einzelne Custom Controller adressiert werden, an eine

Listing 5.1: Abfrage für die Zuordnung eines *Activity_Terminated*-Events einer Prozessaktivität

```
SELECT csi.id
FROM CScopeInstance csi, CScope cs
WHERE csi.processInstanceId = 0
  AND cs.processQName = "http://www.example.com/processes:p1"
  AND cs.XPath = "/process"
  AND csi.cscopeId = cs.id
LIMIT 1;
```

temporäre Queue (siehe Abschnitt 3.1.1 auf Seite 35). Im Pluggable Framework enthalten Incoming Events, die Outgoing Events an spezifische Custom Controller auslösen, eine Referenz auf eine temporäre Queue, über die die Outgoing Events als Antwort gesendet werden. Da es sich bei der Queue, mit der die QueueEventMDB verbunden ist, um eine statische Queue handelt, wird im Folgenden davon ausgegangen, dass das Pluggable Framework im Feld *ReplyTo* die Angabe einer statischen Queue unterstütze. Das Pluggable Framework muss entsprechend angepasst werden.

Die TopicEventMDB subskribiert auf das Topic des Generic Controllers. Eingehende Outgoing Events analysiert die TopicEventMDB und leitet sie weiter. Die TopicQueueMBD muss dabei verschiedene Eventtypen behandeln können. Wird ein Outgoing Event für eine bereits existierende CScopeinstanz an eine StateMachineMDB weitergesendet, verwendet die TopicEventMDB eine serialisierbare Wrapperklasse `EventWrapper`, die die Attribute `event` vom Typ `MessageBase` und `cScopeIntanceId` von Typ `Long` enthält. `MessageBase` ist die oberste Klasse aller Outgoing Events des Pluggable Frameworks [Steo8]. Die Attribute eines Objektes der Klasse `EventWrapper` werden mit dem Outgoing Event und der ID der CScopeinstanz befüllt. Das Objekt wird an die StateMachineMDB gesendet.

Im Folgenden werden die verschiedenen Eventtypen aufgelistet und die Attribute aufgezählt, die für die Zuordnung benötigt werden. Tabelle 5.1 auf der nächsten Seite stellt alle Möglichkeiten dar, welche Eventtypen welchen Automatentypen zugeordnet werden können.

- **Activity_Terminated**: Dieses Event wird in allen Automaten nur für die vom Automaten koordinierte CScopeinstanz behandelt (vgl. Kapitel 4 auf Seite 59). Bei der Zuordnung des Events zu einer CScopeinstanz müssen zwei Fälle unterschieden werden: Ist processInstanceId null, dann handelt es sich um eine `<process>`-Aktivität [Steo8]. Die Zuordnung erfolgt dann über die Attribute `processQName` und `XPath` des CScopes. Dabei muss `processQName` mit `processName` aus dem Event übereinstimmen und `XPath` `"/process"` sein. Die zugehörige JPQL-Abfrage wird in Listing 5.1 definiert[1]. Ist pro-

[1] `LIMIT 1` wird hier als Optimierung gesetzt, da die Abfrage auch ohne Limit nur ein Resulat liefern würde.

Outgoing Events (alphabetisch)	**Fall 1, WS-BA ECR initiator**	<Scope>, <process> ICR new	<invoke> ICR participating	<invoke> ICR new	**Fall 2, WS-BA ECR participant**	<process> ICR new	<scope> ICR new	**Fall 3, WS-AT ECR initiator**	<scope>, <process> ICR new (intern)	<invoke> ICR participating	<invoke> ICR new	**Fall 4, WS-AT ECR participant**	<scope>, <process> ICR new
Acitivity_Terminated		ja	ja	ja		ja	ja		ja	ja	ja		ja
Activity_Executed		ja	ja	ja		ja	ja		ja	ja	ja		ja
Activity_Faulted		--	--	--		ja	ja		--	--	--		--
Activity_Ready		neu	neu	neu		neu	neu		neu	neu	neu		neu
Instance_Completed		--	ja	ja		--	--		--	--	--		--
Message_Modification		--	ja	ja		--	--		--	ja	ja		--
Message_Prepared		--	ja	ja		--	--		--	ja	ja		--
Process_Deployed		--	--	--		--	--		--	--	--		--
Scope_Compensated		ja	--	--		--	ja		--	--	--		--
Scope_Compensating		ja	ja	ja		--	ja		--	--	--		--
Scope_Compl._W._Fault		--	--	--		ja	ja		--	--	--		--
Scope_Handling_Fault		ja	ja	ja		--	ja		ja	ja	ja		ja
Variable Modification		--	--	--		ja	ja		--	--	--		ja

Legende: ja = StateMachineMDB benötigt Eventtyp; neu = StateMachineMDB erzeugt CScopeInstance

Tabelle 5.1: Zuordnungsmöglichkeiten zwischen Events und Automatentypen

cessInstanceId nicht null, kann die richtige CScopeinstanz anhand der `activityID` des Events zugeordnet werden.

- **Activity_Executed:** Das Event für die eigene CScopeinstanz wird von allen Automaten benötigt. Zusätzlich wird es im Automat für koordinierte <scope>-Aktivitäten mit ECR `participating` aus Fall 2 behandelt (siehe auch Abschnitt 4.6.3 auf Seite 87). Die Zuordnung zu Automaten, deren CScopeinstanz das Event betrifft, erfolgt wie für das *Activity_Terminated*-Event. Eine weitere Abfrage sucht nach CScopeinstanzen der identischen Prozessinstanz, deren CScope ECR `participating`, ICR `new` und einen Coordination Type aus WS-BA enthält.
- **Activity_Faulted:** wie *Activity_Terminated*.
- **Activity_Ready:** Dieses Event wird keiner CScopeinstanz zugeordnet, sondern es wird eine CScopeinstanz erzeugt. Dabei wird ein neuer Eintrag in der Tabelle mit den entsprechenden Daten, je nach Fall, erzeugt. Kann ein *Activity_Ready*-Event einer `<process>`-Aktivität über `processQName` keinem CScope zugeordnet werden, so wurde

das *Process_Deployed*-Event des Prozessmodells nicht behandelt (siehe Abschnitt 5.6 auf Seite 117).

- **Instance_Completed:** Dieses Event wird für die Automaten von CScopeinstanzen benötigt, die innerhalb der selben Prozessinstanz liegen, deren CScopes `<invoke>`-Aktivitäten mit ECR `initiator` sind und deren Participants mit WS-BA koordiniert werden (siehe Abschnitte 4.5.2 auf Seite 68 und 4.5.3 auf Seite 71). Diese Kriterien können auf mehrere CScopeinstanzen zutreffen, für jede ist das Event an die passende StateMachineMDB weiterzuleiten.
- **Message_Modification:** wie *Activity_Terminated*, jedoch ist `processInstanceId` niemals null, da es sich um eine `<invoke>`-Aktivität handelt.
- **Message_Prepared:** wie *Activity_Terminated*, jedoch ist `processInstanceId` niemals null, da es sich um eine `<invoke>`-Aktivität handelt.
- **Process_Deployed:** Dieses Event wird an die *DeploymentMDB* (siehe auch 5.6 auf Seite 117) weitergesendet. Der Wrapper-Klasse wird nicht benötigt, da keine ID einer CScopeinstanz mitgesendet werden muss.
- **Scope_Compensated:** wie *Activity_Terminated*, jedoch ist `processInstanceId` niemals null, da es sich um keine `<process>`-Aktivität handeln kann, da diese nach der BPEL-Spezifikation [OAS07] keinen Compensation-Handler besitzt.
- **Scope_Compensating:** wie *Activity_Terminated*, jedoch ist `processInstanceId` niemals null, da es sich um keine `<process>`-Aktivität handeln kann, da diese nach der BPEL-Spezifikation [OAS07] keinen Compensation-Handler besitzt.
- **Scope_Complete_With_Fault:** wie *Activity_Terminated*.
- **Scope_Fault_Handling:** wie *Activity_Terminated*.
- **Variable_Modification:** wie *Activity_Terminated*, jedoch ist `processInstanceId` niemals null, da es sich um eine `<receive>`- oder `<pick>`-Aktivität handelt.

Eine Zuordnung eines Events zu einem bestimmten Automatentypen (in Form einer StateMachineMDB) wird jeweils anhand der SCP vorgenommen, die sich als Attribute in der Entität des der CScopeinstanz über Relation SI of S zugeordneten CScope wiederfinden.

Ein Outgoing Event, das eine Blockade in der BPEL-Engine zur Folge hat (siehe Abschnitt 3.1.1 auf Seite 35), wird von der EventTopicMDB durch Senden eines Incoming Events aufgelöst, wenn das Outgoing Event keiner CScopeinstanz zugeordnet werden kann, die von einem Automaten koordiniert wird.

5.3 Web Services der Mapping-Engine und Zuordnung

Für WS-BA und WS-AT wird jeweils eine Klasse erstellt, die per JAX-WS einen Web Service anbietet, der die Operationen zum Empfang aller in den jeweiligen Coordination

Protokollen definierten Messages anbietet. Beim Aufruf eines dieser (Coordination) Web Services, wird ein Objekt der serialisierbaren Klasse `AssignedCoordinationMessage` erzeugt. Dieses Objekt enthält den Namen der Coordination Message, der als WS-Addressing *Action* [W3C04] beim Aufruf übermittelt wird. Anhand des Namen wird im zugehörigen Automaten die passende Transition bestimmt. Um den passenden Automaten zu bestimmen, enthält `AssignedCoordinationMessage` die ID des Coordination Partners aus Abschnitt 5.1, die in den Reference Properties der Message enthalten ist (vgl. Abschnitt 4.3 auf Seite 61). Da ein Coordination Partner genau einer CScopeinstanz zugeordnet ist, kann die StateMachineMDB anhand dieser ID die zu koordinierende CScopeinstanz ermitteln. Wie in 5.2 auf Seite 112 wird anhand von `ecr`, `icr` und `coordinationType` des zugehörigen CScopes bestimmt, an welche StateMachineMDB dieses Objekt gesendet werden muss.

Ist der CScope der CScopeinstanz des Participants in der Internal Coordination Role (ICR) `participating` und der External Coordination Role (ECR) `initiator` eine `<invoke>`-Aktivität, so muss wie folgt unterschieden werden:

- Koordiniert die CScopeinstanz mit einem Coordination Type aus WS-BA (Fall 1), so behandelt der Automat der CScopeinstanz die Coordination Message selbst, solange er sich in der Ausführung oder Kompensierung befindet. Andernfalls werden die Coordination Messages vom Automaten des Effective Scope (siehe Definition 3.3.9 auf Seite 56) behandelt.
- Koordiniert die CScopeinstanz mit einem Coordination Type aus WS-AT (Fall 3), so werden alle Coordination Messages vom Automaten des Effective Scopes behandelt.

Die Funktion *children* aus [KML08], die in der Definition des Effective Scopes die direkt enthaltenen Scopes zurückliefert, wird über die Relation nested (siehe Datenmodell in Abschnitt 5.1 auf Seite 111) realisiert.

Für die Registrierung von Participants, sowie für die Entgegennahme von `RegisterResponse`-Messages wird je ein weiterer Web Service benötigt. Die Messages Register und RegisterResponse werden jeweils an die zugeordneten StateMachineMDB weitergeleitet. Die Zuordnung erfolg wie in Abschnitt 4.3 auf Seite 61 beschrieben. Zur Erinnerung: Register-Messages werden jeweils vom Automat der CScopeinstanz behandet, wobei in den Transitionstabellen aus Kapitel 4 auf Seite 59 für eine bessere Übersicht Messages, die keinen Transitionen im Automaten auslösen, folglich auch Register-Messages, solange diese keinen Zustandwechsel implizieren, nicht aufgeführt werden.

5.4 Senden von Coordination Messages

Für das Aufrufen von Web Servies der Coordination Partners wird die WSRequestMDB verwendet. In ihrer Queue erhält sie Messages, die als Coordination Messages an Coordinators

und Participants weitergeleitet werden. Messages die in die Queue der WSRequestMDB eingestellt werden, enthalten die Endpunktreferenz des Coordination Partners.

Eine Nachricht an die Queue der MDB enthält als einziges Element die Coordination Message. Diese enthält bereits die Endpunktreferenz des Coordination Partners. Anhand des WS-Addressing-Elements `Action` [W3C04] wird die aufzurufende Operation bestimmt.

5.5 Automaten

Für jeden der Automatentypen, die in Kapitel 4 auf Seite 59 beschrieben wurden, wird eine MDB implementiert. Die Automatentypen wurden in Kapitel 4 ausführlich erläutert. Die textuell und tabellarisch beschriebenen Transitionen werden für die Implementierung in Java umgesetzt.

5.6 Deployment von Prozessen

Das Event `Process_Deployed` enthält das serialisierte BPEL-Prozessmodell und die zugehörigen WSDL-Dateien. Wird es durch die `TopicEventMDB` empfangen, wird es in die Queue der DeploymentMDB weitergeleitet. Die DeploymentMDB führt einen Durchlauf durch das im Event enthaltene BPEL-Prozessmodell durch und erzeugt für `<process>`-, `<scope>`- und `<invoke>`-Aktivitäten je eine CScope-Entität. Sie setzt dabei implizite Scope Coordination Properties (SCP) nach den in Abschnitt 3.3.4 auf Seite 56 festgelegten Regeln. Um an explizite SCP zu kommen, ruft die MDB einen Policy Service auf, der für deployte Prozessmodelle anhand der QNames des Prozessmodells die in Abschnitt 3.4 auf Seite 57 beschriebenen Policy-Attachments zur Verfügung stellt. In dieser Implementierung wird als `expressionLanguage` ausschließlich XPath 1.0 unterstützt.

Im Eventmodell nach [Ste08] ist keine Blockade oder Ausführsperre für das Event *Process_Deployed* vorgesehen. Das kann theoretisch dazu führen, dass die erste Instanz bereits ausgeführt wird, bevor das Event von `BPELParsingMDB` empfangen wurde und diese die CScopes aus dem Prozessmodell extrahiert hat. Es wird in dieser Arbeit davon ausgegangen, dass dieser Fall nicht eintritt.

5.7 Transaktionales Verhalten

Die Implementierung verwendet das Transaktionsmanagement des Application-Servers. Transaktionen sind container-managed [Gon09]. Die WSRequestMDB darf Messages erst an Web Services weiterleiten, wenn die zugehörige Transaktion erfolgreich war. Dadurch wird

vermieden, dass Coordination Partners Messages erhalten, die später zurückgezogen werden müssten. Ebenso werden Messages der Outgoing Queue erst an den Generic Controller gesendet, wenn die zugehörige Transaktion erfolgreich war. Durch das Verwenden des Transaktionsmanagements werden Race Conditions vermieden, die zu Lost Update in den CScopeinstanzen führen würden, wenn beispielsweise gleichzeitig ein Outgoing Event und eine Coordination Message für diese CScopeinstanz eintreffen.

6 Verwandte Arbeiten

In diesem Kapitel werden Arbeiten vorgestellt, die ebenfalls über die Verbindung von WS-Coordination mit BPEL berichten. Dabei gibt es insbesondere mit [TMW+04] und [TKM04] Überschneidungen. Im Folgenden werden diese Arbeiten vorgestellt.

6.1 Transaction Policies für Service-oriented Computing

Tai et al. beschreiben in [TMW+04] Transaction Policies. Transaction Policies können entweder für Clients (*Client Transaction Policies*) oder für Provider (*Provider Transaction Policies*) festgelegt werden. Eine Client Transaction Policy definiert, welche Anforderungen und Fähigkeiten ein Client bezüglich Reliable Messaging und kompensations-basierten Transaktionen beim Aufruf eines Providers besitzt. Eines Provider Transaction Policy bestimmt Anforderungen und Fähigkeiten eines Provider, mit denen Aufrufe behandelt werden müssen bzw. behandelt werden können. Werden die beiden Policies miteinander verglichen und wird innerhalb der jeweiligen Anforderungen und Fähigkeiten eine Kombination gefunden, die beiden Seiten entspricht, wird diese Kombination *Transaction Coupling Mode* genannt. Tai et al. beschreiben den Transaction Coupling Mode als einen Vertrag, der zwischen Provider und Consumer abgeschlossen wird. Vertragsgegenstand ist der zu verwendende Transaktionsmechanismus. Um Transaction Policies mit den Partnern zu assoziieren, schlagen Tai et al. die Mechanismen aus WS-Policy [W3C07c] vor. Für *Direct Transaction Processing*, beispielsweise WS-AT [OAS09a], *Queued Transaction Processing*, beispielsweise WS-RM [IBM05], sowie *Compensation-based Transation Processing*, beispielsweise WS-BA [OAS09b], werden Schemen für Client- und Provider Transaction Policies definiert. Tai et al. zeigen, wie sich mit WS-Policy Attachment Policies an die Elemente aus WSDL und BPEL anhängen lassen. Sie erläutern Verfahren, wie Transaction Coupling Modes während des Deployments und während der Laufzeit berechnet werden können (Matching) und beschreiben einen Prototypen, der die Verfahren implementiert.

In dieser Arbeit wird die Idee des Policy Attachments von Tai et al. übernommen. Es wird ein eigenes XML-Element `CoordinatedScope` definiert (vgl. Abschnitt 3.4 auf Seite 57), das speziell für die in Abschnitt 3.3 auf Seite 53 erläuterten Coordinated Scopes definiert wurde. Die Policies mit `<CoordinatedScope>` werden in dieser Arbeit ausschließlich an Scopealikes, also an `<process>`- , `<scope>`- und `<invoke>`-Aktivitäten angehängt. In [TMW+04] werden

Scopes nicht explizit unterschieden. Zusätzlich werden Policies in [TMW+04] an BPEL-Partnerlinks angehängt.

6.2 Komposition von Coordinated Web Services

Tai et al. beschreiben in [TKM04] Ansätze über die Komposition von BPEL Services in Kombination mit dem Framework WS-Coordination [OAS09c], insbesondere mit den Standards WS-Atomic Transaction [OAS09a] und WS-Business Activity [OAS09b]. Tai et al. verfolgen dabei zwei Ziele:

- ein prozessbasiertes *Web Service Composition Model* anzubieten, das – im Gegensatz zu vorhandenen Modellen mit unzureichender Unterstützung für externe Koordinierung – die Integration verschiedener Coordination Protokolle unterstützt.
- dabei die Komposition von Web Services mit vorhandenen Web Services Standards zu erreichen, ohne neue Kompositionstechniken einführen zu müssen.

Als Motivation führen Tai et al. die Notwendigkeit auf, *Production Workflows* mit Web Services zu realisieren. Production Workflows sind Workflows, die Geschäftslogik definieren und implementieren und durch nicht-funktionale Anforderungen wie Zuverlässigkeit, Transaktionen, usw. ergänzt werden. Die nicht-funktionalen Anforderungen sollen dabei durch Middleware realisiert werden.

Um die Teilnahme an Koordinationsaktivitäten von Web Services und BPEL-Prozessmodellen zu ermöglichen, schlagen Tai et al. die Verwendung von Policy Attachments [W3C07b] und den in [TMW+04] (vgl. Abschnitt 6.1 auf der vorherigen Seite) vorgestellten *Coordination Policies* vor.

In BPEL-Prozessen empfehlen Tai et al. diese Policy Attachments an BPEL-Partnerlinks und -Scopes anzuhängen, und so entweder einen Coordination Context mit einzelnen Partnern bzw. mit mehreren Partnern (Coordinated Scope) zu teilen. Sie deuten Ansätze an, wie die Koordination realisiert werden kann, ohne dabei auf genaue Verfahren einzugehen.

In dieser Arbeit werden Scopes und die scopeähnlichen Aktivitäten `<process>` und `<invoke>` per Attachment zu Coordinated Scopes erweitert. Auf das Attachment von Partnerlinks wird dabei nicht eingegangen. Während Tai et al. die Annahme setzen, dass Coordinated Scopes nicht geschachtelt werden dürfen, wird in dieser Arbeit gezeigt, wie und unter welchen Voraussetzungen dies möglich ist (vgl. Abschnitt 4.9 auf Seite 106). Dabei wird insbesondere auf die Wechselwirkungen zwischen WS-AT und WS-BA eingegangen.

6.3 Externalisierung von BPEL Scopes mit extended WS-BA

Pottinger et al. zeigen in [PML07], wie die komplette transaktionale Verwaltung von BPEL aus der BPEL-Maschine auf einen Coordinator übertragen werden kann. Sie zeigen damit eine weitere Möglichkeit auf, wie Participants mit WS-BA koordiniert werden können. In dieser Arbeit wird dafür die in Abschnitt 4.6 auf Seite 80 vorgestellte Verfahrensweise erläutert.

7 Zusammenfassung und Ausblick

In der Vorliegenden Arbeit wurde gezeigt, wie eine BPEL-Workflowmaschine und insbesondere Apache ODE um die Unterstützung von auf WS-Coordination basierten externen Transaktionen erweitert werden kann. Dazu wurden zunächst Grundlagen geschaffen, die für eine Realisierung benötigt wurden. Es wurden Möglichkeiten vorgestellt und verglichen, wie Informationen über Prozesse, Instanzen und Aktivitäten aus der Workflow-Engine extrahiert werden können und wie der Ablauf selbiger beeinflusst werden kann (Abschnitt 3.1 auf Seite 26). In Abschnitt 3.2 auf Seite 40 wurden Implementierungen von Coordination Middleware verglichen.

Als nächstes wurden die scopeähnlichen Aktivitäten `<process>`, `<scope>` und `<invoke>` (Scopealikes) formal um Scope Coordination Properties zu Coordinated Scopes ergänzt (Abschnitt 3.3 auf Seite 53). Dabei wurden die Internal Coordination Role und die External Coordination Role eingeführt, die das Verhalten der Coordinated Scopes gegenüber des umschließenden Scopes bzw. des Coordination Partners beschreiben. Es wurde ein Mechanismus erläutert, wie Scope Coordination Properties mittels Policy Attachment an Scopealikes angefügt werden können (Abschnitt 3.4 auf Seite 57) und implizite Scope Coordination Properties für Scopealikes definiert, die nicht explizit per Policy mit Scope Coordination Properties versehen wurden.

Im folgenden Schritt wurden Mappingverfahren zwischen BPEL und den WS-Coordiation-Protokollen ausführlich beschrieben (Kapitel 4 auf Seite 59). Dazu wurden vier Fälle definiert:

1. Scopes, die als Initiator Participants mit WS-Business Activity (WS-BA) koordinieren (Abschnitt 4.5 auf Seite 64)
2. Scopes, die als WS-BA-Participants von außen koordiniert werden (Abschnitt 4.6 auf Seite 80)
3. Scopes, die als Initiator Participants mit WS-Atomic Transaction (WS-AT) koordinieren (Abschnitt 4.7 auf Seite 94)
4. Scopes, die als WS-AT-Participants von außen koordiniert werden (Abschnitt 4.8 auf Seite 103)

Dabei wurde detailliert auf die verschiedenen Aktivitäten und ihr Internal Coordination Roles in den Fällen eingegangen und jeweils Automaten erläutert, die die Mappingverfahren realisieren. Es wurde untersucht, welche Beschränkungen dabei hingenommen werden

müssen und welche Wechselwirkungen beim Nesting verschiedenartiger Coordination Types auftreten (Zusammenfassung siehe Abschnitt 4.9 auf Seite 106). Insbesondere wurde gezeigt, dass in einer Instanz die WS-BA-Erweiterung Participant Triggered Compensation aus [KML09] nicht beliebig eingesetzt werden kann (vgl. Beweis 1 auf Seite 92).

Schließlich wurde in Kapitel 5 auf Seite 109 dargelegt, wie ein Prototyp implementiert werden kann und die dafür notwendigen Entitäten erläutert.

Ausblick Es wurde in dieser Arbeit davon ausgegangen, dass ein Scope jeweils nur einen Coordination Type unterstützt, der bereits zur Zeit des Deployments feststeht. In [TMW+04] und [TKM04] werden Ansätze vorgestellt, wie dies auch während des Lebenszyklus einer Prozessinstanz zwischen der Workflowmaschine und ihren Coordination Partnern ausgehandelt werden kann. Auf diesen Ansätzen basierend kann untersucht werden, inwiefern sie auf diese Arbeit übertragen werden können und wie z. B. Coordination Contexts mit verschiedenen Coordination Types von ein und demselben Scope behandelt werden können.

In Abschnitt 4.4 auf Seite 62 wurde eine Erweiterung des Eventmodells aus [Ste08] um ein Modell für `<invoke>`-Aktivitäten vorgeschlagen. In Apache ODE wird durch eine Erweiterung [Apa10f] die Behandlung von Message-Headers unterstützt. Diese Arbeit basiert auf dieser Unterstützung, da ein Coordination Context ohne diese Erweiterung nicht aus den Application Messages extrahiert werden könnte. Aus diesem Grund wird an dieser Stelle eine zusätzliche Erweiterung für das Eventmodell vorgeschlagen, sodass in nachrichtenempfangenden Aktivitäten die Möglichkeit des Zugriffs auf die Nachrichtenteile besteht.

Auch können weitere Mappingverfahren für andere Coordination Types und Protokolle entwickelt werden.

In Abschnitt 4.7.2 auf Seite 94 wurde für AT-Scopes gefordert, dass diese Isolated Scopes sein müssen. Diese Einschränkung wird getroffen, um die Atomizität des Scopes zu gewährleisten. Isolated Scopes dürfen keine weiteren Isolated Scopes beinhalten. Es bleibt zu untersuchen, inwiefern diese Einschränkung aufgeweicht werden kann und welche anderen Annahmen dazu getroffen werden müssen.

Literaturverzeichnis

[Apa10a] Apache Software Foundation. Apache ODE Management API. `http://ode.apache.org/management-api.html`, 2010. (Zitiert auf Seite 38)

[Apa10b] Apache Software Foundation. Apache ODE Management BPEL Compliance. `http://ode.apache.org/ws-bpel-20-specification-compliance.html`, 2010. (Zitiert auf Seite 24)

[Apa10c] Apache Software Foundation. Apache Orchestration Architektur. `http://ode.apache.org/architectural-overview.html`, 2010. (Zitiert auf den Seiten 6, 23 und 24)

[Apa10d] Apache Software Foundation. Apache Orchestration Director Engine (ODE). `http://ode.apache.org`, 2010. (Zitiert auf den Seiten 11 und 23)

[Apa10e] Apache Software Foundation. Atomic Scopes in Apache ODE. `http://ode.apache.org/atomic-scopes-extension-for-bpel.html`, 2010. (Zitiert auf Seite 94)

[Apa10f] Apache Software Foundation. Headers Handling in Apache ODE. `http://ode.apache.org/headers-handling.html`, 2010. (Zitiert auf den Seiten 62 und 124)

[EH07] H. Erven, G. Hicker. *Web-Transaktionen: Erweiterung der WS-BusinessActivity-Spezifikation um WS-BusinessActivity–Initiator und Implementierung der beiden Protokolle im Open Source-Projekt Apache Kandula*. Master's thesis, Technische Universitaet Wien, 2007. URL `http://www.big.tuwien.ac.at/teaching/theses/ma/17.html`. (Zitiert auf Seite 44)

[Fin09] S. Fink. *Realisierung von erweiterten Transaktionskoordinatoren auf Basis von WS-Coordination*. Master's thesis, Universität Hamburg, 2009. (Zitiert auf Seite 51)

[Fow02] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. (Zitiert auf Seite 42)

[Gon09] A. Goncalves. *Beginning Java EE 6 Platform with GlassFish 3: From Novice to Professional*. Apress, 2009. (Zitiert auf Seite 117)

[Gra81] J. Gray. The transaction concept: virtues and limitations (invited paper). In *VLDB '1981: Proceedings of the seventh international conference on Very Large Data Bases*, pp. 144–154. VLDB Endowment, 1981. (Zitiert auf Seite 21)

[Hol95] D. Hollingsworth. Workflow Management Coalition – The Workflow Reference Model. Technical report, Workflow Management Coalition, 1995. (Zitiert auf den Seiten 36 und 38)

[HRF10] M. Husemann, M. von Riegen, S. Fink. TracG Sourceforge. http://sourceforge.net/projects/tracg/, 2010. (Zitiert auf Seite 51)

[HRR07] M. Husemann, M. V. Riegen, N. Ritter. Transactional Coordination of Dynamic Processes in Service-Oriented Environments. pp. 1024–1031. IEEE Computer Society, 2007. doi:10.1109/ICWS.2007.180. (Zitiert auf Seite 51)

[HRRF10] M. Husemann, M. von Riegen, N. Ritter, S. Fink. TracG. http://vsis-www.informatik.uni-hamburg.de/projects/tracg/, 2010. (Zitiert auf Seite 50)

[HW03] G. Hohpe, B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. (Zitiert auf Seite 110)

[IBM05] IBM, BEA, Microsoft, TIBCO Software. Web Services Reliable Messaging (WS-RM). 2005. URL http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-rm/ws-reliablemessaging200502.pdf. (Zitiert auf Seite 119)

[Kan10] Kandula. The Apache Software Foundation. http://ws.apache.org/kandula/, 2010. (Zitiert auf Seite 43)

[KKL07] R. Khalaf, D. Karastoyanova, F. Leymann. Pluggable Framework for Enabling the Execution of Extended BPEL Behavior. In *Proceedings of the 3rd International Workshop on Engineering Service-Oriented Application (WESOA'2007)*. Unbekannt, 2007. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2007-99&engl=0. (Zitiert auf den Seiten 11 und 35)

[KKS+06] D. Karastoyanova, R. Khalaf, R. Schroth, M. Paluszek, F. Leymann. BPEL Event Model. Technischer Bericht Informatik 2006/10, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Universität Stuttgart, Institut für Architektur von Anwendungssystemen, 2006. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2006-10&engl=0. (Zitiert auf den Seiten 6, 26, 27, 28, 32, 37, 40, 62 und 68)

[KL04] D. Kossmann, F. Leymann. Web Services. *Informatik Spektrum*, 27(2):117–128, 2004. doi:10.1007/s00287-004-0378-9. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=ART-2004-23&engl=1. (Zitiert auf Seite 13)

[KL06] R. Khalaf, F. Leymann. Role-based Decomposition of Business Processes using BPEL. In *International Conference on Web Services (ICWS 2006)*, pp. 770–780. IEEE Computer Society, 2006. doi:10.1109/ICWS.

2006.56. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2006-83&engl=. (Zitiert auf Seite 26)

[KLN+06] D. Karastoyanova, F. Leymann, J. Nitzsche, B. Wetzstein, D. Wutke. Parameterized BPEL Processes: Concepts and Implementation. In S. Dustdar, J. L. Fiadeiro, A. P. Sheth, editors, *Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pp. 471–476. Springer, 2006. doi:10.1007/11841760_41. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2006-82&engl=0. (Zitiert auf Seite 26)

[KML08] O. Kopp, R. Mietzner, F. Leymann. Abstract Syntax of WS-BPEL 2.0. Technischer Bericht Informatik 2008/06, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Universität Stuttgart, Institut für Architektur von Anwendungssystemen, 2008. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2008-06&engl=0. (Zitiert auf den Seiten 53, 55 und 116)

[KML09] O. Kopp, R. Mietzner, F. Leymann. The Influence of an External Transaction on a BPEL Scope. In *On the Move to Meaningful Internet Systems: OTM 2009*, volume 5870 of *Lecture Notes in Computer Science*, pp. 381–388. Springer Berlin / Heidelberg, 2009. doi:10.1007/978-3-642-05148-7_27. (Zitiert auf den Seiten 6, 80, 85, 87 und 124)

[KMWL09] O. Kopp, D. Martin, D. Wutke, F. Leymann. The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages. *Enterprise Modelling and Information Systems*, 4(1):3–13, 2009. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=ART-2009-10&engl=0. (Zitiert auf Seite 20)

[KWM+08] O. Kopp, B. Wetzstein, R. Mietzner, S. Pottinger, D. Karastoyanova, F. Leymann. A Model-Driven Approach to Implementing Coordination Protocols in BPEL. In *1st International Workshop on Model-Driven Engineering for Business Process Management (MDE4BPM 2008)*, volume 17 of *Lecture Notes in Business Information Processing*, pp. 188–199. Springer-Verlag, 2008. doi:10.1007/978-3-642-00328-8_19. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2008-72&engl=0. (Zitiert auf Seite 52)

[LLM+08] T. van Lessen, F. Leymann, R. Mietzner, J. Nitzsche, D. Schleicher. A Management Framework for WS-BPEL. In *Proceedings of the 6th IEEE European Conference on Web Services 2008*, pp. 187–196. IEEE Computer Society, 2008. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2008-85&engl=0. (Zitiert auf den Seiten 36, 37 und 38)

[LR00] F. Leymann, D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, 2000. (Zitiert auf Seite 21)

[Mav10] Maven. The Apache Software Foundation. http://maven.apache.org, 2010. (Zitiert auf Seite 44)

[Mel07] I. Melzer. *Service-orientierte Architekturen mit Web Services: Konzepte – Standards – Praxis*. Spektrum, Heidelberg, 2007. (Zitiert auf Seite 13)

[Mie06] R. Mietzner. *Extraction of WS-Business Activity from BPEL 1.1*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2006. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2444&engl=0. (Zitiert auf den Seiten 26 und 49)

[Mül06] T. Müller. *Protokollbeschreibungen für die Koordination zwischen Web Services*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2006. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2434&engl=0. (Zitiert auf den Seiten 47 und 48)

[NLL08] J. Nitzsche, T. van Lessen, F. Leymann. WSDL 2.0 Message Exchange Patterns: Limitations and Opportunities. In *Proceedings of the 3rd International Conference on Internet and Web Applications and Services (ICIW 2008)*. IEEE, 2008. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2008-33&engl=0. (Zitiert auf den Seiten 14 und 76)

[OAS04] OASIS. Universal Description, Discovery and Integration (UDDI). Version 3.0.2. 2004. URL http://uddi.org/pubs/uddi_v3.htm. (Zitiert auf Seite 14)

[OAS06] OASIS. Web Services Resource. 2006. URL http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf. (Zitiert auf Seite 36)

[OAS07] OASIS. Web Services Business Process Execution Language. Version 2.0. 2007. URL http://docs.oasis-open.org/wsbpel/2.0/. (Zitiert auf den Seiten 8, 9, 19, 22, 71 und 115)

[OAS09a] OASIS. Web Services Atomic Transaction (WS-AtomicTransaction). Version 1.2. 2009. URL http://docs.oasis-open.org/ws-tx/wsat/2006/06. (Zitiert auf den Seiten 6, 9, 15, 17, 18, 19, 98, 104, 110, 119 und 120)

[OAS09b] OASIS. Web Services Business Activity Framework (WS-BusinessActivity). Version 1.2. 2009. URL http://docs.oasis-open.org/ws-tx/wsba/2006/06. (Zitiert auf den Seiten 6, 9, 15, 18, 76, 80, 95, 98, 110, 119 und 120)

[OAS09c] OASIS. Web Services Coordination (WS-Coordination). Version 1.2. 2009. URL http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec.html. (Zitiert auf den Seiten 9, 15, 17, 59, 81, 83, 88, 110 und 120)

[Ora10] Oracle. Glassfish Application Server. https://glassfish.dev.java.net/, 2010. (Zitiert auf Seite 110)

[OWT06a] OASIS-WSRF-TC. Web Services Resource Framework. 2006. URL http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf. (Zitiert auf Seite 36)

[OWT06b] OASIS-WSRF-TC. Web Services Resource Property. 2006. URL http://docs.oasis-open.org/wsrf/wsrf-ws_resource_properties-1.2-spec-os.pdf. (Zitiert auf Seite 37)

[Pal07] M. Paluszek. *Coordinating Distributed Loops and Fault Handling, Transactional Scopes using WS- Coordination protocols layered on WS-BPEL services*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2007. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2586&engl=0. (Zitiert auf Seite 48)

[PG08] R. M. Parizi, A. A. A. Ghani. An Ensemble of Complexity Metrics for BPEL Web Processes. In *SNPD '08: Proceedings of the 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pp. 753–758. IEEE Computer Society, Washington, DC, USA, 2008. doi:http://dx.doi.org/10.1109/SNPD.2008.152. (Zitiert auf Seite 42)

[PML07] S. Pottinger, R. Mietzner, F. Leymann. Coordinate BPEL Scopes and Processes by Extending the WS-Business Activity Framework. In R. Meersman, Z. Tari, editors, *Proceedings of the 15th International Conference on Cooperative Information Systems (CoopIS 2007)*, volume 4803 of *Lecture Notes in Computer Science*, pp. 336–352. Springer, 2007. doi:10.1007/978-3-540-76848-7_22. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2007-90&engl=0. (Zitiert auf den Seiten 21, 49 und 121)

[Saa00] T. Saaty. *Fundamentals of the Analytic Hierarchy Process*. RWS Publications, 4922 Ellsworth Avenue, Pittsburgh, PA 15413, 2000. (Zitiert auf Seite 41)

[SM05] P. Sauter, I. Melzer. A Comparison of WS-BusinessActivity and BPEL4WS Long-Running Transaction. In *Proc. of Kommunikation in Verteilten Systemen (KIVS*. 2005. (Zitiert auf den Seiten 17 und 81)

[Ste08] T. Steinmetz. *Ein Event-Modell für WS-BPEL 2.0 und dessen Realisierung in Apache ODE*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2008. URL http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2729&engl=0. (Zitiert auf den Seiten 6, 26, 27, 29, 30, 31, 32, 33, 34, 35, 37, 39, 40, 53, 62, 68, 110, 113, 117 und 124)

[TKM04] S. Tai, R. Khalaf, T. Mikalsen. Composition of coordinated web services. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pp. 294–310. Springer-Verlag New York, Inc., New York, NY, USA, 2004. (Zitiert auf den Seiten 119, 120 und 124)

[TMW+04] S. Tai, T. Mikalsen, E. Wohlstadter, N. Desai, I. Rouvellou. Transaction policies for service-oriented computing. volume 51, pp. 59–79. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2004. doi:http://dx.doi.org/10.1016/j.datak.2003.03.001. (Zitiert auf den Seiten 57, 119, 120 und 124)

[Trö08] P. Tröger. *Dynamische Ressourcenverwaltung für dienstbasierte Software-Systeme*. Cuvillier Verlag, 2008. (Zitiert auf Seite 14)

[TS07] A. S. Tanenbaum, M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, Upper Saddle River, NJ, second edition, 2007. (Zitiert auf den Seiten 18 und 99)

[Vet06] T. Vetter. *Anpassung und Implementierung verschiedener Transaktionsprotokolle auf WS-Coordination*. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2006. URL `http://www.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2386&engl=0`. (Zitiert auf Seite 46)

[W3C04] W3C. Web Services Addressing (WS-Addressing). Version 1.0. 2004. URL `http://www.w3.org/Submission/ws-addressing/`. (Zitiert auf den Seiten 15, 61, 116 und 117)

[W3C07a] W3C. Web Services Definition Language (WSDL). Version 2.0. 2007. URL `http://www.w3.org/TR/wsdl20/`. (Zitiert auf Seite 13)

[W3C07b] W3C. Web Services Policy Attachment (WS-Policy Attachment). Version 1.5. 2007. URL `http://www.w3.org/TR/2007/REC-ws-policy-attach-20070904/`. (Zitiert auf den Seiten 14, 57 und 120)

[W3C07c] W3C. Web Services Policy (WS-Policy). Version 1.5. 2007. URL `http://www.w3.org/TR/ws-policy/`. (Zitiert auf den Seiten 14 und 119)

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D. F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005. (Zitiert auf den Seiten 13, 14 und 19)

[WDGW08] M. Weidlich, G. Decker, A. Großkopf, M. Weske. BPEL to BPMN: The Myth of a Straight-Forward Mapping. In R. Meersman, Z. Tari, editors, *OTM Conferences (1)*, volume 5331 of *Lecture Notes in Computer Science*, pp. 265–282. Springer, 2008. URL `http://dblp.uni-trier.de/db/conf/otm/otm2008-1.html#WeidlichDGW08`. (Zitiert auf Seite 77)

[WVKG08] T. Wang, J. Vonk, B. Kratz, P. Grefen. A survey on the history of transaction management: from flat to grid transactions. volume 23, pp. 235–270. Springer Netherlands, 2008. doi:10.1007/s10619-008-7028-1. URL `http://www.springerlink.com/content/0667826840552700/`. (Zitiert auf Seite 15)

[Zam04] S. Zambrovski. *Entwurfskonzepte und Implementierungsstrategien fur das WS-BusinessActivity Framework*. Master's thesis, Technische Universität

Hamburg-Harburg, 2004. URL http://jwst.sourceforge.net/resources/publications/wsba-report.pdf. (Zitiert auf Seite 50)

[Zam10] S. Zambrovski. JWST. http://jwst.sourceforge.net/, 2010. (Zitiert auf Seite 49)

Alle URLs wurden zuletzt am 05.07.2010 geprüft.

Erklärung

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

(Sebastian Henke)