Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Diplomarbeit Nr. 2429

Generierung von BPEL mit Hilfe von koordinierten Kommunikations-Graphen auf Basis transaktionaler Protokolle für Web Services

Michael, Sabine

Studiengang:	Informatik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. Stefan Pottinger
begonnen am:	01. Januar 2006
beendet am:	31. Mai 2006
CR-Klassifikation:	D.2.2, D.2.12, H.3.4, H4.1

Inhaltsverzeichnis

Abstract

Im Rahmen der vorliegenden Diplomarbeit, mit dem Titel „Generierung von BPEL mit Hilfe von koordinierten Kommunikations-Graphen auf Basis transaktionaler Protokolle für Web Services“, soll durch eine selbst entworfene Diagrammart die Möglichkeit der Modellierung von Protokollgraphen untersucht werden, mit deren Hilfe Code in der Business Execution Process Language (BPEL) generiert werden kann.

Der koordinierte transaktionale Kommunikations-Graph (Coordinated Transactional Communication-Graph, kurz CTC-Graph genannt) muss Zustände und Nachrichtenflüsse zwischen den Kommunikationspartnern in einem koordinierten Web Services Umfeld modellieren und beschreiben können.

Programmatisch soll die Modellierung des CTC-Graphen mit Hilfe eines Eclipse-Plug-ins unterstützt werden. Der Graph stellt visuell das Kommunikationsprotokoll im koordinierten Web Services Umfeld dar.

Eine Analyse der Ergebnisse sowie ein Ausblick auf mögliche Erweiterungen der CTC-Graphen und der BPEL-Generierung bilden den Abschluss dieser Arbeit.

1 Einführung

1.1 Das Web Services Umfeld

Unter einem Web Service wird eine Software-Anwendung verstanden, die mit einem Uniform Ressource Identifier (URI) eindeutig identifizierbar ist und deren Schnittstellen als XML-Artefakte (vgl. Spezifikationen [28] und [29]) definiert, beschrieben und gefunden werden können.

Gem. „Web Services Plattform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-Reliable Messaging, and More" [31] und „World Wide Web Consortium" [32] ist unter einem Web Service die direkte Interaktion zwischen mindestens zwei Maschinen zu verstehen, d.h. es gibt keine „menschliche Benutzerschnittstelle". Der Austausch von Nachrichten erfolgt über internetbasierte Protokolle.

Die Web Services-Architektur besteht aus drei wesentlichen Funktionsträgern [4]:

1. Dienstanbieter (Service Provider): Er bietet seine Dienste über das Web an. Hierfür beschreibt und annonciert er diese Dienste. Gleichzeitig übernimmt der Dienstanbieter die Implementierung und Wartung des angebotenen Dienstes.

2. Dienstnachfrager (Service Consumer): Er verwendet den vom Dienstanbieter angebotenen Dienst. Mit der Hilfe eines Dienstmaklers (s.u.) sucht der Dienstnachfrager nach einem passenden Anbieter. Anschließend kann er nun den benötigten Dienst in seinen eigenen Diensten und Applikationen verwenden.

3. Dienstmakler (Broker): Er bietet die Speicherung von Dienstbeschreibungen des Dienstanbieters an, verwaltet und kategorisiert diese. Des Weiteren ermöglicht er das automatisierte Auffinden existierender Dienste für den Dienstnachfrager.

In dieser Diplomarbeit liegt der Schwerpunkt zum Einen auf der automatisierten Zusammenarbeit von Web Services Partnern (im Rahmen der Business Process Execution Language) und zum Anderen auf der Koordination der Kommunikation sowie der Atomizität und Kompensierung von Interaktionen zwischen den oben beschriebenen Partnern (Web Services Transactions).

Um einen Überblick in die einzelnen Bestandteile der bevorstehenden Aufgabe zu gewinnen, wird nachfolgend auf die Spezifikationen von Web Services Transactions und **B**usiness **P**rocess **E**xecution **L**anguage (kurz **BPEL**) eingegangen. Die Sprache und ihre Konstrukte gehören zu den wichtigsten Aspekten für die Arbeiten an den zu entwerfenden koordinierten transaktionalen Kommunikations-Graphen und der automatisierten Übersetzung des Modells in

BPEL-Code im Eclipse-Plug-in. Deshalb wird der Fokus in Kapitel 1 bewusst auf BPEL gerichtet.

Alle Zusammenfassungen, auch die ausführlichere Ausarbeitung zu BPEL in Kapitel 1.1.2, dienen lediglich dem besseren Verständnis der bevorstehenden Aufgaben und erheben keinerlei Anspruch auf Vollständigkeit. Zur Vertiefung und für weitere Recherchen einzelner Themen wird auf die genannten Nachschlagewerke im „Literaturverzeichnis“ hingewiesen. Durch Ziffern in eckigen Klammer soll der Bezug auf die im Literaturverzeichnis genannten Werke hergestellt werden.

1.1.1 Web Services Transactions (WS-Coor / WS-AT / WS-BA)

Die nachfolgend aufgeführten Spezifikationen haben für die vorliegende Arbeit besondere Relevanz:

1. Web Services Coordination
2. Web Services Atomic Transaction
3. Web Services Business Activity

Sie werden im Folgenden gemeinsam unter dem Sammelbegriff „Web Services Transactions“ (vgl. „Web Services Transactions specifications“ [28]) geführt.
Unter Transaktionen werden analog zur Datenbanken-Welt unter anderem die zuverlässigen, konsistenten und atomar ausgeführten Aufgaben zwischen Kommunikationspartnern verstanden.

Transaktionen sollten die sog. **ACID-Eigenschaften** aufweisen. **ACID** ist die Abkürzung der Anfangsbuchstaben der Begriffe **A**tomicity (Atomizität), **C**onsistency (Konsistenz), **I**solation (Isolation), **D**urability (Dauerhaftigkeit). Unter diesen vier Eigenschaften von Transaktionen wird gemeinhin Folgendes verstanden:

1. Atomizität: Eine Transaktion ist eine atomare Verarbeitungseinheit; sie wird entweder vollständig oder überhaupt nicht ausgeführt.
2. Konsistenz: Eine Transaktion ist konsistenzbewahrend wenn ihre vollständige Ausführung die Datenbank von einem konsistenten Zustand in einen anderen überführt.
3. Isolation: Eine Transaktion sollte so ausgeführt werden, als ob sie isoliert von anderen Transaktionen ist, d. h. die Ausführung einer Transaktion sollte nicht von anderen, gleichzeitig ablaufenden Transaktionen gestört werden.
4. Dauerhaftigkeit: Die durch eine bestätigte Transaktion in die Datenbank geschriebene Änderung muss in jedem Fall weiterhin in der Datenbank fortbestehen. Sie darf nicht aufgrund eines Fehlers verloren gehen.

Der Abschluss einer Transaktion kann auf zwei Arten erfolgen (vgl. „Grundlagen von Datenbanksystemen“ [15]):

1. Commit: Signalisiert die erfolgreiche Beendigung der Transaktion. Änderungen (Aktualisierungen), die von der Transaktion ausgeführt wurden, werden permanent in die Datenbank überführt und können nicht rückgängig gemacht werden.
2. Rollback (oder Abort): Signalisiert den Abbruch der Transaktion. Änderungen, die durch die Transaktion bisher vorgenommen wurden, werden rückgängig gemacht.

In Hinblick auf Web Services, soll damit erreicht werden, dass alle Beteiligten einer Transaktion zu jedem Zeitpunkt über den gleichen Informationsstand verfügen, was den Ausgang dieser anbelangt.

Sollte aus verschiedenen Ursachen (bspw. interner Fehler, Hardware-Fehler oder aus anderen Gründen) ein Partner nicht in der Lage sein, seinen Teil der Aufgabe zu erfüllen, so wird die komplette Bearbeitung abgebrochen und evtl. bereits vorgenommene Änderungen, die im Rahmen dieser gemeinsamen Aufgabe bereits erfolgt sind, wieder rückgängig gemacht.

Die für die Beschreibung des transaktionalen Verhaltens besonders wichtigen Spezifikationen „Web Services Atomic Transaction“ [6], „Web Services Coordination“ [7] und „Web Services Business Activity“ [8] werden im Folgenden näher erläutert.

1.1.1.1 Web Services Coordination

Im Web Services Umfeld wird die Interaktion von mehr als zwei Kommunikationspartnern mit Hilfe von **W**eb **S**ervices **Coor**dination - im Folgenden **WS-Coor** genannt – koordiniert und bewerkstelligt. Detaillierte Informationen hierzu sind in der Spezifikation [7] zu finden.

Alle beteiligten Kommunikationspartner erhalten eine der zwei vordefinierten Rollen:

1. Koordinator (coordinator): Die Funktion des Koordinators besteht darin, die anstehenden Aufgaben und deren Einzelberechnungen zu verwalten. Nur mit Hilfe des Koordinators kann der erwünschte transaktionale Ausgang der Aufgabe sichergestellt werden.
2. Teilnehmer (participants): Die Teilnehmer führen die Berechnungen durch, die ihnen im Rahmen der Transaktion zugeteilt werden. Des Weiteren helfen die Teilnehmer dem Koordinator bei der Entscheidung, ob die Ergebnisse ihrer Berechnungen festgeschrieben werden sollen oder nicht. Ist eine Transaktion fehlerfrei ausgeführt worden, so sind alle Änderungen festgeschrieben und die Transaktion war erfolgreich. Sind Fehler aufgetreten, werden alle bis dahin erfolgten Änderungen rückgängig gemacht und die Transaktion endet unvollzogen.

Die Spezifikation von WS-Coor [7] schreibt vor, wie eine Anwendung, die mehrere Kommunikationspartner in eine Berechnung einbeziehen möchte, den

Koordinator aktiviert und wie dieser zwischen der Anwendung und den Kommunikationspartnern zu interagieren hat.
Die Aufgaben des Koordinators bestehen zum Einen aus dem Aushandeln der Einzelheiten der Kommunikation mit den Teilnehmern z.B. Kommunikations-Protokoll, Transaktions-Protokoll, etc. und zum Anderen aus der Koordination der Berechnungen bis hin zur Entscheidung, ob die Transaktion erfolgreich oder nicht erfolgreich beendet wurde.
Die Entscheidung welches Transaktions-Protokoll verwendet werden soll, ist nicht Bestandteil der WS-Coord Spezifikation. Hierfür wurden die nachfolgend beschriebenen Spezifikationen (Web Services Atomic Transaction und Web Services Business Activity) definiert, welche je nach Bedarf und Transaktions-Fall verwendet werden können.

1.1.1.2 Web Services Atomic Transaction

Mit **W**eb **S**ervices **A**tomic **T**ransaction (im Folgenden als **WS-AT** abgekürzt) werden kurzlebige und schnell abschließbare Transaktionen im Web Service Umfeld beschrieben, vgl. Spezifikation [6]. Die Vorgehensweisen und das Verhalten einzelner Teilnehmer einer Transaktion werden klar definiert, um die erwünschten Transaktionseigenschaften zu gewährleisten.

Von elementarer Bedeutung ist die exakte Definition der Verhaltensmuster und –regeln zwischen allen beteiligten Kommunikationspartnern. Im Fehlerfall muss die Transaktion sofort abgebrochen und alle Änderungen rückgängig gemacht werden bzw. bei einer erfolgreichen Transaktion die Änderungen gespeichert und die Transaktion/ Kommunikation ordnungsgemäß beendet werden.

Die Vereinbarung zwischen den Kommunikationspartnern wird mittels eines – aus den Verteilten Systemen bekannten und bewährten „2 Phase-Commit Protokoll“ (2PC) Konzeptes durchgeführt.

Folgendes Schaubild soll den Ablauf des 2PC verdeutlichen:

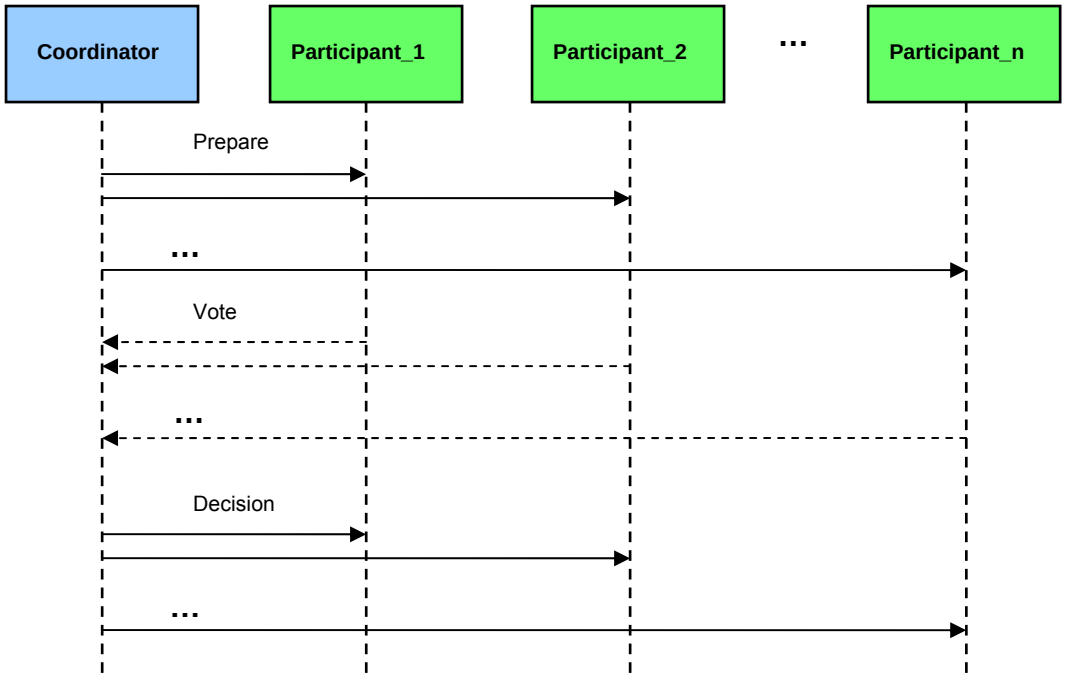


Abb. 1-1: Ablauf des „2 Phase-Commit Protokoll“ (2PC)

Nach der Abarbeitung der Aufgaben einer Transaktion, läuft ein 2PC nach folgendem Schema ab:

1. Der Koordinator befragt alle Teilnehmer nach ihrer Bereitschaft die Transaktion zu beenden.
2. Jeder der Teilnehmer antwortet auf die Frage entweder mit einer positiven oder negativen Antwort. Bei der positiven Antwort sind keine Fehler aufgetreten und die Änderungen können festgeschrieben werden. Bei einer negativen Antwort, also im Falle des Auftretens von Fehlern bei der Berechnung oder dem Ausbleiben einer Antwort (Teilnehmer ist ausgefallen) können die Änderungen rückgängig gemacht werden.
3. Der Koordinator entscheidet dann anhand der Antworten, ob die Änderungen der Transaktion festgeschrieben werden sollen, gdw. alle Teilnehmer eine positive Antwort gesendet haben, oder ob alle Änderungen wieder rückgängig gemacht werden müssen, gdw. mindestens ein Teilnehmer eine negative Antwort oder keine Antwort verschickt hat vgl. Weerawarana et al in [31].

WS-AT 2 Phase Commit Diagramm mit Nachrichten

Das nachfolgende Diagramm aus der WS-AT-Spezifikation [6] verdeutlicht das oben beschriebenen 2PC-Protokoll in seinen Grundzügen. Zusätzlich können dem Diagramm Informationen bezüglich der Systemzustände, die für die koordinierte Einheit von zentraler Bedeutung sind, entnommen werden. Auch die Nachrichten der jeweiligen Kommunikationspartner (participant oder coordinator), werden im Diagramm verdeutlicht.

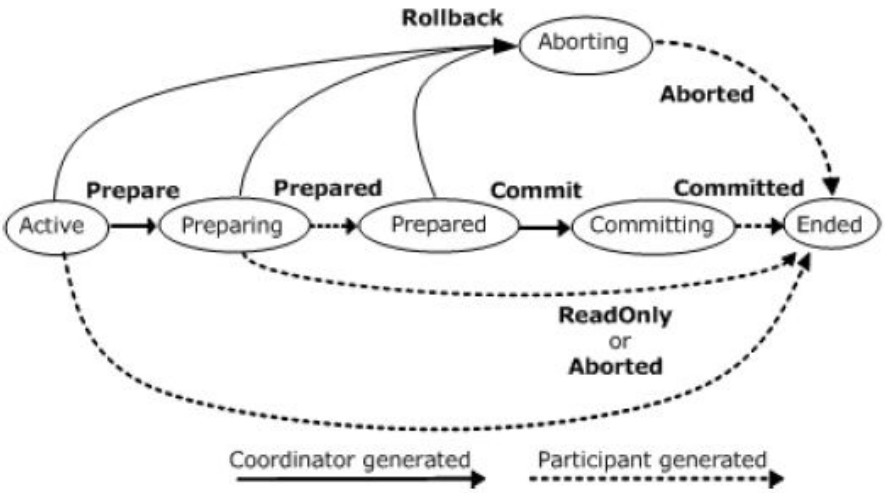


Abb. 1-2: WS-AT 2PC Diagramm mit Nachrichten (Quelle WS-AT Spezifikation [6])

Ein System, das sich gerade mit der Abarbeitung einer Transaktion befasst, befindet sich im *„Active“* Zustand.
Sind die Berechnungen abgeschlossen, so ist auch die Transaktion zu Ende. Im Anschluss werden die einzelnen Kommunikationspartner befragt, ob die Änderungen festgeschrieben werden können.

Hierfür schickt der Koordinator eine Nachricht an alle Teilnehmer und bittet sie um ihre Antwort auf die Frage, ob Berechnungen erfolgreich waren oder Fehler aufgetreten sind. Das System befindet sich nun im Zustand *„Preparing“*.
Erhält der Koordinator alle Antworten, wechselt das System in den *„Prepared“*-Zustand. Ab diesem Zeitpunkt entscheidet der Koordinator alleinig über den Ausgang der Transaktion.

Die Entscheidung teilt er den Teilnehmern mittels einer Nachricht mit. Abhängig von der Entscheidung wechselt das System dann in den entsprechenden Zustand *(„Aborting“* / *„Committing“*). Nachdem die Teilnehmer die Entscheidung des Koordinators ausgeführt haben („Commit“ / „Abort“ / „Rollback“), wechselt das System in den Zustand *„Ended“*.

Ist bei dem Koordinator selbst, bei der transaktionsinitiierenden Anwendung oder bei einem Kommunikationspartner ein Fehler aufgetreten, bleibt die Befragung aus. In diesem Fall bricht der Koordinator die Transaktion ab und informiert die Teilnehmer über den Fehler. Anschließend findet ein Zustandsübergang in den Zustand *„Ended“* statt – vgl. WS-AT Spezifikation [6].

1.1.1.3 Web Services Business Activity

Die Web Services Business Activity Spezifikation [8] definiert und beschreibt die Ausprägung der möglichen Kommunikationsprotokolle im koordinierten Web Services Umfeld, analog der Web Services Atomic Transaction Spezifikation.

Während die Web Services Atomic Transactions Spezifikation [6] kurzlebige und schnell abschließbare Transaktionen zur Aufgabe hat, werden durch die **W**eb **S**ervices **B**usiness **A**ctivity (im Folgenden **WS-BA** genannt) langlebige d.h. nicht schnell abschließbare Transaktionen im Web Service Umfeld beschrieben.

Für die Verdeutlichung einer langlebigen Transaktion kann folgendes Beispiel angeführt werden: Koordinator „A“ startet eine Transaktion T_a um bestimmte Berechnungen und Anfragen verschiedener Teilnehmer (hier zwei Teilnehmer „B“ und „C“) durchführen zu lassen. B muss, um die Berechnungen von T_a durchführen zu können, selbst andere verteilte Dienste in Anspruch nehmen. B startet – nun selbst in der Rolle des Koordinators – seinerseits eine Transaktion T_b, um die Anfragen von T_a zu abzuarbeiten. Eine solche Verschachtelung von Koordinationskontexten ist nicht unüblich und völlig korrekt. Sie kann beliebig fortgeführt werden. Es ist anzunehmen, dass B für die Bearbeitung seiner Teilaufgaben von T_a mehr Zeit benötigen wird als C. B ist letztendlich auf die Ergebnisse der Teilnehmer seiner Transaktion T_b angewiesen, die ebenfalls Zeit für die Abarbeitung ihrer Aufgaben benötigen.

Die Vorgehensweise und das Verhalten der einzelnen Transaktionsteilnehmer müssen zwingend, wie bereits in Kapitel 1.1.1.1 beschrieben, klar definiert sein, um die erwünschten Transaktions-Eigenschaften gewährleisten zu können, vgl. WS-BA Spezifikation [8].

Wichtig bei den Business Activities sind die Verhaltensmuster und -regeln der einzelnen Kommunikationspartner im Fehlerfall.

Da es sich bei WS-BA um langlebige Transaktionen handelt und infolgedessen nicht alle Teilnehmer (participants) bis zum endgültigen Abschluss der Transaktion verfügbar sind, arbeitet dieses Protokoll mit dem Konzept der Kompensation (compensate); d.h. Änderungen werden bei den Teilnehmern einer Transaktion festgeschrieben, müssen allerdings im Fehlerfall rückgängig gemacht werden. Wie in der WS-BA Spezifikation [8] eingehend beschrieben, resultiert aus der Protokollierung aller Änderungen ein hoher Aufwand an Vorbereitungs- und Verwaltungsarbeit.

Bei WS-BA werden zwei verschiedene Formen des Transaktionsabschlusses unterschieden. Diese beiden Formen sind für den weiteren Verlauf der Arbeit wichtig und werden nachfolgend erläutert:

1. WS-BA with Participant Completion
2. WS-BA with Coordinator Completion

1.1.1.3.1 WS-BA with Participant Completion Diagramm mit Nachrichten

Das nachfolgende Diagramm verdeutlicht die Idee, die sich hinter dem WS-BA with Participant Completion Protokoll verbirgt.

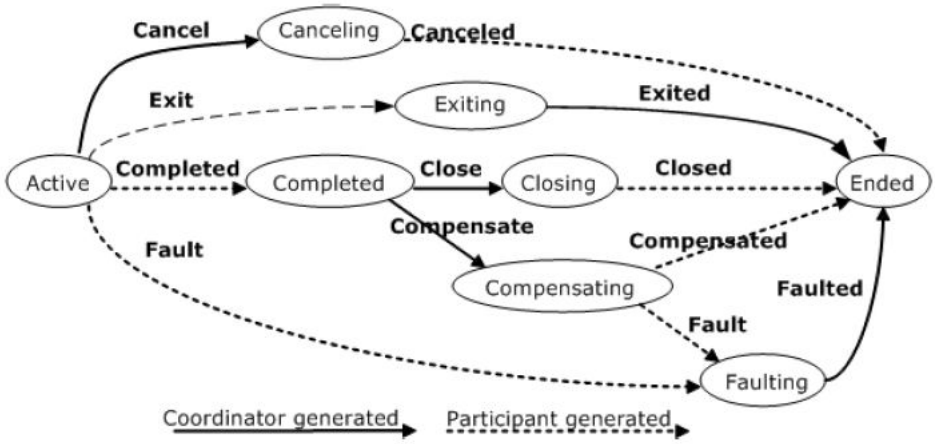


Abb. 1-3: WS-BA with Participant Completion Diagramm mit Nachrichten (Quelle WS-BA Spezifikation [8])

Das in Abbildung 1-3 dargestellte Diagramm visualisiert das Konzept des WS-BA with Participant Completion. Es zeigt die Zustände, welche die komplett zu koordinierende Einheit einnehmen kann, sowie die Nachrichten, die die jeweiligen Transaktionspartner (in der Rolle als Teilnehmer „participant“ oder des Koordinator „coordinator“) zu bestimmten Zeitpunkten verschicken können.

Das Besondere an dem Participant Completion Protokoll ist, dass sich verschiedene Transaktionsteilnehmer zu jedem gewünschten Zeitpunkt aus der Transaktion verabschieden können. Infolgedessen bekommt die Protokollierung von Änderungen der laufenden Transaktion durch den Koordinator ein besonderes Gewicht. Das Protokoll versetzt den Koordinator, im Falle einer fehlgeschlagenen Transaktion in die Lage, an alle Teilnehmer, also auch an diejenigen, die zu diesem Zeitpunkt nicht mehr aktiv sind, die nötigen „`compensate`“ Nachrichten zu senden. So können die bereits durchgeführten Änderungen rückgängig gemacht werden (vgl. WS-BA Spezifikation [8]).

1.1.1.3.2 WS-BA with Coordinator Completion Diagramm mit Nachrichten

Das in Abbildung 1-4 gezeigte Diagramm verdeutlicht die Idee, die sich hinter dem WS-BA with Coordinator Completion verbirgt.

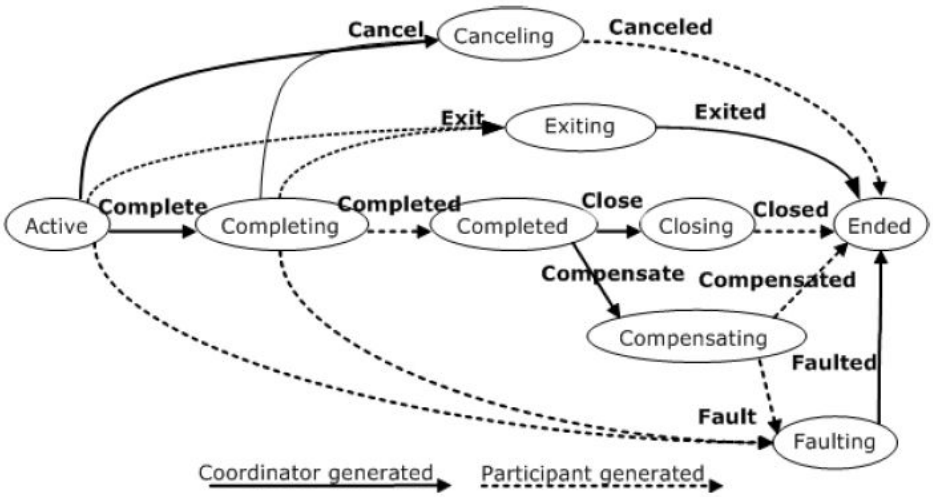


Abb. 1-4: WS-BA with Coordinator Completion Diagramm mit Nachrichten (Quelle WS-BA Spezifikation [8])

Das Diagramm visualisiert die Zustände, die eine komplett zu koordinierende Einheit einnehmen kann sowie die von den beteiligten Transaktionspartnern (ob in der Rolle eines Teilnehmers „participant“ oder des Koordinators „coordinator“) verschickten Nachrichten.

Im Gegensatz zum WS-BA with Participant Completion Protokoll können sich die Transaktionsteilnehmer nicht zu jedem Zeitpunkt aus der Transaktion ausklinken. Diese Aufgabe obliegt dem Koordinator. Nur dieser kann den Befehl „beenden“ erteilen. Somit muss sich der Teilnehmer darauf verlassen, dass der Koordinator ihm zum richtigen und bestenfalls frühstmöglichen Zeitpunkt mitteilt, dass er nicht mehr für die laufende Transaktion benötigt wird.

Auch bei diesem Protokoll ist es wichtig, dass der Koordinator weiterhin die Änderungen, die im Rahmen der Transaktion durchgeführt wurden, protokolliert.
Im Falle einer fehlgeschlagenen Transaktion muss der Koordinator die erforderlichen „`compensate`“ Nachrichten an alle Teilnehmer (auch an die, die der Koordinator selbst aus der Transaktion verabschiedet hat) senden (vgl. WS-BA Spezifikation [8]).

1.1.2 Business Process Execution Language

Da die Entwicklung von Web Services immer weiter voranschreitet und die Akzeptanz bei Entwicklern und Industrie immer größer wird, steigt das Bedürfnis

nach einer gemeinsamen universellen Sprache zur Beschreibung von Geschäftsprozessen.

In der Anfangsphase wurden von IT-Unternehmen eigenständige Versuche gestartet, um diese Lücke zu schließen (siehe auch Weerawarana et al [31]). Jedoch präsentierte erst im Jahre 2003 eine Gruppe von Firmen (darunter IBM, Microsoft, BEA) eine zweckmäßige Lösung für dieses Problem in Form der **B**usiness **P**rocess **E**xecution **L**anguage (kurz **BPEL,** vgl. BPEL Spezifikation [1]).

BPEL ist eine XML-basierte Sprache zur Beschreibung von Geschäftsprozessen, deren einzelne Aktivitäten durch Web Services implementiert sind. Weiterhin kann mit Hilfe von BPEL ein Web Service selbst beschrieben werden.

Das Web Services Umfeld verlangt nach diversen Eigenschaften, die beim Entwurf von BPEL berücksichtigt wurden:

- Geschäftsprozesse enthalten datenabhängiges Verhalten.
- Die Fähigkeit zur Definition von außerordentlichen Bedingungen und deren Konsequenzen (sog. Recovery-Sequenzen) ist ebenso wichtig wie die Beschreibung eines fehlerfreien Geschäftsprozesses.
- Langlebige Interaktionen beinhalten oft mehrere, manchmal auch verschachtelte Abarbeitungsschritte, in denen jeder einzelne Schritt seine eigenen Anforderungen bzgl. der Daten besitzt. Geschäftsprozesse verlangen häufig eine partnerübergreifende Übereinkunft betreffend des Ausgangs (Erfolg oder Misserfolg) aller möglichen Schritte eines Vorgangs.

Die BPEL Spezifikation [1] definiert zwei Basis-Konzepte, die bei der Benutzung der Sprache verwendbar sind:

1. Abstract process
2. Executable process

1.1.2.1 Abstract Process

In Abstract BPEL wird eine Geschäftsprozess-Rolle definiert. Verwendet werden alle BPEL-Konzepte. Allerdings wird die Datenverarbeitung abstrahiert d.h. weniger ausführlich und konkret behandelt. Damit ist die Möglichkeit gegeben Außenstehenden eine Sicht auf die Prozesse zu gewähren, ohne interne Berechnungen offen legen zu müssen und dem Risiko von Datenmanipulationen auszusetzen. Abstrakte Prozesse behandeln nur protokoll-relevante Daten.

BPEL bietet gem. der Spezifikation [1] die Möglichkeit protokollrelevante Daten als „message properties“ (Nachrichten-Eigenschaften) zu kennzeichnen. Darüber hinaus bietet ein abstract process die Möglichkeit nichtdeterministische Datenwerte zu verwenden, um interne Aspekte der Verarbeitung nicht offenbaren zu müssen.

1.1.2.2 Executable Process

Die Prozesslogik und Zustände definieren genau die Vorgänge und Interaktionen der einzelnen Geschäftspartner und deren Interaktionsprotokoll.

Anders als bei einem abstract process wird der executable process nicht verwendet, um einen kleinen und oberflächlichen Einblick in die Geschäftsprozesse zu gewähren. In den executable processes werden alle einzelne Bearbeitungsschritte, Datenmanipulationen und der Nachrichtenaustausch beschrieben. Dieser erzeugte Code wird anschließend in Maschinen eingesetzt. So kann der beschriebene Geschäftsprozess durchgeführt werden vgl. Spezifikation [1].

1.1.2.3 Struktur von BPEL

Um die o.g. Aufgaben ausführen zu können, besitzt BPEL einen speziell strukturierten Prozess-Aufbau sowie diverse Sprachkonstrukte. Im Folgenden wird eine Übersicht der Basisstruktur dieser Sprache und der wichtigsten Konstrukte und ihrer Funktionalität gegeben.

Die Basisstruktur eines BPEL-Prozesses kann vereinfacht in 3 Abschnitte eingeteilt werden:

- Allgemeiner Teil mit Sprach- und Verhaltensinformationen zum auszuführenden BPEL-Prozess.
- Definition der Konstrukte, die für die Konnektivität, den Datenaustausch und die Kommunikation zwischen Web Services notwendig sind.
- Eigentliche Prozess-Steuerung (sog. activity), welche die Verarbeitungsschritte und Einzelheiten der Berechnungen definiert, siehe auch Spezifikation [1].

1.1.2.3.1 BPEL-Struktur-Abschnitte und ihre Sprachkonstrukte

Nachdem die Basisstruktur von BPEL-Prozessen beschrieben wurde, werden hier die Sprach-Konstrukte der verschiedenen Abschnitte und deren Zweck erklärt.

Abschnitt 1: allgemeine Prozess-Informationen[1]

```
<process name="ncname" targetNamespace="uri"
    queryLanguage="anyURI"?
    expressionLanguage="anyURI"?
    supressJoinFailure="yes | no"?
    enableInstanceCompensation="yes | no"?
    abstractProcess="yes | no"?
    xmlns="http://schemas.xmlsoap.org/ws/2003/03/business
-process/">
[Abschnitt2 / Abschnitt3]
</process>
```

Wie man in diesem allgemeinen Beispiel erkennen kann, werden im oberen Abschnitt die allgemeinen Definitionen des Prozesses als Attribute im Sprachkonstrukt „`process`“ angegeben.

Anfangs werden Angaben zum Prozessnamen und dessen Namensraum gemacht, auf diese Angaben wird aufgrund der selbsterklärenden Attribute hier nicht weiter eingegangen mehr hierzu siehe W3C Recomendation [27]. Die Fragezeichen zwischen den Attributen, verdeutlichen welche der Attribute fakultativ sind. Wird eines dieser Attribute weg gelassen, so wird automatisch ein Default-Wert eingesetzt.

Die weiteren Attribute sind aufschlussreicher, weil sie den allgemeinen Typ und das Verhalten des Business-Prozesses definieren (siehe auch Spezifikation [1]).

- „`queryLanguage="anyURI""` (default: XPath): Bestimmt die verwendete Abfragesprache und dient u.a. zur Selektion von Eigenschafts-Definitionen.
- „`expressionLanguage="anyURI""` (default: XPath): Dieses Attribut gibt die verwendete Sprache an.
- „`supressJoinFailure="yes|no""` (default: no): Bestimmt, ob die Fähigkeit von BPEL einen Fehler des Typs „joinFailure“ zu überschreiben, eingesetzt wird. Ein Fehler, der in einer verschachtelten Unter-Aktivität auftritt, kann vom Entwickler unterdrückt werden. So wird verhindert, dass Fehler an die nächst höher liegende Aktivität weitergereicht wird.
- „`enableInstanceCompensation="yes|no""` (default: no): Dieses Attribut legt fest, ob eine Instanz des Prozesses als Ganzes - auf einer plattformspezifischen Art - kompensiert werden soll oder nicht.
- „`abstractProcess="yes|no""` (default: no): Dieses Attribut gibt an, ob ein Prozess als abstrakt (statt ausführbar) definiert wird oder nicht.

[1] Auszug aus dem Abschnitt "The Structure of a Business Process" der BPEL-Spezifikation (Quelle [1]).

Abschnitt 2: allgemeine Definitionen zu Konnektivität, Datenaustausch und Kommunikation zwischen Web Services [2]

```
<partnerLinks> ?
    <partnerLink name="ncname" partnerLinkType="qname"
myRole="ncname"? partnerRole="ncname"?>+
    </partnerLink>
</partnerLinks>

<partners> ?
    <partner name="ncname">+
    <partnerLink name="ncname" />+
    </partner>
</partners>
<variables> ?
    <variable name="ncname" properties="qn-list"
type="qname"? element="qname"? />+
</variables>
<correlationSets> ?
    <correlationSet name="ncname" properties="qn-list"/>+
</correlationSets>
<faultHandlers> ?
    <catch faultName="qname"? faultVariable="ncname"?>*
         Activity
    </catch>
<catchAll>?
         Activity
    </catchAll>
</faultHandlers>
<compensationHandler>?
Activity
</compensationHandler>
<eventHandlers>?
    <onMessage partnerLink="ncname" portType="qname"
              Operation="ncname" variable="ncname"?>
       <correlations>?
         <correlation set="ncname" initiate="yes|no"?>+
       </correlations>
       Activity
</onMessage>
<onAlarm for="duration"? until="deadline"?> *
   Activity
</onAlarm>
</eventHandlers>
```

[2] Auszug aus dem Abschnitt "The Structure of a Business Process" der BPEL-Spezifikation (Quelle [1]).

Wie anhand dieses allgemeinen Beispiels zu erkennen ist, werden in diesem Abschnitt diverse Definitionen bzgl. Konnektivität, Kommunikation und Datenaustausch gezeigt. Die Fragezeichen hinter einzelnen Strukturelementen zeigen an, welches der Elemente jeweils fakultativ ist. Lässt man eines dieser Attribute entfallen, wird kein Gebrauch dieses Konstruktes gemacht, d.h. das Verhalten dieser Elemente ist nicht definiert.

Alle weiteren Attribute sind obligatorisch, denn sie definieren die Kommunikationspartner und -wege sowie das Fehlerverhalten des Business-Prozesses gem. BPEL Spezifikation [1].

- „`partnerLinks`“: Mit diesem BPEL-Befehl ist es möglich die Fähigkeit einer Beziehung zu einem Kommunikationspartner festzulegen. Die Rollen der Kommunikationspartner können als Parameter mit angegeben werden.
- „`variables`“: Dienen der Speicherung und Handhabung von Daten, die über Nachrichten an Kommunikationspartner verschickt werden.
- „`correlationSets`“: Mittels des correlationSets können Nachrichten den korrekten Instanzen eines Prozesses weitergegeben werden.
- „`faultHandlers`“: Bestimmt das Verhalten von BPEL zur Bearbeitung Behebung eines bestimmten Fehlers. Enthalten ist mindestens ein faultHandler, z.B. der Standard-FaultHandler.
- „`compensationHandlers`“: Das Kompensierungsverhalten, d.h. welche Werte sollen wie zurückgesetzt werden, wird an dieser Stelle festgelegt.
- „`eventHandlers`“: Mit Hilfe dieses Konstruktes können eintreffende Ereignisse (z.B. ein Alarm oder eine bestimmte Nachricht) korrekt bearbeitet werden bzw. die gewünschten Bearbeitungen angestoßen werden. Enthält mindestens ein „`onMessage …`“- oder „`onAlarm …`“-Handler.

Abschnitt 3: Prozess-Verlauf, Prozess-Steuerung[3]

```
[Abschnitt1 / Abschnitt2]
       Activity
```

In diesem Abschnitt findet die eigentliche Abarbeitung des BPEL-Prozesses statt. Abschnitt 1 und Abschnitt 2 sind die Vorbereitung bzw. Voraussetzungen, damit die eigentlichen Aufgaben auch korrekt ausgeführt werden können. So wurden im Vorfeld die Rahmenbedingungen geschaffen, damit der BPEL-Prozess kontrolliert und korrekt ablaufen kann.

Als `Activity` kommen folgende Aktionen, sowohl in „Abschnitt 2: allgemeine Definitionen zu Konnektivität, Datenaustausch und Kommunikation zwischen Web Services“ als auch in „Abschnitt 3: Prozess-Verlauf, Prozess-Steuerung“, in Frage:

[3] Auszug aus dem Abschnitt "The Structure of a Business Process" der BPEL-Spezifikation (Quelle [1]).

Tab. 1 BPEL-Befehle – „*activity*“

BPEL-Befehl	Erläuterung
`<receive>`	blockiere Empfänger bis gewünschte Nachricht eintrifft
`<reply>`	auf eine empfangene Nachricht antworten
`<invoke>`	Nachricht verschicken, dabei kann es sich um eine Einweg-Nachricht handeln oder ein request/response Nachrichtenpaar
`<assign>`	Wertzuordnung für Variable
`<throw>`	Ausnahme / Fehler Weitergabe
`<terminate>`	sofortiger Abbruch der Bearbeitungen
`<wait>`	eine festgelegte Zeitspanne abwarten
`<empty>`	keine Aktion wird ausgeführt - auch als Platzhalter verwendet – ermöglicht die Synchronisation
`<sequence>`	sequentielle Bearbeitung von Aktionen
`<switch>`	erlaubt die gezielte Auswahl eines geeigneten Bearbeitungszweiges
`<while>`	Schleifenkonstrukt, Berechnung erfolgt solange die Bedingung erfüllt ist
`<pick>`	erlaubt das Anhalten einer Aktion für einen bestimmten Zeitintervall oder bis zum Eintreffen einer speziellen Nachricht
`<flow>`	parallele Bearbeitung von Aktionen
`<scope>`	verschachtelte Aktivität mit eigenem Geltungsbereich
`<compensate>`	ermöglicht das Aufrufen der Kompensierung (erfolgte Aktionen und Bearbeitungen werden rückgängig gemacht), die in einem inneren Geltungsbereich die ansonsten nicht von der automatischen Kompensierung mitberücksichtigt worden wäre.

Nachdem nun die Struktur von BPEL-Prozesse eingeführt ist, können konkret die Unterschiede zwischen den executable- und den abstract-Prozessen im Hinblick auf Erweiterungen von Sprachkonstrukten und deren Verwendung analysiert werden. Diese Erweiterungen betreffen meist den executable Prozess. Es werden zusätzlich zu den unten aufgelisteten Erweiterungen weitere Fehlermeldungen definiert, die im Fehlerfall vom Programmierer aufgefangen und behandelt werden können.

Tab. 2 – Erweiterungen in BPEL für executable und abstrakten Prozesse [1]

BPEL-Strukturen	executable Prozess Erweiterungen	abstract Process Erweiterungen
Expressions[4]	zur Extraktion gewünschter Objekteigenschaften von Variablen	-
Umgang mit Variablen	Variablen müssen vor der ersten Verwendung initialisiert werden	Variablen müssen weder angegeben noch initialisiert werden
Wertzu-weisungen	Variablen können Werte anderer Variablen zugewiesen werden (Variablen müssen initialisiert worden sein)	Erlauben die Modellierung eines nicht deterministischen Verhaltens abstrakter Prozesse
Correlation	Eigenschaftswerte der `<correlationSets>` müssen für alle Nachrichten aller Operationen innerhalb der Laufzeit eines Bereiches (`scope`) identisch sein	-
Web Services Operationen	die Erweiterungen bei den Operationen betreffend dem Auffangen von Fehlern bei nicht 100%-ig definiertem Verhalten wie: a) 2 Bereiche erwarten die gleiche Nachricht. b) mehrere Antworten von synchronen Nachrichten fallen aus, so entsteht ein nicht definiertes Verhalten beim Eintreffen einer Antwort-Nachricht. c) die `inputVariable` ist in einem executable Prozess nicht optional, wird dieses Attribut vergessen, muss dieser Fehler aufgefangen werden.	-

[4] Expressions dienen der Extraktion von gewünschten Objekteigenschaften der Variablen, z.B. bestimmter Daten oder Eigenschaften. Sind die gewünschten Objekteigenschaften nicht in einer Variablen enthalten, müssen auch Fehlermeldungen und -behebung (bpws:selectionFailure) im Prozess definiert sein.

Terminierung einer Dienst-Instanz	mit <terminate> können sofort alle Ausführungen einer Prozessinstanz (jeweils dort wo Befehl aufgerufen wurde) terminiert werden	-
Kompen-sierung	<compensationHandler> dürfen nur einmal während der Ausführung einer Prozessinstanz aufgerufen werden	-
Event Handler	<eventHandler> behandeln vorhandene Ambiguitäten beim Empfang oder Senden von <onMessage> Nachrichten	-

Die ausführliche Behandlung der Business Process Execution Language soll helfen die bevorstehenden Aufgaben, nämlich dem Entwurf von CTC-Graphen und die Anpassung der zu entwerfenden Graphen an die Möglichkeiten und Strukturkonstrukte der BPEL Sprache zu bewältigen.

1.2 Das Eclipse-Umfeld

Eclipse ist eine Open Source Community mit Fokus auf erweiterbare Software-Entwicklungswerkzeuge und Anwendungsplattformen. Eclipse stellt eine erweiterbare Werkzeugplattform für den gesamten Lebenszyklus der Softwareentwicklung bereit und bietet Werkzeuge für Modellierung, Test und Performance, Business Intelligence und Embedded Entwicklung, IDEs für Java, C++ und weitere Sprachen sowie Werkzeuge und eine Plattform für Rich Client Applikationen. Weitere Informationen hierzu sind auf der Eclipse Homepage [10] zu finden.

Die Erweiterung der Eclipse Plattform geschieht mit Hilfe von sog. Plug-ins. Sie bringen auf eine einfache Art und Weise zusätzliche Funktionen und Bearbeitungsmöglichkeiten in die Plattform ein.
Die Eclipse-Plattform selbst ist modular aufgebaut, d.h. die Grundfunktionen sind in einem „Kern“ eingeschlossen. Die Hauptaufgabe des Kerns besteht im Aufrufen und Verwalten der oben angesprochenen Plug-ins, siehe auch Daum [9] i.V.m. der Eclipse Homepage [10].

Die Plug-ins bieten die Möglichkeit eine Eclipse-Umgebung um ausgewählte Funktionen zu erweitern. Somit kann die Plattform mit Werkzeugen (angeboten in Form von Plug-ins) angereichert werden, die bei der Lösung bestimmter Aufgaben unterstützend wirken können.

Zwischenzeitlich sind diverse Plug-ins erhältlich, die das Arbeiten mit Eclipse vereinfachen und neue Aspekte hinzufügen.
Als Beispiel für eine sinnvolle Erweiterung von Eclipse kann an dieser Stelle das kommerzielle (aber in der Grundversion kostenlose) UML – Plug-in „OMONDO“ genannt werden. Details hierzu unter The Live UML Company Homepage [24]. Dieses Plug-in erleichtert mit seiner einfachen Bedienung und gut strukturierten graphischen Oberfläche die Erstellung von UML Klassendiagrammen. OMONDO ist ein graphisches Editor-Plug-in für Eclipse auf Basis von **E**clipse **M**odelling **F**ramework (**EMF**) und **G**raphical **E**diting **F**ramework (**GEF**).

Weil beide Frameworks für die Entwicklung des CTC-Graphen Plug-ins (vgl. Kapitel 3) verwendet werden, folgt eine kurze Erläuterung. Auf eine tiefergehende Erklärung und Analyse beider Rahmenwerke (EMF / GEF) wird hier jedoch verzichtet, für weitere Informationen zu den einzelnen Themen sei auf das „Literaturverzeichnis“ (Seite 107 ff.) hingewiesen.

1.2.1 Eclipse Modelling Framework

Das **E**clipse **M**odelling **F**ramework (im Folgenden **EMF** genannt) ist ein Rahmenwerk, das die Möglichkeit der Modellierung und der Code-Generierung zur Erstellung von datenstruktur-abhängigen Tools und Applikationen bietet (vgl. Eclipse Documents [12]).
Von der in XMI[5] beschriebenen Modell-Spezifikation, bietet EMF verschiedene Tools und Laufzeitunterstützung, um dazugehörige Java-Klassen zu erzeugen.

EMF besteht im Wesentlichen aus drei Teilen:

1. **EMF**: Das Herzstück des Frameworks enthält ein Metamodell (Encore) zur Beschreibung von Modellen und für deren Laufzeitunterstützung (z.B. Änderungsmeldung, effiziente API zur Manipulation der EMF-Objekte etc.).

2. **EMF.Edit**: Das EMF.Edit Framework enthält generische, wieder verwendbare Klassen zur Erstellung eines Editors für beliebige EMF-Modelle.

3. **EMF.Codegen**: Die EMF Codegenerierungs-Einheit kann alles erstellen, was für einen Editor eines beliebigen EMF-Modells benötigt wird.

1.2.2 Graphical Editing Framework

Das **G**raphical **E**diting **F**ramework (im Folgenden **GEF** genannt) ermöglicht Entwicklern die Erzeugung komplexer graphischer Editoren auf der Grundlage von vorhandenen Applikations-Modellen. GEF basiert auf einer Model-View-

[5] XML Metadata Interchange (XMI), http://www.omg.org/technology/documents/formal/xmi.htm , accessed 28. May 2006

Controller (MVC) Architektur und gewährleistet damit eine klare Trennung zwischen Programmlogik, Funktionalität und Darstellungselementen.

Die MVC-Architektur besteht aus 3 Komponenten:

1. **Model**: Neben der Verwaltung und Repräsentation der Daten einer Anwendung führt das Model auch alle Änderungen der Daten aus. Das Model liefert auf Anfrage den Zustand der Daten und reagiert auf Befehle den Zustand zu ändern. Das Model kennt kein konkretes View und kann mit beliebig vielen Views verbunden sein. Über einen Verweis auf den aktuell verwendeten View können vom Model Änderungen darin vorgenommen werden.

2. **View-Objekt:** Das View-Objekt übernimmt die grafische Darstellung der Daten, in dem es eine mögliche visuelle Abbildung des Models darstellt. Ein View-Objekt ist dabei immer mit genau einem Model verbunden. Für den Fall, dass sich das Model ändert, erstellt das View-Objekt automatisch eine neue passende Darstellung des Models und stellt diese grafisch dar. Das View-Objekt besitzt immer eine Referenz auf ein konkretes Model. Über diese Referenz kann es die Methoden des Models direkt aufrufen.

3. **Controller:** Der Controller definiert die Art und Weise wie ein Benutzer mit der Applikation interagieren kann. Er nimmt Eingaben vom Benutzer entgegen und bildet diese auf Funktionen des Models oder des Views ab. Um diese Funktionalität zu gewährleisten, muss der Controller zwingend Zugriff auf das Model und den View erhalten, Näheres gem. Baray [3].

GEF besteht aus zwei Plug-ins:

1. **org.eclipse.draw2d**: Das Draw2D-Plug-in bietet Layout und Rendering Werkzeuge zur Darstellung der graphischen Elemente.

2. **org.eclipse.gef**: Das GEF-Plug-in bietet die Möglichkeit der Darstellung und Manipulation der mit Draw2D-Plug-in erzeugten graphischen Elemente. Selektions-, Erzeugungs-, Verbindungs- und Markierungswerkzeuge wie auch eine Palette zur Verwendung dieser Werkzeuge stehen zur Verfügung. Zwei unterschiedliche Viewer (zur Graphischen- oder Baumstruktur-Darstellung) werden ebenfalls vom org.eclipse.gef Plug-in angeboten. Darüber hinaus bietet dieses Plug-in einen Controller-Framework zur Abbildung eines Modells in einem bestimmten View. Undo- /Redo-Kommandos werden mit Hilfe eines Commando Stacks unterstützt. Wie die Beschreibung des GEF-Plug-ins schon erahnen lässt, benötigt es org.eclipse.draw2d vgl. Eclipse Documents [13].

2 Koordinierte transaktionale Kommunikations-Graphen und Anregungen aus der Graphentheorie

Wie in der Einleitung bereits dargelegt, besteht ein Teil der Aufgabenstellung dieser Arbeit darin, ein geeignetes Modell für den Entwurf der koordinierten transaktionalen Kommunikations-Graphen zu konzipieren. Die bereits beschriebenen Abbildungen 1-2, 1-3 ,1-4 und 1-5 sind bei der formalen Beschreibung und Erstellung der CTC-Graphen behilflich.

An dieser Stelle kann bereits erwähnt werden, dass sich alle bestehenden Graphentypen nur bedingt für die Aufgabe der CTC-Graphen eignen, was die Konzeption eines eigenen Graphen-Typs notwendig macht.
Für die Formale Beschreibung und Konzeption der CTC-Graphen (Zustände, Zusammenhänge, Nachrichtenfluss etc.) könnten die nachfolgend aufgeführten etablierten und bekannten Ansätze der Graphentheorie als Ideen-Lieferanten dienen.

Folgende Graphen-Arten werden hierbei analysiert:

1. Zustandsdiagramme
2. Aktivitätsdiagramme
3. Sequenzdiagramme
4. Kommunikationsdiagramme (ehem. Kollaborationsdiagramme)
5. Petrinetze

Zustandsdiagramme, Aktivitätsdiagramme, Sequenzdiagramm und Kollaborationsdiagramme gehören zu der Gruppe der Verhaltensdiagramme. Sie sind in der Lage die dynamischen Aspekte eines Systems zu modellieren.
Die aufgeführten Graphen gehören zu den in der **U**nified **M**odelling **L**anguage[6] (**UML**) vgl. OMG Homepage [22] und UML Resource Page [26] aufgeführten dreizehn Diagramm-Arten. Die im Rahmen dieser Arbeit angefertigten Skizzen, Beschreibungen etc. wurden mit der neuste UML-Version – nämlich UML2 vgl. Oestereich [23] Rupp et al [25] – angefertigt.

Von zentraler Bedeutung bei der Erstellung von CTC-Graphen ist deren Übersichtlichkeit. In der vorliegenden Arbeit wurde darauf besonderes Augenmerk, sowohl bei der Analyse der bereits existierenden Modelle als auch beim Entwurf der CTC-Graphen gelegt. Eine gute Übersicht ist die Vorraussetzung für das schnelle und klare Verständnis des dargestellten Sachverhaltes und schafft darüber hinaus die Möglichkeit der expliziten Einbindung benötigter Daten bspw. WSDL-Fragmenten oder die besondere Attributierung von Graphen-Elementen. Je eindeutiger und detaillierter die Graphen-Zustände, Nachrichten bzw. der Nachrichtenfluss dargestellt werden, desto genauer kann anschließend die Abbildung des Graphen in BPEL-Code erfolgen.

[6] **UML** – Modellierungssprache für Software und andere Systeme

Im folgenden Abschnitt werden die oben aufgelisteten Graphen-Modelle kurz beschrieben. Insbesondere auf den Aufbau und Merkmale wird hierbei eingegangen. Anschließend wird analysiert inwieweit sich diese Graphen-Modelle für die vorliegende Arbeit als hilfreich erweisen könnten.

2.1 Zustandsdiagramme

Zustandsdiagramme (auch Zustandsübergangsdiagramm genannt) sind wohl die bekannteste und am häufigsten verwendete Diagramm-Art. Sie zeichnen sich durch kompakte und überschaubare Darstellung komplexer Sachverhalte aus.

Anhand eines Zustandsdiagrammes können beispielsweise die verschiedenen Zustände (z.B. Änderung bestimmter Attributwerte), die ein Objekt im Laufe seines Lebens einnehmen kann und Ereignisse, die auf Grund einer Zustandsänderung eintreten können, dargestellt werden (vgl. Oestereich [23]).

In UML beschreibt ein Zustandsdiagramm einen erweiterten endlichen Automaten, der eine Menge von Zuständen (Startzustand, ein oder mehrere Übergangszustände. Endzustände) besitzt und sich zu jedem Zeitpunkt in einer Menge endlicher Zustände befindet. Der Übergang von einem Zustand in den Nächsten erfolgt ohne zeitliche Verzögerung und folgt einer vorgegebenen Zustandsübergangsfunktion (vgl. Rupp et al [25]).

Erweiterte endliche Automaten bieten die Möglichkeit das Systemverhalten in immer kleinere und einfachere Teile zu zerlegen, was bei endlichen Automaten nicht realisierbar ist. Dies bringt insbesondere Erleichterungen bei der Codierung und im Testing mit sich.

Ein weiterer Vorteil erweiterter endlicher Automaten besteht darin, parallel ablaufende Zustandsautomaten zu modellieren, was gem. „UML 2 glasklar – Praxiswissen für die UML-Modellierung und -Zertifizierung“ [25] für den Entwurf von Verteilten Systemen durchaus von Nutzen sein kann.

2.1.1 Formale Beschreibung, Graphische Bestandteile und Aufbau von Zustandsdiagrammen

Formal lassen sich Zustandsdiagramme beschreiben durch:

- Eine endliche, nicht leere Menge von Zuständen Q – sog. Knoten
- Eine endliche, nicht leere Menge von Ereignissen M (Eingabealphabet) – sog. Kanten
- Eine Zustandsüberführungsfunktion δ: Q x M $\rightarrow$ Q
- Einen Anfangszustand q_0 ($q_0 \in Q$)
- Eine Menge von Endzuständen Q_e ($Q_e \subseteq Q$)

Im nachfolgenden Beispiel „einfacher Drucker“ soll die prinzipielle Funktion von Zustandsdiagrammen erläutert werden.

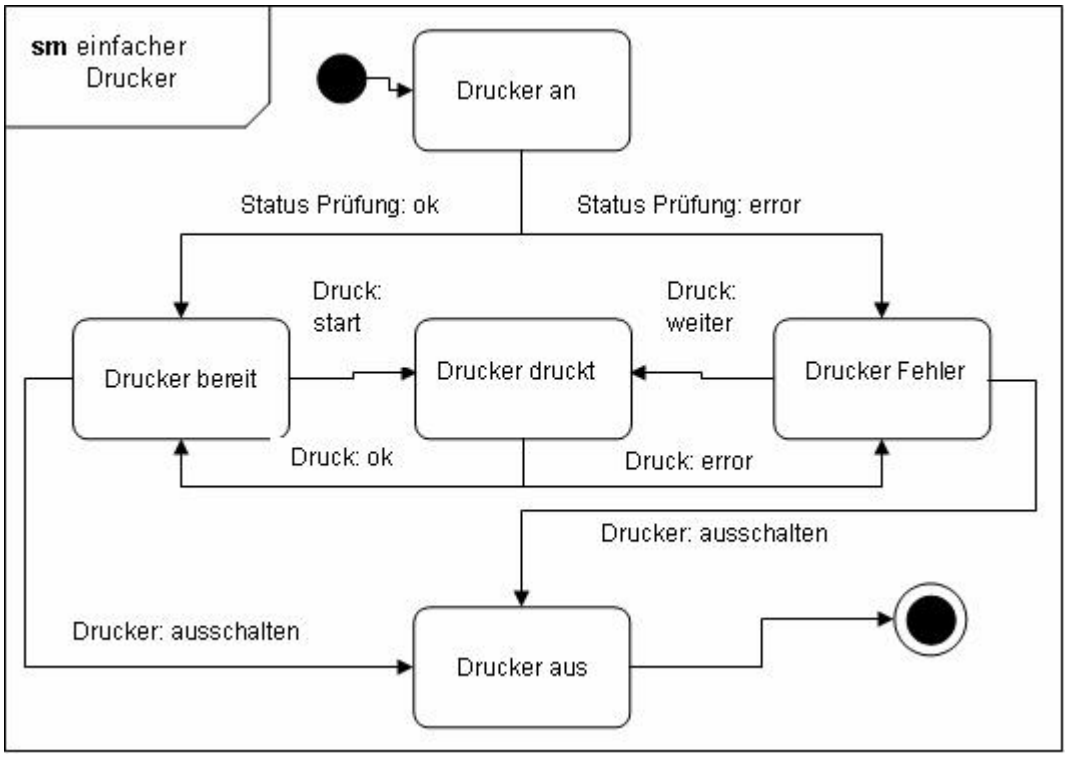


Abb. 2-1: Zustandsdiagramm - Beispiel "einfacher Drucker"

Abbildung 2-1 zeigt ein Zustandsdiagramm mit einem Kopf- und einem Inhaltsbereich. Der Kürzel „sm“ im Kopfbereich des Diagramms, weist auf einen Zustandsautomaten – auf Englisch „**s**tate **m**achine“ – hin.

Die möglichen Zustände werden in diesem Zustandsdiagramm durch ein Rechteck mit abgerundeten Ecken (hier: „Drucker an“, „Drucker bereit“, etc.) dargestellt. Zustandsübergänge werden durch gerichtete Kanten dargestellt.
Die Kanten sind jeweils mit Ereignissen beschriftet, die zum jeweiligen Zustandübergang führen können (hier: „Status Prüfung: ok“, „Drucker ausschalten“, etc.).

Der Startpunkt bei der Traversierung eines Zustandsdiagramms ist stets der Anfangszustand q_0. Von q_0 ausgehend, gelangt man mit den vordefinierten Ereignissen M und der Zustandsüberführungsfunktion δ: Q x M → Q zu weiteren definierten Zuständen der Menge Q. Am Ende einer korrekten Eingabe gelangt man zu einem Endzustand aus Q_e.

Zur Visualisierung von verschiedenen Zuständen (z.B. Änderung von Attributwerten) eines bestimmten Objektes werden Zustandsdiagramme verwendet.

Wie anhand dieses Beispiels verdeutlicht, sind Zustandsdiagramme eine einfache und intuitive Art der Modellierung. Darüber hinaus eignen sie sich für die Visualisierung (endlicher) Automaten.

2.2 Aktivitätsdiagramme

Aktivitätsdiagramme sind eine spezielle Art von Zustandsdiagrammen, die überwiegend für die Modellierung von Aktivitäten herangezogen werden. Mit Hilfe dieser Diagramme können dynamische Aspekte des zu modellierenden Systems angezeigt und beschrieben werden.

Unter einer Aktivität versteht man einen Zustand mit einer internen Aktion und einer oder mehrerer ausgehender Transitionen, welche jeweils durch definierte Bedingungen voneinander unterschieden werden. Jede Aktivität stellt einen einzelnen Ablaufschritt dar (gem. Oestereich [23]).

Mit Aktivitätsdiagrammen wird der Ablauf eines Anwendungsfalls, einer Klasse oder einer Operation beschrieben. Es eignet sich jedoch nur zur Modellierung aller Aktivitäten interner Ablaufmöglichkeiten eines Systems vgl. Rupp et al [25].

Legt man bei der Modellierung primär Wert auf Zustände und die Übergänge zwischen ihnen – weniger auf die Aktivitäten und den darin enthaltenen Aktionen – empfiehlt es sich eine andere Diagrammart zur Visualisierung des gewünschten Sachverhaltes zu verwenden (z.B. Zustandsdiagramme).

2.2.1 Formale Beschreibung, Graphische Bestandteile und Aufbau von Aktivitätsdiagrammen

Gem. Oestereich [23] i.V.m. Rupp et al [25] werden Aktivitätsdiagramme wie folgt formal beschrieben:

- Eine endliche, nicht leere Menge von Objektknoten Q – sog. Knoten
- Eine endliche, nicht leere Menge von Aktionen A – sog. Kanten
- Eine Anfangsaktivität a_0 ($a_0 \in$ A)
- Eine Endaktivität a_e ($a_e \in$ A)
- Eine endliche, nicht leere Menge Kanten K

Im nachfolgenden Diagramm werden die elementaren Funktionen eines Aktivitätsdiagramms anhand des Beispiels „Datei drucken“ verdeutlicht.

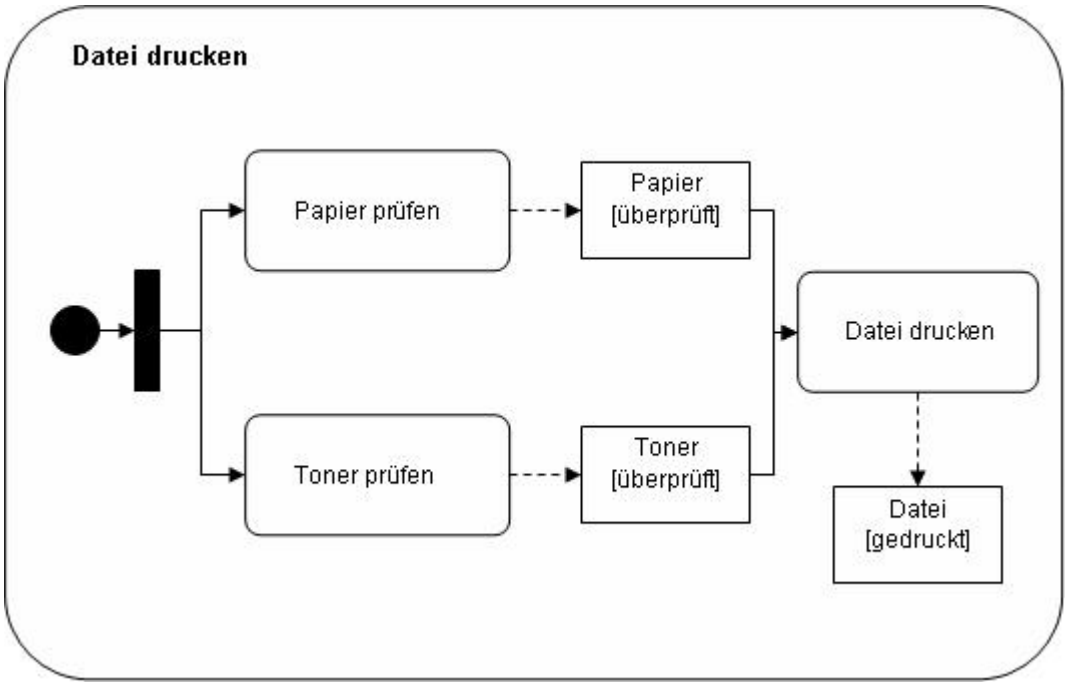


Abb. 2-2: Aktivitätsdiagramm - Beispiel „Datei drucken“

Abbildung 2-2 zeigt ein UML2 Aktivitätsdiagramm mit einem Kopf- und einem Inhaltsbereich. Das in früheren UML-Versionen übliche Schlüsselwort „act“ im Kopfbereich wurde bei UML2 abgeschafft, dafür ist nun das gesamte Diagramm von einem Rechteck mit abgerundeten Kanten eingefasst. Im Kopfbereich befindet sich der Name der modellierten Aktivität (hier: „Datei drucken“).

Objektzustände im Aktivitätsdiagramm werden als Rechtecke dargestellt, die den Namen des Zustandes und in eckigen Klammern den Objektzustand enthalten (hier: „Papier [überprüft]“, etc.).
Aktivitäten werden als Rechtecke mit abgerundeten Ecken dargestellt (hier: „Toner prüfen“, etc.).

Aktivitäten und Objektzustände werden durch gestrichelte Kanten (sog. Activity Edges – hier z.B. zwischen „Papier prüfen“ und „Papier [überprüft]“) verbunden. Aktivitäten werden untereinander mit durchgezogenen gerichteten Kanten verbunden. Der Übergang wird implizit durch den Abschluss der Aktivität ausgelöst.

Des Weiteren könnten die ausgehenden gerichteten Kanten mit Bedingungen versehen werden, diese sind dann in eckigen Klammern jeweils angeheftet (in Abb. 2-2 nicht dargestellt). Bei den Bedingungen handelt es sich um boolsche Ausdrücke. Anstatt Bedingungen, die eine Startaktivität verlassen, direkt an die Kanten zu binden, wird eine nicht gefüllte Raute gezeichnet, von der aus die Kanten mit ihren verschiedenen Bedingungen, zu den jeweiligen Zielaktivitäten, abgehen.

Kanten können auch synchronisiert und geteilt werden. Diese Vorgänge werden durch breite senkrechte Balken dargestellt, die auf Kanten treffen oder von denen abgehen (siehe Abb. 2-2: Aktivitätsdiagramm-Beispiel „Datei drucken“).

Der Startpunkt bei der Traversierung eines Aktivitätsdiagramms ist immer die Anfangsaktivität a_0, die über eine Kante evtl. mit einem Objektknoten verbunden ist.

Von a_0 ausgehend gelangt man nach Beendigung der Aktion (unter Beachtung der vorgegebenen boolschen Bedingungen) zu weiterführenden Aktionen. Dabei können sich auch Objektzustände ändern. Die Objektzustände oder ihre Änderung sind allerdings nicht das Hauptaugenmerk von Aktivitätsdiagrammen. Objektzustände dienen lediglich der Vervollständigung oder dem besseren Verständnis des Diagramms (vgl. Oestereich [23]).
Am Ende der Bearbeitung gelangt man zu einer Endaktivität a_e, spätestens an diesem Punkt erreicht das Objekt einen weiteren, nämlich den letzten Objektzustand.

Wie bereits durch die beschriebene Vorgehensweise bei der Traversierung eines Aktivitätsdiagramms verdeutlicht wird, benötigen Aktivitätsdiagramme im Gegensatz zu Zustandsdiagrammen keine Zustandsübergangsfunktion. Bei dieser Diagrammart werden elementare Aktionen vernetzt, die ihrerseits mit Kontroll- und Datenflüssen verbunden sind. Die Semantik der Aktivitätsdiagramme beschreibt kausale Zusammenhänge.

Einleitung Interaktionsdiagramme

Wie in der Einleitung bereits angekündigt, werden auch Sequenz- und Kommunikationsdiagramme (ehem. Kollaborationsdiagramme) analysiert.

Diese beiden Diagramm-Typen sind Teil einer eigenen „Diagramm-Unterklasse“, den sog. Interaktionsdiagrammen. In Interaktionsdiagrammen sind Ereignisse und Ereignismengen Auslöser für den Kommunikationsablauf.

Die wichtigsten Bestandteile der Interaktionsdiagramme sind (gem. Rupp et al [25]):

1. Lebenslinien
2. Kommunikationspartner
3. Interaktionen (das Zusammenspiel der Kommunikationspartner)
4. Nachrichten
5. Sprachmittel für die Flusskontrolle

2.3 Sequenzdiagramme

Ein Sequenzdiagramm ist die graphische Darstellung einer Interaktion und spezifiziert den Austausch von Nachrichten zwischen Objekten, die im Diagramm als Lebenslinien (siehe Abb. 2-3) dargestellt sind.

Eine Sequenz ist hier eine Reihe von Nachrichten, die eine ausgewählte Menge von Objekten in einer zeitlich begrenzten Situation austauschen.

2.3.1 Formale Beschreibung, Graphische Bestandteile und Aufbau von Sequenzdiagrammen

Formal lassen sich Sequenzdiagramme beschreiben durch (vgl. Oestereich [23]):

- Eine endliche, nicht leere Menge von Objekten O – sog. Knoten
- Eine endliche, nicht leere Menge von Interaktionen I – sog. Kanten
- Evtl. einem Startobjekt s_0 ($s_0 \in O$)

Im nachfolgenden Diagramm werden die elementaren Funktionen eines Sequenzdiagramms anhand des Beispiels „Ausdruck“ verdeutlicht.

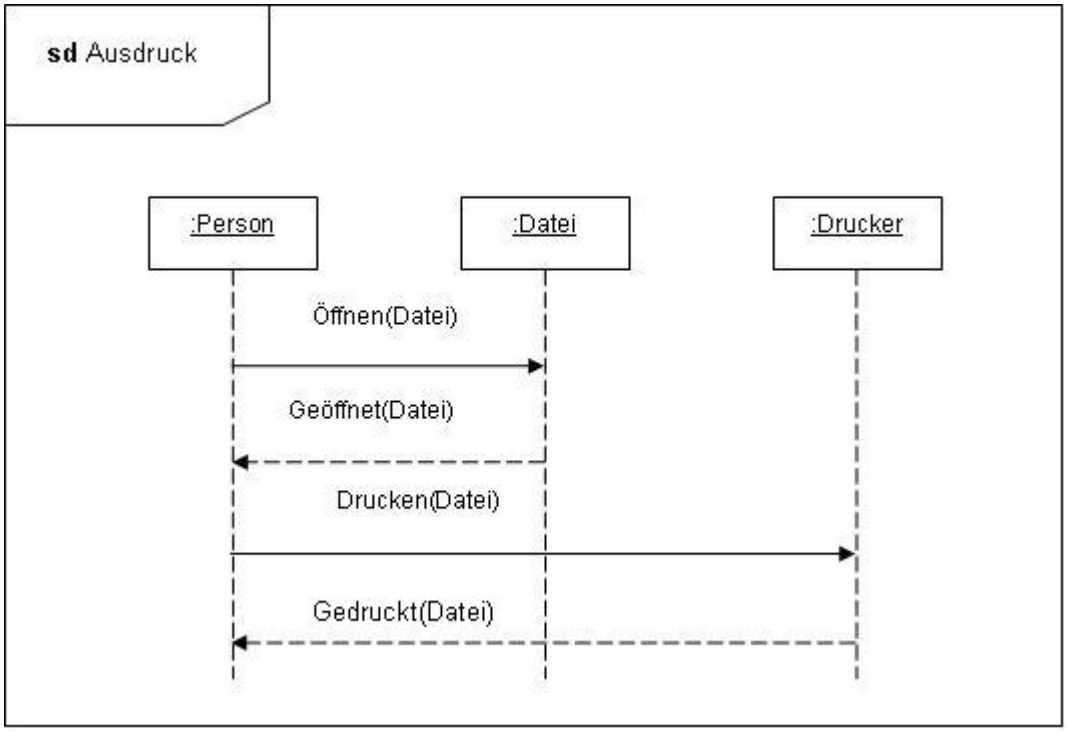


Abb. 2-3: Sequenzdiagramm - Beispiel "Ausdruck"

Die Abbildung 2-3 zeigt ein UML2 Sequenzdiagramm mit einem Kopf- und einem Inhaltsbereich. Das Schlüsselwort im Kopfbereich ist bei einem Sequenzdiagramm (auf Englisch „**s**equence **d**iagramm“) „sd“ (oder „interaction“).
Im Inhaltsbereich sind drei Objekte (:Person, :Datei, :Drucker) mit ihren Lebenslinien und deren Interaktionen untereinander dargestellt. Die zu erfüllende Aufgabe in diesem Sequenzdiagramm, ist das Ausdrucken einer beliebigen Datei (:Datei) eines Anwenders (:Person) an einem bestimmten Drucker (:Drucker).
Um diese Aktion durchführen zu können, sind die Aktionen „Öffnen(Datei)“ und „Drucken(Datei)“ in genau dieser Reihenfolge auszuführen. Die Aktionen „Geöffnet(Datei)“ und „Gedruckt(Datei)“ sind die jeweiligen Reaktionen auf die vom Anwender ausgeführten Arbeitsschritte.

Startpunkt bei der Traversierung eines Sequenzdiagramms ist entweder das Startobjekt s_0 oder eine Interaktion, die ein Objekt aktiviert (siehe Abb. 2-3: Sequenzdiagramm - Beispiel "Ausdruck"). Von diesem ersten aktivierten Objekt wird die Bearbeitung angesteuert, dabei werden Nachrichten an andere Objekte gesendet.

Wie bereits angedeutet, steht bei Sequenzdiagrammen der zeitliche Verlauf der Nachrichten im Vordergrund. Die durch Nachrichten verursachten Zustandsübergänge sind irrelevant. Interaktionen sind kompliziert und müssen sehr stark gesteuert werden. Die strukturelle Verbindung zwischen den Kommunikationspartnern ist uninteressant. Sie zeigt lediglich Ablaufdetails.

Die Objekte werden mit senkrechten gestrichelten Lebenslinien dargestellt. Der zeitliche Verlauf der Nachrichten wird hierdurch hervorgehoben. Die Zeit läuft von oben nach unten ab. Über der Lebenslinie steht ein Rechteck, das sog. Objektsymbol, welches den Namen des Objektes enthält.

Die Nachrichten werden als gerichtete Kanten zwischen den Objekt-Linien dargestellt. Auf ihnen wird die Nachricht in Form von *nachricht(parameter)* notiert. Die Antwort auf eine gesendete Nachricht wird in Textform *antwort:=nachricht()* oder ebenfalls als Pfeil dargestellt. Nachrichten können zusätzlich mit boolschen Bedingungen in der Schreibweise *[Bedingung] nachricht()* angereichert werden. Diese werden in Abhängigkeit von den Bedingungen gesendet.

Die gestrichelte Lebenslinie kann durch breite, nicht ausgefüllte, senkrechte Balken überlagert werden. Diese Balken symbolisieren den Steuerungsfokus. Er gibt an, welches Objekt gerade mit der Ausführung von Aktionen, der sog. Aktionssequenz aktiv ist. Am (linken oder rechten) Rand des Steuerungsfokus können frei formulierte Erläuterungen, Zeitanforderungen, Zusicherungen etc. gem. Rupp et al [25] notiert werden.

Ebenfalls erwähnenswert ist die Tatsache, dass mit Hilfe von Sequenzdiagrammen auch das Erzeugen und das Entfernen von Objekten darstellbar ist. Die Konstruktion eines neuen Objektes kann durch eine Nachricht (Erschaffungsnachricht), die auf ein Objektsymbol trifft, angezeigt werden. Die Destruktion eines Objektes wird durch ein Kreuz am Ende der Lebenslinie abgebildet. Nach der Destruktion der Lebenslinie können keine zeitlichen Ereignisse an die Linie gerichtet werden bzw. dort eintreffen.

Auch die Darstellung von Iterationen ist bei Sequenzdiagrammen möglich. Hierzu wird das Iterationssymbol „*“ (sog. Iterationsmarke) vor der Nachricht verwendet.

Ein Sequenzdiagramm stellt in der Regel einen Weg durch einen Entscheidungsbaum innerhalb eines Systemablaufes dar. Es eignet sich nicht als Übersicht für alle Entscheidungsmöglichkeiten. Für diese Zwecke eignen sich Aktivitätsdiagramme oder Zustandsdiagramme besser (vgl. Oestereich [23]).

Ein weiterer Vorteil dieser Diagrammart liegt darin, dass Synchrone-, Asynchrone- und Antwort-Nachrichten in Sequenzdiagrammen dargestellt werden können. Die Nachrichten werden jeweils durch gerichtete Kanten mit geschlossenen Pfeilspitzen (Synchrone-Nachrichten), gerichtete Kanten mit offenen Pfeilspitzen (Asynchrone-Nachrichten) bzw. gerichtete gestrichelte Kanten mit geschlossener Pfeilspitze (Antwort-Nachrichten) dargestellt (vgl. Rupp et al [25]).

2.4 Kommunikationsdiagramme

Kommunikationsdiagramme, früher auch als Kollaborationsdiagramme bezeichnet, sind eng mit den Sequenzdiagrammen verwandt. Im Grunde zeigen beide Diagramm-Arten den gleichen Sachverhalt, jedoch jeweils aus einer anderen Perspektive.

Bei einem Kommunikationsdiagramm stehen die Objekte und ihre Zusammenarbeit an Stelle von Abläufen der Kommunikationspartner im Vordergrund, zwischen ihnen werden ausgewählte Nachrichten angezeigt. Der zeitliche Verlauf der Kommunikation zwischen den Objekten, der bei den Sequenzdiagrammen im Vordergrund steht, wird bei den Kommunikationsdiagrammen durch die Nummerierung der Nachrichten verdeutlicht.

Kommunikationsdiagramme sind übersichtlicher und einfacher zu modellieren als Sequenzdiagramme. Diese Art von Diagrammen ist bei Architektur- und Designprinzipien klar zu bevorzugen. Bemerkenswert ist, dass keine Äquivalenz zwischen Kommunikationsdiagramme und Sequenzdiagramme besteht. Weiteres siehe hierzu unter „UML 2 glasklar – Praxiswissen für die UML-Modellierung und -Zertifizierung“ [25].

Damit zwei Objekte miteinander kommunizieren können, muss der Sender der Nachricht eine Referenz auf das Empfängerobjekt haben, dies wird graphisch als eine Verbindungslinie zwischen den Kommunikationspartnern dargestellt.
Die Objektverbindung kann grundsätzlich oder temporär bzw. lokal vorhanden sein. Ohne dass eine Assoziation vorhanden ist, kann sich das Objekt stets selbst Nachrichten schicken, sofern man dies in Anspruch nehmen möchte.

Die zeitliche Abfolge der Nachrichten, die Namen und Antworten und ihre möglichen Argumente werden im Kommunikationsdiagramm dargestellt. Analog zu Sequenzdiagrammen, die eng mit Kommunikationsdiagrammen verwandt sind, können auch bei dieser Diagramm-Art Iterationen bzw. Nachrichten-Schleifen dargestellt werden (vgl. Oestereich [23]).

2.4.1 Formale Beschreibung, Graphische Bestandteile und Aufbau von Kommunikationsdiagrammen

Formal lassen sich Kommunikationsdiagramme beschreiben durch:

- Eine endliche, nicht leere Menge von Objekten O – sog. Knoten
- Eine endliche, nicht leere Menge von (ausgewählten) Nachrichten N – sog. Kanten
- Eine Startnachricht n_0 ($n_0 \in N$)

Im nachfolgenden Diagramm werden die elementaren Funktionen eines Kommunikationsdiagramms anhand des Beispiels „Kinobesuch“ verdeutlicht (gem. Oestereich [23]).

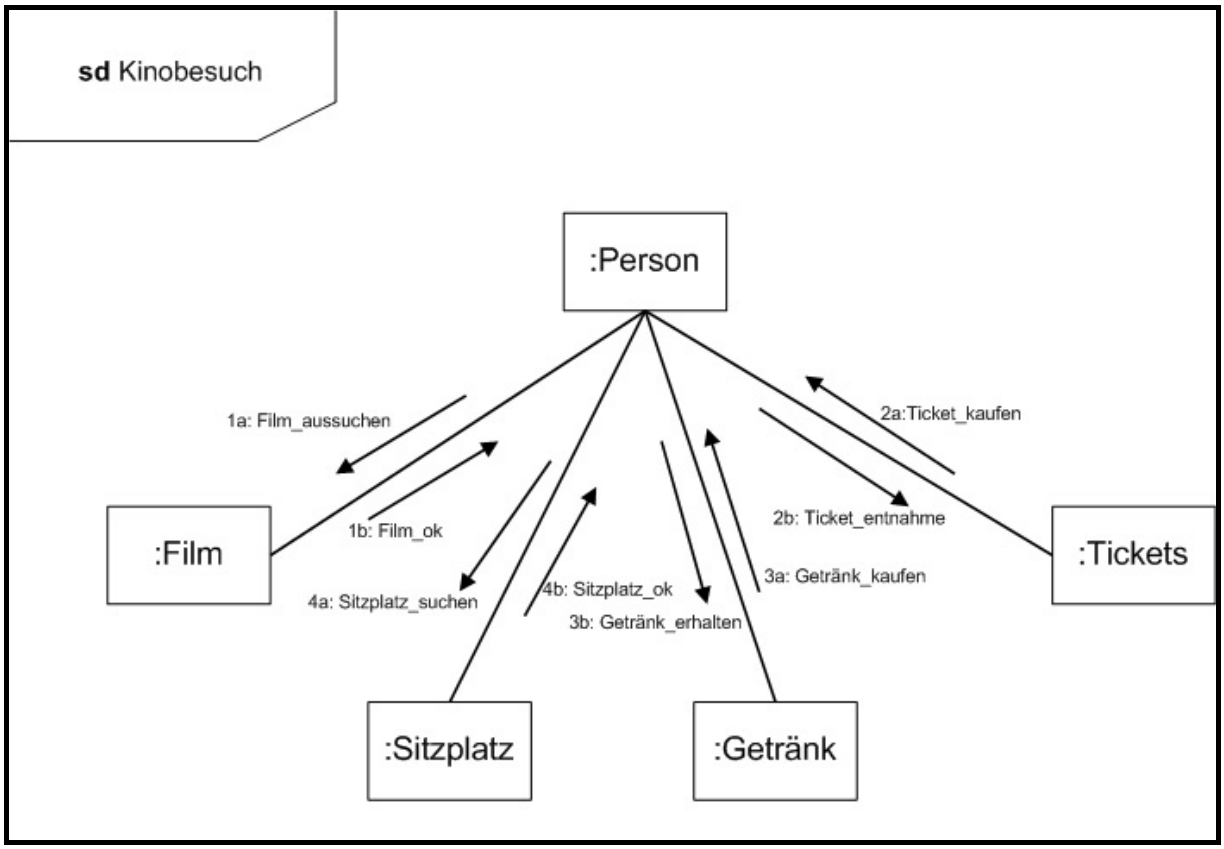


Abb. 2-4: Kommunikationsdiagramm - Beispiel "Kinobesuch"

Abbildung 2-4 zeigt ein UML2-Kommunikationsdiagramm mit einem Kopf- und einem Inhaltsbereich. Das Schlüsselwort im Kopfbereich lautet bei einem Kommunikationsdiagramm „sd“ (oder „interaction“) – hierdurch besteht Verwechslungsgefahr mit Sequenzdiagrammen. Die Lebenslinie der Objekte (:Person, :Film, :Sitzplatz, :Getränk, :Tickets) wird nicht gezeichnet sondern nur Rechtecke, die ihren Namen enthalten. Es wird eine Verbindungslinie zwischen den interagierenden Objekten gezogen. Entlang der Verbindungslinie werden die Aktionen beschrieben. Die Reihenfolge der ablaufenden Aktionen wird mittels der eindeutigen und aufsteigenden Nummerierung verdeutlicht („Film_aussuchen“-Aktion bevor „Ticket_kaufen“-Aktion).

Ein weiterer Unterschied zu den Sequenzdiagrammen liegt darin, dass bei Kommunikationsdiagrammen keine sichtbare Unterscheidung zwischen Synchronen- und Asynchronen-Nachrichten getroffen wird. Auch verloren gegangene Nachrichten werden nicht modelliert (vgl. Rupp et al [25]). Die Pfeilrichtungen der Aktionen zeigen lediglich die Richtung der Interaktion an.

Der Startpunkt bei der Traversierung eines Kommunikationsdiagramms ist die Startnachricht n_0. Sie aktiviert das erste Objekt, das anschließend mit der Bearbeitung beginnt. Das erste Objekt sendet dann Nachrichten an andere Objekte und startet damit die Kommunikation.

Ähnlich wie in einem Sequenzdiagramm, werden in einem Kommunikationsdiagramm Objekte als Lebenslinien dargestellt, an denen oben ein Rechteck angebracht ist. Im Gegensatz zu einem Sequenzdiagramm, wird die

gestrichelte Lebenslinie im Kommunikationsdiagramm, welche für die Zeitachse während des Austauschs von Nachrichten steht, nicht angezeigt. Das Objekt wird lediglich durch ein Rechteck mit seinem Namen darin dargestellt.

Eine Nachricht wird als kurzer Pfeil abgebildet, dessen Richtung vom Sender zum Empfänger zeigt. Die Nachricht „wandert" entlang einer Verbindungslinie zwischen den Objekten. Der Pfeil ist unter anderem mit einer Sequenznummer beschriftet.

Die erste Nachricht erhält die Sequenznummer 1. Trifft eine Startnachricht von außerhalb ein (z.B. durch einen Prozess der die Kommunikation initiiert), erfolgt die Darstellung ohne Sequenznummer. Eine Startnachricht kann graphisch auch wahlweise von einem Akteur-Symbol ausgehen (in Abb. 2-4: Kommunikationsdiagramm - Beispiel "Kinobesuch" nicht dargestellt).

Folgende Syntax liegt der Nachrichtenbezeichnung, dem sog. Sequenzbezeichner, zugrunde:

Vorgänger-Bedingung SequenzNr. Antwort:= NachrichtBez(Parameterliste)

Die einzelnen Bestandteile haben folgende Bedeutung:

- *Vorgänger-Bedingung*: Eine Aufzählung der Sequenznummern der bereits gesendeten Nachrichten. Diese werden durch Kommas getrennt. So kann die Synchronisation herbeigeführt werden. Mit einem „/" wird die Aufzählung der Sequenznummern beendet. Die Vorgänger-Bedingung ist optional (Bsp.: 1.1, 2.3 /) – ähnlich der Nummerierung der Kapitel dieser Diplomarbeit.

- *Sequenzausdruck*: Um die Reihenfolge der Nachrichten anzuzeigen, werden sie aufsteigend nummeriert. Sofern sich eine Nachricht im gleichen Kontext befindet wie die empfangene Nachricht, erhält sie eine durch einen Punkt „." getrennte Unter-Sequenznummer. So kann die Verschachtelung und Abhängigkeit der Nachrichten dargestellt werden (Bsp.: Nachricht 1.2.3 folgt der Nachricht 1.2.2 – beide wurden während der Interpretation von Nachricht 1.2 gesendet). Anstelle von Nummern sind auch Zeichenfolgen zur Verwendung denkbar. Sofern der Sequenzausdruck angegeben ist, wird er mit einem Doppelpunkt „:" abgeschlossen.

- *Antwort*: Die von einer Nachricht gelieferte Antwort kann mit einem Namen versehen werden. Dieser Name kann wiederum als Argument in anderen Nachrichten verwendet werden. Der Gültigkeitsbereich dieser Namen ist, analog zu lokalen Variablen, innerhalb der zu sendenden Nachricht. Des Weiteren können die Namen die gesamte Nachricht und nicht nur ein Teil davon enthalten.

- *NachrichtenBez(Parameterliste)*: Die Nachrichten Bezeichnung ist in der Regel gleich lautend wie die Operation, die diese Nachricht interpretiert. Angegeben wird die Signatur der verwendeten Operation.

Iterationen werden, wie auch bei Sequenzdiagrammen, mit einem Stern „*“ markiert. Eine parallele Ausführung wird durch zwei senkrechte Linien „||“ gekennzeichnet.

Mittels von Pseudo-Code oder tatsächlichen Programmiersprachen können Bedingungen an die jeweiligen Nachrichten angeheftet werden. Ist eine bestimmte Bedingung erfüllt, wird die zugehörige Nachricht versandt. So ist es möglich, diverse individuelle Szenarien und generelle Interaktionsstrukturen darzustellen.

Auch bei Kommunikationsdiagrammen können Objekte erzeugt oder zerstört werden. Dies kann mit den jeweiligen Kennzeichnungen <<*new*>> bzw. <<*destroyed*>> verdeutlicht werden. <<*transient*>> werden kurzlebige Objekte genannt, die nicht gespeichert und jeweils nach Bedarf erzeugt und wieder zerstört werden können.

Eine bemerkenswerte Eigenschaft der Kommunikationsdiagramme besteht darin, dass auf Rollennamen (Attributnamen) zurückgegriffen werden kann, um die Beziehung zwischen zwei benachbarten und miteinander kommunizierenden Objekten darzustellen.

Einer der folgenden Stereotype in Tabelle XY kann verwendet werden um den Rollennamen (Attributnamen) zu verdeutlichen (vgl. Oestereich [23]).

Tab. 3 - Stereotype von Rollenamen in Kommunikationsdiagramme

Stereotype	Erklärung
<<association>>	Der Objektbeziehung liegt eine Assoziation, Aggregation oder Komposition zugrunde. Dies ist der Standardfall, die Angabe ist nicht zwingend erforderlich.
<<global>>	Das empfangende Objekt ist global.
<<local>>	Das empfangende Objekt ist lokal in der sendenden Operation und somit entweder <<*new*>> oder <<*transient*>>.
<<parameter>>	Das empfangende Objekt ist ein Parameter in der sendenden Operation.
<<self>>	Das empfangende Objekt ist das sendende Objekt.

2.5 Petrinetze

Ein Petrinetz ist ein bipartiter und gerichteter Graph, der aus zwei Objektsorten besteht: Aus Stellen (places) und Transitionen (transitions). Stellen und Transitionen können durch gerichtete Kanten verbunden werden (Flussrelation F).

Es gibt keine direkte Verbindung zwischen zwei Stellen oder zwei Transitionen. Stellen vertreten die passiven bzw. Transitionen die aktiven Bestandteile von Petrinetzen.
Die Stellen repräsentieren lokale Zustände des Systems, die während des Ablaufs (des Systems) erfüllt werden könnten.

Die Transitionen stellen Aktionen oder Ereignisse dar, für deren Ausführung bzw. Eintreten bestimmte Zustände Voraussetzung sind. Nach dem Auftreten dieses Vorganges können bestimmte Zustände erreicht werden. In Bezug auf Transitionen können Zustände auch als Bedingungen aufgefasst werden. Der Übergang zwischen den verschiedenen Zuständen des Systems durch Transitionen wird auch als „schalten“ bezeichnet. Die dadurch eintretende Veränderung der Zustände wird durch die Flussrelation F (s.u. Formale Beschreibung) codiert.

2.5.1 Formale Beschreibung, Graphische Bestandteile und Aufbau von Petrinetzen

Formal lassen sich Petrinetze beschreiben durch:

- Eine endliche, nicht leere Menge von Stellen S
- Eine endliche, nicht leere Menge von Transitionen T
- Eine endliche, nicht leere Menge von Kanten (Flussrelation); $F \rightarrow (SxT \cup TxS)$
- Eine Kapazitätsfunktion für jede Stelle K: $S \rightarrow N\backslash\{0\}$
- Eine Gewichtsfunktion, für die Kosten der Kanten W: $F \rightarrow N\backslash\{0\}$
- Eine anfängliche Markenbelegung m_0 ; Abbildung $S \rightarrow N$

Zusätzlich gilt: $0 \leq M(s_i) \leq K(s_i)$ und $S \cap T=\varnothing$ (gem. Marczinzik [19]).

Das nachfolgende Diagramm zeigt die prinzipielle Funktionsweise eines Petrinetzes mittels des Beispiels „Kunde / Schalter Zuordnung“.

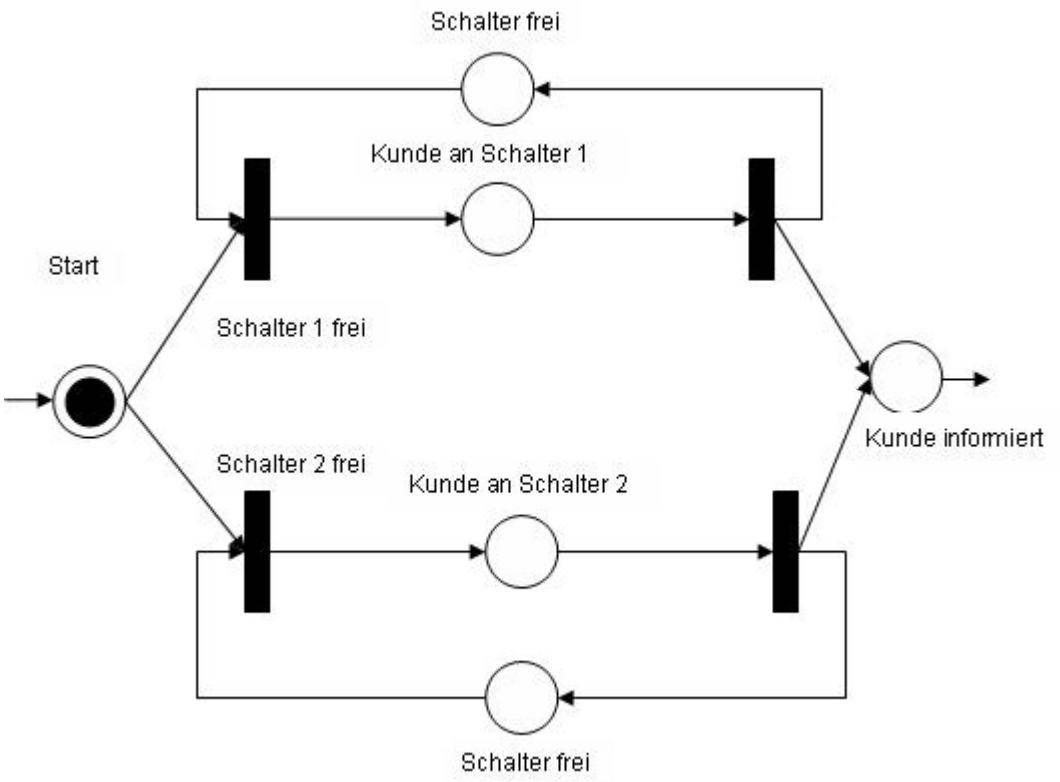


Abb. 2-5: Petri-Netz - Beispiel "Kunde / Schalter Zuordnung "

Stellen werden als Kreise dargestellt und die Marken als gefüllte, schwarze Kreise darin eingetragen. Die Belegung der Stellen heißt Markierung. Sie stellt den Zustand des Petrinetzes dar und ist als Momentaufnahme des Petrinetzes zu verstehen.
Die Kapazität wird als Zahl an der jeweiligen Stelle notiert, ebenso wie der Bezeichner, der lediglich zum besseren Verständnis des Graphen beitragen soll. Wird keine Kapazität für eine Stelle notiert, so ist seine Kapazität unendlich groß.

Die Transitionen werden als Rechtecke im Petrinetz dargestellt. Sie zeigen senkrecht zu einer Hauptrichtung. Im Netz können Transitionen auch mit einem Namen versehen werden.

Flussrelationen werden als Pfeil (gerichtete Kante) zwischen Stellen und Transitionen abgebildet. Das Kantengewicht ist ein Faktor der beim Schalten einer Transition auf den Markenfluss wirkt. Es wird auf der jeweiligen Kante notiert. Wird das Gewicht nicht explizit angegeben, wird 1 als Wert angenommen.

Stellen und Transitionen werden durch gerichtete Kanten verbunden. Zwei Stellen oder zwei Transitionen dürfen nicht durch eine gerichtete Kante miteinander verbunden werden.

Startpunkt bei der Traversierung eines Petrinetzes ist immer an der Startmarkierung m_0.

Des Weiteren sind S und T disjunkt ($S \cap T = \emptyset$). Den Stellen werden zwei natürliche Zahlen (Abbildungen K und m_0) und den Kanten wird eine natürliche Zahl (sog. Kantengewicht; Abbildung W) zugeordnet.

Transitionen sind aktiviert bzw. schaltbereit, wenn sich in allen Eingangsstellen mindestens so viele Marken befinden, wie die Transition an Kosten verursachen und alle Ausgangsstellen noch genug Kapazität haben, um die neuen Marken aufzunehmen. Aktivierte Transitionen können zu einem beliebigen Zeitpunkt schalten.
Schaltet eine Transition, werden entsprechend der Kantengewichte aus ihren Eingangsstellen Marken entnommen und bei den Ausgangsstellen ebenfalls entsprechend der Kantengewichte Marken hinzugefügt.
Wenn im Netz keine Transition mehr aktiviert ist, ist das Netz tot (vgl. Marczinzik [19]).

Petrinetze bieten analog zu Aktivitätsdiagrammen folgende Möglichkeiten:

- Verzweigung (fork) von Bearbeitungswegen
- Zusammenführung (join) von Bearbeitungswegen
- Synchronisation von Prozessen

Während eine Verzweigung stets mehr als einen Nachfolger anbietet, kann eine Transition bei der Zusammenführung von Bearbeitungswegen erst schalten, wenn alle zu ihr führenden Zweige beendet wurden. Diese Transition besitzt logischerweise nur einen Nachfolger.

Die Synchronisation ist streng genommen nicht mehr als eine Zusammenführung mit sofortiger Verzweigung der Bearbeitungswege. Bei einer Synchronisation wartet eine Transition darauf, dass alle zu synchronisierenden Prozesse bis zu einem festgelegten Punkt fortgeschritten sind und startet erst dann die Fortsetzung für alle beteiligten Zweige.

Die vorgestellten Konzepte und Möglichkeiten von Petrinetzen gleichen der Verzweigung/Zusammenführung bzw. Synchronisation von Aktivitätsdiagrammen. Diese Ähnlichkeit liegt darin begründet, dass bei der Überarbeitung der Aktivitätsdiagramme in UML2, diverse Aspekte von Petrinetzen mit eingeflossen sind.

2.6 Entwurf eines geeigneten Diagramms für CTC-Graphen

Vor dem Entwurf einer neuen Diagrammart, müssen nachfolgend aufgeführte Fragestellungen genauer analysiert werden. Die Antworten auf die gestellten Fragen helfen die optimale Lösung für diese bevorstehende Aufgabe, der Modellierung eines koordinierten transaktionalen Kommunikations-Graphen und dessen anschließende Übersetzung in BPEL, zu finden.

Erste und wichtigste Frage für die bevorstehende Modellierung von transaktionalen Kommunikations-Graphen (Graphen, die die transaktionale Kommunikation zwischen Teilnehmern eines koordinierten Web Services Prozesses darstellen) lautet:

Können die koordinierten transaktionalen Kommunikations-Graphen mit Hilfe der analysierten Diagrammarten beschrieben werden?

Wird diese Frage eindeutig mit „Nein“ beantwortet, stellen sich folgende weiterführende Fragen:

- Welche einzelnen Bestandteile der analysierten Diagrammarten, könnten für den Entwurf der CTC-Graphen nützlich sein?
- Was sollte bei der Konzeption der koordinierten transaktionalen Kommunikations-Graphen für Web Services (CTC-Graphen) berücksichtigt werden? (vgl. Kapitel 2.6.2)
- Wie lassen sich die koordinierten transaktionalen Kommunikations-Graphen formal beschreiben und wie könnten sie letztendlich aussehen? (vgl. Kapitel 2.6.3)

Zur ersten Frage:

Wie bereits in der Einleitung von Kapitel 2 ausgeführt, eignen sich die oben dargestellten Diagramme nicht um CTC-Graphen zu beschreiben. Die analysierten Diagrammarten können nicht alle nötigen Anforderungen für eine geeignete Modellierung von koordinierten transaktionalen Kommunikationen im Web Service Umfeld erfüllen. Die erste Frage wäre somit eindeutig mit „Nein“ zu beantworten.

Zu den weiteren Fragen:

Weil die analysierten Diagrammarten als CTC-Graphen ungeeignet sind, muss nun geklärt werden, wie und ob Elemente und Konzepte der untersuchten Graphen in die neu zu entwerfenden Graphen einfließen können. Es wird nach Parallelen und Gemeinsamkeiten gesucht, die die Modellierung und den Entwurf der CTC-Graphen und deren Formale Beschreibung vereinfachen und unterstützen können.

Mit Hilfe der Abbildungen 1-2 und 1-3 soll die Formale Beschreibung für CTC-Graphen schrittweise ausgearbeitet werden. Als erster Schritt muss die Frage geklärt werden, welche allgemeine Grundform die Graphen besitzen sollen. Kann diese Frage eindeutig beantwortet werden, sind die genaueren Bestandteile der CTC-Graphen (Knoten- und Kantenarten) anhand von allgemeinen Strukturanalysen der Pattern und deren korrekte BPEL-Übersetzung zu definieren. Diese Vorgehensweise mündet in die Formale Beschreibung der CTC-Graphen und ihren Einschränkungen.

Die Analyse der Ausgangsgraphen unter Einbeziehung verschiedener Modellierungs- und Implementierungsaspekte hat gezeigt, dass zwei unterschiedliche Lösungsalternativen für den Entwurf eines CTC-Graphen in Betracht kommen.

Alternative 1

Graph besitzt:

- einen Knotentyp (den allgemeinen Systemzustand)
- zwei unterschiedliche Kantentypen (spezielle Nachrichtentypen jeweils für coordinator und participant)

Alternative 2

Graph besitzt:

- verschiedene Knotentypen (spezielle Zustände für coordinator und participant)
- einen Kantentyp (die ausgetauschten Nachrichten)

Die folgende sorgfältige Untersuchung beider Möglichkeiten zeigt, dass Alternative 1 die adäquateste Grundlage für die Modellierung der CTC-Graphen ist.

Untersuchung der Alternative 1:

Einerseits ist die optische Ähnlichkeit mit den Ausgangsgraphen sichtbar und dient so dem besseren Verständnis der Weiterverfolgung und dem Ausbau des Konzeptes eines Graphen. Der Graph ist in der Lage, die in den Spezifikationen verdeutlichten Sachverhalte, widerzuspiegeln. Andererseits kann so die Übersicht der Graphen bewahrt werden, was zu einer einfacheren und leichter verständlichen Übersetzung in BPEL führt.
Diese leicht erlernbare Struktur der Graphen und das hieraus resultierende Verständnis ist ein wichtiger Blickwinkel bei der Entscheidung über die graphische Darstellung der CTC-Graphen. Die angestrebte Symmetrie bei der Übersetzung von CTC-Graphen in BPEL-Dateien ist gleichfalls ein wichtiger Gesichtspunkt, der hier berücksichtigt werden kann.

Der Wunsch nach einer executable-BPEL Ausgabedatei wurde bereits angesprochen und mittels kompakteren Knoten – alias „Systemzuständen“ – ist die exakte Übersetzung von Knoten in „`scope`“ BPEL Konstrukte möglich.

Die unterschiedlichen Kantentypen sollten auch optisch voneinander zu unterscheiden sein. So ist die Zuordnung der Nachrichten zu ihren Sendern klar erkennbar.
Im Hinblick auf die generierten BPEL-Dateien, können diese einen ähnlichen Aufbau und eine gewisse Symmetrie, zu den CTC-Graphen aufweisen. Die Abbildung von Systemzuständen in scopes fördert sowohl die Erzeugung wie auch das Verständnis der vom Plug-in automatisch generierten Dateien.

Im folgenden Graphen wird vereinfacht das Konzept der Systemzustände (als einziger Knotentyp) und der unterschiedlichen Nachrichtentypen (als zwei verschiedene Kantentypen) veranschaulicht.

Die Idee wird am folgenden Beispiel verdeutlicht. Der Einfachheit halber werden die nachfolgenden Punkte zusammengefasst dargestellt:

- die berechnenden Zustände als ein einziger „Compute“-Zustand
- unterschiedliche Berechnungsnachrichten als Compute“- bzw. „Result“-Nachrichten
- die unterschiedlichen Entscheidungsnachrichten (Commit / Abort) als „Decision“-Nachricht
- Fehler- oder Sonderfälle werden nicht explizit berücksichtigt

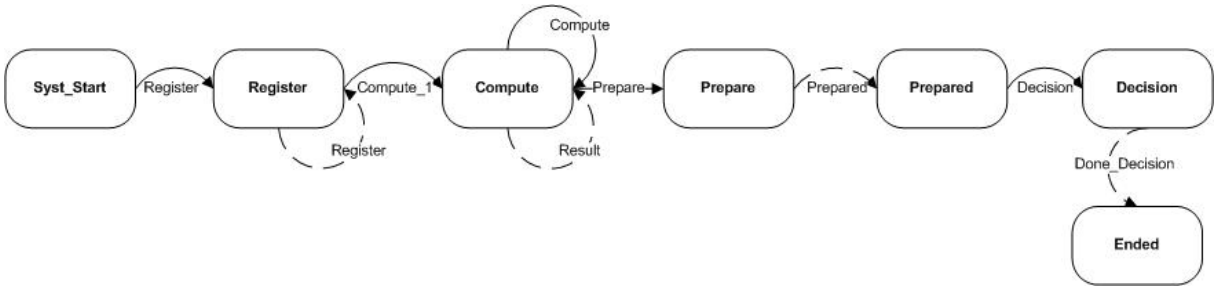


Abb. 2-6: Beispiel-Graph für Alternative 1 mit einem Systemzustand und mehreren Nachrichtenarten [7]

Die vom Koordinator versandten Nachrichten sind unter Zuhilfenahme von durchgezogenen gerichteten Kanten dargestellt. Nachrichten, die von den Teilnehmern versandt werden, sind durch gestrichelte gerichtete Kanten abgebildet.

Untersuchung der Alternative 2:

Die zweite Alternative (mehrere Knotentypen und ein Kantentyp) erscheint auf den ersten Blick als durchführbar. Bei näherer Betrachtung wird jedoch schnell deutlich, dass diese Alternative eine ungeeignete Entwurfsgrundlage darstellt.

Die vielen Knotentypen, die selbstverständlich aus Gründen der Übersichtlichkeit in einer anderen Weise abgebildet werden, stellen mangels der fehlenden Übersichtlichkeit eine große Fehlerquelle dar.
Im Hinblick auf die zu erzeugenden BPEL-Dateien wird dieser Typ keine große Ähnlichkeit oder Symmetrie mit dem Ausgangsgraphen aufweisen. Dies erschwert das Verständnis und damit das spätere Handling der Dateien.

Im folgenden Beispiel-Graphen wurden die weiter oben verwendeten Vereinfachungen und Zusammenfassungen ebenfalls genutzt (siehe Erläuterungen zu Abb. 2-6: Beispiel-Graph für Alternative 1 mit einem Systemzustand
und mehreren Nachrichtenarten).

[7] Dieser Beispielgraph, unterstützt graphisch die oben genannten Aspekte. Manche der hier verwendeten Konstrukte (z.B. loops) werden – wg. Beschränkungen in BPEL - evtl. nicht Teil der CTC-Graphen. Abbildung 2-6 soll lediglich das Verständnis für die Überlegungen und Erläuterungen wecken und nicht das Maß für das noch zu entwickelnde Ergebnis darstellen.

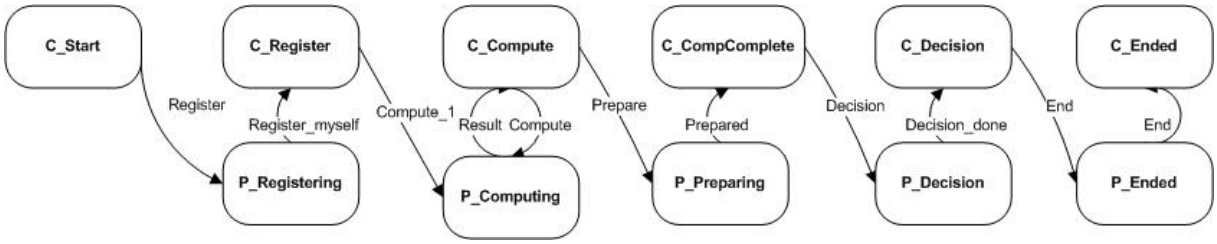


Abb. 2-7: Beispiel-Graph für Alternative 2 mit mehreren Zustandskategorien und einer Nachrichtenart

Nachdem die Frage der groben Ausgangsform von CTC-Graphen geklärt wurde, sollen die nachfolgenden Überlegungen und Untersuchungen eine Verfeinerung des Modells herbeiführen.

Zunächst wurde hier der Frage der Analogien zwischen den weiter oben analysierten Graphen-Modellen nachgegangen. Die einzelnen Modelle wurden hierfür in Bezug auf die Möglichkeiten der Konzept-Übernahme für die zu entwerfenden koordinierten transaktionalen Kommunikations-Graphen untersucht. Dieser Analyse der Analogien folgt die Erläuterung der Grundzüge der Übersetzung von CTC-Graphen in BPEL-Code. Dabei werden die am häufigsten verwendeten Konstrukte der Graphentheorie, wie in einem sequentiellen, bedingungsgeknüpften oder schleifen-ähnlichen Zustandsübergang, analysiert und deren Übersetzung in BPEL veranschaulicht.

Alle Analysen münden schließlich in die Konzeption der Formalen Beschreibung von CTC-Graphen. Die Formale Beschreibung dient als Grundlage für den Entwurf des Klassendiagramms für CTC-Graphen. Das Klassendiagramm für CTC-Graphen stellt das Fundament für das CTC-Graphen Editor Plug-in dar.

2.6.1.1 Zustandsdiagramme als Ideenlieferant für CTC-Graphen

Die Grundlagen der Zustandsdiagramme sind unkompliziert strukturiert und gut verständlich. Allerdings eignen sie sich nur bedingt für die Modellierung der CTC-Graphen.

Wie im Unterkapitel 1.1.1.1 „Web Services Coordination“ bereits beschrieben, ist das WS-Coor ein Umfeld in dem verschiedene Partner mittels Nachrichten miteinander kommunizieren und dadurch Veränderungen des Systemzustandes verursachen.

Wie Zustandsdiagramme, haben auch CTC-Graphen eine Zustandsüberführungsfunktion. Sie wurde bereits in der Formalen Beschreibung der Zustandsdiagramme (vgl. Kapitel 2.1.1) definiert. So ist es möglich alle eingegebenen Wörter in Hinsicht auf ihre Akzeptanz, nämlich ob vom Startzustand aus durch die Eingabe der Endzustand erreicht werden kann, zu prüfen. Die Systemzustände von CTC-Graphen könnten durchaus durch Modellierungskonzepte der Zustände von Zustandsdiagrammen beschrieben werden.

Bedeutendster und gewichtigster Grund, weshalb Zustandsdiagramme nicht ohne weiteres für die Modellierung von CTC-Graphen herangezogen werden können ist, dass CTC-Graphen mit Nachrichten – insbesondere mit verschiedenen Arten von Nachrichten – arbeiten. Weitere Informationen zu den Nachrichten folgen in Kapitel 2.6.3.2. Auch können Zustandsdiagramme die Eingabe, die einen Zustandsübergang anstößt, nicht in verschiedene Klassen/Kategorien untergliedern. Die nötige Unterscheidung zwischen Coordinator-Nachrichten und Participant-Nachrichten ist in Zustandsdiagrammen somit nicht gegeben.

In Hinblick auf die Generierung des BPEL-Codes sind die, in einem Zustandsdiagramm darstellbaren, Informationen zu unpräzise und deshalb nicht ausreichend für den gewünschten Umfang der automatisierten Übersetzung.

Generell gilt, dass für eine vollständige Beschreibung eines BPEL-Prozesses die eigentliche Programm-Logik, die bspw. Berechnungen durchführt, benötigt wird. Diese Aspekte müssen in die generierten BPEL-Dateien vom Anwender selbst eingefügt werden.Die generierten BPEL-Dateien sind das Rückrad der modellierten Transaktion und gewährleisten das Zusammenspiel der koordinierten transaktionalen Kommunikation.

Trotzdem werden die Zustände, wie sie in Zustandsdiagrammen definiert und verwendet werden, für die Modellierung des Systemzustands der CTC-Graphen benötigt und auch übernommen.

2.6.1.2 Aktivitätsdiagramme als Ideenlieferant für CTC-Graphen

Aktivitätsdiagramme bieten ebenfalls nicht die benötigten Konstrukte für die Beschreibung von CTC-Graphen, obwohl es einige Parallelen und Ähnlichkeiten zwischen den beiden Graphen-Typen gibt.

Als erstes fällt auf, dass Aktivitätsdiagramme mit nur wenigen Zuständen arbeiten, die – wie später unter Formale Beschreibung der CTC-Graphen verdeutlicht – dringend für die Modellierung von Systemzuständen benötigt werden.

Wie bereits unter Kapitel 2.2 beschrieben, werden die einzelnen Aktivitäten des Diagramms Schritt für Schritt durchlaufen. Sobald eine Aktion abgeschlossen ist, folgt die nächste Aktion. Dies macht die Modellierung, trotz der Möglichkeit mit boolschen Bedingungen zu arbeiten, etwas aufwändiger.
Die Nachrichten, die im CTC-Graphen ausgetauscht werden, ähneln Aktivitäten. Diese Aktivitäten sind aber nicht das Hauptaugenmerk dieser neuen Graphen-Art.

Abschließend lässt sich festhalten, dass die Systemzustände der zu entwerfenden CTC-Graphen zwar ein ähnliches Verhalten wie die Aktivitäten eines Aktivitätsdiagramms aufweisen, dennoch werden bei CTC-Graphen andere Strukturen benötigt und eingesetzt.

2.6.1.3 Sequenzdiagramme als Ideenlieferant für CTC-Graphen

Sequenzdiagramme eignen sich ebenfalls nur bedingt für die Modellierung von CTC-Graphen. Die Übersichtlichkeit des gesamten Graphen würde durch die Modellierung von Systemzuständen unter Zuhilfenahme von Lebenslinien stark eingeschränkt.

Bemerkenswert bei Sequenzdiagrammen, sind die gesendeten Nachrichten bzw. das Verhalten, das von ihnen initiiert wird. Wie in der Formalen Beschreibung gesehen (vgl. Kapitel 2.3Sequenzdiagramme), werden die Zustandsübergänge jeweils von Nachrichten ausgelöst. Bei Sequenzdiagrammen besteht zwar die Möglichkeit Nachrichten in Bezug auf ihr Sende-/Antwortverhalten zu differenzieren (Synchrone-, Asynchrone- und Antwortnachrichten), die Möglichkeit unterschiedliche Nachrichtenformen darzustellen ist jedoch nicht gegeben.

Das Modellierungskonzept von synchronen und asynchronen Nachrichten ist das wichtigste Konzept, das aus den Sequenzdiagrammen in die Modellierung der CTC-Graphen einfließen könnte.
Es wäre bei Protokollgraphen möglich, beide verschiedene Nachrichten-Typen (Coordinator- und Participant-Nachrichten) auch als synchrone oder asynchrone Nachrichten zu spezifizieren (asynchrones BPEL-Konstrukt „`invoke`“/ „`receive`“; synchrones BPEL-Konstrukt „`invoke`“ / „`reply`“). Die Notation mit der geschlossenen Pfeilspitze bzw. offenen Pfeilspitze könnte z.B. ebenfalls in die Notation der CTC-Graphen übernommen werden.

Ob sich dieses Feature (synchrone / asynchrone Nachrichten) im endgültigen Konzept der CTC-Graphen durchsetzen wird, bleibt an dieser Stelle noch offen. Festzuhalten ist jedoch, dass ein solches Feature für die CTC-Graphen nützlich sein könnte. Werden die gesamten Daten, die für die Ausführung von BPEL-Codes benötigt werden betrachtet, so kann bereits in den WSDL-Files eine Unterscheidung zwischen synchronen und asynchronen Nachrichten entdeckt werden. Das Attribut „`input`“ / „`output`“ bei der Definition der Operationen von portTypes, welches auf eine Synchronität hindeuten, kann dort vorgefunden werden.

2.6.1.4 Kommunikationsdiagramme als Ideenlieferant für CTC-Graphen

Weitere Aspekte, als die bei der Ausführung über Sequenzdiagramme als Ideenlieferant für CTC-Graphen (vgl. Kapitel 2.6.1.3) genannten, zeigen sich bei Kommunikationsdiagrammen nicht. Kommunikationsdiagramme sind, wie in Kapitel 2.4 bereits erläutert, sehr eng mit den Sequenzdiagrammen verwandt.

Der wesentliche Unterschied zwischen Kommunikations- und Sequenzdiagrammen besteht im Modellierungs-Fokus (d.h. in der Perspektive). Die Perspektive liegt bei Sequenzdiagrammen insbesondere auf dem zeitlichen Verlauf der Nachrichten. Bei Kommunikationsdiagrammen stehen Objekte und

ihre Zusammenarbeit, also die Kommunikation oder auch die sog. Kollaboration, im Vordergrund.

Kommunikationsdiagramme sind übersichtlicher als Sequenzdiagramme. Dies könnte die Modellierung der CTC-Graphen ebenfalls übersichtlicher gestalten. Allerdings fehlen auch den Kommunikationsdiagrammen die benötigten unterschiedlichen Nachrichten-Typen.

Die bei Sequenzdiagrammen so hilfreiche Modellierung von synchronen und asynchronen Nachrichten fehlt hier gänzlich. Kommunikationsdiagramme werden somit wenig zum Entwurf der CTC-Graphen beisteuern können.

2.6.1.5 Petrinetze als Ideenlieferant für CTC-Graphen

Als Ideenlieferanten erscheinen Petrinetze oberflächlich betrachtet hilfreich, da sie mit zwei verschiedenen Objekttypen (Stellen und Transitionen) arbeiten. Mit Hilfe eines Petrinetzes könnten Systemzustände modelliert werden, dennoch besteht weiterhin die Problematik der zwei unterschiedlichen Nachrichten-Typen. Zum einen die Menge der Koordinator-Nachrichten und zum anderen die Menge der Teilnehmer-Nachrichten – mehr hierzu folgt in Kapitel 2.6.3 „Formale Beschreibung der CTC-Graphen“.

Die verschiedenen Objekttypen (Stellen und Transitionen) und deren unterschiedliche Darstellungsform, dienen hier als Ideenlieferant, wie die unterschiedlichen Nachrichten-Typen dargestellt bzw. eingesetzt werden könnten.

Abschließend lässt sich festhalten, dass Petrinetze keine zusätzlichen neuen Erkenntnisse für den Entwurf der CTC-Graphen liefern können. Dies mag auf den ersten Blick verwundern, wenn jedoch berücksichtigt wird, dass viele Ideen und Bestandteile von Petrinetzen bereits in die Überarbeitung der Aktivitätsdiagramme (von UML 1.x in UML2) geflossen sind, relativiert sich diese Aussage.

2.6.2 Modellierungsgrundlagen für Koordinierte Transaktionale Kommunikations-Graphen für Web Services

Wie bereits Eingangs in Kapitel 2 aufgeführt, wird nach der Analyse existierender graphentheoretischer Ansätze im Hinblick auf deren Eignung für die Modellierung der CTC-Graphen, eine Analyse der graphischen Struktur-Elemente und deren Übersetzung in BPEL durchgeführt. Alle diese Aspekte bilden die Grundlage für die Formale Beschreibung von koordinierten transaktionalen Kommunikations-Graphen.

Die nachfolgend untersuchten Struktur-Elemente werden nicht komplett in den CTC-Graphen einfließen. Die Analyse der Struktur-Elemente dient der letzten Eignungsprüfung für die Übersetzung in BPEL. Hierbei soll auch das Verständnis des generierten BPEL-Codes eine wichtige Rolle spielen.

Der Aufbau von koordinierten transaktionalen Kommunikations-Graphen soll der UML2 Notation ähneln. Diese Ähnlichkeit wird sich auch im Aufbau und im Aussehen der Strukturelemente widerspiegeln.

Zu Beginn soll mit Hilfe einer Abbildung das Aussehen und auch einige der zukünftigen graphischen Elemente vorgestellt werden.

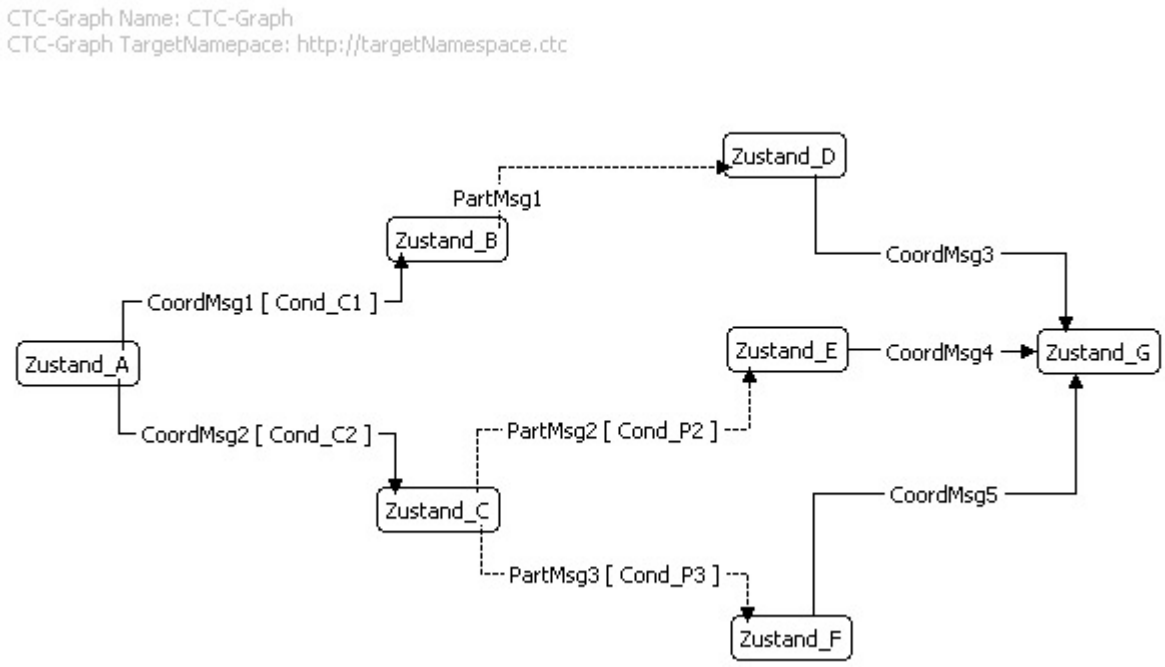


Abb. 2-8: koordinierte transaktionale Kommunikations-Graphen für Web Services- Modellierungselemente [8]

Abbildung 2-8 stellt ein allgemeines Beispiel der CTC-Graphen dar. Die Systemzustände werden durch Rechtecke mit abgerundeten Ecken dargestellt. Nachrichten, die vom Koordinator gesendet werden, sind als durchgezogene, gerichtete Kanten abgebildet. Stammen die Nachrichten vom Teilnehmer, werden sie durch gestrichelte, gerichtete Kanten dargestellt. Abhängig von der Art der gesendeten Koordinator- bzw. Participant-Nachricht, werden die Kanten durch zusätzliche optische Merkmale voneinander unterschieden (vgl. Kapitel 3.2.3).

Bei der Analyse der existierenden graphentheoretischen Modelle hat sich gezeigt, dass für den Entwurf von CTC-Graphen sowie der darauf folgenden BPEL-Übersetzung, eine Fülle an Informationen benötigt werden.
Nachfolgend soll das Übersetzungskonzept des Graphen in BPEL und dessen Folgen für den Entwurf der Graphen aufgezeigt werden.

Mit Hilfe eines einfachen Beispiels kann die gerade angesprochene Informationsvielfalt innerhalb der CTC-Graphen veranschaulicht werden. Wie bereits ausgeführt, muss schon bei der Modellierung von CTC-Graphen die Möglichkeit gegeben sein, die Nachrichtenarten so zu modellieren, dass sofort deutlich wird, ob eine Nachricht vom Koordinator (coordinator) oder vom

[8] Beispiel für einen koordinierten transaktionalen Kommunikations-Graphen modelliert mit Hilfe des CTC-Graph Editors (eigener Eclipse-Plug-in) – weitere Informationen folgen in Kapitel 3.

Teilnehmer (participant) gesendet wurde. Diese Unterscheidung wird nicht aus Schönheitsgründen gemacht, sondern vereinfacht die Generierung der jeweiligen BPEL-Codes. Es wird eine BPEL-Datei für den Koordinator und jeweils eine für die Teilnehmer generiert.

Die benötigte Information, in Bezug auf die BPEL-Übersetzung, wird wie folgt beschrieben.
Erforderliche Eigenschaften für die anschließende Übersetzung in BPEL:

- Möglichst einfache und intuitiv verständliche Abbildung der CTC-Zustände in BPEL-Code.
- Effektive und effiziente Attributierung von graphischen Elementen für eine möglichst vollständige und korrekte Übersetzung in BPEL (z.B. portTypes, partnerLinks, etc. für die verschiedenen gesendeten Nachrichten).
- Möglichkeit einer, wenn auch nur indirekten, Darstellung von „`flow`“ oder „`sequence`“ Konstrukten im BPEL Prozess in Bezug auf Senden und Empfangen von Nachrichten.
- Miteinbeziehen von möglichen BPEL-Nachrichten Versendearten („`invoke`“ / „`reply`“ / „`receive`“) in der Modellierung von CTC-Graphen.

Insgesamt sollte der generierte BPEL-Code gut leserlich und leicht verständlich sein, damit seine anschließende Ergänzung mit der eigentlichen Programm-Logik zügig voranschreiten kann.

Die zu ergänzende Prozess-Logik kann von der aktuellen Version des Plug-ins nicht automatisch erstellt werden. Sie findet lediglich mit der Übersetzung des CTC-Graphen in BPEL statt. Die fehlende Logik muss, wie bereits in Kapitel 2.6.1.1 verdeutlicht, vom Programmierer selbst eingefügt werden. Dies könnte als letzter Schritt in der Programmierung des vollständigen und funktionsfähigen (executable) BPEL-File betrachtet werden.
Die Unterstützung und die automatische Generierung der Programm-Logik ist nicht Gegenstand der vorliegenden Aufgabenstellung. Hierzu ist eine Ergänzung mit Hilfe von anderen Werkzeugen erforderlich.
Auch soll im Rahmen dieser Diplomarbeit kein WSDL-Editor entworfen oder in das zu entwerfende Plug-in integriert werden. Die benötigten WSDL-Files für den Koordinator- oder Teilnehmer-BPEL-Code müssen vom Benutzer mit Hilfe anderer Programme erzeugt und korrekt eingesetzt werden.

Die Diplomarbeit bietet zahlreiche Erweiterungs- und Verfeinerungsmöglichkeiten. Näheres hierzu folgt in Kapitel 4.

2.6.2.1 Graphische Konstrukte und der dazugehörige BPEL-Code

Durch das Zeichnen verschiedener Skizzen der möglichen Ausprägungen der CTC-Graphen wurde erkannt, dass sich diverse graphische Konstrukte (sog. Patterns) wiederholen.

Bei der Unterteilung dieser Skizzen, die mit Hilfe von CTC-Graphen modellierbar sind, kristallisieren sich folgende Konstrukte heraus:

1. Sequentielles Senden von Nachrichten
2. Bedingungsgeknüpftes Senden von Nachrichten
3. Paralleles Senden von Nachrichten (Broadcast-Nachricht) und Antworten auf die Broadcast-Nachrichten
4. Einfache-Schleife (Problemfall)
5. Schleife (Problemfall)

Die erwähnten Konstrukte, nachfolgend auch Pattern genannt, stellen die kleinsten Kommunikations-Elemente dar mit denen alle CTC-Graphen modelliert werden können. Diese Patterns dürfen nicht als atomare Strukturen der CTC-Graphen betrachtet werden, das sind weiterhin die Knoten und Kanten. Mittels der Patterns soll in diesem Abschnitt die Übersetzung der CTC-Graphen in BPEL-Code analysiert und verdeutlicht werden.
Bei dieser Analyse werden sich auch die speziellen Ausprägungen der Kanten (bedingungsgeknüpfte Kanten, Kanten für paralleles Senden von Nachrichten, etc.) herauskristallisieren, die später als atomare Strukturen für die Modellierung der CTC-Graphen im Eclipse-Plug-in zur Verfügung stehen sollen.

Zustände werden aus Gründen der Übersichtlichkeit als „scope"-Aktivitäten in BPEL übersetzt. Die spätere Bearbeitung des Codes soll hierdurch vereinfacht werden. Die Zustände werden so auf eine sinnvolle und effektive Weise in BPEL abgebildet. Die Suche nach einer bestimmten Stelle im BPEL-Code für Zwecke der Überarbeitung oder das Einfügen von Programmlogik soll hierdurch erleichtert werden.
Die vorgestellten Code-Artefakte (im Folgenden auch BPEL-Snippets gennant) gelten stellvertretend für die Übersetzung der graphischen Konstrukte in BPEL. Die verwendeten Rollen des Coordinators oder Participants, wie auch der Nachrichtenaustausch, sind eine mögliche Ausprägung des Systems.

Anmerkung: Für die Analyse der späteren generierten BPEL-Dateien wurde mit ActiveWebflow™ Professional Designer Version 1.5.0 [9] gearbeitet. Mit Hilfe dieser ActiveWebflow-Dateien konnte die Struktur wohlgeformter BPEL-Dateien auf die vom entwickelten Plug-in generierten BPEL-Code übertragen werden.

Konstrukt 1: Sequentielles Senden von Nachrichten

Das sequentielle Senden von Nachrichten gilt als ein weit verbreitetes Kommunikationsmuster. Es ist eines der unkompliziertesten graphischen Konstrukte.
Der einfache Nachrichtenaustausch zwischen Kommunikationspartnern wird hier dargestellt, wobei jede ausgetauschte Nachricht eine Zustandsänderung des Systems bewirkt.

[9] ActiveWebflow ein BPEL-Generierungs-Tool von ActiveEndpoints (http://www.active-endpoints.com/ - zuletzt verfolgt am 23. May 2006).

Der Code zu dem graphischen Konstrukt folgt jeweils nach der Abbildung.

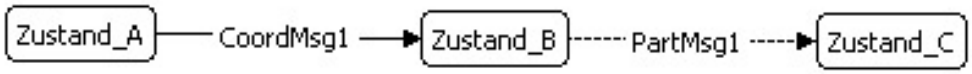


Abb. 2-9: Sequentielles Senden von Nachrichten in CTC-Graphen [10]

Beispielcode (BPEL-Snippet) für den Aktions-Abschnitt der Koordinator BPEL-Datei:

```
<sequence>
   <scope name="Zustand_A">
      <sequence>
         <empty>
            <!-- add computations here -->
         </empty>
         <invoke name="CoordMsg1" partnerLink="pL_Name"
portType="pT_Name" operation="op_Name"
inputVariable="inputVar">
            <!-- faults, correlations etc. add here -->
         </invoke>
      </sequence>
   </scope>
   <scope name="Zustand_B">
      <sequence>
         <empty>
            <!-- add computations here -->
         </empty>
      </sequence>
   </scope>
   <scope name="Zustand_C">
      <sequence>
         <receive name="PartMsg1" partnerLink="pL_Name"
portType="pT_Name" operation="op_Name"
variable="inputVar">
            <!-- faults, correlations etc. add here -->
         </receive>
         <empty>
            <!-- add computations here -->
         </empty>
      </sequence>
   </scope>
</sequence>
```

[10]Beispiel für die Sequenz in einem koordinierten transaktionalen Kommunikations-Graphen modelliert mit Hilfe des CTC-Graph Editors (eigener Eclipse-Plug-in) – weitere Informationen folgen in Kapitel 3.

Beispielcode (BPEL-Snippet) für den Aktions-Abschnitt der Participant BPEL-Datei:

```
<sequence>
   <scope name="Zustand_A">
      <sequence>
         <empty>
            <!-- add computations here -->
         </empty>
      </sequence>
   </scope>
   <scope name="Zustand_B">
      <sequence>
         <receive name="CoordMsg1" partnerLink="pL_Name"
portType="pT_Name" operation="op_Name"
variable="inputVar">
            <!-- faults, correlations etc. add here -->
         </receive>
         <empty>
            <!-- add computations here -->
         </empty>
         <invoke name="PartMsg1" partnerLink="pL_Name"
portType="pT_Name" operation="op_Name"
inputVariable="inputVar">
            <!-- faults, correlations etc. add here -->
         </invoke>
      </sequence>
   </scope>
   <scope name="Zustand_C">
      <sequence>
         <empty>
            <!-- add computations here -->
         </empty>
      </sequence>
   </scope>
</sequence>
```

Konstrukt 2: Bedingungsgeknüpftes Senden von Nachrichten

Auch dieses Konstrukt ist in der Graphentheorie weit verbreitet. Das Senden einer Nachricht wird an bestimmte Bedingungen geknüpft. Die damit verursachte Änderung des Systemzustandes ist somit an die Bedingung des Sendens geknüpft.

Eine solche Bedingung könnte bspw. eine Abfrage auf bestimmte Werte sein: Ist der Wert einer Variablen > X, dann soll Nachricht „CoordMsg1“ versandt werden und damit Zustand_B als Folgezustand erreicht werden, andernfalls soll Nachricht „CoordMsg2“ versandt werden und Zustand_C als Folgezustand erreicht werden.

Eine Möglichkeit das bedingungsgeknüpfte Senden von Nachrichten in BPEL zu übersetzen, wäre der Gebrauch des BPEL-Konstruktes „`switch`“. Allerdings kann dasselbe Verhalten auch mit Hilfe eines „`flow`“ Konstruktes in Verbindung mit „`links`“ (auch Link-Conditions genannt) erreicht werden.
Mit Hilfe der Link-Conditions in einem „`flow`“ wird immer nur ein Bearbeitungszweig aktiviert, somit kann auf das aufwändige „`switch`“ Konstrukt verzichtet werden.

Der dazugehörige Code zu diesem graphischen Konstrukt folgt nach der Abbildung 2-10.

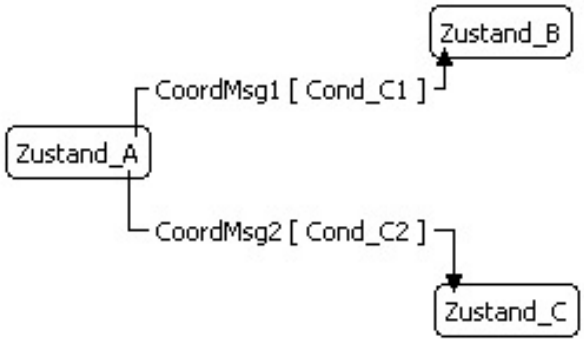


Abb. 2-10: Bedingungsgeknüpftes Senden von Nachrichten in CTC-Graphen [11]

Beispielcode (BPEL-Snippet) für den Aktions-Abschnitt der Koordinator BPEL-Datei:

```
<sequence>
  <scope name="Zustand_A">
     <sequence>
        <empty>
           <!--  add computations here -->
        </empty>
        <flow name="CoordMsg1_flow">
           <link name="CoordMsg1"/>
           <link name="CoordMsg2"/>
           <empty name="Add your instructions here to
activate link conditions">
              <source linkName="CoordMsg1"
transitionCondition="Cond_C1"/>
              <source linkName="CoordMsg2"
transitionCondition="Cond_C2"/>
           </empty>
```

[11] Beispiel für das bedingungsgeknüpfte Senden von Nachrichten in einem koordinierten transaktionalen Kommunikations-Graphen modelliert mit Hilfe des CTC-Graph Editors (eigener Eclipse-Plug-in) – weitere Informationen folgen in Kapitel 3.

```
            <invoke name="CoordMsg1"
partnerLinkType="pL_Name" portType="pT_Name"
operation="op_Name" inputVariable="inputVar1">
                <target linkName="CoordMsg1"/>
                <!-- faults, correlations etc. add here -->
                </invoke>
                <invoke name="CoordMsg1"
partnerLinkType="pL_Name" portType="pT_Name"
operation="op_Name" inputVariable="inputVar1">
                <target linkName="CoordMsg2"/>
                <!-- faults, correlations etc. add here -->
            </invoke>
         </flow>
      </sequence>
   </scope>
   <scope name="Zustand_B">
      <sequence>
         <empty>
            <!--  add computations here -->
         </empty>
      </sequence>
   </scope>
   <scope name="Zustand_C">
      <sequence>
         <empty>
            <!--  add computations here -->
         </empty>
      </sequence>
   </scope>
</sequence>
```

Beispielcode (BPEL-Snippet) für den Aktions-Abschnitt der Participant BPEL-Datei:

```
<sequence>
   <scope name="Zustand_A">
      <sequence>
         <empty>
            <!--  add computations here -->
         </empty>
      </sequence>
   </scope>
   <scope name="Zustand_B">
      <sequence>
         <receive name="CoordMsg1" partnerLink="pL_Name"
portType="pT_Name" operation="op_Name"
variable="inputVar1">
            <!-- faults, correlations etc. add here -->
         </receive>
         <empty>
```

```
            <!--  add computations here -->
         </empty>
      </sequence>
   </scope>
   <scope name="Zustand_C">
      <sequence>
         <receive name="CoordMsg2" partnerLink="pL_Name"
portType="pT_Name" operation="op_Name"
variable="inputVar2">
            <!-- faults, correlations etc. add here -->
         </receive>
         <empty>
            <!--  add computations here -->
         </empty>
      </sequence>
   </scope>
</sequence>
```

Konstrukt 3: Paralleles Senden von Nachrichten (Broadcast-Nachricht) und Antworten auf die Broadcast-Nachrichten

Weil bei CTC-Graphen mit Systemzuständen gearbeitet wird, ist die Möglichkeit einer parallelen Ausführung von Zuständen nicht gegeben. Ein System kann sich zu einem bestimmten Zeitpunkt nur in einem wohl definierten Zustand befinden.

Allerdings ist die Möglichkeit der Parallelität auch hier gegeben, weil mit Parallelität nicht nur die Ausführung von Zuständen gemeint ist sondern auch das parallele Senden von Nachrichten (sog. Broadcast-Nachrichten).
Diese Parallelität für das Senden von Nachrichten wird allerdings nur an den Stellen der CTC-Graphen benötigt, wenn der Koordinator eine bestimmte Nachricht gleichzeitig an alle Teilnehmer einer Transaktion senden möchte. Zum Beispiel beim Senden von „`commit`“ oder „`abort`“ Nachrichten werden diese Broadcast-Nachrichten benötigt. Mit dieser Art von Nachrichten wird üblicherweise das 2PC vom Koordinator eingeleitet oder dessen Ergebnis bekannt gegeben.

Die Parallelität beim Empfangen von Antwort-Nachrichten auf einen bestimmten Broadcast wird ebenfalls benötigt. Somit kann der Koordinator erst alle Antworten sammeln, um abschließend festzustellen, ob die Änderungen der Transaktion festgeschrieben werden können oder nicht.

Als Lösungsansatz für das parallele Senden von Nachrichten wäre denkbar, dass vor Beginn der eigentlichen Transaktion, also bereits während der Registrierungsphase aller Teilnehmer, der Koordinator eine vollständige Liste der Teilnehmer erstellt. Diese Teilnehmerliste müsste in einer globalen Variablen gespeichert werden, bspw. in einem Array (im Folgenden Teilnehmer-Array genannt). So könnte innerhalb verschiedener „`scope`“-Aktivitäten darauf zugegriffen werden. Dieser Ansatz ist jedoch nicht durchführbar, weil BPEL von Haus aus keine Array-Funktionen vorsieht. Mit Hilfe von XSD (gem. XSD Schema

Definition [33]), könnte allerdings ein solches Teilnehmer-Array beschrieben werden.

Um gewährleisten zu können, dass die Teilnehmer-Arrays immer das gleiche Schema haben, wird an dieser Stelle das WSDL-Konstrukt „`types`“ aufgeführt, dass die entsprechende Array-Typ-Beschreibung für die verwendete Koordinator WSDL-Datei verdeutlicht (vgl. Array-Beschreibung [20]).

```
<types>
   <xsd:schema
    targetNamespace="http://new.webservice.namespace"
    xmlns:typens="http://new.webservice.namespace">
      <xsd:complexType name="TRANSPART">[12]
         <xsd:sequence>
            <xsd:element name="partnerLink"
             type="xsd:string"/>
            <xsd:element name="portType"
             type="xsd:string"/>
            <xsd:element name="operation"
             type="xsd:string"/>
         </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType
           name="ArrayOfTRANSPART">
         <xsd:complexContent>
            <xsd:restriction base="soapenc:Array">
               <xsd:attribute ref="soapenc:arrayType"[13]
                    wsdl:arrayType="thisns:TRANSPART[]"/>
            </xsd:restriction>
         </xsd:complexContent>
      </xsd:complexType>
   </xsd:schema>
</types>
```

Die Problematik dieses Ansatzes stellt sich wie folgt dar:

- Findet keine korrekte Befüllung des Teilnehmer-Arrays statt, können die weiterführenden Berechnungen nicht stattfinden.
- Die oben verwendeten XSD Simple Types zur Speicherung von `partnerLink`, `portType` und `operation` sind evtl. nicht kompatibel mit den von BPEL erwarteten Einträge (bspw. bei einer „`invoke`“-Aktion). Gravierende Fehler bei der Ausführung des generierten BPEL-Codes könnten hierdurch verursacht werden.

Um diese Fehlerquellen und die darin enthaltenen Unsicherheiten zu vermeiden, wurde eine zweite Lösung für das parallele Senden von Nachrichten entwickelt.

[12] Verwendeter Namespace: `xmlns:xsd="http://www.w3.org/2001/XMLSchema"`
[13] Verwendeter Namespace: `xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"`

Generell gilt, dass jeder Teilnehmer eine eigene BPEL-Datei für die Ausführung einer bestimmten Transaktion benötigt. In ihr werden die individuellen Bearbeitungsschritte aufgeführt. So ist es möglich, dass der Benutzer des CTC-Graph Editors indirekt eingeben kann, wie viele Teilnehmer er für eine bestimmte Transaktion erwartet (vgl. Kapitel 3.3.3). Dabei wird die Anzahl der erwartenden Teilnehmer erfasst und die korrekte Anzahl an Teilnehmer-BPEL-Files generiert.

Der dazugehörige Code (BPEL-Snippet) zu dem graphischen Konstrukt folgt nach der Abbildung.

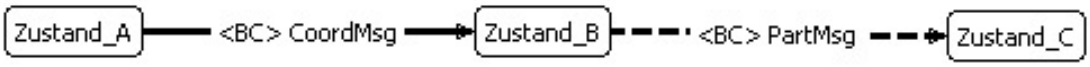


Abb. 2-11: Paralleles Versenden von Nachrichten in CTC-Graphen (Broadcast-Nachrichten an alle Participants)[14] und die dazugehörige Antwort-Nachricht der Participants

Beispielcode (BPEL-Snippet) für den Aktions-Abschnitt der Koordinator BPEL-Datei für das Senden einer Broadcast Nachricht (Anzahl Teilnehmer = n):

```
<!-- declare variable that holds the number of
participants (numberOfParticipants) and to count the
number of times the message was sent (sentCounter) -->
<assign name="Assign_While_Variables">
   <copy>
      <from expression="n"/>
      <to variable="numberOfParticipants"/>
   </copy>
   <copy>
      <from expression="1"/>
      <to variable="sentCounter"/>
   </copy>
</assign>
<while condition="bpws:getVariableData('sentCounter')
&lt;= bpws:getVariableData('numberOfParticipants')">
   <sequence>
      <flow>
         <links>
            <link linkName="link_1"/>
            <link linkName="link_2"/>
              ...
            <link linkName="link_n"/>
         </links>
         <empty>
```

[14] Beispiel für das Senden und Antworten auf Broadcast-Nachrichten in einem koordinierten transaktionalen Kommunikations-Graphen modelliert mit Hilfe des CTC-Graph Editors (eigener Eclipse-Plug-in) – weitere Informationen folgen in Kapitel 3.

```
            <source name="link_1"
transitionCondition="bpws:getVariableData('sentCounter')
= 1"/>
            <source name="link_2"
transitionCondition="bpws:getVariableData('sentCounter')
= 2"/>
              ...
            <source name="link_n"
transitionCondition="bpws:getVariableData('sentCounter')
= n"/>
         </empty>
         <invoke name="Msg_1" partnerLink="pL_Name_1"
portType="pT_Name" operation="op_Name" inputVariable="
inVar_1">
            <target linkName="link_1"/>
            <!-- faults, correlations etc. add here -->
         </invoke>
         <invoke name="Msg_2" partnerLink="pL_Name_2"
portType="pT_Name" operation="op_Name" inputVariable="
inVar_2">
            <target linkName="link_2"/>
            <!-- faults, correlations etc. add here -->
         </invoke>
          ...
         <invoke name="Msg_n" partnerLink="pL_Name_n"
portType="pT_Name" operation="op_Name" inputVariable="
inVar_n">
            <target linkName="link_n"/>
            <!-- faults, correlations etc. add here -->
         </invoke>
      </flow>
      <assign>
         <copy>
            <from
query="bpws:getVariableData('sentCounter') = + 1"
variable="sentCounter"/>
            <to variable="sentCounter"/>
         </copy>
      </assign>
   </sequence>
</while>
```

Beispielcode (BPEL-Snippet) für den Aktions-Abschnitt einer BPEL-Datei für das Empfangen von Teilnehmer-Antworten auf eine durch den Koordinator geschickten Broadcast Nachricht (Anzahl Teilnehmer = n):

```
<!-- declare variable that holds the number of
participants (numberOfParticipants) and to count the
number of times the message was sent (sentCounter) -->
<assign name="Assign_While_Variables">
```

```
   <copy>
      <from expression="n"/>
      <to variable="numberOfParticipants"/>
   </copy>
   <copy>
      <from expression="1"/>
      <to variable="sentCounter"/>
   </copy>
</assign>
<while condition="bpws:getVariableData('sentCounter')
&lt;= bpws:getVariableData('numberOfParticipants')">
   <sequence>
      <flow>
         <links>
            <!-- one link condition for each part. -->
            <link linkName="link_1"/>
            <link linkName="link_2"/>
              ...
            <link linkName="link_n"/>
         </links>
         <empty>
            <source name="link_1"
transitionCondition="bpws:getVariableData('sentCounter')
= 1"/>
            <source name="link_2"
transitionCondition="bpws:getVariableData('sentCounter')
= 2"/>
              ...
            <source name="link_n"
transitionCondition="bpws:getVariableData('sentCounter')
= n"/>
         </empty>
         <invoke name="Msg_1" partnerLink="pL_Name"
portType="pT_Name" operation="op_Name"
inputVariable="invar_1">
            <target linkName="link_1"/>
            <!-- faults, correlations etc. add here -->
         </invoke>
         <invoke name="Msg_2" partnerLink="pL_Name"
portType="pT_Name" operation="" inputVariable="inVar _2">
            <target linkName="link_2"/>
            <!-- faults, correlations etc. add here -->
         </invoke>
          ...
         <invoke name="Msg_n" partnerLink="pL_Name"
portType="pT_Name" operation="" inputVariable="inVar _n">
            <target linkName="link_n"/>
            <!-- faults, correlations etc. add here -->
         </invoke>
      </flow>
      <assign>
```

```
         <copy>
            <from
query="bpws:getVariableData('sentCounter') = + 1"
variable="sentCounter"/>
            <to variable="sentCounter"/>
         </copy>
      </assign>
   </sequence>
</while>
```

Wie bereits ausgeführt, besteht die Übersetzung des parallelen Versendens von Nachrichten in BPEL nur zu einem Teil aus den oben dargestellten BPEL-Codes. Das Versenden der Nachrichten an sämtliche Transaktions-Teilnehmer setzt voraus, dass die Anzahl der erwarteten Teilnehmer einer Transaktion bereits vom Benutzer des CTC-Graph Editors genannt werden kann.

An dieser Stelle soll auch kurz die Migration zu BPEL 2.0 kurz angesprochen werden. Die neuesten Working Drafts der BPEL-Spezifikation 2.0 [2], enthalten das neue BPEL-Konstrukt: „`forEach`“. Dieses Konstrukt könnte das parallele Versenden von Nachrichten in CTC-Graphen an alle Teilnehmer einer Transaktion deutlich vereinfachen.
Im Rahmen dieser Arbeit wurde allerdings bewusst auf dieses neue BPEL-Konstrukt verzichtet. Dieser Verzicht basiert auf den gängigen Berechnungsgrundlagen von Technologiezyklen aus Forschung und Industrie [17] und auf die – zum jetzigen Zeitpunkt noch nicht eingeführte – Spezifikation von BPEL 2.0.
Der BPEL-Code, der mit Hilfe des Eclipse-Plug-ins erzeugt wird, soll ohne große Änderungen (ausgenommen Programmlogik) auf möglichst vielen Maschinen ausführbar sein. Nur wenige Hersteller unterstützen bereits die neueste BPEL-Version (BPEL 2.0). Bei [2] handelt sich um einen Working Draft und nicht um die fertige BPEL 2.0 Spezifikation. Da die meisten Web Services Umgebungen noch mehrere Jahre BPEL 1.1 kompatibel sein werden, schränkt diese Entscheidung die Anwendung des CTC-Graphen und des CTC-Graph Editors nicht ein.

Konstrukt 4: Einfache Schleife

Die Schleife, die nur einen Zustand umfasst, ermöglicht es innerhalb eines Systemzustandes Nachrichten (ob Koordinator- oder Teilnehmer-Nachrichten) zu verschicken, ohne damit einen Zustandsübergang hervorzurufen. Dieses Konstrukt kann hilfreich sein, wenn noch Daten einer Berechnung ausgetauscht werden sollen, das System aber im gleichen Zustand verweilt.

Hierbei kann es sich sowohl um eine einzelne Nachricht wie auch um eine beliebige, jedoch bereits definierte, Anzahl von Nachrichten handeln.
Diese hier beschriebene „Ein-Zustand-Schleife“ kann mit Hilfe von Bedingungen modelliert werden. Diese Bedingungen werden verwendet, um die Anzahl der Iterationen zu steuern.

Die nachfolgende Abbildung zeigt eine einfache Schleife ohne Bedingungen. Wird die Schleife öfters verwendet, ist zusätzlich die Angabe einer Abbruchbedingung notwendig. Diese Bedingung kann auch optisch an den Pfeil angeheftet werden.

Eine Abbruchbedingung könnte hier bspw. eine Abfrage auf bestimmte Werte sein. Ist der Wert einer Variablen > X, dann soll Nachricht „Msg1“ versandt werden, damit wird Zustand_A nicht verlassen. Ist diese Bedingung nicht mehr erfüllt wird eine weitere Nachricht „Msg2“ (in der folgenden Abb. nicht eingezeichnet) versandt und Zustand_A wird verlassen.

Der dazugehörige Code (BPEL-Snippet) zu dem graphischen Konstrukt folgt nach der Abbildung.

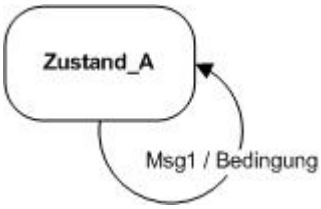


Abb. 2-12: CTC-Graphen : einfache Schleife

Beispielcode (BPEL-Snippet) für den Aktions-Abschnitt der Koordinator BPEL-Datei:

```
<scope name="Zustand_A">
  <variables>
     <variable name="firstEntry" type="xsd:boolean"/>
     <variable name="Msg1Sent" type="xsd:boolean"/>
  </variables>
  <sequence>
     <assign name="firstEntryToDo">
        <copy>
           <from expression="true()"/>
           <to variable="firstEntry"/>
        </copy>
     </assign>
     <while name="loopActivitiesZustand_A"
          condition="bpws:getVariableData('firstEntry')
              or bpws:getVariableData('Msg1Sent')" >**
        <sequence>
           <assign name="firstEntryDone">
              <copy>
                 <from expression="false()"/>
                 <to variable="firstEntry"/>
              </copy>
           </assign>
           <empty name="placeComputationsHere"/>
           <flow>
              <links>
                 <link name="Msg1ToBeSent"/>
              </links>
```

```
                <invoke name="Msg1"
                        partnerLink=”pL_Name”
                        portType=”pT_Name”
                        operation=”op_Name”>
                   <target linkName="Msg1ToBeSent"/>
                   <!—- faults, correlations, etc. -->
                </invoke>
                <assign name="stayInLoop">
                   <source linkName="Msg1ToBeSent"/>
                   <copy>
                      <from expression="true()"/>
                      <to variable="Msg1Sent"/>
                   </copy>
                </assign>
             </flow>
          </sequence>
       </while>
    </sequence>
</scope>
```

** Die while-Schleife wird nur verwendet, wenn die Nachricht mehr als einmal versendet werden soll.

Beispielcode (BPEL-Snippet) für den Aktions-Abschnitt der Participant BPEL-Datei:

```
<scope name="Zustand_A">
     <variables>
        <!-- message to stay in loop arrived -->
        <variable name="Message2Loop"
                  type="xsd:boolean"/>
        <variable name="firstEntry"
                  type="xsd:boolean"/>
     </variables>
     <eventHandlers> **
        <onMessage partnerLink="pL_Name"
                   portType="pT_Name"
                   operation="op_Name">
           <!-- message to stay looping arrived -->
           <assign name="goOnLooping">
              <copy>
                 <from expression="true()"/>
                 <to variable="Message2Loop"/>
              </copy>
           </assign>
        </onMessage>
     </eventHandlers>
     <sequence>
        <!-- in first entry message causing loop to go on
hasn't arrived -->
        <assign name="loopYN">
```

```
            <copy>
               <from expression="true()"/>
               <to variable="Message2Loop"/>
            </copy>
         </assign>
         <assign name="firstEntryToDo">
            <copy>
               <from expression="true()"/>
               <to variable="firstEntry"/>
            </copy>
         </assign>
         <while name="loopActivitiesZustand_A"
           condition="checkeVariablen">**
            <sequence>
               <assign name="firstEntryDone">
                  <copy>
                     <from expression="false()"/>
                     <to variable="firstEntry"/>
                  </copy>
               </assign>
               <switch>
                  <!-- message to loop arrived - then
                       receive it -->
                  <case
      condition="bpws:getVariableData('Message2Loop')">
                     <receive name="receiveCoordMsg1"
                              partnerLink="pL_Name"
                              portType="pT_Name"
                              operation="op_Name">
                       <!-- faults, correlations etc. -->
                     </receive>
                  </case>
               </switch>
               <empty name="placeComputationsHere"/>
            </sequence>
         </while>
      </sequence>
   </scope>
</sequence>
```

** Die while-Schleife wird nur verwendet, wenn die Nachricht mehr als einmal versendet werden soll.

Konstrukt 5: Schleife

Eine Schleife, die mehr als einen Zustand umfasst, bietet die Möglichkeit, innerhalb eines Systems verschiedene Zustände mehrfach zu besuchen.

Diese Schleife wird in der Graphentheorie gerne und oft verwendet, sie stellt jedoch gerade im Hinblick auf die BPEL-Übersetzung ein Problemfall dar.

Die Ursache dieser Problematik ist, dass der BPEL-Code vom Koordinator und den Teilnehmern einer Transaktion sequentiell abgearbeitet wird. Ein Sprungelement – das gerade bei älteren Assembler-Programmiersprachen zu finden ist – wie „goto“ wurde allerdings nicht in der BPEL-Spezifikation [1] definiert.

Die Arbeiten von J. Koehler et al. [18] und R.F. Hauser [16] beschäftigen sich mit dieser Thematik, indem sie Grundzüge aus dem Informatik-Bereich Compilerbau (sog. „T1–T2 Analyse“) auf diese Problematik erfolgreich anwenden.

Mit Hilfe der im Paper [16] von Hauser entwickelten B- und C-Regeln, sind Schleifen durchaus in BPEL zu übersetzen.

An dieser Stelle wird die Übersetzung in BPEL komplexer, weil unter Zuhilfenahme von „while“-Schleifen die Graphen in BPEL-Code umgewandelt werden. Die Zustände („scope“-Aktivitäten) müssen geschachtelt werden, damit die Schleifen korrekt implementiert werden können.

Wie anhand des BPEL-Snippets zu erkennen ist (siehe BPEL-Snippet nach Abb. 2-13: CTC-Graphen : Loop (strukturierte Schleife)“), soll für die Darstellung der Schleifen-Konstrukte eine ähnliche Übersetzungsmethode (gem. Hauser[16]) verwendet werden, wie die hier bereits durchgeführte BPEL-Übersetzung. Die Überlegungen und Regeln passen so dementsprechend gut zu den bereits eingeführten Übersetzungen der CTC-Graphen.

Die Paper „Transforming Unstructured Cycles in Sequential Flow Graphs“ [16] und „Declarative Techniques For Model-Driven Business Process Integration“ [18] können an dieser Stelle die Grundzüge der Loop-Übersetzung in BPEL mittels des Aktions-Abschnittes der Koordinator BPEL-Datei unterstützen.

Der dazugehörige Code (BPEL-Snippet) für das graphische Konstrukt folgt nach der Abbildung:

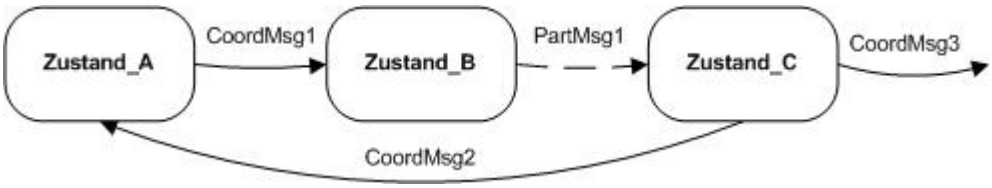


Abb. 2-13: CTC-Graphen : Loop (strukturierte Schleife)

Beispielcode (BPEL-Snippet) für den Aktions-Abschnitt der Koordinator BPEL-Datei:

```
<sequence>
   <scope name="Zustand_A">
      <variables>
         <variable name="firstLoopEntry"
                   type="xsd:boolean"/>
         <variable name="stayInLoop"
```

```
                type="xsd:boolean"/>
      </variables>
      <!-- start loop between “Zustand_A” and
       “Zustand_C” -->
      <sequence>
         <!-- firstEntry=true() ; after getting into
          while new assignment firstEntry=false() -->
         <assign name="firstLoopEntryToBeDone">
            <copy>
               <from expression="true()"/>
               <to variable="firstLoopEntry"/>
            </copy>
         </assign>
         <while name="loopAC"
                condition="KeepLoopGoing” >
            <sequence>
               <assign name="firsLooptEntryDone">
                  <copy>
                     <from expression="false()"/>
                     <to variable="firstLoopEntry"/>
                  </copy>
               </assign>
               <empty name="placeComputationsHere"/>
               <invoke name="CoordMsg1"
                       partnerLink="pL_Name"
                       portType=”pT_Name”
                       operation=”op_Name”>
                   <!—- faults, correlations, etc. -->
               </invoke>
               <scope name="Zustand_B">
                  <empty
                     name="placeComputationsHere"/>
               </scope>
               <scope name="Zustand_C”>
                  <sequence>
                     <receive name="PartMsg1"
                              partnerLink="pL_Name"
                              portType=”pT_Name”
                              operation=”op_Name”
                              variable=”PartMsg1”>
                     </receive>
                     <flow>
                        <links>
                           <link name="stayL"/>
                           <link name="leaveL"/>
                        </links>
                        <assign name="stay" >
                           <source linkName="stayL"
                           transitionCondition="C1"/>
                           <copy>
                             <from
```

```
                                    expression="true()"/>
                                  <to
                                    variable="stayInLoop"/>
                               </copy>
                            </assign>
                            <assign name="leave" >
                               <source linkName="leaveL"
                               transitionCondition="C2"/>
                               <copy>
                                 <from
                                  expression="false()"/>
                                 <to
                                  variable="stayInLoop"/>
                               </copy>
                            </assign>
                            <invoke name="CoordMsg2"
                                   partnerLink="pL_name"
                                   portType="pT_Name"
                                   operation="op_Name">
                               <target linkName="stayL"/>
                            </invoke>
                            <empty
                           name="placeComputationsHere"/>
                            <invoke name="CoordMsg3"
                                   partnerLink="pL_Name"
                                   portType="pT_Name"
                                   operation="op_Name">
                               <target
                                 linkName="leaveL"/>
                            </invoke>
                         </flow>
                      </sequence>
                   </scope>
                </sequence>
             </while>
          </sequence>
       </scope>
    </sequence>
```

Der BPEL-Code zeigt die bereits beschriebenen Probleme bei der Übersetzung von Schleifen in BPEL. Die Datei ist schwer verständlich und bei größeren Schleifen, wäre allein die Verschachtelungstiefe der „scope“–Aktivitäten und der darin enthaltenen Aktionen nicht mehr überschaubar. Die Korrektheit der Übersetzung von CTC-Graphen in BPEL wäre somit nicht mehr gewährleistet. Diese Gründe und Überlegungen fließen in die Formale Beschreibung der CTC-Graphen mit ein (vgl. Kapitel 2.6.3.1).

Erzeugter Code und seine Zusammensetzung:

In BPEL-Snippets werden Daten verwendet, die Verknüpfungen zum eigenen BPEL-File wie auch zum WSDL-File aufweisen. Diese Angaben werden vom Anwender gemacht und vom Programm verwendet, um das BPEL-File so vollständig wie möglich zu generieren.

In welcher Form die benötigten Daten angegeben werden, wird sich bei der Konzeptions- und Implementierungsphase mit Eclipse, EMF und GEF im Detail zeigen. Möglich wäre im Plug-in eine Eingabe ausgewählter Attribute, die an graphische Elemente gekoppelt werden (z.B. Zustände oder Nachrichten-Kanten).
Als weitere Möglichkeit ist auch eine „indirekte Eingabe“ denkbar. Dabei können Daten zuerst einem graphischen Element zugeordnet und anschließend ausgelesen werden. Die Daten müssten dem Programm in festgelegten Formaten übergeben werden z.B. WSDL-Code Fragmente. Die ausgelesenen Informationen werden in diesem Fall bei der anschließenden Übersetzung der CTC-Graphen in BPEL verwendet, um den BPEL-Code zu generieren und zu vervollständigen.

Daten wie Partner-Links, die für das Senden oder Empfangen einer Nachricht benötigt werden, sind am Anfang einer BPEL-Datei anzugeben. In ihrer Definition wird der im WSDL-File verankerte „`partnerLinkType`“ benötigt und muss vom Anwender geliefert werden (Verknüpfung zwischen „`partnerLink`“ und „`partnerLinkType`“).

Des Weiteren muss der Bezug zwischen Variablen und ihren im WSDL-File verankerten Typen vom Anwender korrekt hergestellt werden (bei der Eingabe von „`messageType`“).

2.6.3 Formale Beschreibung der CTC-Graphen

Durch den Vergleich und die Analyse der Diagramme der WS-AT und der WS-BA Spezifikationen [6] [8], werden die Gemeinsamkeiten deutlich aufgezeigt, die so zur Formalen Beschreibung der CTC-Graphen führen.

Unter Zuhilfenahme der Formalen Beschreibung können in einem zweiten Schritt auch die oben beschriebenen Diagrammarten und deren Bestandteile als Modellierungshilfe herangezogen werden.

Formal lassen sich die Diagramme in WS-AT und WS-BA Spezifikationen beschreiben durch:

- Eine endliche, nicht leere Menge von System-Zuständen Q – sog. Knoten
- Eine endliche, nicht leere Menge von Coordinator-Nachrichten M_c – sog. Kanten
- Eine endliche, nicht leere Menge von Participant-Nachrichten M_P – sog. Kanten
- Eine eindeutige Zustandüberführungsfunktion δ: $Q \times M \rightarrow Q$
- Einen eindeutigen Startzustand q_0 ($q_0 \in Q$)

- Einen oder mehrere Endzustände Q_e ($Q_e \subseteq Q$)

Mit einem eindeutigen Startzustand q_0 ($q_0 \in Q$) ist ein Knoten, der keine eingehenden Kanten besitzt, gemeint.

Des Weiteren sind M_c und M_P disjunkt ($M_c \cap M_P = \emptyset$). Die Gesamtheit der verschickten bzw. empfangenen Nachrichten, also die Kantenmenge wird als M bezeichnet (wobei $M = M_c \cup M_P$).

Der CTC-Graph wird wie folgt definiert:

- (N, E) wobei
- $E \subseteq N \times N$, N = Menge aller Knoten und
- E = M, d.h. die Menge der Kanten repräsentieren.

Die Zusatzbeschränkung, dass zwischen den Zuständen die Kantentypen immer alternierend sein müssen, erscheint auf den ersten Blick logisch, sie kann jedoch bei detaillierter Analyse nicht aufrechterhalten werden. Betrachtet man die Ausgangsdiagramme noch einmal intensiver, erkennt man in Abb. 1-2: WS-AT 2PC Diagramm mit Nachrichten (Quelle WS-AT Spezifikation [6]), dass z.B. zwei aufeinander folgende Koordinator-Nachrichten verschickt werden können, die jeweils einen Zustandsübergang mit sich bringen. Wird im Zustand *„Active“* gestartet und mittels der Koordinator-Nachricht „Prepare“ in den *„Preparing“*-Zustand gewechselt, kann ein Zustandsübergang in den *„Aborting“*-Zustand mit der Koordinator-Nachricht „Rollback“ erfolgen.

Der Startpunkt bei der Traversierung der CTC-Graphen ist immer der Anfangszustand q_0. Mit Hilfe von Coordinator- bzw. Participant-Nachrichten können die anderen Systemzustände, entsprechend der Zustandsüberführungsfunktion $\delta: Q \times M \rightarrow Q$, erreicht werden.

Die hier ausgearbeitete Formale Beschreibung ist der kleinste gemeinsame Nenner, unter dem die Graphen weiter ausgebaut bzw. mit weiteren Eigenschaften und Attributen versehen werden können.

Gerade im Hinblick auf die Aufgabenstellung, der Modellierung einer Graphen-Art und der anschließende Übersetzung in BPEL, werden weiterhin Verfeinerungen in Bezug auf die Kanten-Arten für CTC-Graphen und damit auch einhergehende Modelleinschränkungen durchgeführt.

2.6.3.1 Allgemeine Einschränkungen der Kanten in CTC-Graphen

Nach der genauen Betrachtung der möglicherweise in Frage kommenden Strukturen der Kommunikations-Pattern und der Grundzüge der BPEL-Übersetzung von CTC-Graphen werden Einschränkungen bei den zu modellierenden Kanten gemacht. Sie werden im Folgenden einer genaueren Analyse unterzogen:

1. CTC-Graphen sind azyklisch

Obwohl die theoretischen Grundzüge für die Übersetzung einer Schleife in BPEL bereits bekannt (vgl. Hauser [16] und Koehler et al [18]) und durchaus nachvollziehbar sind, ist die Umsetzung der Theorie (B- und C-Regeln) insbesondere in Bezug auf die Verwendung in einem koordinierten, transaktionalen Web Services Umfeld nur schwer strukturierbar und unüberschaubar.
Bei Übersetzung eines einfachen Loops in Coordinator- bzw. Participant-BPEL-Code wächst der Umfang des Codes rapide an. So geht die angestrebte leicht lesbare und verständliche Struktur des generierten BPEL-Codes verloren.

2. CTC-Graphen sind deterministisch

Jede Kante in einem CTC-Graphen ist eindeutig durch ihren Quell- und Zielknoten bestimmt, folglich dürfen keine Kanten denselben Quell- und Zielknoten besitzen. Wie im folgenden Unterkapitel weiter ausgeführt, können nur mehrere Kanten aus einem Knoten führen, wenn sie an eine Bedingung geknüpft sind. Anhand der erfüllten Bedingung kann, mit Hilfe der Zustandsübergangsfunktion, ein Zielknoten ermittelt werden – somit ist der CTC-Graph deterministisch.

Im CTC-Graph folgt unter den gleichen Voraussetzungen stets die gleiche Anweisung. Zu jeder Zeit ist der nachfolgende Abarbeitungsschritt (hier der nachfolgende Systemzustand) eindeutig festgelegt.

3. CTC-Graphen unterstützen keine synchrone Kommunikation

Die von BPEL unterstützte synchrone Kommunikation („`reply`“) wurde bei der Implementierung des Plug-ins angestrebt. Für die korrekte Modellierung und anschließenden Verwendung der synchronen Kommunikation, würden Verifikationsmechanismen benötigt, die den Rahmen des Programmier-Aufwands der Diplomarbeit im Bereich „GEF“ übersteigen. Um die Aufgabe der „Generierung von BPEL-Code aus einem CTC-Graphen“ nicht zu gefährden, wurde auf die synchrone Kommunikation verzichtet.

Diese Einschränkung wirkt sich in keinster Weise nachteilig auf die Mächtigkeit der CTC-Graphen aus. Mittels BPEL-Konstrukte „`correlationSets`“, die vom Anwender im generierten BPEL-Code ergänzt werden können, kann das gleiche synchrone Verhalten abgebildet werden.

2.6.3.2 Verfeinerung in Bezug auf die Kanten-Arten der CTC-Graphen

Nach der Analyse der möglicherweise auftretenden Strukturen während des Nachrichten-Austauschs von CTC-Graphen, können die Kanten-Arten (Coordinator-Nachrichten- und Participant-Nachrichten-Menge) in verschiedene Nachrichten-Typen unterteilt werden. Jeder der nachfolgend beschriebenen

Typen besitzt bestimmte Restriktionen, die die Formale Beschreibung einschränken, aber auch ihre Gültigkeit gewährleisten.

Die Nachrichten, die der Koordinator versendet, die sog. Coordinator-Nachrichten, bestehen aus den folgenden 4 Nachrichten-Typen:

1. Einfache Coordinator-Nachricht

Die einfache Coordinator-Nachricht stellt einen einfachen Zustandsübergang dar. Ohne weitere Bedingungen wird die Nachricht an einen Empfänger verschickt. So erfolgt der Übergang von einem Zustand in den nächsten Zustand. Der Zustandsübergang kann nur dann korrekt erfolgen, wenn keine weiteren Möglichkeiten gegeben sind diesen Zustand zu verlassen. Verlässt eine einfache Coordinator-Nachricht einen Zustand, so kann keine weitere Nachricht den Zustand verlassen.

2. Bedingungsgeknüpfte Coordinator-Nachricht

Die bedingungsgeknüpfte Coordinator-Nachricht stellt einen Zustandsübergang dar, der an die Erfüllung dieser bestimmten Bedingung geknüpft ist. Abhängig von der Erfüllung dieser Bedingung, wird eine festgelegte Nachricht verschickt. So erfolgt der Übergang von einem Zustand in den nächsten Zustand. Besitzt eine ausgehende Kante keine Bedingung oder verlassen zwei Kanten denselben Zustand mit derselben Bedingung, kann die Determiniertheit der Zustandsüberführungsfunktion nicht gewährleistet werden. Verlässt eine bedingungsgeknüpfte Coordinator-Nachricht einen Knoten, so können nur weitere Kanten dieses Typs denselben Knoten verlassen.

3. Einfache Coordinator-Broadcast-Nachricht

Die einfache Coordinator-Broadcast-Nachricht stellt einen einfachen Zustandsübergang dar. Ohne weitere Bedingungen wird diese Nachricht an mehrere Empfänger (z.B. alle Teilnehmer einer Transaktion) verschickt. So erfolgt auch der Übergang von einem Zustand in den nächsten Zustand. Der Zustandsübergang erfolgt nur dann korrekt, wenn keine anderen Möglichkeiten gegeben sind diesen Zustand zu verlassen. Verlässt eine einfache Coordinator-Broadcast-Nachricht einen Knoten, so kann keine weitere Kante denselben Knoten verlassen.

4. Bedingungsgeknüpfte Coordinator-Broadcast-Nachricht

Die bedingungsgeknüpfte Coordinator-Broadcast-Nachricht stellt einen Zustandsübergang dar, der an die Erfüllung einer bestimmten Bedingung geknüpft ist. In Abhängigkeit von der Erfüllung der angegebenen Bedingung, wird eine bestimmte Nachricht an mehrere Empfänger verschickt (z.B. an alle Teilnehmer einer Transaktion) dann erfolgt der Übergang von einem Zustand in den nächsten Zustand. Gemäß der Formalen Beschreibung für CTC-Graphen

dürfen keine zwei Bedingungen gleich sein, die vom selben Zustand abgehen oder wenn eine der Kanten keine Bedingung besitzt. Damit ist die Eindeutigkeit der Zustandsüberführungsfunktion gewährleistet. Verlässt eine bedingungsgeknüpfte Coordinator-Broadcast-Nachricht einen Knoten, so können nur weitere Kanten dieses Typs denselben Knoten verlassen.

Analog zu den o.g. Coordinator-Nachrichten lassen sich Nachrichten, die von Teilnehmern versendet werden (sog. Participant-Nachrichten) ebenfalls in 4 Nachrichten-Typen unterteilen:

1. Einfache Participant-Nachricht

Die einfache Participant-Nachricht stellt einen einfachen Zustandsübergang dar. Ohne weitere Bedingungen wird die Nachricht an einen Empfänger verschickt, so erfolgt auch der Übergang von einem Zustand in den nächsten Zustand. Der Zustandsübergang kann nur dann korrekt erfolgen, wenn keine weiteren Möglichkeiten gegeben sind, ihn zu verlassen. Verlässt eine einfache Participant-Nachricht einen Knoten, so kann keine weitere Kante denselben Knoten verlassen.

2. Bedingungsgeknüpfte Participant-Nachricht

Die bedingungsgeknüpfte Participant-Nachricht stellt einen Zustandsübergang dar, der an die Erfüllung einer bestimmten Bedingung geknüpft ist. In Abhängigkeit der Erfüllung der angegebenen Bedingung wird eine bestimmte Nachricht verschickt. So kann der Übergang von einem Zustand in den nächsten Zustand erfolgen. Gemäß der Formalen Beschreibung für CTC-Graphen dürfen weder zwei Bedingungen, die denselben Zustand verlassen, gleich sein noch darf eine der Kanten keine Bedingung besitzen. Letzteres gewährleistet die Eindeutigkeit der Zustandsüberführungsfunktion. Verlässt eine bedingungsgeknüpfte Participant-Nachricht einen Knoten, so können nur weitere Kanten dieses Typs denselben Knoten verlassen.

3. Einfache Participant-Broadcast-Antwortnachricht

Die einfache Participant-Broadcast-Antwortnachricht stellt einen einfachen Zustandsübergang dar. Sie wird als Antwort auf eine vorangegangene Coordinator-Broadcast-Nachricht ohne weitere Bedingungen an den Coordinator verschickt. So erfolgt der Übergang von einem Zustand in den nächsten Zustand. Der Zustandsübergang kann nur dann korrekt erfolgen, wenn keine weitere Möglichkeit gegeben ist, diesen Zustand zu verlassen. Verlässt eine einfache Participant-Broadcast-Antwortnachricht einen Knoten, so kann keine weitere Kante diesen Knoten verlassen.

4. Bedingungsgeknüpfte Participant-Broadcast- Antwortnachricht

Die bedingungsgeknüpfte Participant-Broadcast-Antwortnachricht stellt einen Zustandsübergang dar, der an die Erfüllung einer bestimmten Bedingung geknüpft ist. Abhängig von der Erfüllung der angegebenen Bedingung wird die Nachricht des Koordinators als Antwort auf eine vorangegangene Coordinator-Broadcast-Nachricht verschickt. Der Übergang von einem Zustand in den nächsten Zustand kann dann erfolgen Gemäß der Formalen Beschreibung für CTC-Graphen dürfen weder zwei Bedingungen, die demselben Zustand verlassen gleich sein noch darf eine der Kanten keine Bedingung besitzen. Letzteres gewährleistet die Eindeutigkeit der Zustandsüberführungsfunktion. Verlässt eine bedingungsgeknüpfte Participant-Broadcast-Antwortnachricht einen Zustand, so können nur weitere Nachrichten dieses Typs diesen Zustand verlassen.

Alle soeben beschriebenen Kanten-Typen und ihre Einschränkungen stellen sicher, dass zu jedem Zeitpunkt nur ein möglicher Zustandsübergang aktiv ist. Damit ist die Eindeutigkeit der Zustandüberführungsfunktion gewährleistet.

2.7 Übersetzungsstrategie von CTC-Graphen in BPEL

An dieser Stelle wird die Übersetzungsstrategie von CTC-Graphen in BPEL, wie sie auch in Kapitel 2.6.2 „Modellierungsgrundlagen für Koordinierte Transaktionale Kommunikations-Graphen für Web Services“ verwendet wurde, zusammenfassend dargestellt.

- Die System-Zustände der CTC-Graphen werden in das BPEL-Konstrukt „`scope`“ übersetzt. Jeder Zustand besitzt einen Namen, der dem „`scope`“-Attribut „`name`“ zugewiesen wird. Zusätzlich zu dem eben genannten Attribut, bieten System-Zustände auch die Möglichkeit der Gedächtnisstütze in Form von Kommentaren in den „`scope`“-Gültigkeitsbereich generierter BPEL-Dateien einzufügen.

- Das Senden einer Nachricht stellt den Zustandsübergang dar, deshalb gilt für die BPEL-Übersetzung des CTC-Graphen folgende strikte Aktionsreihenfolge:

    1. Die BPEL-Operation „`invoke`“, die das Senden einer Nachricht veranlasst, muss die letzte Aktivität in einem „`scope`“ sein, weil danach einen Zustandswechsel erfolgt. Weitere Aktionen und Berechnungen dürfen danach nicht mehr in die BPEL-Datei eingefügt werden.
    2. Die BPEL-Operation „`receive`“, die den Empfang einer Nachricht bewirkt, muss zwingend die erste Aktion in einem „`scope`“ sein. Die Nachricht, die den Zustandsübergang ausgelöst hat, wird vom Adressaten empfangen und der Inhalt der Nachricht kann zur

Weiterbearbeitung freigegeben werden. Vor dem „receive“ sind keine Aktionen oder Berechnungen in die BPEL-Datei einzufügen.

3. CTC-Graphen unterstützen keine synchrone Kommunikation (vgl. Kapitel 2.6.3.1), deshalb wird die BPEL-Operation „reply“ bei der Übersetzung nicht verwendet.

- Zwischen den BPEL-Operationen „receive“ und „invoke“ werden im generierten Code leere Aktionen, sog. „empty“-Aktivitäten, eingefügt. Sie vervollständigen die executable BPEL-Datei und dienen als Platzhalter für Berechnungen, die der Anwender dem Prozess hinzufügen möchte.

In „Appendix A – Beispiele CTC-Graphen“ sind zwei ausführliche Beispiel-Graphen und die dazugehörigen übersetzten BPEL-Dateien aufgeführt. Die in diesem Kapitel zusammengefassten Modellierungsansätze und Übersetzungsstrategien können anhand dieser Beispiele im praktischen Einsatz analysiert werden.

3 Eclipse Implementierung

Die Implementierung des Plug-ins erfolgt in Einzelschritten, die im Folgenden nachvollzogen werden. Mit Hilfe von Beispiel-Code oder Screenshots werden Ideen, Konzepte und Vorgehensweisen eingeführt und im Detail erläutert.

Folgende Versionen wurden bei der Programmierung verwendet:

- Eclipse SDK Version: 3.1.2
- EclipseUML Free Edition Version: 2.1.0 (OMONDO)
- Graphical Editing Framework Version: 3.1.1
- Eclipse Modelling Framework Version: 2.1.1
- EMF Edit Version: 2.1.1
- EMF Template Code Generator Version: 2.1.2

In Anlehnung an die Vorgehensweisen bei „Using GEF with EMF“ [11] und „Erstellung eines graphischen Editor-Plug-ins mit EMF und GEF“ [14] läuft die Implementierung nach dem folgenden systematischen Handlungsschema ab:

1. **EMF-Modell**: Entwurf eines EMF-Modells für CTC-Graphen und Generierung der dazugehörigen Java-Klassen. Hierfür verwendete Hilfsmittel: “OMONDO-Plug-in” [24], “EMF-Dokumentation” [12] und “Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework” [21].

2. **CTC-Graph Editor**: Konzeption und Implementierung des graphischen Editors für CTC-Graphen. Verwendete Hilfsmittel: „GEF-Dokumentation“ [13], „Maßgeschneiderte graphische Editoren mit dem Graphical Editing Framework“ [5] und „Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework“ [21].

3. **BPEL-Generator**: Konzeption und Implementierung des BPEL-Übersetzers und dessen Integration in den CTC-Graph Editor. Verwendete Hilfsmittel: „BPEL-Spezifikation“ [1] und BPEL-Code. Der BPEL-Code wurde mit dem ActiveWebflow™ Professional Designer Version 1.5.0 zur Analyse der korrekten Anwendung der Übersetzungsstrategie von CTC-Graphen modelliert (vgl. Kapitel 2.7).

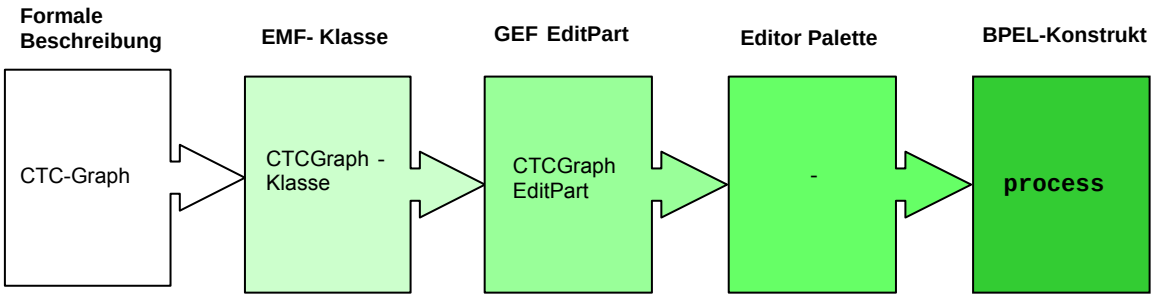
3.1 Von der Formalen Beschreibung zur Übersetzung

An dieser Stelle werden die Parallelen zwischen der Formalen Beschreibung, den Klassen des EMF-Modells, den GEF EditParts, die Modellierungswerkzeuge in der Editor Palette und den BPEL-Konstrukten anhand einer schematischen Darstellung verdeutlicht.

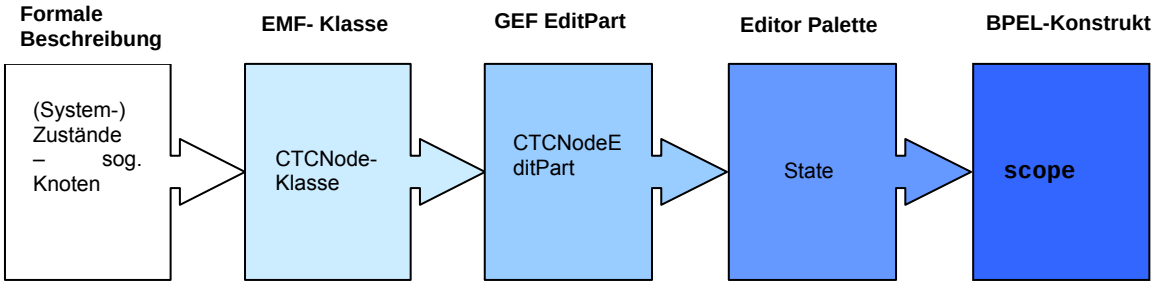
Dieses Verständnis ist von zentraler Bedeutung für die folgenden Schritte der Konzeption und Implementierung des CTC-Graph Editors und der Übersetzung der CTC-Graphen in BPEL-Code.

Das Schaubild zeigt die schrittweise, aufeinander aufbauende Vorgehensweise der vorliegenden Arbeit.

1. Coordinated Transactional Communitcation-Graph:



2. Systemzustand (sog. Knoten):



3. Coordinator- und Participant-Nachrichten (sog. Kanten):

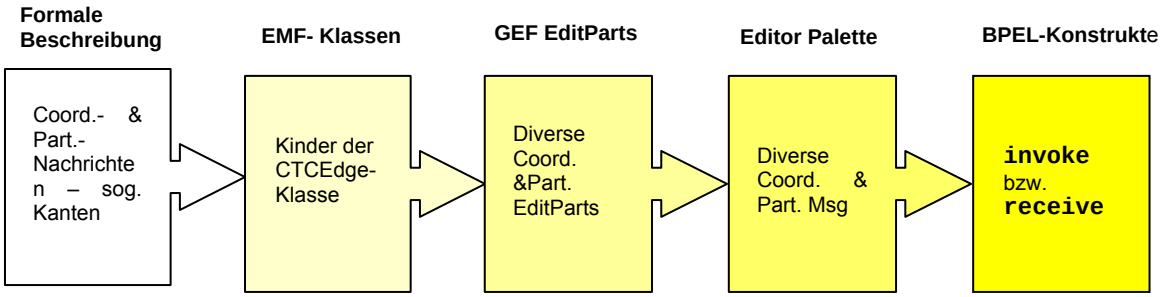


Abb. 3-1: Vorgehensweise ab Formaler Beschreibung bis BPEL-Konstrukt

3.2 Das EMF-Modell

Unter Zuhilfenahme der in Kapitel 2.6 durchgeführten Analyse der benötigten Elemente für CTC-Graphen, wurde mit dem „OMONDO-Plug-in“ [24] ein EMF-Klassendiagramm für CTC-Graphen erstellt.
Die automatische Codegenerierung des EMF-Modells aus dem entworfenen Klassendiagramm erleichtert die Erzeugung der benötigten Java-Klassen, die im Laufe der Implementierung des CTC-Graph Editor Plug-ins benötigt werden.

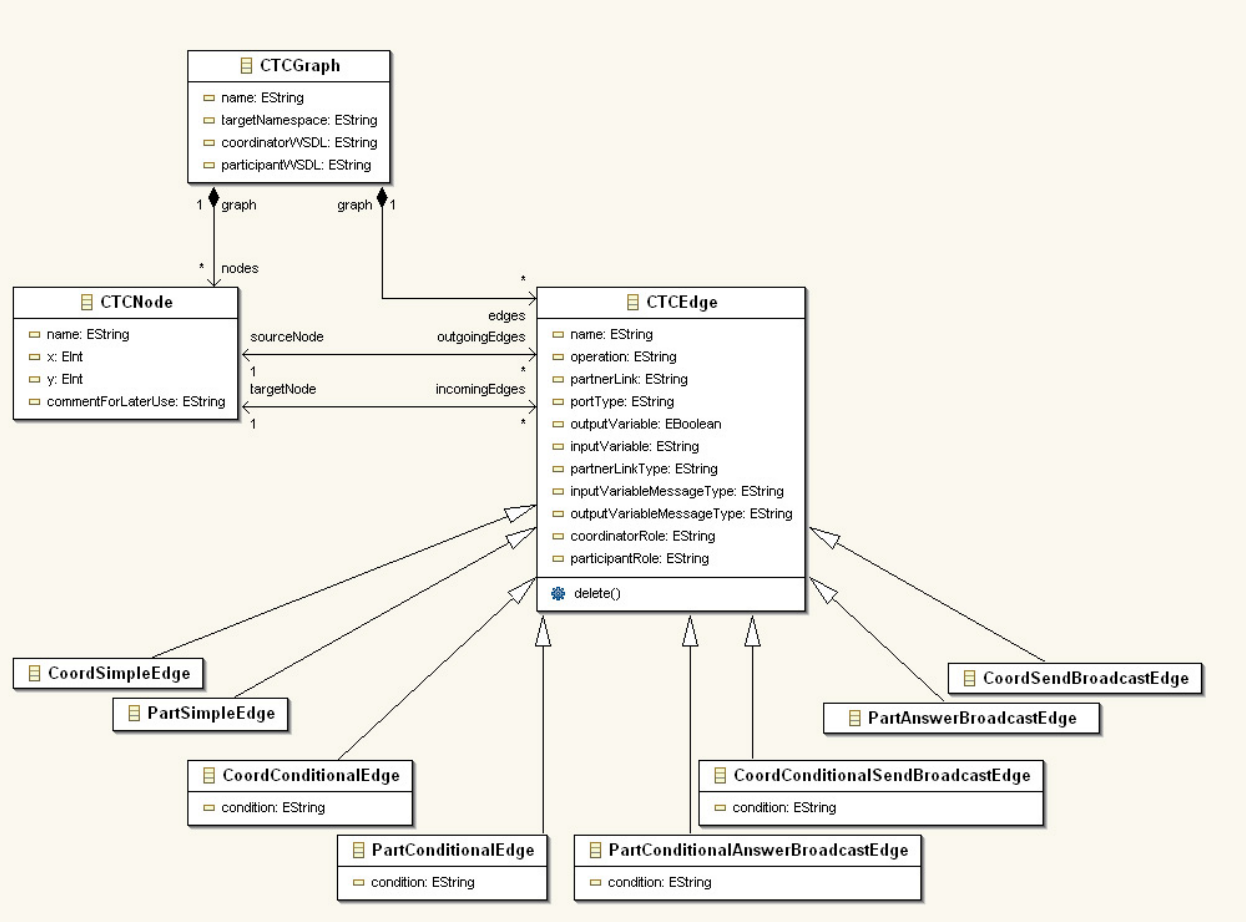


Abb. 3-2: EMF-Klassendiagramm für CTC-Graphen [15]

Das abgebildete Klassendiagramm zeigt die modellierten Klassen mit ihren Attributen, Assoziationen und Generalisationen. Die Klassen bilden die Grundstruktur der CTC-Graphen ab. Sie stellen die Grundlage für die Programmierung der zu modellierenden Graphen im Editor dar.

Attribute verdeutlichen einzelne spezielle Eigenschaften einer Klasse. Die im EMF-Klassendiagramm gewählten Attribute (mit Ausnahme der in GEF benötigten Positionierungs-Attribute „x“ und „y“ für die CTCNode-Klasse) werden mit Hilfe der BPEL-Spezifikation [1] im Hinblick auf die spätere BPEL-Code Generierung bestimmt. Die Namen der Attribute sind ebenfalls eindeutig und lassen sich weitgehend deckungsgleich in BPEL-Struktur-Attribute

[15] Screenshot aus der Eclipse Umgebung mit dem verwendeten OMONDO-Plug-in [23]

übersetzen. Attribute werden als Einträge mit Namen und Typen in den Rechtecken, die eine Klasse darstellen, abgebildet.

Assoziationen verdeutlichen Zusammenhänge, Verhältnisse und Rollen der Klassen des EMF-Klassendiagramms und werden durch gerichtete Kanten mit offener Pfeilspitze dargestellt. Die Beschriftung der Kanten zeigt in welchem Verhältnis und in welcher Rolle die einzelnen Klassen zueinander stehen. Ein Beispiel für eine Assoziation ist das Verhältnis zwischen der CTCGraph-Klasse und der CTCNode-Klasse.

Generalisationen verdeutlichen Verallgemeinerungen verschiedener (spezialisierter) Klassen zu einer gemeinsamen „Oberklasse“. Diese Oberklasse enthält alle gemeinsamen Attribute der spezialisierten Unterklassen. Sämtliche Attribute, die lediglich von den Unterklassen selbst benötigt und verwendet werden, bleiben als Attribute allein dieser Unterklasse bestehen. Generalisationen werden durch gerichtete Pfeile mit offener Pfeilspitze im EMF-Klassendiagramm dargestellt. Ein Beispiel für eine Generalisation ist das Verhältnis zwischen der PartSimpleEdge-Klasse und ihrer Oberklasse CTCEdge.

Es folgt eine Aufstellung der verwendeten EMF-Datentypen. Dies soll das Verständnis für die Attribute der drei EMF-Klassen fördern.

Datentyp	Erläuterung
EString	stellt eine beliebige Zeichenkette dar
Eint	stellt einen Zahlenwert dar
EBoolean	stellt einen Zweiwertigen Datentyp dar, mit den möglichen Ausgabewerten „`true`“ oder „`false`“

Das EMF-Modell besteht aus den im Folgenden beschriebenen drei „Hauptklassen“.

3.2.1 Die CTCGraph-Klasse

In der Formalen Beschreibung in Kapitel 2.6.3 wurde bereits ausgeführt, dass ein CTC-Graph (CTCGraph-Klasse) aus mehreren Knoten (CTCNode-Klasse) verbunden durch Kanten (CTCEdge-Klasse) besteht. Die Beziehungen zwischen Graph/Knoten und Graph/Kanten können dem EMF-Klassendiagramm entnommen werden.

Des Weiteren können aus dem Klassendiagramm die Attribute der Klasse CTCGraph gelesen werden. Der CTCGraph soll später in die BPEL Anfangsstruktur „`process`“ übersetzt werden. Laut BPEL Spezifikation [1] sind die Attribute „`name`“, „`targetNamespace`“ und die URI für die zu verwendende WSDL-Datei (in den Attributen „`coordinatorWSDL`“ und „`participantWSDL`“ des CTCGraphen) für jeden BPEL-Prozess wesentlich und nicht fakultativ. Deshalb wurden genau diese vier Attribute in die CTCGraph-Klasse eingefügt.

Alle vier Attribute sind vom Typ EString. Dieser kann die gewünschten Attribut-Werte entgegen nehmen.
Diese Attribute wurden während der Modellierung der CTCGraph-Klasse beachtet, weil sie Pflichteinträge der zu generierenden BPEL-Dateien darstellen. Die Werte der Attribute können vom Anwender angegeben werden. Erhalten diese Attribute keinen Namen, so erhalten sie einen leeren String („“) als default-Wert in den erzeugten beiden Dateien. Ein leerer String als Wert der soeben genannten Pflicht-Attribute könnte zu Schwierigkeiten beim Import der generierten BPEL-Dateien in BPEL-Editoren bzw. BPEL-Designer führen. Eine nicht-leere Eingabe ist in diesem Fall sehr zu empfehlen.

Eine Überprüfung des Inhaltes der Attribute ist jedoch in dieser Version des Editors nicht vorhanden. Eine solche Erweiterung wäre denkbar und nützlich – hierzu folgt mehr in Kapitel 4 „Zusammenfassung und Ausblick“.

Das Attribut „`participantWSDL`“ nimmt eine sehr spezielle und bedeutende Rolle bei der Übersetzung des CTC-Graphen in BPEL ein. Der Benutzer kann mehrere durch Semikolon („;“) getrennte WSDL-URIs angeben. Diese Auflistung wird bei der BPEL-Übersetzung nicht nur verwendet um die richtige Anzahl von Teilnehmer-BPEL-Dateien zu erzeugen sondern ist auch wichtig für die korrekte Funktionsweise des Sendens und Empfangens von Broadcast-Nachrichten (vgl. Kapitel 2.6.2.1).

Die CTCGraph-Klasse ist eine Art Container, in dem die einzelnen Elemente (Knoten und Kanten) eingefügt werden können. Die CTCGraph-Klasse und ihre Instanzen werden gerne auch als Eltern (parent) der Systemzustände (node) und Kanten (edge) bezeichnet.

3.2.2 Die CTCNode-Klasse

Die CTCNode-Klasse stellt die Knoten des CTC-Graphen dar. Die Knoten des Graphen repräsentieren die Systemzustände in denen sich das transaktionale System befindet.

Auch diese Klasse enthält Attribute, die für die spätere Übersetzung in BPEL notwendig sind.
Wie in Kapitel 2.6.2 bereits erläutert, werden die Systemzustände in dem BPEL-Konstrukt „`scope`“ übersetzt. Damit ist eine klare und einfache Strukturierung der generierten BPEL-Dokumente sichergestellt. Ein „`scope`“ benötigt einen Namen, der mit Hilfe des Attributes „`name`“ (vom Typ EString) vom Benutzer abgefragt bzw. angegeben wird. Die CTCNode-Klasse bietet ein zusätzliches Attribut namens „`commentForLaterUse`“ (ebenfalls vom Typ EString). Dieses Attribut soll den Anwendern die Möglichkeit bieten, eigene Kommentare zu den erzeugten Systemzuständen in die BPEL-Datei einzufügen. Dies kann als Gedächtnisstütze und anschließende Hilfe bei der Ergänzung der Programm-Logik dienen.

Die CTCNode-Klasse besitzt Assoziationen zu den Klassen CTCGraph und CTCEdge.
Ein Graph kann viele (in Abb. 3-2 mit „*“ gekennzeichnet) Knoten („nodes“) enthalten. Die Knoten (CTCNode) ihrerseits können nur zu einem (in Abb. 3-2 mit „1“ gekennzeichnet) Graphen („graph“) eine Beziehung haben – was die Gewichtung der Kantenspitzen im EMF-Klassendiagramm erklärt.

Die Beziehung zwischen Knoten und Kanten (CTCEdge) ist wie folgt: Ein Knoten kann mehrere (in Abb. 3-2 mit „*“ gekennzeichnet) ausgehende Kanten („outgoingEdge“) wie auch mehrere (in Abb. 3-2 mit „*“ gekennzeichnet) eingehende Kanten („incomingEdge“) besitzen. Eingeschränkt wird diese Aussage lediglich durch die Art der Kanten (vgl. Kapitel 3.2.3).

3.2.3 Die CTCEdge-Klasse

Die CTCEdge-Klasse stellt die Kanten des CTC-Graphen dar. Diese Kanten repräsentieren die Aktionen, die für Zustandsübergänge verantwortlich sind.

CTCEdge stellt das Versenden bzw. das Empfangen von Nachrichten dar und wird dementsprechend entweder als ein „`invoke`“ oder als ein „`receive`“ in BPEL abgebildet. Eine Zusammenfassung der Übersetzungsstrategie befindet sich in Kapitel 2.7.

Jedes Versenden oder Empfangen von Nachrichten benötigt in BPEL weitere Informationen zur Sicherstellung, dass die Nachricht den richtigen Empfänger zugestellt wird.
Diese allgemein geltenden Informationen sind:

- Name der Sende-/Empfangs-Aktion: Er wird mit Hilfe des Attributes „`name`“ vom Benutzer abgefragt und eingegeben. Der Typ dieses Attributes ist EString, mit ihm kann der gewünschte Name der Nachricht als Attribut-Wert angegeben werden.
- Der Name des Partner-Links für das Senden / Empfangen von Nachrichten wird mit Hilfe des Attributes „`partnerLink`“ vom Benutzer abgefragt/eingegeben. Der Typ dieses Attributes ist ebenfalls EString.
- Der Name des Partner-Link Typs des angegebenen „`partnerLink`“ (s.o.) wird ebenfalls vom Benutzer abgefragt. Der Typ des Attributes „`partnerLinkType`“ ist vom Typ EString.
- Für die vollständige Definition der „`partnerLink`“ in einer BPEL-Datei werden auch die Rollenangaben der Kommunikationspartner benötigt. Sie können anhand der Attribute „`coordinatorRole`“ und „`participatRole`“ vom Benutzer angegeben werden. Beide Attribute sind vom Typ EString. Mit ihm können die gewünschten Rollennamen eingetragen werden.
- Der Name des für die Sende-/Empfangs-Aktion verwendeten Port-Typen wird mittels des Attributes „`portType`“ vom Benutzer abgefragt und eingegeben. Auch bei dem Attribut „`portType`“ ist der angegebene Typ ein EString.

- Der Name der eingesetzten Operation für das Senden und Empfangen von Nachrichten wird über das Attribut „operation“ vom Benutzer abgefragt. „Operation“ ist ebenfalls von Typ EString.
- Die Pflicht-Variable „inputVariable“ dient zur Zwischenspeicherung der zu versendenden Nachrichten und ist vom Typ EString. Der Name der Variablen kann frei gewählt werden.
- Die fakultative „outputVariable“, soll das Ergebnis entgegennehmen falls eine Antwort auf die gesendete Nachrichte erwartet wird. Die Antwort wird mittels Attribut „outputVariable“ vom Benutzer abgefragt und eingegeben. Hierbei handelt es sich nicht um einen Attribut des Typs EString sondern um den Typen EBoolean. Mit diesem Attribut gibt der Benutzer an, ob eine solche Variable benötigt wird („true“) oder nicht („false“). Als Name erhält die Variable den Namen der Nachricht, so wird die Zugehörigkeit der Variablen zur Nachricht verdeutlicht.
- Für die vollständige und korrekte Deklaration der Variablen benötigt BPEL zwei Nachrichten-Typen: den sog. „inputVariableMessageType“ für die „inputVariable“ wie auch den „outputVariableMessageType“ für die „outputVariable“. Die Typenangaben können ebenfalls vom Benutzer abgefragt und eingegeben werden. Die Attribute „inputVariableMessageType“ und „outputVariableMessageType“ sind beide vom Typ EString.

Auch an dieser Stelle ist eine Überprüfung des Inhaltes der Attribute nicht in dieser Version des Editors vorhanden. Eine solche Erweiterung wäre denkbar und nützlich – hierzu folgt mehr in Kapitel 4 „Zusammenfassung und Ausblick“.

Das Versenden einer Nachricht stellt die letzte Aktion in einem Systemzustand dar. Anschließend erfolgt der Zustandsübergang. Die erste Aktion in dem neuen Zustand ist der Empfang einer Nachricht (vgl. Kapitel 2.7). Der Empfang wird nicht explizit im Graphen abgebildet. Die Aktion „receive“ enthält ebenfalls alle benötigten Informationen der Attribute, welche die zugehörige Kante enthält („name“, „partnerLink“, „portType“, „operation“, „inputVariable“). Die Attribut-Werte von „receive“ sind identisch mit den Attribut-Werten des zuvor stattgefundenen „invoke“.

Die CTCEdge-Klasse besitzt als einzige die Methode „delete()“. Sie ist für die spätere Implementierung des CTC-Graph Editors bedeutend. Die „delete()“-Methode kann, wie der Name bereit verdeutlicht, Kanten des CTC-Graphen löschen.

Weil die „delete()“–Methode die einzige Ergänzung der erzeugten Klassen des EMF-Klassendiagramms und einfach verständlich ist, wird sie an dieser Stelle kurz vorgestellt. Das folgende Beispiel verdeutlicht insbesondere die Vorgehensweise – speziell den Zugriff auf die Attribut-Werte einer Klasse. Die Implementierung dieser Methode war für die persönliche Aufwandsabschätzung der bevorstehenden Programmieraufgabe des CTC-Graph Editors in Bezug auf die Manipulation der CTC-Graph-Daten von großem Nutzen.

```
public void delete()[16]
{
    // get source node, target node and parent graph
    CTCGraph parent = this.getGraph();
    CTCNode source = this.getSourceNode();
    CTCNode target = this.getTargetNode();

    // delete the edge
    source.getOutgoingEdges().remove(this);
    target.getIncomingEdges().remove(this);
    parent.getEdges().remove(this);
}
```

Wie bereits im vorgehenden Kapitel 2.6.3 angedeutet, benötigt ein CTC-Graph unterschiedliche Kanten-Arten, welche die Nachrichten, die vom Koordinator oder von Teilnehmern versandt werden, darstellen. Diese Unterscheidung zwischen den Absendern einer Nachricht wurde auch bei dem EMF-Klassendiagramm beachtet. Gleiches gilt für die unterschiedlichen Arten von Nachrichten.

Folgende Generalisationen der CTCEdge-Klasse kristallisierten sich mit Hilfe der Analysen in Kapitel 2.6 „Entwurf eines geeigneten Diagramms für CTC-Graphen“ heraus:

1. CoordSimpleEdge
2. PartSimpleEdge
3. CoordConditionalEdge
4. PartConditionalEdge
5. CoordSendBroadcastEdge
6. PartAnswerBroadcastEdge
7. CoordConditionalSendBroadcastEdge
8. PartConditionalAnswerBroadcastEdge

3.2.3.1 CoodSimpleEdge-Klasse

Diese Klasse repräsentiert die einfachste Art von Nachrichten, die ein Koordinator verschicken kann. Diese einfache Nachrichtenart benötigt keine zusätzlichen Attribute zu den o.g. Standart-Attributen. Im CTC-Graph Editor wird diese Klasse durch eine durchgezogene, gerichtete und mit Namen beschriftete Kante dargestellt. Der Übergang zu einem genau vordefinierten Systemzustand erfolgt automatisch nach dem Versenden einer Nachricht.

[16] In CTC-Graphen werden die Kanten als Verbindung zwischen den Ziel- und Quellknoten angegeben. In der `delete()`-Methode wird diese Referenz genutzt um die Kante zu entfernen. Werden die Referenzen der zu löschenden Kante aus Ziel- und Quellknoten und anschließend auch aus dem CTC-Graphen gelöscht, so ist sie nicht mehr existent.

3.2.3.2 PartSimpleEdge-Klasse

Analog zur CoordSimpleEdge-Klasse stellt auch die PartSimpleEdge-Klasse die einfachste Art von Nachrichten dar, die ein Teilnehmer einer Transaktion dem Koordinator schicken kann. Diese einfache Nachrichtenart wird durch eine gestrichelte, gerichtete und mit Namen beschriftete Kante im CTC-Graphen abgebildet. Der Übergang zu einem genau vordefinierten Systemzustand erfolgt automatisch nach dem Versenden der Nachrichten.

3.2.3.3 CoodConditionalEdge-Klasse

Die CoordConditionalEdge-Klasse stellt die Klasse der bedingungsgeknüpften Nachrichten des Koordinators dar. Diese Art der bedingungsgeknüpften Nachrichten wird im CTC-Graphen durch eine durchgezogene, gerichtete Kante veranschaulicht, die mit einem Namen und der in eckigen Klammern gefassten Bedingung versehen ist. Der Übergang zu einer bestimmten Menge an vordefinierten Systemzuständen erfolgt erst nachdem, die angegebene Bedingung ausgewertet wurde. Anschließend kann entschieden werden, welche Nachricht versandt wird.

3.2.3.4 PartConditionalEdge-Klasse

Analog zu der bereits beschriebenen CoordConditionalEdge-Klasse stellt die PartConditionalEdge-Klasse das Versenden bedingungsgeknüpfter Nachrichten Seitens der Teilnehmer dar. Im CTC-Graphen ist diese Nachrichtenart durch gestrichelte, gerichtete Kanten abgebildet, die mit einen Namen und der in eckigen Klammern gefassten Bedingung versehen sind. Der Übergang zu einer bestimmten Menge vordefinierter Systemzustände erfolgt erst nachdem die angegebene Bedingung ausgewertet wurde. Anschließend wird entschieden, welche Nachricht zu verschicken ist.

3.2.3.5 CoodSendBroadcastEdge-Klasse

Die CoordSendBroadcastEdge-Klasse stellt die Art von Nachrichten dar, die ein Koordinator verwenden kann, um eine Broadcast-Nachricht zu verschicken. Diese Art der Nachrichten wird vom Koordinator bei bestimmten Anlässen verwendet, um allen Teilnehmern einer Transaktion dieselbe Nachricht zu senden. Die CoordSendBroadcastEdge-Klasse wird im CTC-Graphen durch eine breite, durchgezogene Kante mit offener Pfeilspitze veranschaulicht. Des Weiteren hilft auch die Beschriftung der Kanten bei der Unterscheidung dieses Nachrichten-Typs. Die Kante erhält neben der Namensbeschriftung auch ein Präfix „<BC>“ (als Kurzform von Broadcast). Dieses Präfix dient lediglich der optischen Darstellung und wird nicht für die Übersetzung in BPEL verwendet. Ähnlich wie bei den CoordSimpleEdge Nachrichten-Typ erfolgt nach dem Versandt der Nachrichten automatisch der Übergang in einen genau vordefinierten Systemzustand.

3.2.3.6 PartAnswerBroadcastEdge-Klasse

Die PartAnswerBroadcastEdge-Klasse stellt die Antwortnachrichten dar, welche die Teilnehmer einer Transaktion dem Koordinator schicken können. Diese Nachrichten werden nicht spontan versandt, sondern sind Antworten der Teilnehmer auf eine vorangegangene CoordSendBroadcastEdge-Nachricht. Die PartAnswerBroadcastEdge-Klasse wird im CTC-Graphen durch eine breite, gestrichelte Kante mit offener Pfeilspitze dargestellt. Auch hier hilft die Beschriftung der Kanten bei der Unterscheidung des Nachrichten-Typs. Die Kante erhält neben der Namensbeschriftung auch das Präfix „<BC>“. Dieses Präfix dient lediglich der optischen Darstellung und wird nicht für die Übersetzung in BPEL verwendet. Ähnlich wie bei den PartSimpleEdge Nachrichten-Typen erfolgt nach dem Versandt der Nachrichten automatisch der Übergang in einen genau vordefinierten Systemzustand.

3.2.3.7 CoordConditionalSendBroadcastEdge-Klasse

Analog zu den Unterschieden zwischen der CoordSimpleEdge-Klasse und der CoordConditionalEdge-Klasse bietet diese Klasse die Möglichkeit Broadcast-Nachrichten bedingungsgeknüpft zu verschicken.
Die CoordConditionalSendBroadcastEdge-Klasse wird im CTC-Graphen durch eine breite, durchgezogene Kante mit offener Pfeilspitze dargestellt. Auch hier hilft die Beschriftung der Kanten bei der Unterscheidung des Nachrichten-Typs. Die Kante erhält neben der Namensbeschriftung und dem Präfix „<BC>“ eine in eckigen Klammern eingefasste Bedingung. Auch hier dient das Präfix lediglich der optischen Darstellung und wird nicht für die Übersetzung in BPEL verwendet. Ähnlich wie bei den CoordConditionalEdge Nachrichten-Typen erfolgt der Übergang zu einer bestimmten Menge vordefinierter Systemzustände erst nachdem die angegebene Bedingung ausgewertet wurde. Anschließend wird entschieden, welche Nachricht zu verschicken ist.

3.2.3.8 PartConditionalAnswerBroadcastEdge-Klasse

Analog zu den Unterschieden im Zusammenhang zwischen der PartSimpleEdge-Klasse und der PartConditionalEdge-Klasse bietet diese Klasse die Möglichkeit Antworten auf Broadcast-Nachrichten bedingungsgeknüpft zu verschicken. Die PartConditionalAnswerBroadcastEdge-Klasse wird im CTC-Graphen durch eine breite, gestrichelte Kante mit offener Pfeilspitze dargestellt. Auch hier hilft die Beschriftung der Kanten bei der Unterscheidung des Nachrichten-Typs. Die Kante erhält neben der Namensbeschriftung und dem Präfix „<BC>“ eine in eckigen Klammern eingefasste Bedingung. Auch hier dient das Präfix dient lediglich der optischen Darstellung und wird nicht für die Übersetzung in BPEL verwendet. Ähnlich wie bei den PartConditionalEdge Nachrichten-Typen erfolgt der Übergang zu einer bestimmten Menge vordefinierter Systemzustände erst nachdem die angegebene Bedingung ausgewertet wurde. Anschließend wird entschieden, welche Nachricht zu verschicken ist.

3.2.4 Erfahrungen mit EMF.Codegen

Unter Zuhilfenahme der in Kapitel 1.2.1 bereits erwähnten EMF Code-Generierungsfunktion wurde das Modell in Java-Klassen übersetzt. Mittels des modellierten EMF-Klassendiagramms wurden die nötigen Klassen-Definitionsdateien erzeugt, die zu einem späteren Zeitpunkt im Editor verwendet werden können.

Die EMF.Codegen-Funktionalität erspart dem Anwender aufwendige Programmierarbeiten. Die hieraus resultierende Zeitersparnis wurde später für die Ergänzung der zu implementierenden Methode und des CTC-Graph Editors benötigt.

Weil die CTCEdge-Klasse lediglich eine Methode („`delete()`“) besitzt, konnten die Arbeiten am EMF-Klassendiagramm und der Klassen-Java-Dateien zügig abgeschlossen werden.

Die generierten Dateien werden in einem Package, in diesem Fall dem „CTCGraphModel“-Package, erzeugt. Die Repräsentation der einzelnen Klassen werden in gleichnamigen Java-Klassen erzeugt (z.B. CTCGraph.java, CTCNode.java, …). Die eigentlichen Implementierungen der Klassen-Repräsentationen sind im Package „impl“ (in diesem Fall „CTCGraphModel/impl“-Package) enthalten.
Zusätzlich wird auch ein „util“-Package generiert, welches eine „AdapterFactory“ und einen „ModelSwitch“ enthält. Diese Klassen sind wichtig für die Erzeugung der einzelnen Instanzen und zur Navigation innerhalb der einzelnen Instanzen in einem Graphen.

Das EMF-Klassendiagramm bildet die Grundlage für die Implementierung des CTC-Graph Editors.

3.3 Der CTC-Graph Editor

Der CTC-Graph Editor und seine Klassen basieren auf dem oben beschriebenen EMF-Modell. Wie bereits geschildert, erleichtern die von GEF angebotenen Funktionalitäten die Programmierung eines graphischen Editors.

Das modellierte EMF-Klassendiagramm und die dazugehörigen generierten Java-Klassen ermöglichen dem Programmierer, sich bei der Implementierung eines GEF-Editors lediglich auf dessen Funktionalität und Präsentation konzentrieren zu müssen.

3.3.1 Erfahrungen bei der Einarbeitung in GEF

Die Palette der in GEF angebotenen Funktionalitäten werden in „Eclipse Development using the Graphical Editing Framework and the Eclipse Modelling Framework“ [21] oder auf der Eclipse GEF-Homepage unter „Graphical Editing Framework (GEF) Overview“ [13] behandelt. Als sehr aufwändig hat sich die anschließende Implementierung mit GEF erwiesen. Obwohl im Internet und in Fachzeitschriften eine Vielzahl von Artikeln zum Thema GEF zu finden sind, wird dabei immer auf das EMF/GEF-„Standardwerk“, IBMs Redbook „Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework“ [21], als Quelle zurückgreifen.

Die eingehende Analyse unterschiedlicher mit GEF erzeugter Editoren zeigte auch, dass sich viele auf die in „Eclipse Development using the Graphical Editing Framework and the Eclipse Modelling Framework“ [21] enthaltenen Redbook-Samples anlehnen. Oft bildet das „Network Editor“ Sample die Grundlage dieser Editoren. Der Einstieg über die Grundstrukturen und Vorgehensweisen bei der Implementierung mit GEF erfolgt rasch und ist auch für Anfänger einfach verständlich.

Der im Rahmen dieser Arbeit entwickelte Editor basiert ebenfalls auf den Grundstrukturen des oben genannten „Network Editors“. [21]
Der „Network Editor“ und die darauf aufbauenden Editoren bieten eine solide Implementierung, welche die Struktur eines grafischen Editors und die Vorgehensweisen bei dessen Implementierung verdeutlichen. GEF bietet jedoch darüber hinaus genügend Stolperfallen bei der „einfachen“ Implementierung benötigter Bestandteile eines Editors. Problematisch ist die fehlende Literatur zum Thema GEF. Hier besteht großer Entwicklungsbedarf.

3.3.2 Vorgehensweise und Überlegungen bei der Implementierung des Editors

Wie schon in Kapitel 1.2.2 „Graphical Editing Framework“ beschrieben, verwendet GEF das Model-View-Controller Modell. Der erste Schritt, die Erzeugung eines Models, wurde bereits mit Hilfe des EMF-Klassendiagramms und den generierten und erweiterten CTC-Graph Java-Klassen gemacht. Die View- und Controller-Komponenten werden noch benötigt, um von den in GEF verankerten Funktionalitäten Gebrauch machen zu können.

Die Rolle des Controllers übernimmt in GEF der zu implementierende EditPart.
GEF stellt den Viewer zur Verfügung (z.B. den ScrollingGraphicalViewer), der die Viewer-Komponente des MVC-Entwurfsmusters realisiert. Jedem Viewer wird eine EditPart-Factory zugeordnet. Die Factory übernimmt Aufgaben wie die Herstellung einer konkreten Präsentation für die Elemente des abstrakten Datenmodells.

Für die Implementierung eines GEF-Editors ist das Verständnis folgender Bestandteile und deren Wechselbeziehungen notwendig:

1. EditPart
2. Connections
3. EditPolicy
4. Command
5. Palette

3.3.2.1 EditPart

Aufgabe und Funktionsweise von EditParts:

EditParts übernehmen die Rolle des Controllers in der GEF-MVC-Architektur und sind dementsprechend verantwortlich für Änderungen am Modell. Einzelne komplexe Strukturen des zuvor angefertigten Modells können aus verschiedenen EditParts bestehen oder Referenzen auf andere Strukturen und deren EditParts besitzen.
Ein EditPart ist zuständig für die Abbildung des Models in dem dazugehörigen View. Alle Änderungen am Model werden vom EditPart entgegengenommen, verarbeitet und ggf. erfolgt anschließend eine Aktualisierung des Views. Diese Änderung bildet den aktuellen Model-Zustand ab.

Zuständigkeiten eines EditParts:

- Erzeugung des Figure-Objektes (die sichtbare Repräsentation eines Model-Objektes im View)
- Bestimmung der enthaltenen Objekte im aktuellen View
- Aktualisierung des Figure-Objektes bei Modelländerungen (Änderung der sichtbaren Repräsentation des Modell-Objektes)
- Umsetzung von Objekt-Erzeugungs- und Bearbeitungsanfragen des Benutzers in Commands (siehe Kapitel 3.3.2.5).

Die EditParts werden mit Hilfe von EditPartFactory-Instanzen erzeugt, die je nach Typ des abzubildenden Objektes aus dem Datenmodell eine entsprechende EditPart-Instanz erzeugen. Für die Erzeugung und Verwaltung von Objekt-Instanzen verwenden EditParts eine oder mehrere EditPartPolicies.

Verwendung im CTC-Graph Editor:

Für jede durch EMF.Codegen generierte CTC-Graph Java-Klasse muss ein EditPart implementiert werden. Die implementierten EditParts werden nach ihren EMF-Klassen benannt. Zum Beispiel findet die CTCGraph.java Klasse als CTCGraphEditPart.java ihre Abbildung im GEF-Editor. Dort werden die wichtigsten Eigenschaften und Grundfunktionalitäten des EditParts implementiert, die für einen Controller (im MVC-Model) notwendig sind.

Zu den Grundfunktionalitäten zählen unter anderem:

- der Konstruktor des Objektes
- das Figure-Objekt (die graphische Darstellung des Objekts)
- Verwendung des „`IPropertySource`“-Elements, welches die Objekt-Attribute und die dazugehörigen Werte im PropertyView anzeigen. Mit Hilfe des soeben erwähnten PropertyViews können die Werte der Attribute vom Anwender bearbeitet werden.
- Erzeugung der PropertySource zur Anzeige und Editierung im PropertyView.
- Zuordnung der EditPartPolicies eines EditParts in der zu implementierenden „`createEditPolicies()`“-Methode (mittels „`installEditPolicy(String, EditPolicy)`“).

3.3.2.2 EditPartFactory

Aufgaben und Funktionsweisen der EditPartFactory:

Jede GEF-Applikation benötigt mindestens eine Factory um die EditParts bestimmter modellbasierender Objekte zu erzeugen. Dabei muss in jedem EditPart mindestens die Methode „`createEditPart(EditPart, Object)`“ implementiert werden. Diese Methode erzeugt die gewünschten Objekte.

Verwendung im CTC-Graph Editor:

Die Klasse GraphicalEditPartsFactory.java implementiert die EditPartFactory und erzeugt alle benötigten Elemente wie den Graphen (CTCGraph), seine Knoten (CTCNode) und Kanten (Kinder der CTCEdge-Klasse wie: CoordSimpleEdge, PartSimpleEdge, CoordConditionalEdge, PartConditionalEdge, CoordSendBroadcastEdge, PartAnswerBroadcastEdge, CoordConditionalSendBroadcastEdge, PartConditionalAnswerBroadcastEdge).
Die Implementierung der Klasse ist kurz und übersichtlich und enthält lediglich die Generierung der gewünschten graphischen Elemente (vgl. SourceCode).

3.3.2.3 EditDomain

Aufgaben und Funktionsweisen der EditDomain:

Die EditDomain enthält den kompletten Status einer GEF-Applikation. Die zentrale Komponente bildet der „`CommandStack`“. In ihm sind alle ausgeführten Commands gespeichert. Deshalb ist die EditDomain eng mit dem eigentlichen Grafik-Editor gekoppelt.

Verwendung im CTC-Graph Editor:

Die EditDomain wird in der Klasse AbstractEditorPage.java definiert. Von dieser Klasse wird die eigentliche CTCGraphPage.java abgeleitet. Die allgemeinen Funktionalitäten des Editors werden in AbstractEditorPage festlegt. Zu den Grundfunktionen des Editors zählen z.B.:

- „`doSave()`"-Methode, sie definiert das Verhalten der Speichermethode.
- „`isDirty()`"-Methode, sie zeigt an ob die Änderungen an den Graphen vorgenommen, aber noch nicht festgeschrieben (in Form einer Speicherung der aktuellen Fassung) wurden.
- „`getCommandStack()`"-Methode, für das Speichern aller auf den Graphen aufgeführten Befehle.
- Erzeugung und Positionierung der Palette im Editor mittels der dazugehörigen Java-Klasse zur Bestückung der Werkzeugpalette.
- Methoden zur genauen Beschreibung des Aussehens des CTC-Graph Editors.
- Methoden zur Registrierung und Konfiguration vom EditPartViewer.

3.3.2.4 EditPartPolicy

Aufgaben und Funktionsweisen der EditPartPolicies:

EditPolicies erweitern den Grafik-Editor mit unterschiedlichen Editier-Funktionen. Sie werden separat an einen Editor gekoppelt und in eigene Klassen implementiert. Damit ist die Erweiterbarkeit und Wartungsfreundlichkeit des Editors gewährleistet. Im Editor werden Policies in der Methode „`createEditPolicies()`" und mit dem Aufruf „`installEditPolicy(String, EditPolicy)`" definiert.

In einer EditPolicy besteht die Möglichkeit Commands zu definieren, die ausgeführt werden, sobald eine bestimmte Aktion ausgelöst wurde.

Verwendung im CTC-Graph Editor:

Wie bereits erwähnt, benötigen EditParts mindestens eine Policy, die gewisse Verhalten, Einschränkungen (z.B. bei der Erzeugung bestimmter Kanten) oder auch das Aussehen graphischer Elemente (im Sonderfall Connections[17]) steuert.

Weil bei der Erläuterung die EditParts der CTCGraphEditPart verwendet wurden, wird an dieser Stelle, zur Verdeutlichung der EditPartPolicies, kurz auf die dazugehörigen Policies dieses Objektes eingegangen.

[17] An den Connections wird deutlich, dass die angestrebte klare MVC-Architektur nicht konsequent in GEF durchgesetzt wurde.

Die CTCGraphEditPart-Klasse verwendet zwei Policies:

- Die CTCGraphXYLayoutEditPolicy: Sie steuert das Verhalten für das dazugehörige XY-Layout, welches von GEF angeboten wird und passt die Bedürfnisse des CTC-Graph Editors an das Layout an. Darin werden z.B. die Mindestgröße für Kinder der CTC-Graphen (Knoten oder Kanten) definiert.
- Die CTCGraphEditPolicy: Sie erbt von ContainerEditPolicy, die eine Implementierung der Methode „`getCreateCommand()`“ vorschreibt. Die CTCGraphEditPolicy stellt allerdings eine Ausnahme dar, weil CTC-Graphen nicht direkt vom Benutzer erstellt werden können. Ein Graph wird stattdessen durch die Erzeugung einer „`.ctc`“ Datei im Plug-in generiert. Er wird als Hintergrund des Editors repräsentiert. Deshalb liefert „`getCreateCommand()`“ „`null`“ als Rückgabewert. Hierdurch wird auch die Einschränkung eingehalten, dass CTC-Graphen nicht verschachtelt und damit auch keine Schleifen modelliert werden können. Der Benutzer kann auf diesem Wege bspw. die Einschränkung, dass CTC-Graphen azyklisch sein müssen, nicht umgehen.

3.3.2.5 Commands

Aufgaben und Funktionsweisen der Commands:

Commands werden nach dem Auslösen einer bestimmten Bearbeitungsaktion des Benutzers ausgeführt. Sie werden beispielsweise verwendet, um Elemente aus dem Modell zu löschen oder hinzuzufügen. Diese Aktionen werden mittels der Action-Registry des Editors zur Verfügung gestellt und können dort über den Editor direkt oder über Erweiterungen der Klasse ActionBarContributor hinzugefügt werden.

Die Commands werden im CommandStack des Editors gespeichert. So sind Undo- und Redo-Operationen möglich. Folgende Operationen müssen aus diesem Grund beim Schreiben eines Commands implementiert werden:

- „`undo()`“: Änderungen können rückgängig gemacht werden
- „`redo()`“: Rückgängig gemachte Änderung können erneut vollzogen werden
- „`canUndo()`“: legt fest, ob eine Änderungen rückgängig gemacht werden kann oder nicht
- „`execute()`“: Ausführen von Commands

Verwendung im CTC-Graph Editor:

Die im CTC-Graph verwendeten Aktionen dienen der Erzeugung, Veränderung und dem Löschen graphischer Elemente, den Kind-Objekten des Graphen. Commands können Einschränkungen enthalten, die Auswirkung auf die Erzeugung von Objekten haben (im Sonderfall Connections [17] siehe Seite 80).

Die Erzeugung einer Kante (z.B. CoordSimpleEdge, PartSimpleEdge) oder eines Knoten (CTCNode) wird über Commands gesteuert. Diese Befehle enthalten auch Einschränkungen bzgl. der Erzeugung bestimmter Elemente.
Beispiele für Einschränkungen:

- Keine zwei Kanten besitzen den gleichen Quell- und Zielknoten (Ausnahme bilden die bedingungsgeknüpften Kanten vgl. Kapitel 3.2.3)
- Eine Kante, die eine Antwortnachricht von Teilnehmern auf eine Broadcast-Nachricht darstellt, kann erzeugt werden gdw. bereits eine Broadcast-Nachricht vom Koordinator verschickt worden ist.

3.3.2.6 Palette

Aufgaben und Funktionsweisen der Palette:

Der PaletteViewer beherbergt eine Palette verschiedener Werkzeuge für die Manipulation der Daten in einem Editor.
GEF stellt bereits eine ganze Reihe vorgefertigter Grafikwerkzeuge zur Verfügung. Wird ein Werkzeug betätigt, so wird die zugeordnete Aktion ausgeführt. Auch hier stellt GEF im Package „org.eclipse.gef.ui.actions“ eine große Auswahl vorgefertigter Aktionen bereit (z.B. das „Select“- oder „Marquee“-Werkzeug). Mit Hilfe von Gruppierungen kann die gewünschte Ordnung in die Werkzeugpalette gebracht werden. Icons, welche die angebotenen Aktionen bildlich darstellen, können die Arbeit mit dem Editor erleichtern.

Verwendung im CTC-Graph Editor:

Die Palette ist eines der wichtigsten und am häufigsten verwendeten Bestandteile des CTC-Graph Editors. Sie ist an der linken Seite des Editors verankert und enthält, als Schaltfläche dargestellt, alle Commands zur Erzeugung graphischer Elemente bei der Modellierung eines CTC-Graphen.
In der Palette des CTC-Graph Editors können die Befehle zur Erzeugung von Knoten und Kanten ausgeführt werden.

3.3.2.7 Connections

Aufgaben und Funktionsweisen von Connections:

Connections stellen, wie der Name bereits sagt, Verbindungen zwischen einzelnen Objekten (Objekt-Klassen) her. Sie erweitern den AbstractConnectionEditPart. Connections sind grundsätzlich gerichtet und besitzen immer einen Start- und Endknoten. Hier durch können sie definiert werden. Graphisch sind die Connections an bestimmten Punkten mit den Knoten verbunden, an den sog. Anker-Punkten. Wird ein Start- oder ein Endknoten gelöscht, ist die Verbindung nicht mehr definiert und wird ebenfalls, je nach implementiertem Verhalten, gelöscht.

Dadurch, dass die Verbindungen auch durch EditParts beschrieben werden, besitzen sie ebenfalls eine oder mehrere EditPartPolicies, die ihr Verhalten und Aussehen beschreiben.

Verwendung im CTC-Graph Editor:

Alle Kanten in einem CTC-Graphen werden als Connections implementiert. Die Kanten besitzen eindeutige Quell- und Zielknoten. Die implementierten Connections haben im Allgemeinen einen kurzen und übersichtlichen Quellcode. In ihm sind die zugehörigen Policies und das `IPropertySource`-Element, welches den PropertyView mit den Attributen und Werten bestückt, enthalten.

3.3.3 CTC-Graph Editor verstehen und richtig verwenden

Unter Zuhilfenahme eines Screenshots des CTC-Graph Editors werden die Verwendung sowie wissenswerte und wesentlichen Details des Editors und den zugrunde liegendem GEF-Baustein erläutert.

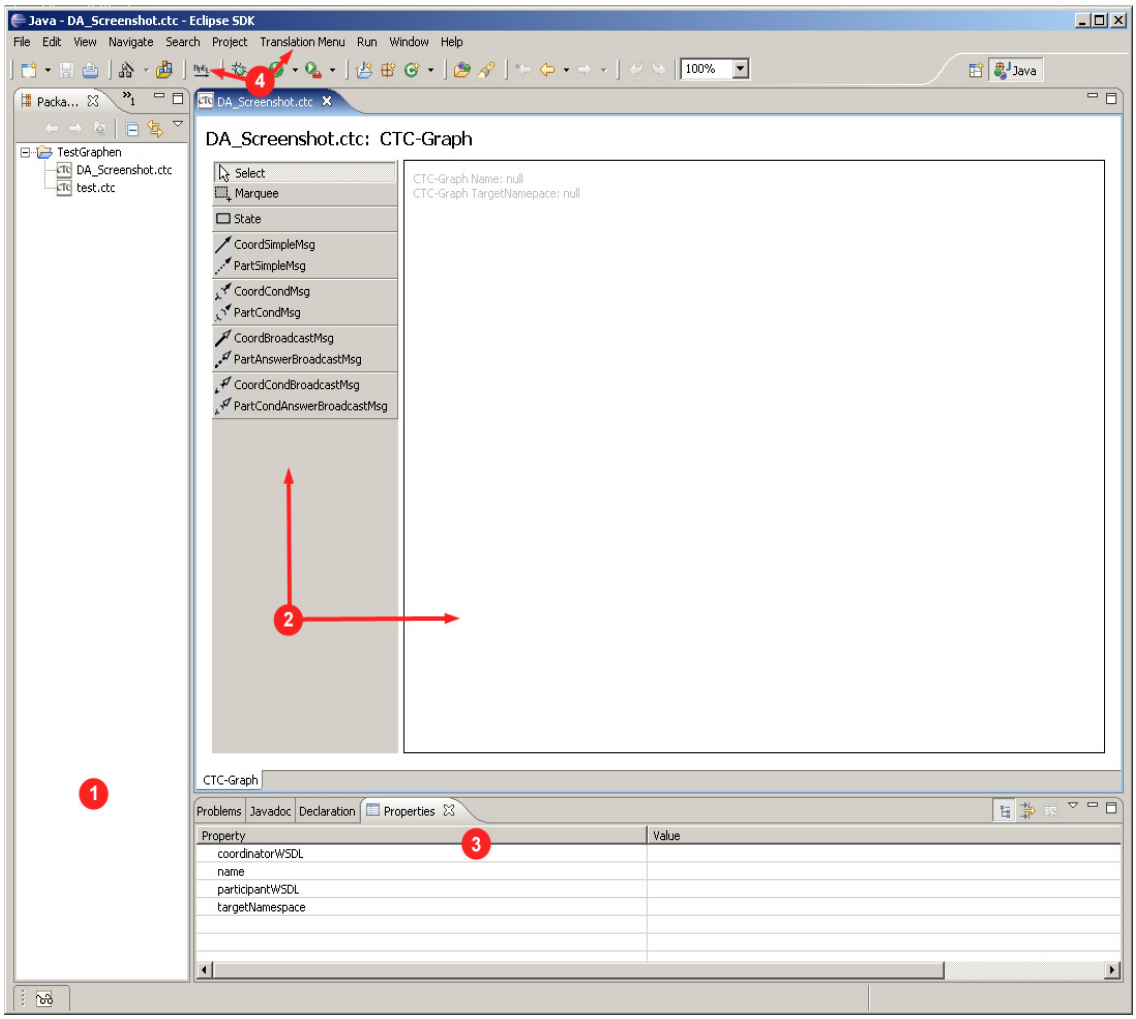


Abb. 3-3: Screenshot des CTC-Graph Editor Plug-ins

Die obige Abbildung zeigt einen Screenshot des CTC-Graph Editors als Plug-in in der Eclipse-Umgebung. Folgende nummerierte Teilbereiche, sind für die korrekte Bedienung des Editors wichtig:

1. PackageExplorer-View
2. CTC-Graph Editor und Werkzeug-Palette
3. Property-View
4. TranslationMenu und „BPEL“-Button in der Shortcut-Leiste

3.3.3.1 Das PackageExplorer-View

Das PackageExplorer-Fenster befindet sich standardmäßig in der Eclipse-Umgebung links außen. Es beinhaltet einen Projekt-Ordner, in dem die Dateien gespeichert werden können.

Wird ein CTC-Graph modelliert, muss eine Datei mit der Endung „.ctc“ bspw. über das Eclipse-Menü File → New → File erzeugt werden. Durch das installierte CTC-Graph Editor- Plug-in, öffnet sich automatisch - nach dem Erstellen der „`.ctc`“-Datei - der Editor und seine Werkzeugpalette mit der neuen Datei.

Gespeicherte Dateien sind ebenfalls im PackageExplorer-Fenster in den entsprechenden Projekt-Ordnern abgelegt. Sie lassen sich jederzeit wieder öffnen und bearbeiten.

3.3.3.2 Der CTC-Graph Editor

Der Editor nimmt standardmäßig den größten Teil der Eclipse-Entwicklungsumgebung ein. Sein Aufbau ist leicht verständlich. In der Kopfzeile des Editors wird der Name der z. Zt. geöffnete Datei angezeigt gefolgt von „: CTC-Graph“.

Auf der linken Seite des Editors liegt die Werkzeugpalette. Mit ihrer Hilfe können alle Werkzeuge zur Modellierung und Bearbeitung von CTC-Graphen selektiert werden. Auf der rechten Seite stellt der schwarz umrandete weiße Hintergrund den leeren CTC-Graphen dar. Die unter Zuhilfenahme der Palette generierten graphischen Elemente sind im Graphen eingebettet und werden intern als Kinder des Graphen behandelt.
Die graue Schrift am oberen rechten Rand des CTC-Graphen stellt eine Hilfe dar und bildet die Attribute „`name`“ und „`targetNamespace`“ des Graphen ab. Sie werden für die BPEL Generierung benötigt.

3.3.3.3 Die Werkzeugpalette

Die Palette enthält allgemeine GEF-Werzeuge (z.B. Select, Marquee) und CTC-Graph-spezifische Werkzeuge (z.B. State, CoordConditionalMsg).

Die GEF-Tools „Select“ und „Marquee“ dienen der Selektion ausgewählter graphischer Elemente und der respektiven Selektion aller graphischen Elemente in einem bestimmten Bereich. Mit diesen Tools können einzelne Elemente zur Bearbeitung, Löschung oder Verschiebung ausgewählt werden.

Unterhalb der hier vorgestellten GEF-Werkzeuge sind weitere Werkzeuge positioniert, die CTC-Graph-Objekte erzeugen können.

Diese Werkzeuge bilden die in CTC-Graph EMF-Klassen erläuterten Klassen CTCNode, CoordSimpleEdge, PartSimpleEdge, CoordConditionalEdge, PartConditionalEdge, CoordSendBroadcastEdge, PartAnswerBroadcastEdge, CoordConditionalSendBroadcastEdge und PartConditionalAnswerBroadcastEdge ab. Restriktionen bzgl. der Erzeugung und der Bedeutung einzelner Attribute, können in Kapitel 3.2 nachgelesen werden. Selbstverständlich sind alle Werte der besprochenen Attribute frei wählbar. Eine syntaktische oder semantische Verifikation der Eingaben ist in dieser Version des Editors nicht vorgesehen. Der Benutzer ist für die Korrektheit und den Gehalt seiner Eingabe selbst verantwortlich.

Wie bereits den Namen der Werkzeuge zu entnehmen ist, haben sie folgende Funktionen:

- **„State“** dient der Erzeugung von Zuständen im CTC-Graphen (CTCNode-Klasse). Soll ein Zustand erzeugt werden, muss die entsprechende Schaltfläche auf der Werkzeugpalette ausgewählt werden. Anschließend wird auf einer beliebigen Stelle im CTC-Graphen geklickt. Diese Stelle dient als Koordinate. Auf ihr wird das Element nach der Erzeugung positioniert. Bevor ein Zustand erzeugt wird, darf der Benutzer einen Namen für den Zustand eingeben. Um die korrekte Funktionalität des BPEL-Files zu gewährleisten, müssen die Zustände einen eindeutigen Namen erhalten. Wird kein Name vom Benutzer eingegeben, so wird vom System automatisch ein Name für den neuen Zustand vergeben.
- **„CoordSimpleMsg“** dient der Erzeugung einer einfachen Koordinator-Nachricht (CoordSimpleEdge-Klasse). Soll die Kante gezeichnet werden, so ist die Schaltfläche anzuklicken und anschließend der gewünschte Quell- und Ziel-Zustand zu selektieren.
- **„PartSimpleMsg“** dient der Erzeugung einer einfachen Teilnehmer-Nachricht (PartSimpleEdge-Klasse). Soll die Kante gezeichnet werden, so ist die Schaltfläche anzuklicken und anschließend der gewünschte Quell- und Ziel-Zustand zu selektieren.
- **„CoordConditionalMsg“** dient der Erzeugung einer bedingungsgeknüpften Koordinator-Nachricht (CoordConditionalEdge-Klasse). Soll die Kante gezeichnet werden, so ist die Schaltfläche anzuklicken und anschließend der gewünschte Quell- und Ziel-Zustand zu selektieren.
- **„PartConditionalMsg“** dient der Erzeugung einer bedingungsgeknüpften Teilnehmer-Nachricht (PartConditionalEdge-Klasse). Soll die Kante gezeichnet werden, so ist die Schaltfläche anzuklicken und anschließend der gewünschte Quell- und Ziel-Zustand zu selektieren.

- **„CoordBoradcastMsg“** dient der Erzeugung einer einfachen Koordinator-Broadcast-Nachricht (CoordSendBroadcastEdge-Klasse). Soll die Kante gezeichnet werden, so ist die Schaltfläche anzuklicken und anschließend der gewünschte Quell- und Ziel-Zustand zu selektieren.
- **„PartAnswerBroadcastMsg“** dient der Erzeugung einer einfachen Teilnehmer -Antwortnachricht auf einem Broadcast des Koordinators (PartAnswerBroadcastEdge -Klasse). Soll die Kante gezeichnet werden, so ist die Schaltfläche anzuklicken und anschließend der gewünschte Quell- und Ziel-Zustand zu selektieren.
- **„CoordCondBroadcastMsg“** dient der Erzeugung einer bedingungsgeknüpften Koordinator-Broadcast-Nachricht (CoordConditionalSendBroadcastEdge -Klasse). Soll die Kante gezeichnet werden, so ist die Schaltfläche anzuklicken und anschließend der gewünschte Quell- und Ziel-Zustand zu selektieren.
- **„PartCondAnswerBroadcastMsg“** dient der Erzeugung einer bedingungsgeknüpften Teilnehmer-Antwortnachricht auf einen Broadcast des Koordinators (PartConditionalAnswerBroadcastEdge-Klasse). Soll die Kante gezeichnet werden, so ist die Schaltfläche anzuklicken und anschließend der gewünschte Quell- und Ziel-Zustand zu selektieren.

Alle soeben beschriebenen Elemente können verschoben oder gelöscht werden. Ein „Resize“ der Zustände ist jedoch nicht möglich. Sie besitzen immer die ideale Größe, die proportional zur Länge ihrer Namen ist. Die erzeugten Nachrichten enrhalten vom System einen eindeutigen Namen. Er kann, wie alle anderen Attribut-Werte, vom Benutzer im Property-View (s.u.) geändert werden.
Kanten können nicht direkt manipuliert werden, weil sie sich über ihren Quell- und Zielknoten definieren und so ihre Länge enthalten ist. Die nachträgliche Änderung von Quell- oder Zielknoten ist nicht vorgesehen.

3.3.3.4 Das Property-View

Das Property-View ist, neben der Werkzeugpalette, eines der wichtigsten Arbeitswerkzeuge im CTC-Graph Editor. Das Fenster liegt unterhalb des CTC-Graph Editor Fensters und enthält Attribute und Werte des aktuell ausgewählten Elementes des CTC-Graphen.

Im Property-View können die Werte verschiedener Attribute des selektierten CTC-Graph Objekts neu festgesetzt werden. Attribut-Werte, die vom Benutzer frei gelassen wurden, werden als leeres String („“) in die BPEL-Datei übertragen. Es empfiehlt sich möglichst genaue Werte für diese zu verwenden, damit die automatisch generierten BPEL-Dateien (Koordinator- und Teilnehmer-BPEL-Dateien) so vollständig wie möglich werden und sich auch problemlos mit beliebigen BPEL-Editoren bzw. BPEL–Designern öffnen lassen. Bspw. erfolgt der Import der mittels CTC-Graph Editor generierten BPEL-Datei in ActiveWebflow™ Professional Designer Version 1.5.0 nur dann problemlos, wenn der Attribut-Wert der zu verwendeten WSDL-Datei nicht leer ist. Ansonsten sind fehlende Werte nachträglich in die Datei mittels eines Editors einzufügen.

Wird das PropertyView-Fenster nicht von Anfang an gezeigt oder wird es versehentlich vom Anwender geschlossen, so kann es jederzeit über das Eclipse-Menü Window → ShowView → Properties wieder eingeblendet werden.

3.3.3.5 Das CTC-Graph TranslationMenu und der „BPEL“-Button in der Shortcut-Leiste

Auf der Eclipse-Menüleiste befindet sich das Translation Menu. Über dieses Menü (Translation Menu → CTC to executable BPEL) kann die Übersetzung des CTC-Graphen in BPEL gestartet werden.
Ein Starten der Übersetzung ist durch das Benutzen des „BPEL“-Buttons () in der Eclipse-Shortcut-Leiste möglich.

Wurde die Übersetzung des CTC-Graphen in der entsprechenden Coordinator- und Participant-BPEL-Datei gestartet, wird der Graph auf einen eindeutigen und einzigen Startknoten (gem. Formale Beschreibung der CTC-Graphen) überprüft. Ist diese Vorraussetzung nicht erfüllt, findet keine Übersetzung statt.

Ist nur ein Startzustand vorhanden, wird dem Benutzer die Möglichkeit gegeben den Pfad unter dem die BPEL-Dateien gespeichert werden sollen und den Namen der zu erzeugenden Dateien (default Wert: `generatedCoordinatorBPELFile.bpel` und `generatedParticipantBPELFile.bpel`) anzugeben. Wird keiner dieser Schritte vom Benutzer abgebrochen, startet die Übersetzung.

Wurden mehrere „`participantWSDL`“ Dateien (durch einen „`;`“ getrennt) vom Benutzer in das entsprechende Feld im Property-View eingetragen, so generiert der Übersetzer mehrere Participant-BPEL-Dateien. Die erzeugten Dateien werden automatisch durchnummeriert. Die Nummerierung fängt bei 1 an (default Ausgabe-Wert: `generatedParticipantBPELFile_1.bpel`, `generatedParticipantBPELFile_2.bpel`, etc.) und endet bei n (wobei n die Anzahl der angegebenen Participant-WSDL-Dateien ist).

Nach Beendigung der Aufgabe wird der Anwender darüber informiert, dass die Dateien erfolgreich erzeugt wurden.

Die BPEL-Dateien sind nun bereit und können mit Hilfe eines BPEL-Editors bzw. BPEL-Designers weiter bearbeitet werden.

3.3.4 Die CTCtoBPEL-Klasse

Das Translation Menu und die CTCtoBPEL-Klasse stellen die Verbindung zwischen den beiden Hauptaufgaben des CTC-Graph Editors dar. Sie sind der Ausgangspunkt für die Übersetzung des modellierten CTC-Graphen in die entsprechenden Koordinator- und Teilnehmer-BPEL-Dateien.

Die CTCtoBPEL-Klasse enthält alle nötigen Aktionen und wesentlichen Vorraussetzungen damit die eigentliche Übersetzung des Graphen in BPEL gestartet werden kann.

Mit den folgenden drei Aktionen wird sichergestellt, dass eine korrekte Übersetzung des CTC-Graphen in BPEL erfolgt:

1. Speicherung der aktuellen Fassung des modellierten CTC-Graphen: ist notwendig, damit alle Änderungen am aktuellen Graphen festgeschrieben werden.
2. Abfrage des Speicherorts und –name der zu generierenden BPEL-Dateien: stellt sicher, dass der Benutzer den gewünschten Zielort und –name der Dateien angeben kann. Diese Daten werden für die Übersetzung benötigt.
3. Überprüfung des Graphen auf einen eindeutigen Startzustand: stellt sicher, dass der modellierte Graph (spätestens während der Übersetzung nach BPEL) nicht gegen die Formale Beschreibung von CTC-Graphen verstößt. Eine frühere Überprüfung dieser Konformität des CTC-Graphen mit der Formalen Beschreibung ist denkbar, aber nicht sinnvoll, weil so der Benutzer die Möglichkeit hat auch unfertige Graphen jederzeit für eine spätere Bearbeitung zu speichern bzw. zu übersetzen.

Bei Abbruch einer der o.g. Aktionen durch den Benutzer (z.B. Speicherung wird nicht gewünscht oder der Name einer zu generierenden Dateien wird nicht angegeben) oder durch das Programm (Graph besitzt mehr als einen Startzustand), meldet das Programm einen Fehler und bricht den Vorbereitungsprozess und damit auch die Übersetzung automatisch ab.

Wurden alle Aktionen erfolgreich durchgeführt, so sind alle nötigen Vorraussetzungen für die Übersetzung des Graphen in BPEL erfüllt. Die Übersetzung wird daraufhin gestartet.

3.4 Der BPEL-Generator

Aufbau und Funktionsweise des BPEL-Generators sind wie folgt:

Im Wesentlichen besteht der BPEL-Generator aus zwei Hauptbestandteilen (im „ctceditor.util.bpel“ – Package zu finden):

1. BPELWriter-Klasse: Sie führt die Logik zur Erzeugung der Koordinator- und Teilnehmer-BPEL-Dateien, basierend auf den modellierten CTC-Graphen.
2. XMLWriter-Klasse: Sie enthält hilfreiche Methoden, welche die Erzeugung der Dateien unterstützten.

3.4.1 BPELWriter-Klasse

Als Modul zur Übersetzung des Graphen in BPEL-Code ist die Klasse BPELWriter von zentraler Bedeutung.
Für die Übersetzung wird eine Instanz dieser Klasse mit folgenden Parametern erzeugt:

- das aktuelle „IWorkbenchWindow“ (entspricht der Eclipse Workbench und wird für Fehlerfeedback und Erfolgsmeldung an den Benutzer benötigt)
- den aktuellen Graphen in Form eines CTCGraph-Objektes
- ein String, welcher das Zielverzeichnis für die zu erzeugenden Dateien enthält.
- ein String-Array (Java-Notation: String[]), welches den aufgesplitteten „participantWSDL” Attribut-Wert enthält („participantWSDL”-String wird an „;” getrennt und in einem String[] eingetragen).

In diesem Konstruktor wird jeweils eine Instanz des XMLWriters für den Coordinator- und eine oder mehrere (je nach Anzahl der Teilnehmer einer Transaktion) für den Participant-BPEL-Code erzeugt und zwischengespeichert.
Die eigentliche Code-Erzeugung wird mit dem Aufruf der CoordBPELWriter- und der PartBPELWriter-Klasse gestartet. Beide Klassen besitzen die gleichen Abarbeitungsschritte, die sich lediglich durch spezielle Coordinator- bzw. Participant spezifische BPEL-Ausführungsschritte unterscheiden.
Nach der Ausgabe des Wurzelelementes „process“ mit den notwendigen Namespace-Shortcut-Deklarationen (BPEL-Schema und dazugehörige WSDL-Datei) sowie dem vorgesehenen „targetNamespace“ wird in drei Schritten der Inhalt vervollständigt:

1. Elemente „partnerLink“
2. Elemente „variable“
3. Elemente „scope“

Im Anschluss werden das Wurzelelement und die XMLWriter-Instanzen geschlossen, d.h. die wohlgeformten XML-Dateien werden abgeschlossen.

Die Ausgabe der Informationen in die BPEL-Dateien für Coordinator und Participant läuft dabei bis auf wenige Ausnahmen parallel ab.

3.4.1.1 Elemente "partnerLink"

Für den Code der Partner-Links werden alle Kanten des Graphen durchlaufen und dem Element „partnerLinks“ für jede Kante ein Kind-Element „partnerLink“ mit den jeweiligen Attributen „name“, „partnerLinkType“, „myRole“ und „partnerRole“ erstellt.

3.4.1.2 Elemente "variable"

Als Kind-Elemente von „variables“ werden die Elemente „variable“ erzeugt.
Hierfür wird die Liste aller Kanten durchlaufen und für jede Kante eine „inputVariable“ ggf. auch eine „outputVariable“ (ihr Wert „true“ ist) erzeugt. Dies ist abhängig davon, ob die jeweiligen Werte an der Kante gesetzt sind bzw. überhaupt möglich sind. Wurde der Wert des Attributs „name“ einer Variablen nicht angegeben, so werden Namen automatisch erzeugt. Jedem „variable“-Element wird ein Attribut namens „messageType“ zugeordnet.

3.4.1.3 Elemente "scope"

Schließlich werden alle Knoten mittels Breitensuche durchlaufen.
Beginnend mit dem, unter Zuhilfenahme der Methode „GraphUtil.findStartNode(CTCGraph)“, ermittelten Startknoten werden alle Knoten des Graphen rekursiv abgearbeitet.
Für jeden Ziel-Knoten einer vom aktuellen Knoten ausgehenden Kante wird eine „scope“- Element erzeugt. Hierbei unterscheidet sich der erzeugte BPEL-Code zwischen Coordinator- und Participant-Code wie folgt:

1. Für jede eingehende Coordinator-Kante wird im Participant-Code und für jede eingehende Participant-Kante im Coordinator-Code ein Element „receive“ am Anfang der „scope“-Aktivität erzeugt (vgl. Kapitel 2.7).

2. Darauf folgt ein XML-Kommentar mit der Aufforderung zur Ergänzung von „faultHandlers“, „correlationSets“ etc. bei der späteren Nachbearbeitung des BPEL-Codes.

3. Anschließend werden die ausgehenden Kanten des aktuellen Knotens durchlaufen: Im Participant-Code wird für jede ausgehende Participant-Kante und im Coordinator-Code für jede ausgehende Coordinator-Code ein Element „invoke“ erzeugt.

Das „scope“-Element wird wieder geschlossen, um die Wohlgeformtheit der Dateien sicherzustellen.

3.4.2 XMLWriter-Klasse

Die Hilfsklasse XMLWriter wird von der BPELWriter-Klasse verwendet, um auf elegante Weise eine wohlgeformte XML-Datei zu erzeugen. Bekannterweise basiert BPEL auf XML und muss deshalb den entsprechenden Vorschriften (u.a. korrekte Verschachtelung der Elemente etc.) folgen.

Folgende Dienste werden von der XMLWriter-Klasse für die BPEL-Übersetzung des CTC-Graphen erbracht:

- Ausgabe der XML-Deklaration
- korrekter Einzug der Elemente entsprechend ihrer Verschachtelungstiefe
- wohlgeformtes Öffnen und Schließen von XML-Tags
- korrektes Erzeugen von XML-Kommentaren
- Ersetzen der XML-Sonderzeichen in Attributswerte und Kommentare durch ihre entsprechenden Entities (d.h. "<" wird ersetzt durch "<", ">" wird ersetzt durch ">" usw.)

Der Implementierungsaufwand für diese Klasse wurde möglichst gering gehalten, daher muss bei ihrer Nutzung folgende Reihenfolge zwingend eingehalten werden:

1. Erzeugen der XMLWriter-Instanz mit Übergabe des Dateinamens.
2. Beliebig häufiges Erzeugen eines Kommentars mit der Methode „`writeComment`“.
3. `startelement` aufrufen, um ein neues Element zu öffnen.
4. Beliebig häufiger Aufruf von „`addAttribute`“ mit Schlüssel und Wert um ein Attribut des gerade erstellten Elements hinzuzufügen. (Hierbei wird jedoch nicht sichergestellt, dass die Schlüssel eindeutig sind.)
5. Beliebig häufiges Erzeugen eines Kommentars mit der Methode „`writeComment`“.
6. Aufruf von „`endelement`“, um das zuletzt geöffnete Element zu schließen.
7. Beliebig häufiges Erzeugen eines Kommentars mit der Methode „`writeComment`“.
8. Schließen der Datei mittels der „`close`“-Methode.

4 Zusammenfassung und Ausblick

4.1 Zusammenfassung

Im Rahmen der Diplomarbeit wurde ein Graphentyp für die Modellierung von transaktionalen Kommunikationsabläufen im Web Services Umfeld, der sog. koordinierte transaktionale Kommunikations-Graph (**C**oordinated **T**ransactional **C**ommunication-Graph kurz **CTC**-Graph), definiert. Dieser Graph wurde speziell darauf ausgerichtet, dass er alle nötigen Informationen enthält, um eine automatische und fast vollständigen Übersetzung in BPEL zu unterstützen.

Der Entwurf und die Konzeption der im Rahmen dieser Arbeit entworfenen Diagrammart wurden mit Hilfe und Erkenntnissen bereits existierender graphentheoretischer Ansätze ausgearbeitet und auf die Bedürfnisse des Web Service Umfeldes, insbesondere auf Web Services Coordination, Web Services Transactions und BPEL, zugeschnitten.

Um die Arbeiten bzgl. der Modellierung von CTC-Graphen zu erleichtern, wurde für diese Arbeit ein Eclipse-Plug-in konzipiert und implementiert.

Das Eclipse-Plug-in (sog. CTC-Graph Editor) übernimmt auch die angesprochene automatisierte Generierung von BPEL-Code. Der entstandene Code spiegelt die Grundzüge der Kommunikation und der Verhältnisse des modellierten Graphs wieder. Der Code kann anschließend – mittels bewährter BPEL-Editoren – bearbeitet werden und erhält somit die erwünschte Programmlogik.

4.2 Ausblick

In dieser Arbeit wurde bereits an verschiedenen Stellen darauf hingewiesen, dass die vorliegenden Erkenntnisse und Implementierung eine Vielzahl von Möglichkeiten für weiterführende Arbeiten bietet. An dieser Stelle findet der interessierte Leser eine Auflistung der weiterführenden Aufgaben. Die Erweiterung des CTC-Graph Modells und Editors können in drei Bereiche unterteilt werden:

1. Weiterführende Aufgaben im Bereich BPEL
2. Weiterführende Aufgaben im Bereich CTC-Graphen und BPEL-Übersetzung
3. Weiterführende Aufgaben im Bereich CTC-Graph Editor

4.2.1 Weiterführende Aufgaben im Bereich BPEL

Obwohl z. Zt. nur eine Entwurfsversion der BPEL 2.0 Spezifikation [2] vorliegt, werden die Unterschiede zu der Vorgängerversion deutlich. Die Änderungen und

Unterschiede werden an dieser Stelle nicht vollständig aufgezählt und erklärt, hierfür wird der Leser auf „Web Services Business Process Execution Language Version 2.0 Comittee Draft, 21th December 2005“ [2] verwiesen.

Jedoch ist es erwähnenswert, dass in BPEL 2.0 neue Konstrukte eingeführt werden, die für CTC-Graphen, bzw. deren Übersetzung in BPEL-Code, künftig hilfreich sein könnten. Mit der neuen BPEL-Version könnte der BPEL-Code der generierten Dateien deutlich verkürzt und optimiert werden. Es folgt eine Auflistung und eine kurze Erläuterung ausgewählter BPEL 2.0 Konstrukte, welche eine Code-Optimierung hervorbringen können.

Ausgewählte Konstrukte[18]:

- „forEach“-Iterator
- „if / then / else“-Verzweigungen

„forEach“-Iterator

```
<forEach counterName=“ncname“ parallel=“yes | no“
   standard-attributes standard-elements>
<iterator>
       <startCounterValue expressionLanguage="anyURI">
          ...
       </startCounterValue>
       <finalCounterValue expressionLanguage="anyURI">
          ...
       </finalCounterValue>
    </iterator>
    <completionCondition>
       <branches expressionLanguage="URI"?
                 countCompletedBranchesOnly="yes|no"?>
           an-integer-expression
       </branches>
    </completionCondition>?
    scope-activity
</forEach>
```

Wie der Name bereits verdeutlicht, handelt es sich bei diesem Konstrukt um einen Iterator, der eine enthaltene Aktivität genau N+1 Mal ausführt, wobei N = „finalCounterValue“- „startCounterValue“ ist.

Das Attribut „parallel=“yes | no““ weist darauf hin, dass die „forEach“ enthaltenen Aktionen auch parallel ausgeführt werden können.

Mögliches Anwendungsgebiet: Für das parallele Versenden von Nachrichten kann auf diese Weise die Teilnehmerliste einfacher abgearbeitet werden (vgl. in Kapitel 3).

[18] Auszug aus dem WorkingDraft von BPEL 2.0 [2]

„if / then / else“-Verzweigungen

```
<if standard-attributes standard-elements>
   <condition expressionLanguage="anyURI"?>
      ... boolean-expression ...
   </condition>
   <then>
     activity
   </then>
   <elseif>*
      <condition expressionLanguage="anyURI"?>
         ... boolean-expression ...
      </condition>
   </elseif>
   <else>?
      activity
   </else>
</if>
```

Bezüglich der Abarbeitung bestimmter Aktionen bzw. dem Senden von Nachrichten kann dieses Konstrukt hilfreich sein. Eine Verkürzung des generierten BPEL-Codes ist hierdurch vorstellbar (vgl. Kapitel 2.6.2).

Mögliches Anwendungsgebiet: Das Konstrukt könnte an Stelle von „flow“ Konstrukten mit Link-Conditions oder auch statt kurzen „switch“ Konstrukten (mit wenigen Auswahlmöglichkeiten) verwendet werden.

4.2.2 Weiterführende Aufgaben im Bereich CTC-Graphen und BPEL-Übersetzung

In Kapitel 2.6.3 „Formale Beschreibung der CTC-Graphen” wurden Einschränkungen bzgl. der Modellierung von CTC-Graphen gemacht. Diese Einschränkungen bieten Ansatzpunkte für Erweiterungen. Werden Verfeinerungen und/oder andere Übersetzungsstrategien der CTC-Graphen in BPEL ausgearbeitet, ist es durchaus vorstellbar, dass so auch die Möglichkeit der Modellierung von Zyklen in CTC-Graphen eingefügt wird.

4.2.3 Weiterführende Aufgaben im Bereich CTC-Graph Editor

Der CTC-Graph Editor stellt einen unkomplizierten Weg dar CTC-Graphen zu modellieren, die anschließend in BPEL-Code übersetzt werden können. Das Plug-in ist funktional aufgebaut. Es bietet wenig Komfort bei der Eingabe und keinerlei Funktionalitäten bei der Bearbeitung des generierten BPEL-Codes.

Die Verfeinerung des CTC-Graph Editors kann in zwei Aspekte unterteilt werden:

1. Erweiterung der Funktionalitäten
2. Optische Erweiterungen

4.2.3.1 Erweiterungen der Funktionalitäten

Eingebetteter Code-Editor

Eine Erweiterung des Editors in Bezug auf die Bearbeitung der Programmlogik des zu generierenden BPEL-Codes ist vorstellbar. Einerseits könnte die Eingabe der fehlenden Programmlogik bereits vor der Erzeugung des BPEL-Codes z.B. in einer Art Wizzard unterstützt werden. Andererseits könnte das Plug-in selbst eine Möglichkeit bieten den soeben generierten BPEL-Codes zu bearbeiten.

Relocate Edges

Bei der Anwendung des CTC-Graph Editors kann bemerkt werden, dass einige Funktionalitäten ergänzt werden könnten. Ein Beispiel hierfür ist ein „relocate“ von Kanten. Es ist dem Benutzer bisher nicht gestattet die Quelle oder das Ziel einer Kante nach ihrer Erzeugung zu ändern.

Stärkere Bindung zwischen CTC-Graph und benötigten WSDL-Dateien

Weitere realisierbare Erweiterungen für den CTC-Graph Editor könnten Funktionalitäten sein, die eine stärkere Bindung zwischen BPEL und den dazugehörigen WSDL-Dateien innehaben. Als Beispiel kann hier eine Funktionalität aufgeführt werden, die bestimmte Daten aus vorhandenen WSDL-Dateien ausliest (z.B. „`partnerLinkType`“, „`operations`“, `portType` oder „`messageTypes`“ Daten aus den entsprechenden coordinator/participant WSDL-Dateien) und diese in bestimmte Felder oder Auswahllisten des Property-Views listet. Der Benutzer könnte so bequem die gewünschten Werte aus einer Auswahlliste selektieren. So wäre es gewährleistet, dass z.B. kein Tippfehler bei der Eingabe von Namen Auswirkungen auf die BPEL-Datei und ihre Ausführung hat.

Synchrone Kommunikation

In Kapitel 2.6.3.1 wurde eine Einschränkung in der Kommunikation gemacht. Wie bereits angesprochen wurde die synchrone Kommunikation aus Gründen des Programmieraufwandes nicht untersucht. Obwohl ein Workaround mit Hilfe von „`correlationSets`“ und asynchronen Nachrichten („`invoke`“ / „`receive`“) das gewünschte Verhalten hervorruft, ist eine Erweiterung des Editors und somit des CTC-Graphen im Hinblick auf synchrone Kommunikation denkbar.

Verifikation der Freitext-Eingaben

Des Weiteren ist eine bessere Verifikation der Freitext-Eingaben z.B. durch gezieltes Unterbinden bestimmter Sonderzeichen, vor allem eine Verifikation in Bezug auf die oben angesprochenen coordinator/participant WSDL-Dateien, denkbar. Dies könnte schneller zu lauffähigen BPEL-Dateien führen. Eine Zeitverkürzung bei der Erzeugung des Grundgerüstes der BPEL-Dateien erscheint möglich, durch den bereits angesprochenen eingebetteten Code-Editor (s.o.). So könnte der CTC-Graph Editor zu einem vollständigeren Tool ausgebaut werden.

Translation Menu Erweiterungen

Für Erweiterungen des CTC-Graph Editors besteht die Möglichkeit der Ergänzung weiterer Funktionalitäten im Translation Menu. Beispielsweise die Übersetzung des Graphen unter der Verwendung von Sprachkonstrukten unterschiedlicher BPEL-Spezifikationen. Solche Erweiterungen können den CTC-Graph Editor zu einem umfassenderen Tool für die Generierung von BPEL-Dateien werden lassen.

4.2.3.2 Optische Erweiterungen

Nicht nur die Verfeinerung und Optimierung der Funktionalitäten des Editors soll an dieser Stelle erwähnt werden, sondern auch im Graphical User Interface (GUI) könnten Verbesserungen vorgenommen werden.

Der Editor leitet sich von der GEF-Klasse MultiPageEditor ab und kann dementsprechend mehrere Seiten bzw. Ansichten unterstützen. Eine sinnvolle Erweiterung des Editors ist der Einbau von zwei zusätzlichen Ansichten. In ihnen könnten die generierten Koordinator- und Teilnehmer-BPEL-Dateien angezeigt und evtl. auch bearbeitet werden.

Durch eine effizientere Anordnung von einzelnen GUI-Elementen oder auch durch das Hinzufügen von Tools in der Editor-Palette könnte eine übersichtlichere und verständlichere Benutzeroberfläche erzeugt werden. Eine Verkürzung der Arbeitszeit durch die verbesserte Übersichtlichkeit des CTC-Graph Editor wäre denkbar.

Die oben genannten Verfeinerungsansätze würden nicht nur eine schnellere, sondern auch eine effektivere und genauere Übersetzung des CTC-Graphen in BPEL unterstützen. Es wäre so möglich den CTC-Graph Editor zu einem umfassenderen Tool weiterzuentwickeln.

Danksagung

An dieser Stelle möchte ich die Personen nennen, die mich mit Rat und Tat bei der Erstellung der vorliegenden Arbeit unterstützt haben.

Mein Dank gilt Nicole Fontanive sowie Carol Rössler und Dr. Andreas Rössler für die Durchsicht des Manuskriptes und die zahlreichen hilfreichen Vorschläge zur sprachlichen Ausgestaltung meiner Diplomarbeit.

Ebenfalls danke ich meinem guten Freund, Frank-Ralph Reiser, der mir zum schnellen Einstieg und Verständnis in Eclipse verholfen hat.

Vielen Dank auch an Prof. Dr. Frank Leymann für die interessante Aufgabenstellung und die freundliche Unterstützung.

Besonderer Dank geht an meinem Betreuer Stefan Pottinger ohne dessen tatkräftige Hilfe und nette Unterstützung diese Arbeit nicht möglich gewesen wäre.

Herzlich bedanken möchte ich mich auch bei Katharina Rössler, den „Towandas reloaded“ und - last but not least - meinen Eltern für die ausgezeichnete „Rückendeckung“.

Abbildungsverzeichnis

Abkürzungsverzeichnis

Abb.	Abbildung
bspw.	beispielsweise
bzgl.	bezüglich
bzw.	beziehungsweise
d.h.	das heißt
etc.	et cetera
evtl.	eventuell
gem.	gemäß
gdw.	genau dann wenn
i.V.m.	in Verbindung mit
o.g.	oben genannte
sog.	so genannt
s.o.	siehe oben
s.u.	siehe unten
Tab.	Tabelle
u.a.	unter anderem
vgl.	vergleiche
z.B.	zum Beispiel
z. Zt.	zur Zeit

Literaturverzeichnis

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, C. K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, S. Weerawarana: *Business Process Execution Language for Web Services Version 1.1. 2003*, online, accessed 23. May 2006: http://www-128.ibm.com/developerworks/library/specification/ws-bpel/

[2] A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. ; Liu, V. Mehta, S. Thatte, P. Yendluri, A.Yiu, A. Alves: *Web Services Business Process Execution Language Version 2.0 Committee Draft, 21th December 2005*, online, accessed 23. May 2006: http://www.oasis-open.org/committees/download.php/16024/wsbpel-specification

[3] C. Baray: *The Model-View-Controller (MVC) Design Pattern*, online, accessed 23. May 2006: http://www.cs.indiana.edu/~cbaray/projects/mvc.html

[4] D. K. Barry: Web Services and Service-Oriented Architectures: The Savvy Manager's Guide. Morgan Kaufmann Publishers 2003

[5] B. Bokowski, F. Gerhardt: *Maßgeschneiderte graphische Editoren mit dem Graphical Editing Framework*, eclipse Magazin, Ausgabe 2.05 , Software & Support Verlag GmbH

[6] L. F. Cabrera, G. Coperland, M. Feingold, R. W. Freund, T. ; Freund, J. Johnson, S. Joyce, C. Kaler, J. Klein, D. Langworhty, M. Little, A. Nadalin, E. Newcomer, D. Orchard, I. Robinson, T. Storey, S. Thatte: *Web Services Atomic Transaction (WS – Atomic Transaction) Version 1.0 . 2005*, online, accessed 23. May 2006: http://www-128.ibm.com/developerworks/webservices/library/specification/ws-tx/#atom

[7] L. F. Cabrera, G. Coperland, M. Feingold, R. W. Freund, T. Freund, J. Johnson, S. Joyce, C. Kaler, J. Klein, D. Langworhty, M. Little, T. Nadalin, E. Newcomer, D. Orchard, I. Robinson, J. Shewchuk, T. Storey: *Web Services Coordination (WS – Coordination) Version 1.0 . 2005*, online, accessed 23. May 2006: http://www-128.ibm.com/developerworks/webservices/library/specification/ws-tx/#coor

[8] L. F. Cabrera, G. Coperland, M. Feingold, R. W. Freund, T. Freund, S. Joyce, J. Klein, D. Langworhty, M. Little, F. Leymann, E. Newcomer, D. Orchard, I. Robinson, T. Storey, S. Thatte: *Web Services Business Activity (WS – Business Activity) Version 1.0 . 2005*, online, accessed 23. May 2006: http://www-128.ibm.com/developerworks/webservices/library/specification/ws-tx/#ba

[9] Daum, B: Java-Entwicklung mit Eclipse 3: Anwendungen, Plug-ins, Rich Clients. dpunkt.verlag 2005

[10] Eclipse.org, online, accessed 23. May 2006: http://www.eclipse.org/

[11] Eclipse Corner Article. *Using GEF with EMF*, online, accessed 23. May 2006: http://www.eclipse.org/articles/Article-GEF-EMF/gef-emf.html

[12] Eclipse Documents. Eclipse Modelling Framework (EMF) Overview, online, accessed 23. May 2006: http://www.eclipse.org/emf/

[13] Eclipse Documents. Graphical Editing Framework (GEF) Overview, online, accessed 23. May 2006: http://www.eclipse.org/gef/

[14] K. Ehrig, C. Ermel, G. Taenzer: *Erstellung eines graphischen Editor-Plug-Ins mit EMF und GEF*, online, accessed 23. May 2006: http://tfs.cs.tu-berlin.de/petrieditor/

[15] R. Elmasri, S. B. Navathte: *Grundlagen von Datenbanksysthemen*. 3. überarbeitete Auflage. Addison-Wesley 2002

[16] R. Hauser: Transforming Unstructured Cycles in Sequential Flow Graphs, Reseach Report, IBM Research GmbH, Juli 2005

[17] Institut für Innovations- und Umweltmanagement - Uni Graz. *Modelle des Innovations- und Technologiemanagements* – Skriptum, Kapitel 2, online, accessed 23. May 2006: http://www-classic.uni-graz.at/inmwww/itm/itm_kapitel_2.pdf ,

[18] J. Koehler, R. Hauser, S. Sendall, M. Wahler: *Declarative techniques for model-driven business process integration*, online, accessed 23. May 2006: http://www.research.ibm.com/journal/sj/441/koehler.pdf , Januar 2005

[19] K. Marczinzik: *Petri-Netze und Fuzzi-Petri-Netze – Einführungskurs mit interaktiven Beispielen*, online, accessed

23. May 2006: http://kik.informatik.fh-dortmund.de/Diplomarbeiten/DA%20Marczinzik/

[20] Microsoft Developer Network, *Web Services Description Language (WSDL) im Überblick,* online, accessed 23. May 2006: http://www.microsoft.com/germany/msdn/library/xmlwebservices/WebServicesDescriptionLanguageWSDLImUeberblick.mspx?mfr=true

[21] W. Moore, D. Dean, A. Gerber, G. Wagenknecht, P. Vanderheyden: *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*, online, accessed 23. May 2006: http://www.redbooks.ibm.com/abstracts/sg246302.html?Open

[22] Object Management Group Homepage, online, accessed 23. May 2006: http://www.omg.org/

[23] B. Oestereich: Objektorientierte Softwareentwicklung mit Unified Modeling Language, 3. Auflage. Oldenburg Verlag 1997

[24] OMONDO – The Live UML Company Homepage, online, accessed 23. May 2006: http://www.omondo.de

[25] C. Rupp, J. Hahn, S. Quentins, M. Jeckle, B. Zengler: *UML 2 gaksklar – Praxiswissen für die UML-Modellierung und – Zertifizierung*. 2. Auflage. Carl Hanser Verlag München Wien 2005

[26] UML Resource Page, online, accessed 23. May 2006: http://www.uml.org/

[27] W3C Recommendation, *Namespaces in XML*, online, accessed 23. May 2006: http://www.w3.org/TR/1999/REC-xml-names-19990114/

[28] W3C Recommendation, *XML Schema Part 1: Structures*, online, accessed 23. May 2006: http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/

[29] W3C Recommendation, *XML Schema Part 2: Datatypes*, online, accessed 23. May 2006: http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

[30] Web Services Transactions specifications, online, accessed 23. May 2006: http://www-128.ibm.com/developerworks/webservices/library/specification/ws-tx/

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

[32] World Wide Web Consortium, online, accessed 23. May 2006: http://www.w3c.org/

[33] XML Schema Definition (XSD), online, accessed 23. May 2006 http://www.w3.org/XML/Schema

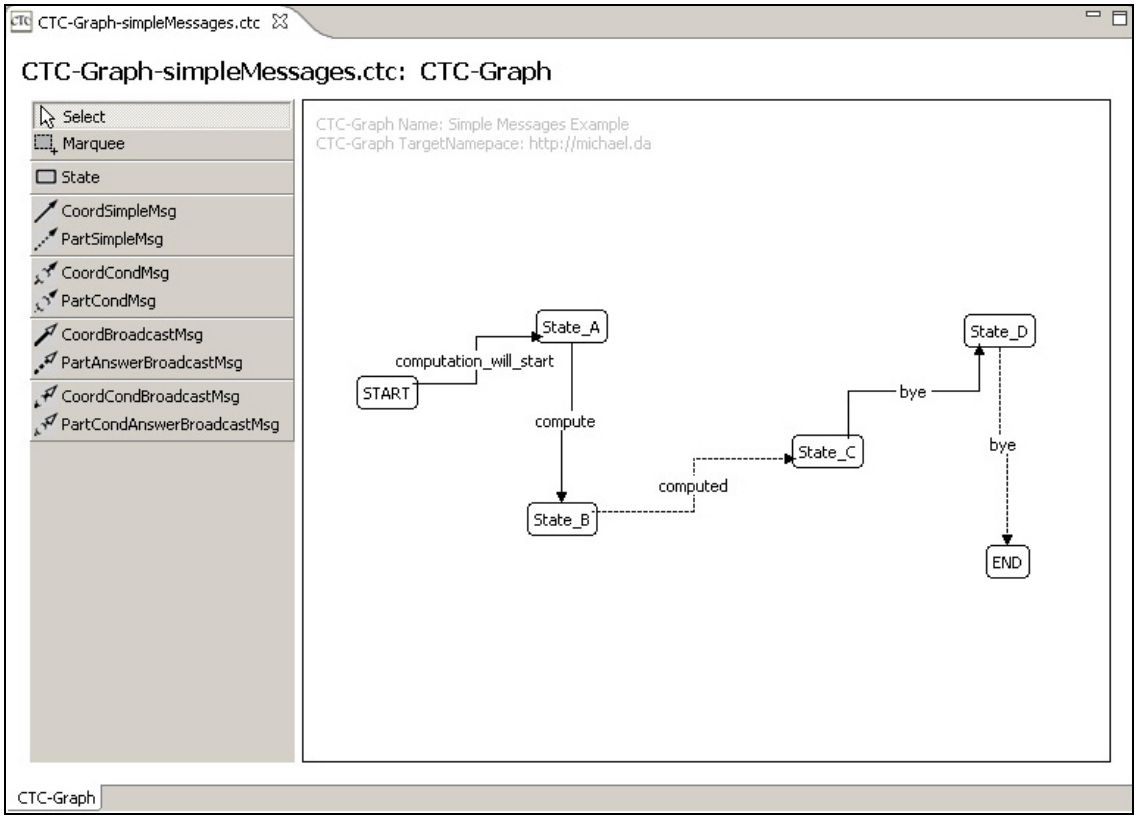
Appendix A – Beispiele CTC-Graphen

In den vorhergehenden Kapiteln wurden die CTC-Graphen und die Übersetzung dieser in BPEL-Code ausführlich besprochen. An dieser Stelle werden Beispielgraphen, die mit dem CTC-Graph Editor erzeugt wurden, präsentiert. Jedem CTC-Graph folgen die vollständigen Coordinator- und Teilnehmer-BPEL-Dateien, die vom BPEL-Generator automatisch erzeugt wurden.

Beispiel 1: Senden und Empfangen einfacher Nachrichten

In diesem Beispiel teilt der Koordinator den Teilnehmern mittels der „computation_will_start“-Nachricht mit, dass eine bestimmte Berechnung gestartet wird. Anschließend erhalten die Teilnehmer die Nachricht „compute“ mit den benötigten Daten. Nach der Berechnung senden die Teilnehmer eine Antwortnachricht „computed“. Weil nach dieser Berechnung nichts mehr zu tun ist, verabschiedet sich der Koordinator von den Teilnehmern mit der „bye“-Nachricht. Die Teilnehmer bestätigen den Empfang dieser Nachricht ebenfalls mit einer „bye“-Nachricht.

CTC-Graph:



In diesem Beispiel werden nur einfache Nachrichten ausgetauscht (CoordSimpleMessage bzw. PartSimpleMessage).

Generierter Koordinator BPEL-Code:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- BPEL-File generated with CTC-Graph Editor v. 1.0;
Diploma-Thesis Sabine Michael, May 2006 -->
<process name="Simple Messages Example"
targetNamespace="http://michael.da"
xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/busines
s-process/" xmlns:lns="http://coord_wsdl_file.here">
  <partnerLinks>
     <partnerLink name="computation_will_start"
partnerLinkType="lns:pL_Type" myRole="coordRole"
partnerRole="partRole"/>
     <partnerLink name="compute"
partnerLinkType="lns:pL_Type" myRole="coordRole"
partnerRole="partRole"/>
     <partnerLink name="computed"
partnerLinkType="lns:pL_Type" myRole="coordRole"
partnerRole="partRole"/>
     <partnerLink name="bye"
partnerLinkType="lns:pL_Type" myRole=""
partnerRole="partRole"/>
     <partnerLink name="bye"
partnerLinkType="lns:pL_Type" myRole="coordRole"
partnerRole="partRole"/>
  </partnerLinks>
  <variables>
     <variable name="inputVar_Name"
messageType="lns:inputVar_Type"/>
     <variable name="inputVar_Name"
messageType="lns:inputVar_Type"/>
     <variable name="var_compute"
messageType="outVar_Type"/>
     <variable name="inVar_Name"
messageType="lns:inVar_Type"/>
     <variable name="inVar_Name"
messageType="lns:inVar_Type"/>
     <variable name="var_bye" messageType="outVar_Type"/>
     <variable name="inVar_Name"
messageType="lns:inVar_Type"/>
  </variables>
  <sequence>
     <scope name="START">
        <sequence>
           <empty>
              <!-- reminder what should be done in state:
START -->
           </empty>
           <invoke name="computation_will_start"
partnerLink="pL_Name" portType="lns:pT_Name"
operation="op_Name" inputVariable="inputVar_Name">
```

```
               <!-- faults, correlations etc. add here -->
            </invoke>
         </sequence>
      </scope>
      <scope name="State_A">
         <sequence>
            <empty>
               <!-- reminder what should be done in state:
State_A -->
            </empty>
            <invoke name="compute" partnerLink="pL_Name"
portType="lns:pT_Name" operation="op_Name"
inputVariable="inputVar_Name"
outputVariable="var_compute">
               <!-- faults, correlations etc. add here -->
            </invoke>
         </sequence>
      </scope>
      <scope name="State_B">
         <sequence>
            <empty>
               <!-- reminder what should be done in state:
State_B -->
            </empty>
         </sequence>
      </scope>
      <scope name="State_C">
         <sequence>
            <receive name="computed" partnerLink="pL_Name"
portType="lns:pT_Name" operation="op_Name"
variable="inVar_Name">
               <!-- faults, correlations etc. add here -->
            </receive>
            <empty>
               <!-- reminder what should be done in state:
State_C -->
            </empty>
            <invoke name="bye" partnerLink="pL_Name"
portType="lns:pt_Name" operation="op_Name"
inputVariable="inVar_Name" outputVariable="var_bye">
               <!-- faults, correlations etc. add here -->
            </invoke>
         </sequence>
      </scope>
      <scope name="State_D">
         <sequence>
            <empty>
               <!-- reminder what should be done in state:
State_D -->
            </empty>
         </sequence>
```

```
      </scope>
      <scope name="END">
         <sequence>
            <receive name="bye" partnerLink="pL_Name"
portType="lns:pT_Name" operation="op_Name"
variable="inVar_Name">
               <!-- faults, correlations etc. add here -->
            </receive>
            <empty>
               <!-- reminder what should be done in state:
END -->
            </empty>
         </sequence>
      </scope>
   </sequence>
</process>
```

Generierter Teilnehmer BPEL-Code:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- BPEL-File generated with CTC-Graph Editor v. 1.0;
Diploma-Thesis Sabine Michael, May 2006 -->
<process name="Simple Messages Example"
targetNamespace="http://michael.da"
xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/busines
s-process/" xmlns:lns="http://part_wsdl_file.here">
   <partnerLinks>
      <partnerLink name="computation_will_start"
partnerLinkType="lns:pL_Type" myRole="partRole"
partnerRole="coordRole"/>
      <partnerLink name="compute"
partnerLinkType="lns:pL_Type" myRole="partRole"
partnerRole="coordRole"/>
      <partnerLink name="computed"
partnerLinkType="lns:pL_Type" myRole="partRole"
partnerRole="coordRole"/>
      <partnerLink name="bye"
partnerLinkType="lns:pL_Type" myRole="partRole"
partnerRole=""/>
      <partnerLink name="bye"
partnerLinkType="lns:pL_Type" myRole="partRole"
partnerRole="coordRole"/>
   </partnerLinks>
   <variables>
      <variable name="inputVar_Name"
messageType="lns:inputVar_Type"/>
      <variable name="inputVar_Name"
messageType="lns:inputVar_Type"/>
      <variable name="inVar_Name"
messageType="lns:inVar_Type"/>
```

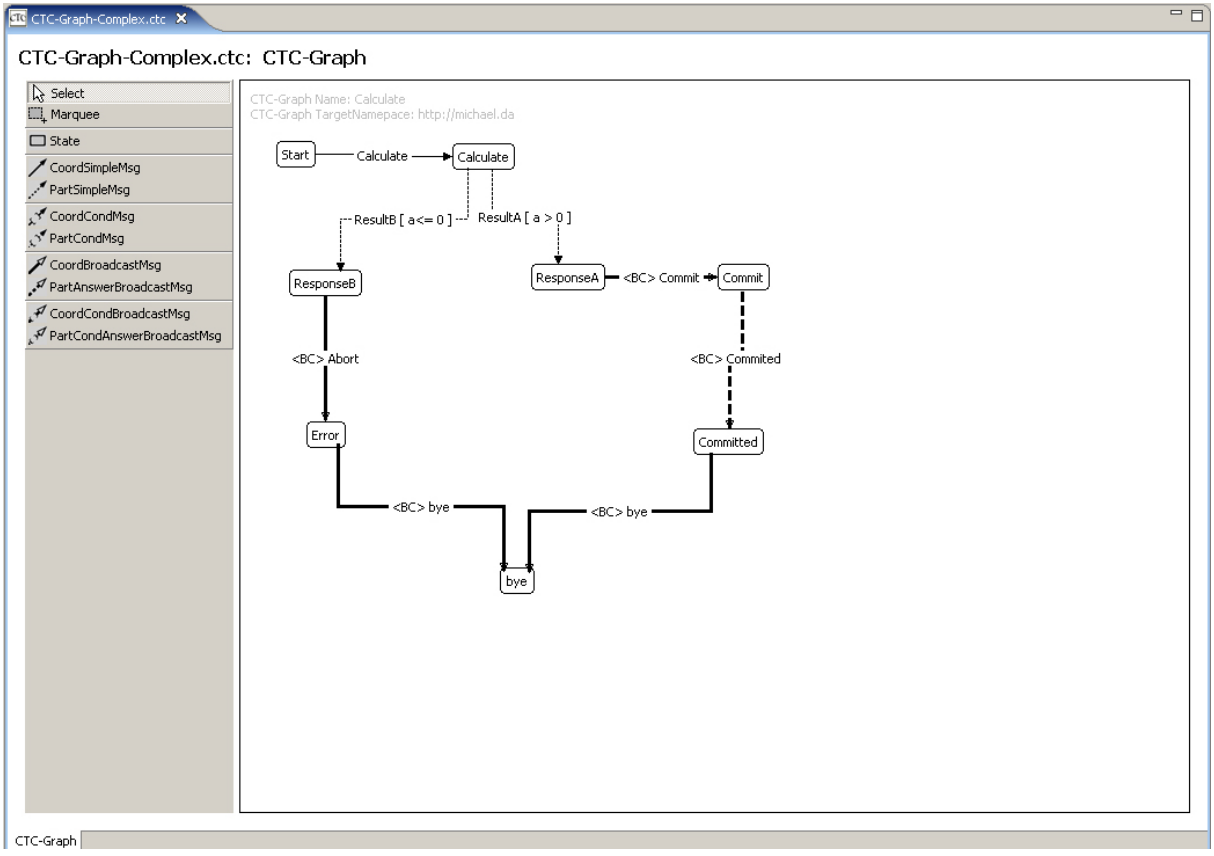
```
      <variable name="inVar_Name"
messageType="lns:inVar_Type"/>
      <variable name="inVar_Name"
messageType="lns:inVar_Type"/>
  </variables>
  <sequence>
      <scope name="START">
         <sequence>
            <empty>
               <!-- reminder what should be done in state:
START -->
            </empty>
         </sequence>
      </scope>
      <scope name="State_A">
         <sequence>
            <receive name="computation_will_start"
partnerLink="pL_Name" portType="lns:pT_Name"
operation="op_Name" variable="inputVar_Name">
               <!-- faults, correlations etc. add here -->
            </receive>
            <empty>
               <!-- reminder what should be done in state:
State_A -->
            </empty>
         </sequence>
      </scope>
      <scope name="State_B">
         <sequence>
            <receive name="compute" partnerLink="pL_Name"
portType="lns:pT_Name" operation="op_Name"
variable="inputVar_Name">
               <!-- faults, correlations etc. add here -->
            </receive>
            <empty>
               <!-- reminder what should be done in state:
State_B -->
            </empty>
            <invoke name="computed" partnerLink="pL_Name"
portType="lns:pT_Name" operation="op_Name"
inputVariable="inVar_Name">
               <!-- faults, correlations etc. add here -->
            </invoke>
         </sequence>
      </scope>
      <scope name="State_C">
         <sequence>
            <empty>
               <!-- reminder what should be done in state:
State_C -->
            </empty>
```

```
         </sequence>
      </scope>
      <scope name="State_D">
         <sequence>
            <receive name="bye" partnerLink="pL_Name"
portType="lns:pt_Name" operation="op_Name"
variable="inVar_Name">
               <!-- faults, correlations etc. add here -->
            </receive>
            <empty>
               <!-- reminder what should be done in state:
State_D -->
            </empty>
            <invoke name="bye" partnerLink="pL_Name"
portType="lns:pT_Name" operation="op_Name"
inputVariable="inVar_Name">
               <!-- faults, correlations etc. add here -->
            </invoke>
         </sequence>
      </scope>
      <scope name="END">
         <sequence>
            <empty>
               <!-- reminder what should be done in state:
END -->
            </empty>
         </sequence>
      </scope>
   </sequence>
</process>
```

Beispiel 2: Senden und Empfangen von einfachen Nachrichten, bedingungsgeknüpften Nachrichten und Commit- / Abort Broadcast-Nachrichten

In diesem Beispiel teilt der Koordinator den beiden Teilnehmern mittels der „Calculate“-Nachricht mit, dass eine bestimmte Berechnung gestartet wird. Abhängig vom Ergebnis der Berechnung senden die Teilnehmer die Nachricht „ResultA“ bzw. „ResultB“ mit den benötigten Daten an den Koordinator. Sobald der Koordinator die Nachricht „ResultB“ erhält, sendet er eine Abort-Broadcast-Nachricht an alle Teilnehmer und beendet anschließend die Transaktion mir einer „bye“-Nachricht, weil „a<=0“. Empfängt der Koordinator allerdings die „ResultA“-Nachricht, ist er mit dem Ergebnis zufrieden. Weil nach dieser Berechnung nichts mehr zu tun ist, sendet er eine „Commit“-Nachricht an alle Teilnehmer. Die Teilnehmer bestätigen den Empfang dieser Nachricht mit einer „Committed“-Nachricht. Nach dem Erhalt der Antwort-Nachrichten aller Teilnehmer, beendet der Koordinator die Transaktion mit einer „bye“-Nachricht.

CTC-Graph:



Im oberen Beispiel werden mehrere Nachrichten-Typen ausgetauscht (CoordSimpleMessage, PartConditionalMessages, CoordBroadcastMessage, PartAnswerBroadcastMessage).

Generierter Koordinator BPEL-Code:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- BPEL-File generated with CTC-Graph Editor v. 1.0 ;
Diploma-Thesis Sabine Michael, May 2006 -->
<process name="Calculate"
targetNamespace="http://michael.da"
xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-
process/" xmlns:lns="http://michael.da"
xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/busines
s-process/"
xmlns:xsd="http://www.w3c.org/2001/XMLSchema">
  <partnerLinks>
     <partnerLink name="Calc_PL"
partnerLinkType="lns:Calc_PL_Type" myRole="coordRole"
partnerRole="partRole"/>
     <partnerLink name="ResultA_PL"
partnerLinkType="lns:ResultA_PL_Type" myRole="coordRole"
partnerRole="partRole"/>
     <partnerLink name="ResultB_PL"
partnerLinkType="lns:ResultB_PL_Type" myRole="coordRole"
partnerRole="partRole"/>
     <partnerLink name="Commit_PL_1"
partnerLinkType="lns:Commit_PL_Type" myRole="coordRole"
partnerRole="partRole"/>
     <partnerLink name="Commit_PL_2"
partnerLinkType="lns:Commit_PL_Type" myRole="coordRole"
partnerRole="partRole"/>
     <partnerLink name="Abort_PL_1"
partnerLinkType="lns:Abort_PL_Type" myRole="coordRole"
partnerRole="partRole"/>
     <partnerLink name="Abort_PL_2"
partnerLinkType="lns:Abort_PL_Type" myRole="coordRole"
partnerRole="partRole"/>
     <partnerLink name="Commited_PL_1"
partnerLinkType="lns:Commited_PL_Type" myRole="coordRole"
partnerRole="partRole"/>
     <partnerLink name="Commited_PL_2"
partnerLinkType="lns:Commited_PL_Type" myRole="coordRole"
partnerRole="partRole"/>
     <partnerLink name="bye_PL_1"
partnerLinkType="lns:bye_PL_Type" myRole="coordRole"
partnerRole="partRole"/>
     <partnerLink name="bye_PL_2"
partnerLinkType="lns:bye_PL_Type" myRole="coordRole"
partnerRole="partRole"/>
     <partnerLink name="null_1"
partnerLinkType="lns:null" myRole="" partnerRole=""/>
     <partnerLink name="null_2"
partnerLinkType="lns:null" myRole="" partnerRole=""/>
  </partnerLinks>
  <variables>
```

```
      <variable name="Calc_inVar"
messageType="lns:Calc_inVar_Type"/>
      <variable name="outVar_Calculate"
messageType="lns:Calc_outVar_Type"/>
      <variable name="ResultA_inVar"
messageType="lns:ResultA_inVar_Type"/>
      <variable name="ResultB_inVar"
messageType="lns:ResultB_inVar_Type"/>
      <variable name="Commit_inVar_1"
messageType="lns:Commit_inVar_Type"/>
      <variable name="Commit_inVar_2"
messageType="lns:Commit_inVar_Type"/>
      <variable name="Abort_inVar_1"
messageType="lns:Abort_inVar_Type"/>
      <variable name="Abort_inVar_2"
messageType="lns:Abort_inVar_Type"/>
      <variable name="Commited_inVar_1"
messageType="lns:Commited_inVar_Type"/>
      <variable name="Commited_inVar_2"
messageType="lns:Commited_inVar_Type"/>
      <variable name="bye_inVar_1"
messageType="lns:bye_inVar_Type"/>
      <variable name="bye_inVar_2"
messageType="lns:bye_inVar_Type"/>
      <variable name="inVar_bye_1"
messageType="lns:null"/>
      <variable name="inVar_bye_2"
messageType="lns:null"/>
      <variable name="numberOfParticipants"
messageType="xsd:int"/>
      <variable name="sentCounter" messageType="xsd:int"/>
   </variables>
   <sequence>
      <scope name="Start">
         <sequence>
            <empty>
               <!-- start calculation -->
            </empty>
            <invoke name="Calculate" partnerLink="Calc_PL"
portType="lns:Calc_PT" operation="Calc_Operation"
inputVariable="Calc_inVar"
outputVariable="outVar_Calculate">
               <!-- faults, correlations etc. add here -->
            </invoke>
         </sequence>
      </scope>
      <scope name="Calculate">
         <sequence>
            <empty>
               <!-- do calculus -->
            </empty>
```

```
         </sequence>
      </scope>
      <scope name="ResponseA">
         <sequence>
            <receive name="ResultA"
partnerLink="ResultA_PL" portType="lns:ResultA_PT"
operation="ResultA_Op" variable="ResultA_inVar">
               <!-- faults, correlations etc. add here -->
            </receive>
            <empty>
               <!-- get response - work -->
            </empty>
            <!-- writeCoordSendBroadcast generated -->
            <assign name="Assign_While_Variables">
               <copy>
                  <from expression="2"/>
                  <to variable="numberOfParticipants"/>
               </copy>
               <copy>
                  <from expression="1"/>
                  <to variable="sentCounter"/>
               </copy>
            </assign>
            <while
condition="bpws:getVariableData('sentCounter') &lt;=
bpws:getVariableData('numberOfParticipants')">
               <sequence>
                  <flow>
                     <links>
                        <link linkName="lk_Commit_1"/>
                        <link linkName="lk_Commit_2"/>
                     </links>
                     <empty>
                        <source name="lk_Commit_1"
transitionCondition="bpws:getVariableData('sentCounter')
= 1"/>
                        <source name="lk_Commit_2"
transitionCondition="bpws:getVariableData('sentCounter')
= 2"/>
                     </empty>
                     <invoke name="Commit_1"
partnerLinkType="Commit_PL_1" portType="lns:Commit_PT"
operation="Commit_Op" inputVariable="Commit_inVar_1">
                        <target linkName="lk_Commit_1"/>
                        <!-- faults, correlations etc. add
here -->
                     </invoke>
                     <invoke name="Commit_2"
partnerLinkType="Commit_PL_2" portType="lns:Commit_PT"
operation="Commit_Op" inputVariable="Commit_inVar_2">
                        <target linkName="lk_Commit_2"/>
```

```
                        <!-- faults, correlations etc. add
here -->
                     </invoke>
                  </flow>
                  <assign>
                     <copy>
                        <from
query="bpws:getVariableData('sentCounter') = + 1"
variable="sentCounter"/>
                        <to variable="sentCounter"/>
                     </copy>
                  </assign>
               </sequence>
            </while>
         </sequence>
      </scope>
      <scope name="ResponseB">
         <sequence>
            <receive name="ResultB"
partnerLink="ResultB_PL" portType="lns:ResultB_PT"
operation="ResultB_Op" variable="ResultB_inVar">
               <!-- faults, correlations etc. add here -->
            </receive>
            <empty>
               <!-- get response - work -->
            </empty>
            <!-- writeCoordSendBroadcast generated -->
            <assign name="Assign_While_Variables">
               <copy>
                  <from expression="2"/>
                  <to variable="numberOfParticipants"/>
               </copy>
               <copy>
                  <from expression="1"/>
                  <to variable="sentCounter"/>
               </copy>
            </assign>
            <while
condition="bpws:getVariableData('sentCounter') &lt;=
bpws:getVariableData('numberOfParticipants')">
               <sequence>
                  <flow>
                     <links>
                        <link linkName="lk_Abort_1"/>
                        <link linkName="lk_Abort_2"/>
                     </links>
                     <empty>
                        <source name="lk_Abort_1"
transitionCondition="bpws:getVariableData('sentCounter')
= 1"/>
```

```
                        <source name="lk_Abort_2"
transitionCondition="bpws:getVariableData('sentCounter')
= 2"/>
                     </empty>
                     <invoke name="Abort_1"
partnerLinkType="Abort_PL_1" portType="lns:Abort_PT"
operation="Abort_Op" inputVariable="Abort_inVar_1">
                        <target linkName="lk_Abort_1"/>
                        <!-- faults, correlations etc. add
here -->
                     </invoke>
                     <invoke name="Abort_2"
partnerLinkType="Abort_PL_2" portType="lns:Abort_PT"
operation="Abort_Op" inputVariable="Abort_inVar_2">
                        <target linkName="lk_Abort_2"/>
                        <!-- faults, correlations etc. add
here -->
                     </invoke>
                  </flow>
                  <assign>
                     <copy>
                        <from
query="bpws:getVariableData('sentCounter') = + 1"
variable="sentCounter"/>
                        <to variable="sentCounter"/>
                     </copy>
                  </assign>
               </sequence>
            </while>
         </sequence>
      </scope>
      <scope name="Commit">
         <sequence>
            <empty>
               <!-- commit -->
            </empty>
         </sequence>
      </scope>
      <scope name="Committed">
         <sequence>
            <flow name="Commited_flow">
               <links>
                  <link linkName="lk_Commited_1"/>
                  <link linkName="lk_Commited_2"/>
               </links>
               <sequence>
                  <flow>
                     <receive name="Commited_1"
partnerLink="Commited_PL_1" portType="lns:Commited_PT"
operation="Commited_Op" variable="Commited_inVar_1">
                        <source linkName="lk_Commited_1"/>
```

```
                        <!-- faults, correlations etc. add
here -->
                    </receive>
                    <receive name="Commited_2"
partnerLink="Commited_PL_2" portType="lns:Commited_PT"
operation="Commited_Op" variable="Commited_inVar_2">
                        <source linkName="lk_Commited_2"/>
                        <!-- faults, correlations etc. add
here -->
                    </receive>
                </flow>
                <empty>
                    <target linkName="lk_Commited_1"/>
                    <target linkName="lk_Commited_2"/>
                </empty>
            </sequence>
        </flow>
        <empty>
            <!-- send bye -->
        </empty>
        <!-- writeCoordSendBroadcast generated -->
        <assign name="Assign_While_Variables">
            <copy>
                <from expression="2"/>
                <to variable="numberOfParticipants"/>
            </copy>
            <copy>
                <from expression="1"/>
                <to variable="sentCounter"/>
            </copy>
        </assign>
        <while
condition="bpws:getVariableData('sentCounter') &lt;=
bpws:getVariableData('numberOfParticipants')">
            <sequence>
                <flow>
                    <links>
                        <link linkName="lk_bye_1"/>
                        <link linkName="lk_bye_2"/>
                    </links>
                    <empty>
                        <source name="lk_bye_1"
transitionCondition="bpws:getVariableData('sentCounter')
= 1"/>
                        <source name="lk_bye_2"
transitionCondition="bpws:getVariableData('sentCounter')
= 2"/>
                    </empty>
                    <invoke name="bye_1"
partnerLink="bye_PL_1" portType="lns:bye_PT"
operation="bye_Op" inputVariable="bye_inVar_1">
```

```
                        <target linkName="lk_bye_1"/>
                        <!-- faults, correlations etc. add
here -->
                     </invoke>
                     <invoke name="bye_2"
partnerLink="bye_PL_2" portType="lns:bye_PT"
operation="bye_Op" inputVariable="bye_inVar_2">
                        <target linkName="lk_bye_2"/>
                        <!-- faults, correlations etc. add
here -->
                     </invoke>
                  </flow>
                  <assign>
                     <copy>
                        <from
query="bpws:getVariableData('sentCounter') = + 1"
variable="sentCounter"/>
                        <to variable="sentCounter"/>
                     </copy>
                  </assign>
               </sequence>
            </while>
         </sequence>
      </scope>
      <scope name="bye">
         <sequence>
            <empty>
               <!-- end communication -->
            </empty>
         </sequence>
      </scope>
      <scope name="Error">
         <sequence>
            <empty>
               <!-- rollback -->
            </empty>
            <!-- writeCoordSendBroadcast generated -->
            <assign name="Assign_While_Variables">
               <copy>
                  <from expression="2"/>
                  <to variable="numberOfParticipants"/>
               </copy>
               <copy>
                  <from expression="1"/>
                  <to variable="sentCounter"/>
               </copy>
            </assign>
            <while
condition="bpws:getVariableData('sentCounter') &lt;=
bpws:getVariableData('numberOfParticipants')">
               <sequence>
```

```
                  <flow>
                     <links>
                        <link linkName="lk_bye_1"/>
                        <link linkName="lk_bye_2"/>
                     </links>
                     <empty>
                        <source name="lk_bye_1"
transitionCondition="bpws:getVariableData('sentCounter')
= 1"/>
                        <source name="lk_bye_2"
transitionCondition="bpws:getVariableData('sentCounter')
= 2"/>
                     </empty>
                     <invoke name="bye_1"
partnerLink="null_1" portType="lns:null" operation=""
inputVariable="inVar_bye_1">
                        <target linkName="lk_bye_1"/>
                        <!-- faults, correlations etc. add
here -->
                     </invoke>
                     <invoke name="bye_2"
partnerLink="null_2" portType="lns:null" operation=""
inputVariable="inVar_bye_2">
                        <target linkName="lk_bye_2"/>
                        <!-- faults, correlations etc. add
here -->
                     </invoke>
                  </flow>
                  <assign>
                     <copy>
                        <from
query="bpws:getVariableData('sentCounter') = + 1"
variable="sentCounter"/>
                        <to variable="sentCounter"/>
                     </copy>
                  </assign>
               </sequence>
            </while>
         </sequence>
      </scope>
   </sequence>
</process>
```

Generierter BPEL-Code Teilnehmer 1:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- BPEL-File generated with CTC-Graph Editor v. 1.0 ;
Diploma-Thesis Sabine Michael, May 2006 -->
<process name="Calculate"
targetNamespace="http://michael.da"
xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-
process/" xmlns:lns="http://one.michael.da"
xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/busines
s-process/"
xmlns:xsd="http://www.w3c.org/2001/XMLSchema">
  <partnerLinks>
     <partnerLink name="Calc_PL"
partnerLinkType="lns:Calc_PL_Type" myRole="partRole"
partnerRole="coordRole"/>
     <partnerLink name="ResultA_PL"
partnerLinkType="lns:ResultA_PL_Type" myRole="partRole"
partnerRole="coordRole"/>
     <partnerLink name="ResultB_PL"
partnerLinkType="lns:ResultB_PL_Type" myRole="partRole"
partnerRole="coordRole"/>
     <partnerLink name="Commit_PL_1"
partnerLinkType="lns:Commit_PL_Type" myRole="partRole"
partnerRole="coordRole"/>
     <partnerLink name="Abort_PL_1"
partnerLinkType="lns:Abort_PL_Type" myRole="partRole"
partnerRole="coordRole"/>
     <partnerLink name="Commited_PL_1"
partnerLinkType="lns:Commited_PL_Type" myRole="partRole"
partnerRole="coordRole"/>
     <partnerLink name="bye_PL_1"
partnerLinkType="lns:bye_PL_Type" myRole="partRole"
partnerRole="coordRole"/>
     <partnerLink name="null_1"
partnerLinkType="lns:null" myRole="" partnerRole=""/>
  </partnerLinks>
  <variables>
     <variable name="Calc_inVar"
messageType="lns:Calc_inVar_Type"/>
     <variable name="ResultA_inVar"
messageType="lns:ResultA_inVar_Type"/>
     <variable name="ResultB_inVar"
messageType="lns:ResultB_inVar_Type"/>
     <variable name="Commit_inVar_1"
messageType="lns:Commit_inVar_Type"/>
     <variable name="Abort_inVar_1"
messageType="lns:Abort_inVar_Type"/>
     <variable name="Commited_inVar"
messageType="lns:Commited_inVar_Type"/>
```

```
      <variable name="bye_inVar_1"
messageType="lns:bye_inVar_Type"/>
      <variable name="inVar_bye_1"
messageType="lns:null"/>
  </variables>
  <sequence>
      <scope name="Start">
         <sequence>
            <empty>
               <!-- start calculation -->
            </empty>
         </sequence>
      </scope>
      <scope name="Calculate">
         <sequence>
            <receive name="Calculate"
partnerLink="Calc_PL" portType="lns:Calc_PT"
operation="Calc_Operation" variable="Calc_inVar">
               <!-- faults, correlations etc. add here -->
            </receive>
            <empty>
               <!-- do calculus -->
            </empty>
            <flow name="ResultA_flow">
               <links>
                  <link linkName="lk_ResultA"/>
                  <link linkName="lk_ResultB"/>
               </links>
               <empty name="Add your instructions here to
activate link conditions">
                  <source linkName="lk_ResultA"
transitionCondition="a &gt; 0"/>
                  <source linkName="lk_ResultB"
transitionCondition="a&lt;= 0"/>
               </empty>
               <invoke name="ResultA"
partnerLink="ResultA_PL" portType="lns:ResultA_PT"
operation="ResultA_Op" inputVariable="ResultA_inVar">
                  <target linkName="lk_ResultA"/>
                  <!-- faults, correlations etc. add here
-->
               </invoke>
               <invoke name="ResultB"
partnerLink="ResultB_PL" portType="lns:ResultB_PT"
operation="ResultB_Op" inputVariable="ResultB_inVar">
                  <target linkName="lk_ResultB"/>
                  <!-- faults, correlations etc. add here
-->
               </invoke>
            </flow>
         </sequence>
```

```
      </scope>
      <scope name="ResponseA">
         <sequence>
            <empty>
               <!-- get response - work -->
            </empty>
         </sequence>
      </scope>
      <scope name="ResponseB">
         <sequence>
            <empty>
               <!-- get response - work -->
            </empty>
         </sequence>
      </scope>
      <scope name="Commit">
         <sequence>
            <receive name="Commit"
partnerLink="Commit_PL_1" portType="lns:Commit_PT"
operation="Commit_Op" variable="Commit_inVar_1">
               <!-- faults, correlations etc. add here -->
            </receive>
            <empty>
               <!-- commit -->
            </empty>
            <invoke name="Commited"
partnerLink="Commited_PL_1" portType="lns:Commited_PT"
operation="Commited_Op" inputVariable="Commited_inVar">
               <!-- faults, correlations etc. add here -->
            </invoke>
         </sequence>
      </scope>
      <scope name="Committed">
         <sequence>
            <empty>
               <!-- send bye -->
            </empty>
         </sequence>
      </scope>
      <scope name="bye">
         <sequence>
            <receive name="bye" partnerLink="bye_PL_1"
portType="lns:bye_PT" operation="bye_Op"
variable="bye_inVar_1">
               <!-- faults, correlations etc. add here -->
            </receive>
            <receive name="bye" partnerLink="null_1"
portType="lns:null" operation="" variable="inVar_bye_1">
               <!-- faults, correlations etc. add here -->
            </receive>
            <empty>
```

```
                <!-- end communication -->
            </empty>
         </sequence>
      </scope>
      <scope name="Error">
         <sequence>
            <receive name="Abort" partnerLink="Abort_PL_1"
portType="lns:Abort_PT" operation="Abort_Op"
variable="Abort_inVar_1">
                <!-- faults, correlations etc. add here -->
            </receive>
            <empty>
               <!-- rollback -->
            </empty>
         </sequence>
      </scope>
   </sequence>
</process>
```

Generierter BPEL-Code Teilnehmer 2:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- BPEL-File generated with CTC-Graph Editor v. 1.0 ;
Diploma-Thesis Sabine Michael, May 2006 -->
<process name="Calculate"
targetNamespace="http://michael.da"
xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-
process/" xmlns:lns="http://two.michael.da"
xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/busines
s-process/"
xmlns:xsd="http://www.w3c.org/2001/XMLSchema">
  <partnerLinks>
     <partnerLink name="Calc_PL"
partnerLinkType="lns:Calc_PL_Type" myRole="partRole"
partnerRole="coordRole"/>
     <partnerLink name="ResultA_PL"
partnerLinkType="lns:ResultA_PL_Type" myRole="partRole"
partnerRole="coordRole"/>
     <partnerLink name="ResultB_PL"
partnerLinkType="lns:ResultB_PL_Type" myRole="partRole"
partnerRole="coordRole"/>
     <partnerLink name="Commit_PL_2"
partnerLinkType="lns:Commit_PL_Type" myRole="partRole"
partnerRole="coordRole"/>
     <partnerLink name="Abort_PL_2"
partnerLinkType="lns:Abort_PL_Type" myRole="partRole"
partnerRole="coordRole"/>
     <partnerLink name="Commited_PL_2"
partnerLinkType="lns:Commited_PL_Type" myRole="partRole"
partnerRole="coordRole"/>
```

```
      <partnerLink name="bye_PL_2"
partnerLinkType="lns:bye_PL_Type" myRole="partRole"
partnerRole="coordRole"/>
      <partnerLink name="null_2"
partnerLinkType="lns:null" myRole="" partnerRole=""/>
   </partnerLinks>
   <variables>
      <variable name="Calc_inVar"
messageType="lns:Calc_inVar_Type"/>
      <variable name="ResultA_inVar"
messageType="lns:ResultA_inVar_Type"/>
      <variable name="ResultB_inVar"
messageType="lns:ResultB_inVar_Type"/>
      <variable name="Commit_inVar_2"
messageType="lns:Commit_inVar_Type"/>
      <variable name="Abort_inVar_2"
messageType="lns:Abort_inVar_Type"/>
      <variable name="Commited_inVar"
messageType="lns:Commited_inVar_Type"/>
      <variable name="bye_inVar_2"
messageType="lns:bye_inVar_Type"/>
      <variable name="inVar_bye_2"
messageType="lns:null"/>
   </variables>
   <sequence>
      <scope name="Start">
         <sequence>
            <empty>
               <!-- start calculation -->
            </empty>
         </sequence>
      </scope>
      <scope name="Calculate">
         <sequence>
            <receive name="Calculate"
partnerLink="Calc_PL" portType="lns:Calc_PT"
operation="Calc_Operation" variable="Calc_inVar">
               <!-- faults, correlations etc. add here -->
            </receive>
            <empty>
               <!-- do calculus -->
            </empty>
            <flow name="ResultA_flow">
               <links>
                  <link linkName="lk_ResultA"/>
                  <link linkName="lk_ResultB"/>
               </links>
               <empty name="Add your instructions here to
activate link conditions">
                  <source linkName="lk_ResultA"
transitionCondition="a &amp;gt; 0"/>
```

```
                  <source linkName="lk_ResultB"
transitionCondition="a&lt;= 0"/>
               </empty>
               <invoke name="ResultA"
partnerLink="ResultA_PL" portType="lns:ResultA_PT"
operation="ResultA_Op" inputVariable="ResultA_inVar">
                  <target linkName="lk_ResultA"/>
                  <!-- faults, correlations etc. add here
-->
               </invoke>
               <invoke name="ResultB"
partnerLink="ResultB_PL" portType="lns:ResultB_PT"
operation="ResultB_Op" inputVariable="ResultB_inVar">
                  <target linkName="lk_ResultB"/>
                  <!-- faults, correlations etc. add here
-->
               </invoke>
            </flow>
         </sequence>
      </scope>
      <scope name="ResponseA">
         <sequence>
            <empty>
               <!-- get response - work -->
            </empty>
         </sequence>
      </scope>
      <scope name="ResponseB">
         <sequence>
            <empty>
               <!-- get response - work -->
            </empty>
         </sequence>
      </scope>
      <scope name="Commit">
         <sequence>
            <receive name="Commit"
partnerLink="Commit_PL_2" portType="lns:Commit_PT"
operation="Commit_Op" variable="Commit_inVar_2">
               <!-- faults, correlations etc. add here -->
            </receive>
            <empty>
               <!-- commit -->
            </empty>
            <invoke name="Commited"
partnerLink="Commited_PL_2" portType="lns:Commited_PT"
operation="Commited_Op" inputVariable="Commited_inVar">
               <!-- faults, correlations etc. add here -->
            </invoke>
         </sequence>
      </scope>
```

```
      <scope name="Committed">
         <sequence>
            <empty>
               <!-- send bye -->
            </empty>
         </sequence>
      </scope>
      <scope name="bye">
         <sequence>
            <receive name="bye" partnerLink="bye_PL_2"
portType="lns:bye_PT" operation="bye_Op"
variable="bye_inVar_2">
               <!-- faults, correlations etc. add here -->
            </receive>
            <receive name="bye" partnerLink="null_2"
portType="lns:null" operation="" variable="inVar_bye_2">
               <!-- faults, correlations etc. add here -->
            </receive>
            <empty>
               <!-- end communication -->
            </empty>
         </sequence>
      </scope>
      <scope name="Error">
         <sequence>
            <receive name="Abort" partnerLink="Abort_PL_2"
portType="lns:Abort_PT" operation="Abort_Op"
variable="Abort_inVar_2">
               <!-- faults, correlations etc. add here -->
            </receive>
            <empty>
               <!-- rollback -->
            </empty>
         </sequence>
      </scope>
   </sequence>
</process>
```

Erklärung

Ich versichere, dass ich diese Arbeit selbständig verfasst und nur die angegebenen Hilfsmittel verwendet habe.

_____ _____

Ort, Datum Sabine Michael