

Institut für Architektur von Anwendungssystemen  
Universitätsstraße 38  
D-70569 Stuttgart

Diplomarbeit Nr. 2341

**Abbildung von EPKs nach BPEL  
anhand des  
Prozessmodellierungswerkzeugs  
Nautilus**

Oliver Kopp

**Studiengang:** Softwaretechnik

**Prüfer:** Prof. Dr. Frank Leymann  
**Betreuer:** M. Sc. Dimka Karastoyanova

**begonnen am:** 25. April 2005  
**beendet am:** 21. Oktober 2005

**CR-Klassifikation:** H.4.1, K.1



# INHALTSVERZEICHNIS

---

|  |           |
|--|-----------|
| <b>1. Einleitung</b>   | <b>5</b>  |
| 1.1. Motivation  | 5         |
| 1.2. Aufbau der Arbeit   | 6         |
| <b>2. Geschäftsprozesse und deren Modellierung</b>             | <b>7</b>  |
| 2.1. Geschäftsprozesse   | 7         |
| 2.2. ARIS  | 9         |
| 2.3. Ereignisgesteuerte Prozessketten                          | 10        |
| <b>3. Metamodell von Nautilus</b>                              | <b>17</b> |
| 3.1. Graphentheoretische Definitionen                          | 17        |
| 3.2. Kompakte Graphen  | 22        |
| 3.3. Die Tiefensuche   | 23        |
| 3.4. Formalisierung der EPKs                                   | 23        |
| 3.5. Elemente jedes Modells                                    | 26        |
| 3.6. Der Modellgraph   | 30        |
| 3.7. Extraktion der EPK-Graphen                                | 48        |
| <b>4. Business Process Execution Language for Web Services</b> | <b>55</b> |
| 4.1. WSDL  | 56        |
| 4.2. Die Säulen von BPEL                                       | 59        |
| 4.3. Hauptbestandteile eines BPEL-Dokuments                    | 59        |
| 4.4. Fehlerbehandlung und Kompensation                         | 65        |
| 4.5. Ereignisbehandlung  | 65        |
| 4.6. flow-Aktivität  | 65        |
| 4.7. Prozesse in BPEL  | 67        |
| <b>5. Mapping des Metamodells nach BPEL</b>                    | <b>69</b> |
| 5.1. Grundsätzliche Umsetzung der Beschriftungen               | 71        |
| 5.2. Gerüst des Prozesses                                      | 71        |
| 5.3. Abbildung der eEPK  | 72        |
| 5.4. Umsetzung von Funktionswegweisern                         | 76        |
| 5.5. Abbildung von Ereignissen                                 | 76        |
| 5.6. Schleifenerkennung  | 77        |

|   |            |
|---|------------|
| 5.7. Abbildung von Operatoren . . . . .                           | 86         |
| 5.8. Generierung der Kanten . . . . .                             | 87         |
| 5.9. Generierung des Eventhandlers . . . . .                      | 89         |
| 5.10. Umsetzung in Correlation Sets . . . . .                     | 90         |
| 5.11. Generierung der WSDL-Datei . . . . .                        | 92         |
| 5.12. Nicht erzeugte Konstrukte . . . . .                         | 92         |
| <b>6. BPEL-Export</b>   | <b>95</b>  |
| 6.1. Die Datenschicht . . . . .                                   | 97         |
| 6.2. Die Modellierungsschicht . . . . .                           | 99         |
| 6.3. Speicherung des Metamodells . . . . .                        | 99         |
| 6.4. Umsetzung des Exports . . . . .                              | 101        |
| 6.5. Onlineprüfung . . . . .                                      | 105        |
| <b>7. Zusammenfassung und Ausblick</b>                            | <b>109</b> |
| <b>A. Tabellen</b>  | <b>111</b> |
| <b>B. Grammatiken</b>   | <b>115</b> |
| <b>C. Oberfläche von Nautilus</b>                                 | <b>117</b> |
| <b>D. Prozesse der Bank AG</b>                                    | <b>119</b> |
| <b>E. Umsetzung des Kreditantragsprozesses</b>                    | <b>123</b> |
| E.1. BPEL-Prozess . . . . .                                       | 123        |
| E.2. WSDL-Datei . . . . .   | 125        |
| E.3. Darstellung in ActiveWebflow Professional Designer . . . . . | 127        |

# EINLEITUNG

---

*Wenn Meißel und Schiffchen von selbst sich bewegten,  
würde die Sklaverei nicht nötig sein!*

*(Aristoteles)*

Geschäftsprozesse sind Verkettungen wertschöpfender Aktivitäten die strategische Bedeutung für das Unternehmen besitzen. So beeinflussen Geschäftsprozesse maßgeblich die entstehenden Kosten und somit den Unternehmensgewinn. In den 90er Jahren lag der Schwerpunkt in der Erfassung von Geschäftsprozessen, um Optimierungspotenziale zu identifizieren. Heute ist die Automatisierung der Geschäftsprozesse durch Maschinen in den Vordergrund gerückt. Als Sprache zur Beschreibung von automatisierten Geschäftsprozessen hat sich die „Business Process Execution Language for Web Services“ (BPEL) durchgesetzt. Bei Entscheidungsträgern findet BPEL allerdings wenig Anklang, da BPEL technisch orientiert ist und sie eine andere Notation von Geschäftsprozessen gewohnt sind. Entscheider verwenden „ereignisgesteuerte Prozessketten“ (EPKs) zur grafischen Modellierung von Geschäftsprozessen.

Durch ereignisgesteuerte Prozessketten können beliebige Prozesse modelliert werden. So sind beispielsweise alle Geschäftsprozesse in SAP R/3 durch ereignisgesteuerte Prozessketten modelliert.

### 1.1. Motivation

Bisher gab es keinen Ansatz, die Welt der ereignisgesteuerten Prozessketten mit der Welt von BPEL automatisiert zu verbinden. Diese Verbindung wird durch die vorliegende Arbeit geschaffen: es werden Ideen entwickelt, wie ereignisgesteuerte Prozessketten ohne Modifikationen nach BPEL abgebildet werden können. Diese Ideen werden anschließend zur Formulierung eines Algorithmus zur automatisierten Umsetzung von ereignisgesteuerten Prozessketten nach BPEL verwendet. Der Algorithmus wird in der Geschäftsprozessmodellierungssoftware Nautilus als Export-Modul realisiert. Somit wird die Verbindung sowohl theoretisch als auch praktisch geschaffen: Entscheider müssen keine neue Notation erlernen und Entwickler erhalten die Spezifikation der zu realisierenden Prozesse in einer maschinennahen Sprache.

### 1.2. Aufbau der Arbeit

Im Kapitel 2 wird zuerst ein einheitliches Verständnis von Geschäftsprozessen geschaffen. Dies umfasst sowohl die Definition von Geschäftsprozessen als auch die ARIS-Methode zur Modellierung von Geschäftsprozessen. Ein Bestandteil der ARIS-Methode ist grafische Darstellung von Prozessen durch ereignisgesteuerte Prozessketten. Zur Modellierung von Prozessen gehört auch die Erfassung der ein- und ausgehenden Information und der beteiligten Mitarbeiter. Eine Erläuterung, wie das Ergebnis der Erfassung mittels ereignisgesteuerter Prozessketten dargestellt werden kann, bildet den Abschluss des Kapitels.

Nautilus ist eine Software zur Erfassung ereignisgesteuerter Prozessketten. Das Metamodell ist im Benutzerhandbuch von Nautilus beschrieben. Allerdings ist das Metamodell nicht formalisiert, so dass es nicht als Ausgangsbasis für einen Algorithmus verwendet werden kann. Deshalb wird das Nautilus-Metamodell im Kapitel 3 formalisiert. Das Ergebnis ist ein „Modellgraph“, der alle Bestandteile einer ereignisgesteuerten Prozesskette enthält.

In dieser Arbeit wird ein Algorithmus definiert, der ereignisgesteuerte Prozessketten in BPEL-Prozesse abbildet. Die Quelle der Abbildung bildet das im Kapitel 3 vorgestellte Metamodell. Zum Verständnis des Ziels der Abbildung werden Web Services gemeinsam mit BPEL in Kapitel 4 vorgestellt.

Mit diesen Grundlagen kann die Abbildung von ereignisgesteuerten Prozessketten nach BPEL beschrieben werden. Die Abbildung der ereignisgesteuerten Prozessketten nach BPEL wird in Kapitel 5 vorgestellt. Um die grundsätzlichen Konzepte zu erläutern, werden zuerst typische Formen und Bestandteile von ereignisgesteuerten Prozessketten zusammen mit ihrer Abbildung nach BPEL aufgeführt. Hierbei werden auch Konstrukte vorgestellt, die sich nicht nach BPEL abbilden lassen. Bei den in der Nautilus-Schulung vorgestellten Modellen bedeutet dies keine Einschränkung, so dass geschlossen werden kann, dass sich gängige Modelle nach BPEL abbilden lassen.

Im Kapitel 6 wird die Implementierung der Abbildung erläutert. Dazu wird zuerst die Architektur von Nautilus und daraufhin die Umsetzung als Export-Modul beschrieben. Durch die Umsetzung werden die Welten der ereignisgesteuerten Prozessketten und Web Services sowohl in der Theorie als auch in der Praxis verbunden.

# GESCHÄFTSPROZESSE UND DEREN MODELLIERUNG

---

In diesem Kapitel werden die für das Verständnis der nachfolgenden Kapitel notwendigen Grundlagen gelegt. Die Darstellung von Geschäftsprozessen mit den Aspekten des Kontroll- und Datenflusses bilden den Anfang des Kapitels. Daraufhin wird der Modellierungsansatz der ARIS-Methode vorgestellt. Die ARIS-Methode wurde für Firmen entwickelt, um ihnen zu ermöglichen, die Komplexität von IT-Projekten zu reduzieren. Zu der Methode gehören ereignisgesteuerte Prozessketten, die es erlauben, Geschäftsprozesse grafisch darzustellen. Diese bilden den Ausgangspunkt der Abbildung nach BPEL, weshalb ihre Bestandteile am Ende des Kapitels erläutert werden.

## 2.1. Geschäftsprozesse

Ein Geschäftsprozess ist eine Menge von Aktivitäten, die in einem Betrieb nach bestimmten Regeln auf ein bestimmtes Ziel hin ausgeführt werden (vgl. [Obe96]).

In Abbildung 2.1 ist ein vereinfachter Geschäftsprozess zur Bearbeitung eines Kreditantrags zu sehen. Sobald ein Kreditantrag eintrifft, wird er nach diesem Muster bearbeitet. Der dargestellte Geschäftsprozess ist ein Modell für eine konkrete Bearbeitung eines Kreditantrags.

Bei einer Geschäftsprozessmodellierung – oder kürzer Prozessmodellierung – werden zwei Aspekte modelliert: Der Ablauf des Prozesses und der Aufbau der Organisation, die an der Ausführung des Prozesses beteiligt ist. In der Aufbauorganisation können neben der längerfristig bestehenden Organisation auch mittelfristige Organisationsstrukturen, wie die Projektarbeit, modelliert werden. Hier werden beispielsweise Stellen oder Mitarbeiter einem Projekt zugeordnet, um die Ressourcen im Unternehmen planen zu können.

In der Ablauforganisation wird der Ablauf eines Prozesses mittels Aktivitäten und Konnektoren beschrieben. Jede Aktivität gibt einen Arbeitsschritt in einem Prozess an. Ein Konnektor besitzt eine Richtung und verbindet eine Aktivität mit einer anderen. Es gibt

## 2. Geschäftsprozesse und deren Modellierung

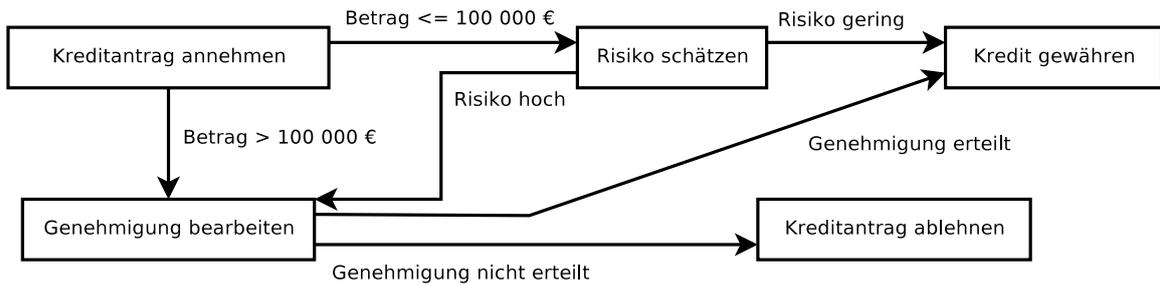


Abbildung 2.1.: Illustrationsprozess Kreditantrag nach [LR99, S. 33 ff.]

zwei Arten von Konnektoren: Kontrollkonnektoren, die der Darstellung des Kontrollflusses dienen und Datenkonnektoren, die zur Darstellung des Datenflusses verwendet werden.

Durch den Kontrollfluss wird die Abfolge der Aktivitäten beschrieben. Kontrollkonnektoren werden dazu verwendet, Aktivitäten zu verbinden. Sie sind mit einer Übergangsbedingung versehen, die angibt, ob von einer Aktivität zur nächsten gegangen werden darf.

Neben der Abfolge der Aktivitäten wird in der Ablauforganisation der Datenfluss beschrieben. Daten können prinzipiell zwischen Aktivitäten oder von Aktivitäten zu Übergangsbedingungen fließen. Der Datenfluss des Kreditantragsprozesses ist in Abbildung 2.2 dargestellt.

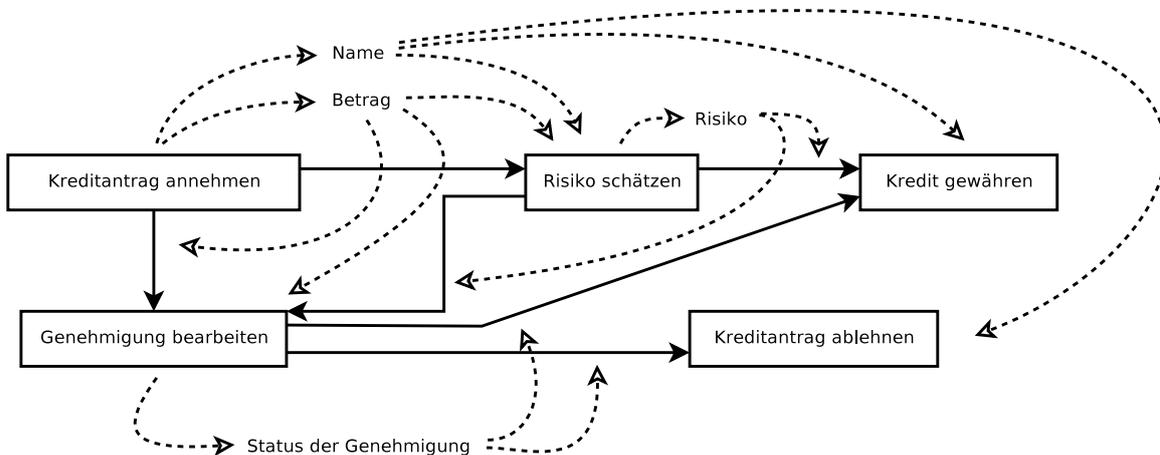


Abbildung 2.2.: Der Datenfluss des Illustrationsprozesses Kreditantrag

Falls bei einer Aktivität ein Fehler auftritt, kann die Behandlung dieses Fehlers durch die Modellierung eines Faulhandlers, der Aktivitäten zur Fehlerbehandlung enthält, modelliert werden. Es gibt Metamodelle, wie beispielsweise die im Abschnitt 2.3 vorgestellte ereignisgesteuerte Prozesskette (EPK), die keine Faulhandler kennen. Hier muss die Fehler-

behandlung im Hauptprozess modelliert werden. Im Falle der EPK beispielsweise durch ein Ereignis „Verarbeitung nicht erfolgreich“.

Es kann notwendig sein, im Prozessmodell anzugeben wie eine oder mehrere Aktivitäten kompensiert werden können. Eine Kompensation beschreibt, wie eine Aktivität rückgängig gemacht werden kann. Im Falle der EPK existiert kein Element, das immer die Beschreibung einer Kompensation anzeigt. Die Kompensation muss ausschließlich mit den Elementen modelliert werden, die auch der Beschreibung eines fehlerfreien und kompensationslosen Prozesses dienen. Die Kennzeichnung als Kompensation erfolgt außerhalb der Elemente der EPK, wie beispielsweise durch die Beschriftung oder durch die in Abschnitt 3.6.15 vorgestellten spezifischen Eigenschaften.

## 2.2. ARIS

ARIS steht für „Architektur integrierter Informationssysteme“ und wurde in [SNZ95] vorgestellt. Informationssysteme sind der informationsverarbeitende Teil eines Unternehmens (vgl. [Rum99, S. 55]). Die Gesamtsicht auf ein Informationssystem wird zur Bewältigung der Komplexität in die vier Sichten Organisationssicht, Funktionssicht, Datensicht und Steuerungssicht aufgeteilt. In der Organisationssicht wird die Organisationsstruktur des Unternehmens beschrieben. Die Funktionssicht enthält alle Funktionen, die als Aktivitäten in Geschäftsprozessen verwendet werden können. Funktionen verwenden Daten, die in der Datensicht zusammengefasst sind. In der Steuerungssicht werden die Bestandteile der anderen Sichten zu Geschäftsprozessen kombiniert.

In jeder Sicht werden drei Beschreibungsschichten unterschieden: das Fachkonzept, das Konzept der Datenverarbeitung (DV-Konzept) und die Implementierung. Die im Fachkonzept erstellten Dokumente dienen der Kommunikation innerhalb von Abteilungen, zwischen Abteilungen und als Schnittstelle zwischen Fachabteilung und IT-Abteilung. Deshalb enthalten Fachkonzepte technische Aspekte, jedoch nur wenige IT-Spezifika. Insbesondere sind Entscheidungen über die Umsetzung nicht enthalten. Falls beispielsweise durch ein Fachkonzept das Onlinebankingsystem einer Bank beschrieben wird, so kann darin die Forderung nach Java als Plattform und Implementierungssprache enthalten sein. Eine Beschreibung der zu entwickelnden Klassen, Methoden und der Interaktion der Instanzen der Klassen wird erst im DV-Konzept oder in der Implementierung gegeben. So ist ein Fachkonzept mit der im Wasserfallmodell des Software-Engineerings gebräuchlichen Spezifikation vergleichbar.

Im DV-Konzept wird eine Umsetzung des Fachkonzepts implementierungsnah beschrieben. An dieser Stelle werden bei der Umsetzung des Onlinebankingsystems die Klassen entworfen, Styleguides für die Implementierung gesetzt und auch die konkrete Laufzeitumgebung festgelegt. Im Wasserfallmodell entspricht diese Stufe dem Entwurf.

Die Realisierung in Software ist ein Bestandteil der Implementierung. Weiterhin gehören zu der Implementierung die Beschaffung von Hardware, der Test der Software, die Integration und der Systemtest.

Die Aufteilung der Gesamtsicht auf ein Informationssystem in vier verschiedene Sichten lässt sich grafisch als Haus darstellen (siehe Abbildung 2.3, vgl. [Sch95]). Die Daten- und Funktionssicht bilden die tragenden Wände, die durch Säulen dargestellt werden. Von ihnen wird das Dach des Hauses, die Struktur der Organisation, getragen. Ohne Daten und Funktionen ist jede Organisation unnötig, da nichts organisiert werden muss. Die Steuerung ist von der Organisation, den Daten und den Funktionen umgeben und schaltet diese zu einem Prozess zusammen.

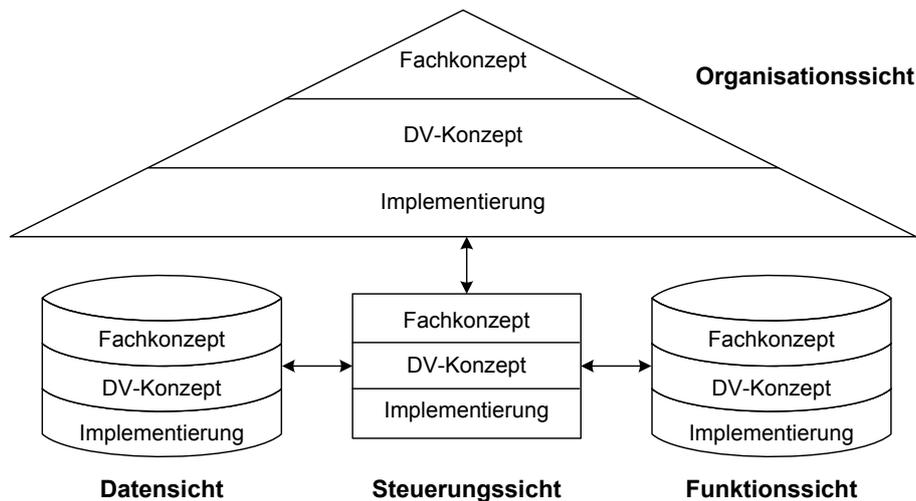


Abbildung 2.3.: Aris-Haus

### 2.3. Ereignisgesteuerte Prozessketten

Ereignisgesteuerte Prozessketten (EPKs) beschreiben den Ablauf eines Prozesses durch die Verbindung von Funktionen, Ereignissen und Operatoren. Sie wurden als Notation von Fachkonzepten der Steuerungssicht in [KNS92] eingeführt und dienen der Kommunikation zwischen Entscheidern sowie zwischen Entwicklern und Entscheidern.

In Abbildung 2.4 ist eine EPK eines beispielhaften Geschäftsprozesses zur Bearbeitung eines Kreditantrags abgebildet. In diesem Prozess beginnt die Abarbeitung bei der Funktion „Kredit Antrag annehmen“. Sobald diese Funktion ausgeführt wurde, ist das Ereignis „Kredit Antrag angenommen“ eingetreten. Der nachfolgende Operator zeigt an, dass eines der beiden Ereignisse eingetreten ist. Entweder ist der Betrag größer oder kleiner-gleich als 100 000 €. Falls der Betrag größer als 100 000 € ist, wird die Funktion zur Bearbeitung der Genehmigung ausgeführt. Falls der Betrag kleiner-gleich als 100 000 € ist, wird das Risiko der Kreditvergabe geschätzt. In dieser ereignisgesteuerten Prozesskette ist keine Information darüber enthalten, welche Daten fließen und von wem die Funktionen ausgeführt werden. Diese Information wird in erweiterter ereignisgesteuerter Prozesskette modelliert. Bevor diese im folgenden Abschnitt beschrieben wird, werden die Eigenschaften der Bestandteile in diesem Abschnitt genauer erläutert.

## 2.3. Ereignisgesteuerte Prozessketten

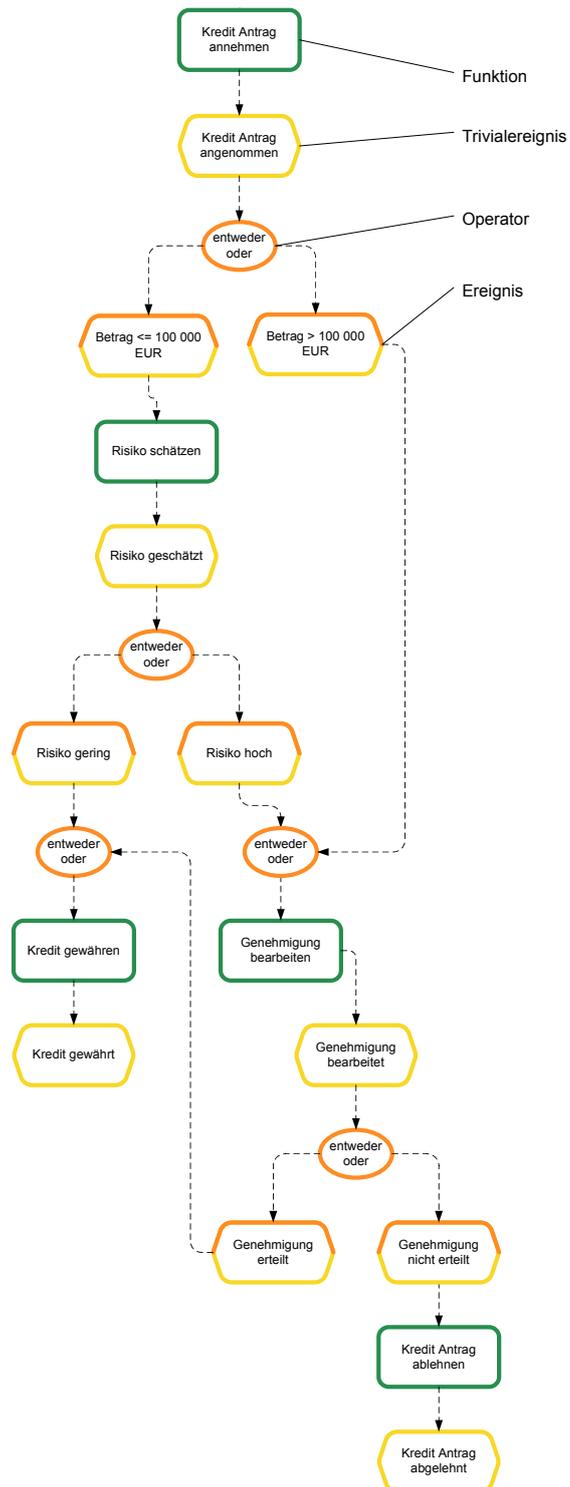


Abbildung 2.4.: EPK-Graphik eines Geschäftsprozesses zur Bearbeitung eines Kreditantrags

## 2. Geschäftsprozesse und deren Modellierung

---

Eine ereignisgesteuerte Prozesskette besteht aus Funktionen, Ereignissen und Operatoren. Die auszuführenden Aktivitäten werden durch die Funktionen beschrieben. Ereignisse bestimmen, wie durch die EPK navigiert wird. Ein Ereignis kann als Folge der Ausführung einer Funktion eintreten oder ein externes Ereignis darstellen. Eine syntaktische Unterscheidung dieser zwei Typen wird nicht getroffen. Die Operatoren dienen der Verknüpfung der Ereignisse und geben weiterhin an, welche Ereignisse eintreten können.

Falls ein Operator mehrere eingehende Kanten hat, wird er join-Operator genannt, da er den Kontrollfluss zusammenfügt. Gehen bei einem Operator mehrere Kanten aus, so wird er fork-Operator genannt, da er den Kontrollfluss auf mehrere Kanten verteilt. Ein Operator kann sowohl ein join- als auch ein fork-Operator sein.

Es gibt drei Arten von Operatoren: und-, oder- und entweder-oder-Operatoren. Diese Arten bestimmen die Semantik von fork- und join-Operatoren. Die Semantik wird ohne Beachtung der Fehlerfälle in Tabelle 2.1 für fork-Operatoren und in Tabelle 2.2 für join-Operatoren beschrieben.

---

| Operator      | Semantik  |
|---------------|---|
| und           | Alle nachfolgenden Pfade werden ausgeführt.               |
| oder          | Mindestens einer der nachfolgenden Pfade wird ausgeführt. |
| entweder-oder | Genau einer der nachfolgenden Pfade wird ausgeführt.      |

---

Tabelle 2.1.: Semantik von fork-Operatoren

---

| Operator      | Semantik  |
|---------------|---|
| und           | Der Knoten wird von allen eingehenden Kanten erreicht.  |
| oder          | Der Knoten wird von mindestens einer eingehenden Kante erreicht.  |
| entweder-oder | Der Knoten wird von genau einer der eingehenden Kanten erreicht.  |
| alle          | Der nachfolgende Knoten wird erst dann ausgeführt, sobald der Knoten von allen durch die Semantik vorgegebenen möglichen eingehenden Kanten erreicht wurde. |

---

Tabelle 2.2.: Semantik von Operatoren als join-Knoten

Bei einem und-Operator, der ein join-Operator ist, kann durch Betrachten der eingehenden Kanten entschieden werden, ob die ausgehenden Kanten verfolgt werden sollen. Dies ist bei dem Oder-Operator nicht der Fall, da erst dann weitergegangen werden kann, wenn keine eingehende Kante mehr ankommen kann. Dies kann jedoch nicht alleine anhand der eingehenden Kanten entschieden werden, sondern es müssen sämtliche Vorgänger überprüft werden. Analog dazu darf die Ausführung bei einem entweder-oder-join-Operator nur dann fortgesetzt werden, wenn der Kontrollfluss an genau einer Kante ankommt, weshalb auch hier sämtliche Vorgänger auf die Möglichkeit der Ausführung überprüft werden müssen. Diese Tatsache wird in [Rum99] als Nicht-Lokalität der oder- und entweder-oder-Operatoren bezeichnet.

Ein Konzept, das durch Nautilus eingeführt wurde, ist die Verarbeitung von zustandsbehafteten Objekten durch Funktionen. Jeder Funktion ist die Verarbeitungsinformation in Form der Angabe von Objekt und Verb zugeordnet. Jedes Ereignis besitzt ein zugehöriges Objekt und einen zugehörigen Zustand. Nach der Verarbeitung eines Objekts durch eine Funktion besitzt das Objekt ein aus dem Verb resultierendes Standardzustand. Falls auf dem Objekt „Kreditantrag“ die Aktion „annehmen“ durchgeführt wird, besitzt das Objekt „Kreditantrag“ danach den Zustand „angenommen“. Ganz unabhängig, ob der Antrag lückenhaft oder das Risiko der Kreditvergabe hoch ist. Deshalb wird das Ereignis, das direkt nach einer Funktion folgt auch „Trivialereignis“ genannt.

Die Verbindung zum Datenfluss wird durch die Namensgebung geschaffen. Hat ein Objekt den gleichen Namen wie ein Medium, so wird angenommen, dass es sich um das gleiche Element handelt. So wird von der Funktion „Kredit Antrag annehmen“ das Medium „Kredit Antrag“ angenommen. Ein Medium beschreibt die Form der Übermittlung von Information (vgl. Tabelle 3.5).

Die Benennung der Elemente erfolgt in „atomaren Begriffen“, wodurch Wiederverwendung von Begriffen ermöglicht wird. Es könnte beispielsweise auch ein Objekt „Kontoeröffnungsantrag“ geben. In atomare Begriffe zerlegt, ist der Name dieses Objekts Konto Eröffnung Antrag. Durch das Vorkommen von „Antrag“ wird gezeigt, dass es sich sowohl bei „Kredit Antrag“ als auch bei „Konto Eröffnung“ um einen „Antrag“ handelt. Aus welchen Komponenten ein Antrag besteht, kann von Fall zu Fall unterschiedlich sein und durch die im Abschnitt 3.6.15 vorgestellten spezifischen Eigenschaften modelliert werden.

Da Trivialereignisse immer nach der Ausführung einer Funktion eintreten, können sie in der Darstellung ereignisgesteuerter Prozessketten weggelassen werden. Weiterhin werden für eine kompakte Darstellung ereignisgesteuerter Prozessketten Operatoren durch die Symbole aus Tabelle 2.3 dargestellt. Für den Kreditantragsprozess aus Abbildung 2.4 ist die kompakte Darstellung in Abbildung 2.5 gezeigt.

| Operator      | Symbol   |
|---------------|----------|
| und           | $\wedge$ |
| oder          | $\vee$   |
| entweder-oder | $\times$ |

Tabelle 2.3.: Symbole der Operatoren

Die Darstellung eines Prozesses kann auf mehrere ereignisgesteuerte Prozessketten verteilt werden. Das Bindeglied bilden dabei die „Funktionswegweiser“. Ein Funktionswegweiser ist eine Funktion, die in einem Prozess verwendet wird, jedoch zu einem anderen Prozess gehört. In Abbildung 2.6 ist ein Prozess zur Kontoeröffnung dargestellt. In ihm wird auf den Akquisitionsprozess verwiesen, falls der Kunde das Angebot ablehnt. Die Funktion „Akquisition durchführen“ ist nicht Teil des Kontoeröffnungsprozesses, sondern des Akquisitionsprozesses. Aus der Sicht des in Abbildung 2.7 dargestellten Akquisitionsprozesses gehört die Funktion „Konto Angebot überwachen“ nicht zu dem Akquisitionsprozess,

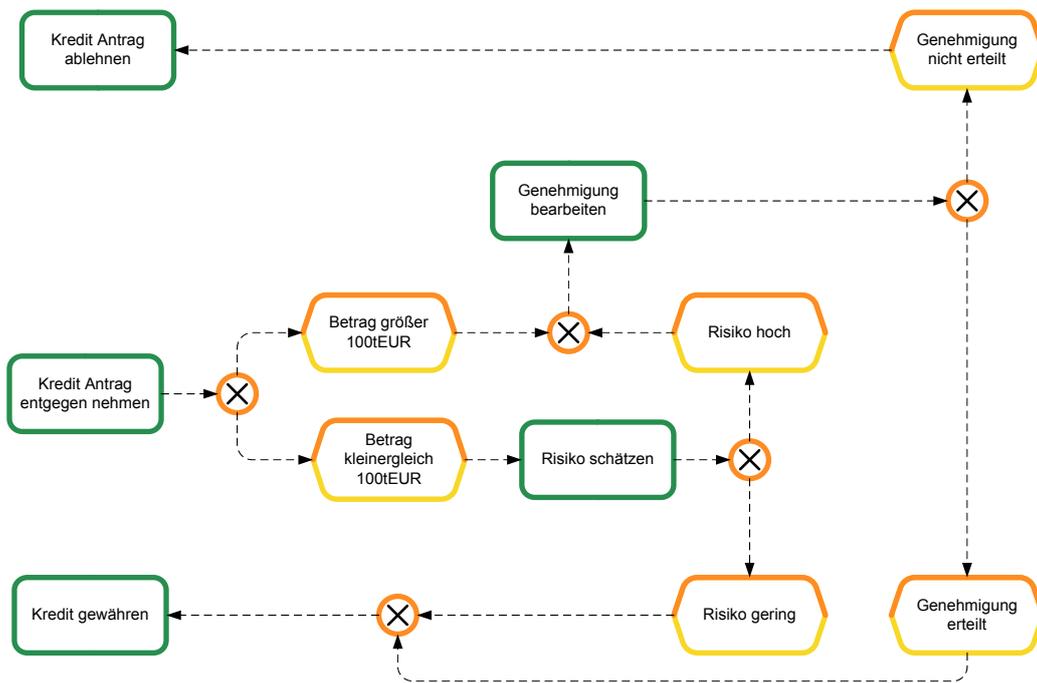


Abbildung 2.5.: Kompakte Darstellung der EPK aus Abbildung 2.4

sondern zu dem Kontoeröffnungsprozess. Funktionswegweiser stellen so die Prozessgrenze dar.

Ein Prozess kann durch mehrere ereignisgesteuerte Prozessketten dargestellt werden. Somit sind Funktionswegweiser die Verknüpfung zwischen den verschiedenen Teilen eines Prozesses und bilden keine Prozessgrenzen. Im Fall der Kontoeröffnung gehört die Akquisition zu dem Prozess der Kontoeröffnung und stellt keinen eigenständigen Prozess dar.

Mit EPKs kann Fehlerbehandlung und Kompensation modelliert werden. Es gibt keine Vorschrift, wie die dazu verwendeten Funktionen als Fehlerbehandlungs- oder Kompensationsfunktionen zu kennzeichnen sind. Deshalb ist es nur mit zusätzlichem Wissen möglich, für eine gegebene EPK zu entscheiden, welche Funktionen Fehlerbehandlungs- oder Kompensationsfunktionen sind.

### 2.3.1. Erweiterte Ereignisgesteuerte Prozessketten

In einer EPK wird der Kontrollfluss eines Prozesses spezifiziert. Der Kontrollfluss kann mit weiteren Informationen, wie dem Datenfluss oder der Einbindung in die Organisationsstruktur, versehen werden. Diese Annotation wird erweiterte ereignisgesteuerte Prozesskette (eEPK) genannt. Die möglichen Annotationen werden im Kapitel 3 vorgestellt. Als Illustration ist in Abbildung 2.3.1 der Informationsfluss des Kreditantragsprozesses als eEPK dargestellt.

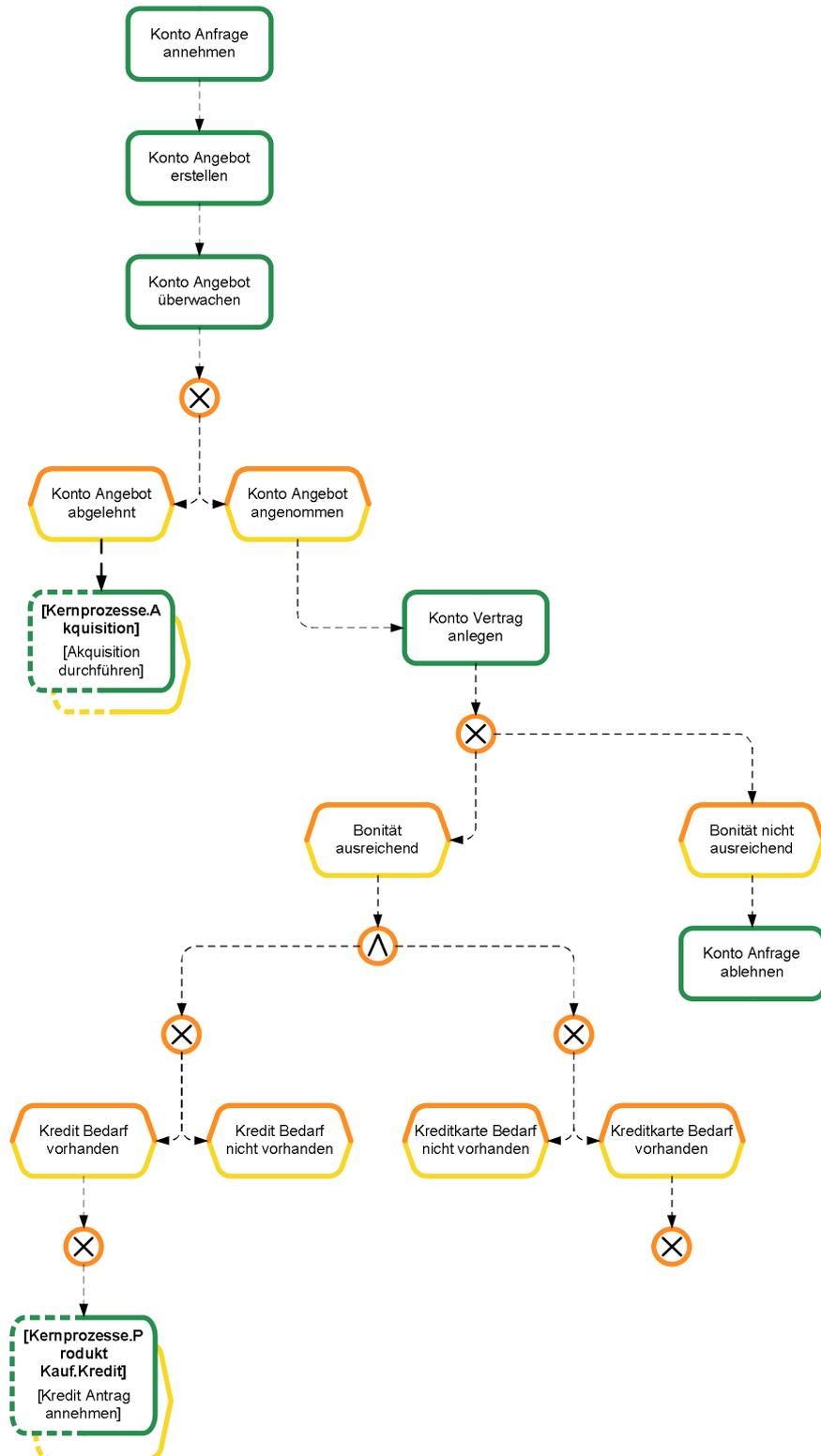


Abbildung 2.6.: Illustration eines Funktionswegweisers in einen anderen Prozess

## 2. Geschäftsprozesse und deren Modellierung

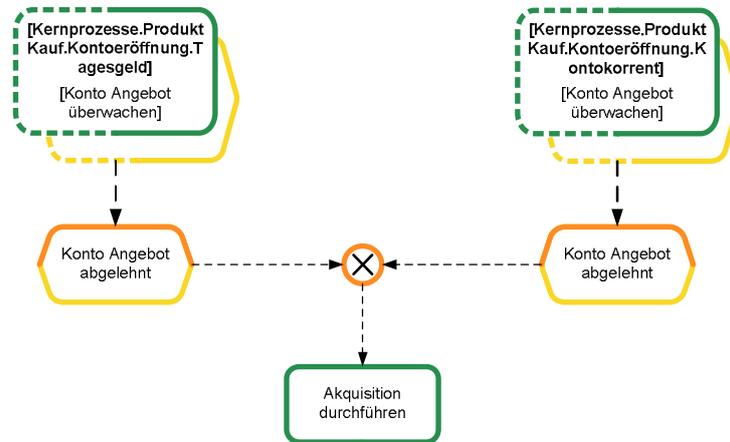


Abbildung 2.7.: Illustration des Auslöseimpulses für eine Funktion durch einen Funktionswegweiser

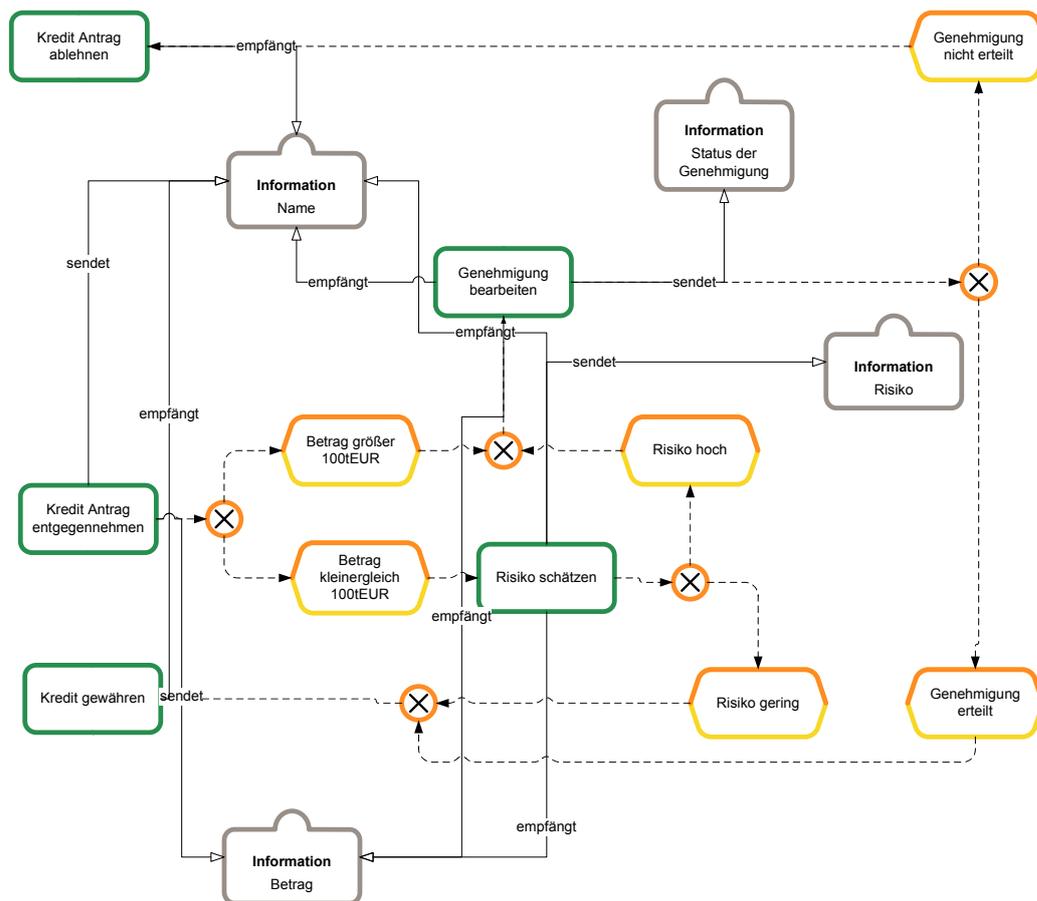


Abbildung 2.8.: Die eEPK des Kreditantragsprozesses (Abbildungen 2.1 und 2.2)

# METAMODELL VON NAUTILUS

---

Mit der Geschäftsprozessmodellierungssoftware Nautilus können Modelle der Geschäftsprozesse einer Firma erstellt werden. In einem Modell sind die Informationen aller Sichten des ARIS-Hauses enthalten.

Meistens ist ein solches Modell firmenspezifisch, jedoch können auch Prozesse anderer Firmen enthalten sein, falls beispielsweise die Interaktion zwischen zwei Geschäftspartnern modelliert werden soll.

Die Basis jedes Modells bilden ereignisgesteuerte Prozessketten, die den Ablauf eines Geschäftsprozesses beschreiben. Zu diesem Ablauf gehören weitere Informationen, wie die gesendeten und empfangenen Informationen, sowie die zugrunde liegende Organisationsstruktur.

Bisher war das Metamodell von Nautilus nur informal im Benutzerhandbuch beschrieben. Um dieses Modell als Grundlage für die Abbildung nach BPEL verwenden zu können, wird es in diesem Kapitel mit Hilfe von Graphen formalisiert. Da Graphen in der Literatur unterschiedlich formalisiert werden, wird am Anfang eine gebräuchliche Notation vorgestellt. Darauf aufbauend wird ein Graph zur Darstellung von ereignisgesteuerten Prozessketten definiert. Anschließend wird das Metamodell von Nautilus formal dargestellt und beschrieben, wie sich die zuvor als Graph formalisierten ereignisgesteuerten Prozessketten darin abbilden lassen.

### 3.1. Graphentheoretische Definitionen

Für die Formalisierung des Metamodelles werden exakte Definitionen von Graphen und ihren Eigenschaften benötigt. Die Graphen selbst werden in der Literatur unterschiedlich formalisiert und es gibt keine einheitliche Syntax. Deshalb folgt hier die Zusammenfassung der an der am Institut für Formale Methoden der Informatik gebräuchlichen Notation.

Ein Graph  $G$  besteht aus Knoten und Kanten. Die Knoten sind Elemente der Knotenmenge  $V$ , die Kanten Elemente der Kantenmenge  $E$ . Eine Kante  $e \in E$  verbindet genau zwei Knoten.

Ein Graph ist gerichtet, falls eine Kante von einem Knoten zu einem Knoten geht, also  $E \in V \times V$ . Dabei wird die erste Komponente des Tupels Quelle und die zweite Komponente Ziel der Kante genannt. In der Literatur werden weiterhin ungerichtete Graphen definiert, die jedoch für diese Arbeit nicht von Bedeutung sind.

Ereignisgesteuerte Prozessketten können durch einem gerichteten Graphen beschrieben werden: Funktionen, Ereignisse und Operatoren bilden hierbei die Knoten. Die Kanten der ereignisgesteuerten Prozesskette bilden die Kanten des Graphen.

Für die weitere Formalisierung von ereignisgesteuerten Prozessketten wird die Definition eines „schlingenfreien Graphen“ benötigt. Eine Kante von und zu dem selben Knoten heisst Schlinge. Man spricht von einem schlingenfreien Graphen, falls es keine Kante von und zu dem selben Knoten gibt, also  $\forall v \in V : (v, v) \notin E$  gilt.

In ereignisgesteuerten Prozessketten dürfen die Knoten nicht beliebig miteinander verbunden werden. Beispielsweise folgt einer Funktion immer ein Ereignis. Alle Bedingungen, die ein Graph erfüllen muss, damit er eine ereignisgesteuerte Prozesskette darstellt, sind in Abschnitt 3.4 aufgeführt und werden dort formalisiert. Bei dieser Formalisierung werden Aussagen über die Vorgänger und die Nachfolger getroffen. Deshalb muss es formale Ausdrucksmittel zur Bestimmung der Vorgänger und Nachfolger geben.

Zu einem Vorgänger führt immer eine Kante, denn sonst wäre es kein Vorgänger. Die Funktion  $\delta^- : V \mapsto E$  ordnet jedem Knoten die eingehenden Kanten zu.

$$\delta^-(v) = \{e | e \in E, \pi_2(e) = v\}.$$

Analog dazu ordnet die Funktion  $\delta^+ : V \mapsto E$  jedem Knoten die ausgehenden Kanten zu:

$$\delta^+(v) = \{e | e \in E, \pi_1(e) = v\}.$$

Die Vorgänger werden durch die Funktion  $adj^- : V \mapsto V$  bestimmt:

$$adj^-(v) = \{\pi_1(\delta^-(v))\}$$

$\delta^-(v)$  gibt die eingehenden Kanten zurück. Jede zurückgegebene Kante hat folglich den Knoten  $v$  als Ziel und einen anderen Knoten  $v'$  als Quelle. Jede Kante ist also von der Form  $(v', v)$ . Deshalb wird die Funktion  $\pi_1$  verwendet, die das erste Element eines Tupels zurück gibt. Dies kann auch für andere Elemente durchgeführt werden, so dass  $\pi_i$  die Projektion auf die  $i$ te Komponente eines Tupels bezeichnet.

Durch  $adj^- : V \mapsto V$  wurden die Vorgänger eines Knotens bestimmt. Analog dazu werden die Nachfolger eines Knotens durch die Funktion  $adj^+ : V \mapsto V$  bestimmt:

$$adj^+(v) = \{\pi_2(\delta^+(v))\}.$$

In einer ereignisgesteuerten Prozesskette können Funktionen und Ereignisse maximal einen Vorgänger und Nachfolger haben. Um diesen formal korrekt durch die Funktionen  $adj^-$  und  $adj^+$  zu erhalten, muss es eine Auswahl des einzigen Elements einer Menge geben. In der „Informal Compiler Algorithm Notation“ (ICAN) wird zur Lösung dieses

Problems der Operator „ $\blacklozenge$ “ eingeführt (vgl. [Mu97, S. 31].  $\blacklozenge$  wählt aus einer Menge ein zufälliges Element aus. Das Ergebnis von  $\blacklozenge\{2, 3, 4, 7\}$  kann 2, 3, 4 oder 7 sein.

Mit diesem Operator kann nun die Funktion  $adj_1^-(v)$  definiert werden. Falls ein Knoten  $v$  genau einen Vorgänger hat, so gibt  $adj_1^-(v)$  diesen zurück. Falls  $v$  keinen oder mehr als einen Vorgänger hat, so gibt  $adj_1^-(v)$  ein Epsilon ( $\varepsilon$ ) zurück:

$$adj_1^-(v) = \begin{cases} \blacklozenge(adj^-(v)) & \text{falls } |\delta^-(v)| = 1 \\ \varepsilon & \text{sonst} \end{cases}$$

Analog dazu bestimmt  $adj_1^+(v)$  den Nachfolger eines Knotens  $v$ , falls  $v$  genau einen Nachfolger hat:

$$adj_1^+(v) = \begin{cases} \blacklozenge(adj^+(v)) & \text{falls } |\delta^+(v)| = 1 \\ \varepsilon & \text{sonst} \end{cases}$$

Bei der Abbildung einer EPK nach BPEL muss der Graph, der die EPK darstellt, modifiziert werden. Zu der Modifikation gehört die Entfernung von Knoten und Kanten aus dem Graphen. Damit die Kanten klar identifizierbar sind, werden die Funktionen  $\delta_1^-(v)$  und  $\delta_1^+(v)$  eingeführt. Falls ein Knoten genau einen Vorgänger hat, bestimmt  $\delta_1^-(v)$  für diesen die Kante zu dem Vorgänger. Analog dazu bestimmt  $\delta_1^+(v)$  für einen Knoten mit genau einem Nachfolger die Kante zu diesem Nachfolger. Falls es keinen oder mehr als einen Nachfolger gibt, wird  $\varepsilon$  zurückgegeben.

$$\delta_1^-(v) = \begin{cases} \blacklozenge(\delta^-(v)) & \text{falls } |\delta^-(v)| = 1 \\ \varepsilon & \text{sonst} \end{cases}$$

$$\delta_1^+(v) = \begin{cases} \blacklozenge(\delta^+(v)) & \text{falls } |\delta^+(v)| = 1 \\ \varepsilon & \text{sonst} \end{cases}$$

In manchen Fällen wird nur eine Teilmenge der Knoten eines Graphen  $G = (V, E)$  betrachtet. Dabei wird eine Knotenmenge  $\tilde{V}$  vorgegeben, auf die der Graph eingeschränkt werden soll. Die Kantenmenge  $E$  wird automatisch zu  $E_{[\tilde{V}]}$  reduziert, wobei die Kanten aus  $E$  entfernt werden, deren Start- oder Zielknoten nicht in  $\tilde{V}$  sind:

Sei  $G = (V, E)$  ein Graph, dann ist  $G' = (V_{[\tilde{V}]}, E_{[\tilde{V}]})$  der auf  $\tilde{V}$  eingeschränkte Graph.

$$E_{[\tilde{V}]} = \left\{ (v_1, v_2) \mid v_1, v_2 \in \tilde{V}, (v_1, v_2) \in E \right\}$$

$[\tilde{V}]$  bezeichnet dabei das Einschränken der ursprünglichen Knotenmenge  $V$  auf  $\tilde{V}$ . Die Kantenmenge  $E'$  muss dabei nicht vorgegeben werden, sondern wird wie oben bestimmt. In der Fachliteratur wird der Graph  $G' = (V_{[\tilde{V}]}, E_{[\tilde{V}]})$  auch „induzierter Teilgraph“ genannt. „induziert“ wird deshalb benutzt, da sich die Kantenmenge aus der Knotenmenge ergibt und diese nicht weiter reduziert wird.

### 3. Metamodell von Nautilus

---

Weiterhin ist es wünschenswert, die Betrachtung auf Knoten  $\tilde{V}$  und Kanten  $\tilde{E}$  einzuschränken:  $\tilde{G} = (V_{[\tilde{V}]}, E_{[\tilde{E}]})$ . Dabei wird zuerst der Graph  $G$  auf  $G' = (V_{[\tilde{V}]}, E')$  reduziert und  $E_{[\tilde{E}]}$  durch den Schnitt von  $E'$  mit  $\tilde{E}$  gebildet:  $E_{[\tilde{E}]} = E' \cap \tilde{E}$ .

Die Funktionen zur Zuordnung der ein- und ausgehenden Kanten werden für den Fall der Einschränkung wie folgt definiert:

$$\begin{aligned}\delta_{[\tilde{E}]}^-(v) &= \delta^-(v) \cap \tilde{E} \\ \delta_{[\tilde{E}]}^+(v) &= \delta^+(v) \cap \tilde{E}\end{aligned}$$

Analog werden die Nachbarschaftsfunktionen für den eingeschränkten Fall definiert:

$$\begin{aligned}adj_{[\tilde{V}]}^+(v) &= adj^-(v) \cap \tilde{V} \\ adj_{[\tilde{V}]}^-(v) &= adj^+(v) \cap \tilde{V}\end{aligned}$$

$adj_{[\tilde{V}]}^*(v)$  bestimmt  $adj^*(V)$  auf  $G_{[\tilde{V}]}$ .

Neben den Nachbarn können in einem Graphen auch „Wege“ beschrieben werden. Ein Weg hat einen Knoten  $v_1$  als Start und einen Knoten  $v_l$  als Ziel. Weiterhin sind in ihm alle Knoten enthalten, die auf dem Weg von  $v_1$  nach  $v_l$  liegen. Formal wird ein Weg durch ein Tupel ausgedrückt. Die Länge des Tupels entspricht der Länge des Wegs. Das erste Tupelelement ist der erste Knoten auf dem Weg, das zweite der zweite bis hin zum letzten Tupelelement, das den letzte Knoten darstellt. Formal ausgedrückt:

In einem Graphen  $G = (V, E)$  ist ein Weg ein Tupel aus Knoten aus  $G$ , wobei die in dem Tupel direkt benachbarten Knoten auch im Graphen benachbart sind: Sei  $(v_1, v_2, \dots, v_l)$  ein Weg, dann gilt

$$\begin{aligned}v_k &\in V, 1 \leq k \leq l \\ (v_i, v_{i+1}) &\in E, 0 < i < l\end{aligned}$$

Mit  $way(v, w)$  wird der Weg von  $v$  nach  $w$  bezeichnet.  $way(v, w)$  ist  $\epsilon$ , falls es keinen Weg von  $v$  nach  $w$  gibt. Ansonsten ist  $way(v, w)$  ein Tupel, das jeden Knoten des Wegs, einschließlich  $v$  und  $w$  enthält. Die Reihenfolge der Elemente des Tupels ist die, wie die Knoten von  $v$  aus auf dem Weg nach  $w$  besucht werden.

Von einem Knoten  $v$  zu einem Knoten  $w$  kann es keinen, einen oder viele Wege geben. Falls die Frage gestellt wird, ob es von einem Knoten  $v$  zum selben Knoten  $v$  immer einen Weg gibt, kann die Antwort je nach Definition eines Wegs „ja“ oder „nein“ lauten. „ja“ lautet sie dann, wenn erlaubt wird, dass bei einem Gang eines Wegs könnte auch keine Kante verwendet werden darf. Das zugehörige Tupel ist dann das leere Tupel  $()$ . „nein“ lautet die Antwort dann, falls nur nicht-leere Wege erlaubt sind. Im Folgenden sollen Wege nur nicht-leere Wege bezeichnen, weshalb es nur dann einen Weg von einem Knoten  $v$  zum selben Knoten  $v$  gibt, falls  $v$  mit sich selbst durch Kanten verbunden ist.

Falls es viele Wege gibt, gibt es darunter einen kürzesten Weg. Ein „kürzester Weg“ ist ein Weg  $W$  von einem Knoten  $v$  zu einem Knoten  $w$ , so dass es keinen anderen Weg  $W'$

gibt, der kürzer ist. Die Länge eines Wegs bemisst sich durch die Anzahl der Knoten, die auf dem Weg liegen. Von einem Knoten  $v$  zu einem Knoten  $w$  kann es mehrere kürzeste Wege geben. Die Funktion  $way_s(x, y)$  dient zur Bestimmung eines kürzesten Wegs von dem Knoten  $x$  zu dem Knoten  $y$ . Die Auswahl aus allen kürzesten Wegen  $minways(x, y)$  von  $x$  nach  $y$  erfolgt beliebig.

$$\begin{aligned} ways(x, y) &= \{(v_1, \dots, v_n) \mid (v_1, \dots, v_n) = way(x, y)\} \\ minways(x, y) &= \{(v_1, \dots, v_m) \mid ((v_1, \dots, v_n) \in ways(x, y)) \wedge \\ &\quad (\forall (v_1, \dots, v_p) \in ways(x, y) : p \geq m)\} \end{aligned}$$

$$way_s(x, y) = \begin{cases} \varepsilon & \text{falls } minways(x, y) = \emptyset \\ \blacklozenge(minways(x, y)) & \text{sonst} \end{cases}$$

Das Metamodell von Nautilus erlaubt es, dass eine EPK, und so auch ihr Graph, nicht zusammenhängend ist. Der Graph zerfällt so in mehrere Zusammenhangskomponenten. Eine Zusammenhangskomponente ist ein Teil eines Graphen, bei dem jeder Knoten mit jedem anderen über einen Weg verbunden ist. Bei einer Verbindung wird die Richtung der Kanten nicht beachtet. Eine Zusammenhangskomponente ist immer maximal. Dies bedeutet, dass es keinen weiteren Knoten im Graphen gibt, die mit den Knoten aus einer Zusammenhangskomponente verbunden sind. Formal ist eine Zusammenhangskomponente ein eingeschränkter Graph  $G'$ , dessen Knoten im Originalgraphen verbunden sind. Die Funktion  $cc(v)$  dient zur bestimmt einer Zusammenhangskomponente, in der der Knoten  $v$  enthalten ist:

$$\begin{aligned} G &= (V, E) \\ cc(v) &= G(V', E_{|V'}), v \in V \\ V' &= \{w \mid \exists way(v, w) \vee \exists way(w, v), w \in V\} \end{aligned}$$

Falls ein Graph nicht in Zusammenhangskomponenten zerfällt, so besteht er aus genau einer Zusammenhangskomponente und wird zusammenhängend genannt.

Ereignisgesteuerte Prozessketten können mit den bis jetzt vorgestellten Formalismen als gerichteter Graph  $G = (V, E)$  formalisiert werden. Die Funktionen und Ereignisse der ereignisgesteuerte Prozesskette des Kreditantragsprozess aus Abbildung 2.5 sind beschriftet. In der Graphentheorie wird die Beschriftung durch die Funktion  $\iota$  formalisiert. Jeder Knoten und jeder Kante kann mittels der Labelfunktion  $\iota : V \cup E \mapsto \mathbf{L}$  mit einer Beschriftung versehen werden.  $\mathbf{L}$  beschreibt die gewählten Labels. Zur Vereinfachung sei  $\iota_{|V}$  der Teil von  $\iota$ , der die Knoten beschriftet und  $\iota_{|E}$  der Teil, der die Kanten beschriftet:

$$\begin{aligned} \iota_{|V}(v) &= \iota(v), v \in V \\ \iota_{|E}(e) &= \iota(e), e \in E \end{aligned}$$

Um die Lesbarkeit von Aussagen über  $\iota$  zu vereinfachen, soll die Aussage  $\iota_{|X} \mapsto Y$  folgende Bedeutung besitzen:

$$\forall x \in X : \iota(x) = y, y \in Y$$

Die Definition eines gelabelten Graphen ergibt sich zu einem Dreiertupel aus den Knoten, Kanten und der Labelfunktion:  $G = (V, E, \iota)$ .

## 3.2. Kompakte Graphen

Im Metamodell gibt es zu jeder Kante von einem Knoten zu einem anderen Knoten immer eine Kante, die die Knoten in entgegengesetzter Richtung verbindet. Damit nicht beide Kanten angegeben werden müssen, werden diese zu einer kompakten Kante zusammengefasst. Der so entstandene Graph wird „kompakter Graph“ genannt. Da die Kanten im Metamodell beschriftet sind, wird die kompakte Darstellung nur für beschriftete Kanten definiert.

Eine kompakte Kante enthält den Quell- und den Zielknoten. Sowohl die Beschriftung der Kante von dem Quell- zu dem Zielknoten als auch die Beschriftung der Kante von dem Ziel- zu dem Quellknoten sind in einem Tripel zusammengefasst:

|                          | Tripelement | Bedeutung                                  |
|--------------------------|-------------|--|
| $(v_1, v_2, (l_1, l_2))$ | $v_1$       | Quellknoten der Vorwärtskante, $v_1 \in V$ |
|                          | $v_2$       | Zielknoten der Vorwärtskante, $v_2 \in V$  |
|                          | $l_1$       | Beschriftung der Vorwärtskante             |
|                          | $l_2$       | Beschriftung der Rückwärtskante            |

Das Tupel  $(l_1, l_2)$  wird auch Beschriftungstupel genannt.

Die kompakte Kantenmenge eines Graphen ist somit:

$$E_k = \bigcup \{ (v_i, v_j, (l_i, l_j)) \mid v_i, v_j \in V \}$$

Die kompakte Labelfunktion eines kompakten Graphen  $G_k = (V_k, E_k, \iota_k)$  beschriftet nur die Knoten, da die Kantenbeschriftungen in der kompakten Kantenmenge enthalten ist.

Ein gerichteter, gelabelter Graph  $G = (V, E, \iota)$  ist zu einem kompakten Graphen  $G_k = (V_k, E_k, \iota_k)$  dual, falls folgende Eigenschaften erfüllt sind:

$$\begin{aligned} V &= V_k \\ |E| &= 2 * |E_k| \\ \forall e_k \in E_k, e_k = (v_i, v_j, (l_i, l_j)), e_i = (v_i, v_j), e_j = (v_j, v_i) : \\ e_i, e_j &\in E \\ \iota(v_i) &= \iota_k(v_i) \\ \iota(v_j) &= \iota_k(v_j) \\ \iota(e_i) &= l_i \\ \iota(e_j) &= l_j \end{aligned}$$

Weiterhin müssen  $\iota$  und  $\iota_k$  minimale Funktionen sein:  $\iota$  muss auf  $V \cup E$  und  $\iota_k$  auf  $E_k$  definiert sein. Anderenfalls wäre die Zuordnung nicht eindeutig und somit auch nicht dual.

### 3.3. Die Tiefensuche

Der Algorithmus der Tiefensuche ist in vielen Lehrbüchern beschrieben. Bei der Abbildung des Metamodells nach BPEL wird die Tiefensuche häufig verwendet, so dass hier ein kurzer Überblick über die wesentlichen Bestandteile gegeben wird.

Die Tiefensuche ist eine bestimmte Art, einen Graphen zu traversieren. Sie wird typischerweise rekursiv umgesetzt. Jede rekursive Funktion, die die Tiefensuche realisiert, hat folgende Eigenschaften:

**Abbruchbedingung des rekursiven Selbstaufrufs** Falls die Tiefensuche zum Finden eines Knotens mit einer bestimmten Eigenschaft verwendet wird, wird diese mit der Bedingung umgesetzt. Die Traversierung wird dann nicht bei den Kindern fortgesetzt. Falls keine Bedingung angegeben wird, bricht die Rekursion bei den Knoten ohne ausgehende Kanten ab.

Im Falle der Traversierung eines Graphen wird jeder bearbeitete Knoten als besucht markiert. Der rekursive Selbstaufruf wird vor oder nach dem Besuchen eines bereits bearbeiteten Knoten abgebrochen.

**Bearbeitung des aktuellen Knoten** Der aktuelle Knoten wird bearbeitet. Dies kann beispielsweise das Schreiben in eine Liste oder das Ändern der Beschriftung sein. Bei der Traversierung eines Graphen wird der Knoten als besucht markiert, um eine doppelte Verarbeitung zu vermeiden und die Terminierung der Tiefensuche zu gewährleisten.

**Rekursiver Selbstaufruf** Die Tiefensuche wird mit jedem Kind als zu besuchenden Knoten rekursiv aufgerufen

### 3.4. Formalisierung der EPKs

Mit Hilfe der Beschriftung können ereignisgesteuerte Prozessketten formalisiert werden. Da jeder Knoten einer EPK neben der Beschriftung noch einen Typ besitzt, ist jede EPK ein Fünftupel  $(V, E, \iota, \mathbb{T}, \mathbb{T}_{OP})$ .  $(V, E, \iota)$ ,  $(V, E, \mathbb{T})$ ,  $(V, E, \mathbb{T}_{OP})$  bilden formal jeweils einen gerichteten, beschrifteten Graphen.  $\iota$  bestimmt zu jedem Knoten die Beschriftung und  $\mathbb{T}$  zu jedem Knoten den Typ. Ein Knoten kann entweder ein Funktionswegweiser, eine Funktion, ein Trivialereignis, ein Ereignis oder ein Operator. Jeder Operator selbst ist entweder ein

oder-, und- oder entweder-oder-Operator (vgl. 2.3). Die Zuordnung zu einem dieser Typen wird durch die Funktion  $\mathbb{T}_{\text{OP}}$  vorgenommen.

$$\begin{aligned}
 K &= (V, E, \iota, \mathbb{T}, \mathbb{T}_{\text{OP}}) \\
 (V, E) &: \text{gerichteter Graph} \\
 \iota &: V \mapsto L(G_2) \\
 \mathbb{T} &: V \mapsto \{\mathbf{function}\uparrow, \mathbf{function}, \mathbf{trivialevent}, \mathbf{event}, \mathbf{operator}\} \\
 \mathbb{T}_{\text{OP}} &: V \mapsto \{\ominus, \otimes, \otimes\} \\
 \mathbb{T}_{\text{OP}}(v) &= \begin{cases} \ominus & \text{falls } v \text{ und-Operator} \\ \otimes & \text{falls } v \text{ oder-Operator} \\ \otimes & \text{falls } v \text{ entweder-oder-Operator} \end{cases}
 \end{aligned}$$

$\iota$  realisiert die Beschriftung. Eine Beschriftung besteht aus Wörtern ohne Zeilenumbruch. Diese Menge wird von der Grammatik  $G_2$  erzeugt, die in Abbildung B.2 definiert ist.

Nicht jeder Graph, der aus Funktionswegweisern, Funktionen, Trivialereignissen, Ereignissen und Operatoren besteht, stellt eine gültige EPK dar. Damit ein Graph  $G = (V, E, \iota, \mathbb{T}, \mathbb{T}_{\text{OP}})$  eine gültige EPK darstellt, müssen folgende Bedingungen erfüllt sein:

1. Jede Funktion hat genau einen Nachfolger. Dieser Nachfolger ist ein Trivialereignis:

$$(\mathbb{T}(v) = \mathbf{function}) \Rightarrow (|\delta^+(v)| = 1) \wedge (\mathbb{T}(\mathit{adj}_1^+(v)) = \mathbf{trivialevent})$$

2. Funktionswegweiser sind besondere Funktionen und bilden die Grenze des Prozesses. Deshalb haben sie entweder genau einen Vorgänger oder genau einen Nachfolger. Dieser Nachfolger ist dann ein Trivialereignis.

$$\begin{aligned}
 (\mathbb{T}(v) = \mathbf{function}\uparrow) &\Rightarrow \\
 &((|\delta^-(v)| = 1) \wedge (|\delta^+(v)| = 0)) \vee \\
 &((|\delta^-(v)| = 0) \wedge (|\delta^+(v)| = 1) \wedge (\mathbb{T}(\mathit{adj}_1^+(v)) = \mathbf{trivialevent}))
 \end{aligned}$$

3. Ein Ereignis besitzt maximal einen Vorgänger und maximal einen Nachfolger:

$$(\mathbb{T}(v) = \mathbf{event}) \Rightarrow (|\delta^-(v)| \leq 1) \wedge (|\delta^+(v)| \leq 1)$$

4. Ereignisse werden durch Operatoren verknüpft oder lösen eine Funktion aus:

$$\begin{aligned}
 (\mathbb{T}(v) = \mathbf{event}) &\Rightarrow \\
 &((|\delta^-(v)| = 1) \Rightarrow (\mathbb{T}(\mathit{adj}_1^-(v)) = \mathbf{operator})) \wedge \\
 &((|\delta^+(v)| = 1) \Rightarrow ((\mathbb{T}(\mathit{adj}_1^+(v)) = \mathbf{operator}) \vee \\
 &\quad (\mathbb{T}(\mathit{adj}_1^+(v)) = \mathbf{function}) \vee \\
 &\quad (\mathbb{T}(\mathit{adj}_1^+(v)) = \mathbf{function}\uparrow)))
 \end{aligned}$$

5. Ein Trivialereignis besitzt eine Funktion oder einen Funktionswegweiser als Vorgänger und maximal einen Nachfolger. Dieser Nachfolger ist dann ein Funktionswegweiser, eine Funktion oder ein Operator.

$$\begin{aligned}
 (\mathbb{T}(v) = \mathbf{trivialevent}) \Rightarrow & \\
 & ((|\delta^-(v)| = 1) \wedge ((\mathbb{T}(adj_1^-(v)) = \mathbf{function}) \vee (\mathbb{T}(adj_1^-(v)) = \mathbf{function}\uparrow))) \wedge \\
 & (((|\delta^+(v)| = 1) \wedge ((\mathbb{T}(adj_1^+(v)) = \mathbf{function}) \vee (\mathbb{T}(adj_1^+(v)) = \mathbf{function}\uparrow)) \vee \\
 & \quad (\mathbb{T}(adj_1^+(v)) = \mathbf{operator}))) \vee \\
 & (|\delta^+(v)| = 0)
 \end{aligned}$$

6. Bei geschachtelten Bedingungen (vgl. Abbildung 3.1) entscheiden Ereignisse, welche Zweige genommen werden und welche Funktionen schließlich ausgeführt werden. Die Ausnahme bilden die und-Operatoren. Da bei ihnen alle Nachfolger ausgeführt werden, muss auf dem Weg zu einer Funktion kein Ereignis besucht werden.

$$\forall f \in V : (\mathbb{T}(f) = \mathbf{function}) \vee (\mathbb{T}(f) = \mathbf{function}\uparrow) \Rightarrow \mathit{cond}(adj^-(f))$$

$$\mathit{cond}(V) = \begin{cases} \text{wahr} & \text{falls } V = \emptyset \\ \forall k \in V : \mathit{cond}'(k) & \text{sonst} \end{cases}$$

$$\mathit{cond}'(k) = \begin{cases} \text{wahr} & \text{falls } \mathbb{T}(k) = \mathbf{event} \\ \mathit{cond}(adj^-(k)) & \text{falls } |adj^+(k)| = 1 \\ \mathit{cond}(adj^-(k)) & \text{falls } (|adj^+(k)| > 1) \wedge (\mathbb{T}_{\text{OP}}(k) = \emptyset) \\ \text{falsch} & \text{sonst} \end{cases}$$

Eine Folgerung aus dieser Regel ist, dass nach entweder-oder- und oder-Operatoren mit mehreren Nachfolgern keine Funktion unter den Nachfolgern sein kann.

7. In einem Prozess werden Aktivitäten durch Funktionen ausgeführt. Ereignisse und Operatoren entscheiden, welche Funktionen ausgeführt werden. Deshalb sind Ereignisse und Operatoren immer mit Funktionen verknüpft:

$$\forall v \in V, \mathbb{T}(v) = (\mathbf{operator} \vee \mathbf{event}) \exists f (\exists \text{way}(v, f) \vee \exists \text{way}(f, v))$$

XOR - AND - p1 z1  
           - p2 z2  
   - OR - p3 z3  
           - p4 z4

Abbildung 3.1.: Eine geschachtelte Bedingung

Falls alle Bedingungen zutreffen, wird  $G = (V, E, \iota, \mathbb{T}, \mathbb{T}_{\text{OP}})$  „EPK-Graph“ genannt.

### 3.5. Elemente jedes Modells

Mit Element werden Knoten des Modellgraphen bezeichnet, die einen Elementtyp besitzen. Jedes Element besitzt genau einen Elementtyp. Die Elementtypen selbst lassen sich in fünf Kategorien unterteilen:

- Grundlegende Elemente (Tabelle 3.1)
- Elemente der ereignisgesteuerten Prozesskette (EPK, Tabelle 3.2)
- Elemente der erweiterten ereignisgesteuerten Prozesskette (eEPK)
  - Tätigkeiten (Tabelle 3.3)
  - Elemente des Datenflusses (Tabelle 3.4)
  - Elemente zur Beschreibung (Tabelle 3.5)
- Elemente zur Strukturierung des Modells (Tabelle 3.6)
- Elemente der Aufbauorganisation (Tabelle 3.7)

---

| Elementtyp     | Beschreibung  |
|----------------|---|
| <b>Begriff</b> | Die Begriffe bilden die Basis für das Labeling, was im Abschnitt 3.6.2 erläutert wird.                                |
| <b>Objekt</b>  | Ein Objekt ein Element eines Geschäftsprozesses, das von Funktionen bearbeitet wird und das einen Zustand haben kann. |
| <b>Verb</b>    | Ein Verb beschreibt, was mit einem Objekt durch eine Funktion durchgeführt wird.                                      |
| <b>Zustand</b> | Ein Zustand beschreibt, welchen Zustand ein Objekt haben kann.  |

---

Tabelle 3.1.: Grundlegende Elemente

| Elementtyp      | Beschreibung  |
|-----------------|---|
| <b>Ereignis</b> | <p>Ein Ereignis stellt nach [KNS92] einen eingetretenen betriebswirtschaftlichen Zustand dar, der durch Ausführung bestimmter Arbeitsschritte eingetreten ist oder bestimmte zukünftige Arbeitsschritte auslöst.</p> <p>Ein Ereignis kann als Übergangsbedingung zwischen zwei Funktionen oder als externes Ereignis, das unabhängig von den Vorgänger-Funktionen und -Ereignissen ist, angesehen werden.</p> |
| <b>Funktion</b> | <p>Eine Funktion stellt eine Aktivität auf einem abstrakten Level dar. Die genau durchzuführenden Schritte werden bei den zugehörigen Tätigkeiten angegeben.</p>  |
| <b>Operator</b> | <p>Operatoren sind Verknüpfungspunkte in einem Geschäftsprozess. Es gibt „oder“- , „und“- und „entweder-oder“-Operatoren. Damit werden Verzweigungen und Zusammenführungen in einem Geschäftsprozess realisiert. Näheres ist in Abschnitt 2.3 ausgeführt.</p>   |

Tabelle 3.2.: Elemente der ereignisgesteuerten Prozesskette

| Elementtyp       | Beschreibung  |
|------------------|---|
| <b>Tätigkeit</b> | <p>Tätigkeiten beschreiben Teilschritte innerhalb einer Funktion. Sie gehören nicht zu der ursprünglichen Definitionen einer EPK, weshalb sie zu der erweiterten ereignisgesteuerten Prozesskette gezählt werden.</p> |

Tabelle 3.3.: Das Element Tätigkeit

### 3. Metamodell von Nautilus

| Elementtyp         | Beschreibung   |
|--------------------|--|
| <b>Beleg</b>       | Belege stellen Informationsträger dar, die physischen Charakter haben und Teil des Datenflusses sind. Beispiele für Belege sind ein Angebot, eine Rechnung oder ein Vertrag.   |
| <b>Dokument</b>    | Dokumente stellen Schriftstücke dar. Im Unterschied zu Belegen sollen diese bei der Modellierung nicht im Datenfluss eines Prozesses verwendet werden. Jedoch können Modelle erstellt werden, in denen über Dokumente Daten ausgetauscht werden. Beispiele für Dokumente sind Gesetzestexte, Formatvorlagen und Verfahrensanweisungen. |
| <b>Information</b> | Informationen sind Teile des Datenflusses und haben keinen physikalischen Charakter, wie beispielsweise eine Kontonummer oder ein Benutzername.  |
| <b>Ware</b>        | Waren sind Bestandteile des Datenflusses und stellen Erzeugnisse eines Unternehmens dar, wie beispielsweise Paletten oder Reifen.  |

Tabelle 3.4.: Elemente des Datenflusses

| Elementtyp           | Beschreibung  |
|----------------------|---|
| <b>Arbeitsmittel</b> | Arbeitsmittel sind Mittel, die zur Durchführung einer Aktivität benötigt werden. Dies kann beispielsweise konkrete Hard- oder Software sein, aber auch Werkzeuge wie beispielsweise ein Hammer.                       |
| <b>Frage</b>         | Fragen werden eingesetzt, um während der Modellierung offene Punkte zu markieren.   |
| <b>Medium</b>        | Ein Medium beschreibt in welcher Form Belege, Dokumente, Informationen oder Waren übermittelt werden. Eine Kontonummer kann beispielsweise auf einem Papierformular oder in einem PDF-Dokument enthalten sein.        |
| <b>Potenzial</b>     | Mit einem Potenzial werden im Modell Verbesserungsmöglichkeiten dokumentiert.   |
| <b>Verfahren</b>     | Verfahren dienen zur Darstellung von Detailinformationen, wie beispielsweise zeitliche Information (wöchentlich), Art der Durchführung (automatisch) oder einer weiteren Detaillierung einer Tätigkeit (LKW beladen). |
| <b>Ziel</b>          | Ein Ziel wird zur Verdeutlichung der Zielsetzung eines Ablaufs eingesetzt, wie beispielsweise die Kundenzufriedenheit sicherzustellen.  |

Tabelle 3.5.: Elemente zur Beschreibung anderer Elemente

| Elementtyp                | Beschreibung   |
|---------------------------|--|
| <b>Geschäftsprozess</b>   | Die EPK bildet den Geschäftsprozess. Ein Geschäftsprozess ist die Zusammenstellung von Funktionen durch Ereignisse und Operatoren zu einem Geschäftsprozess.   |
| <b>Gliederungselement</b> | Durch Gliederungselemente werden Geschäftsprozesse gegliedert. Ein Gliederungselement kann beliebig viele Geschäftsprozesse oder weitere Gliederungselemente enthalten.  |
| <b>Modellelement</b>      | Das Modellelement existiert genau ein Mal pro Modell und dient der Repräsentation des gesamten Modells.  |
| <b>Überbrückungen</b>     | In den Grafiken können Elemente mit einer Überbrückung überbrückt werden, so dass nur das Anfangs- und Endelement der Überbrückung zu sehen ist. Diese Überbrückung dient der Vereinfachung der Darstellung von Prozessen und wird in der Oberfläche von Nautilus nicht reflektiert. |

Tabelle 3.6.: Elemente zur Strukturierung des Modells

| Elementtyp                  | Beschreibung   |
|-----------------------------|--|
| <b>Mitarbeiter</b>          | Ein Mitarbeiter repräsentiert eine konkrete Person, die an den Aktivitäten beteiligt ist.  |
| <b>Organisationseinheit</b> | Eine Organisationseinheit repräsentiert einen Teil eines Unternehmens, wie beispielsweise eine Abteilung, einen Bereich oder einen Standort. |
| <b>Projekt</b>              | Mit Projekten kann eine Projektorganisation modelliert werden.   |
| <b>Stelle</b>               | Eine Stelle beschreibt eine Position oder Rolle von Mitarbeitern in der Aufbauorganisation.  |

Tabelle 3.7.: Bestandteile der Aufbauorganisation

### 3.6. Der Modellgraph

Jedes Modell kann auch als ein Modellgraph  $\mathcal{M}$  aufgefasst werden.

$$\begin{aligned} \mathcal{M} &= (T, \mathbb{T}, \mathbb{T}_{OP}, V, E, \iota, \mathbb{M}, \text{VARIANTS}, \Gamma, \rho \\ &\quad \text{KAT}, \text{conditions}, \text{COND}, \text{conditionfilters}, \mathbb{S}, \mathbb{S}, \iota_B) \\ T &= \{\text{tool, term, voucher, document, event, question, function,} \\ &\quad \text{businessprocess, structuralelement, information,} \\ &\quad \text{datacarrier, employee, modelement, object, operator,} \\ &\quad \text{organizationalunit, potential, project, role, activity, verb,} \\ &\quad \text{process, product, target, state, shortcut}\} \\ V &= \text{tools} \cup \mathcal{B} \cup \text{vouchers} \cup \text{documents} \cup \text{events} \cup \text{questions} \cup \text{functions} \cup \\ &\quad \text{businessprocesses} \cup \text{structuralelements} \cup \text{information} \cup \\ &\quad \text{datacarriers} \cup \text{employees} \cup \text{modellements} \cup \mathcal{O} \cup \text{operators} \cup \\ &\quad \text{organizationalunits} \cup \text{potentials} \cup \text{projects} \cup \text{roles} \cup \text{activities} \cup \mathcal{V} \cup \\ &\quad \text{processes} \cup \text{products} \cup \text{targets} \cup \mathcal{L} \cup \text{shortcuts} \end{aligned}$$

$T$  bezeichnet die Menge aller verwendeten Typen. Zu jedem Typ  $t \in T$  existiert eine Menge  $M_t$ , die alle Elemente des Typs  $t$  enthält. Die Zuordnung von Typen zu Mengen ist in Tabelle 3.8 zu finden.

Da jedes Element nur einen Typ besitzt, sind die zu jedem Typ gehörenden Mengen paarweise disjunkt. Diese Eigenschaft wird durch den Operator  $\cup$  ausgedrückt.

Die Vereinigung der zu jedem Typ gehörenden Mengen ergibt die Knotenmenge  $V$  des Modellgraphens. Jeder Knoten ist ein Element des Modells.  $\mathbb{T}$  ist die Funktion, die jedem Knoten einen Typ zuordnet:  $M_{TOP} : V \mapsto T$ . Die Typen der Operatoren werden analog zu der Beschreibung der ereignisgesteuerten Prozessketten in Abschnitt 3.4 durch die Funktion  $\mathbb{T}_{OP}$  zugeordnet:  $\mathbb{T}_{OP} : V \mapsto \{\emptyset, \emptyset, \otimes\}$ . Zur Vereinfachung der Beschreibung des Metamodells soll die Zugehörigkeit zu einer bestimmten in Tabelle 3.8 aufgeführten Menge den Typ des Knotens ausdrücken. Falls beispielsweise  $v \in \text{functions}$  gilt, dann ist auch  $\mathbb{T}(v) = \text{function}$ .

Jeder Knoten ist entweder ein Master oder eine Variante. Dieses Konzept wird im Abschnitt 3.6.1 erläutert. Um diese Unterscheidung treffen zu können, wird die Funktion  $\text{VARIANTS}$  definiert, die die Varianten einer Knotenmenge bestimmt. Die Funktion  $\mathbb{M}$  bestimmt für jede Variante ihren Master.

Falls  $\mathcal{M} = (T, V, E, \iota, \mathbb{M}, \dots, \iota_B)$  ein Modellgraph ist, dann ist  $G = (V, E, \iota)$  ein kompakter, schlingenfreier Graph. Jede kompakte Kante wird Assoziation genannt. Falls es eine kompakte Kante zwischen zwei Knoten gibt, werden die Elemente „assoziert“ genannt. Die Labelfunktion der Knoten wird im Abschnitt 3.6.2 beschrieben. Die Beschreibung der möglichen Kanten folgt zusammen mit der Labelfunktion der Kanten im Abschnitt 3.6.3.

Die Funktion  $\Gamma$  dient der Ordnung der in einer Funktion verwendeten Tätigkeiten. Jeder Tätigkeit wird eine eindeutige Nummer zugeordnet, welche die Ordnung der Tätigkeiten

| Typ                       | Zugehörige Menge                   | Bezeichnung der Menge      |
|---------------------------|------------------------------------|----------------------------|
| <b>tool</b>               | Menge aller Arbeitsmittel          | <b>tools</b>               |
| <b>term</b>               | Menge aller Begriffe               | <i>B</i>                   |
| <b>voucher</b>            | Menge aller Belege                 | <b>vouchers</b>            |
| <b>document</b>           | Menge aller Dokumente              | <b>documents</b>           |
| <b>event</b>              | Menge aller Ereignisse             | <b>events</b>              |
| <b>question</b>           | Menge aller Fragen                 | <b>questions</b>           |
| <b>function</b>           | Menge aller Funktionen             | <b>functions</b>           |
| <b>businessprocess</b>    | Menge aller Geschäftsprozesse      | <b>businessprocesses</b>   |
| <b>structuralelement</b>  | Menge aller Gliederungselemente    | <b>structuralelements</b>  |
| <b>information</b>        | Menge aller Informationen          | <b>information</b>         |
| <b>datacarrier</b>        | Menge aller Medien                 | <b>datacarriers</b>        |
| <b>employee</b>           | Menge aller Mitarbeiter            | <b>employees</b>           |
| <b>modelelement</b>       | Menge aller Modellelemente         | <b>modelelements</b>       |
| <b>object</b>             | Menge aller Objekte                | <i>O</i>                   |
| <b>operator</b>           | Menge aller Operatoren             | <b>operators</b>           |
| <b>organizationalunit</b> | Menge aller Organisationseinheiten | <b>organizationalunits</b> |
| <b>potential</b>          | Menge aller Potenziale             | <b>potentials</b>          |
| <b>project</b>            | Menge aller Projekte               | <b>projects</b>            |
| <b>role</b>               | Menge aller Stellen                | <b>roles</b>               |
| <b>activity</b>           | Menge aller Tätigkeiten            | <b>activities</b>          |
| <b>verb</b>               | Menge aller Verben                 | <i>V</i>                   |
| <b>process</b>            | Menge aller Verfahren              | <b>processes</b>           |
| <b>product</b>            | Menge aller Waren                  | <b>products</b>            |
| <b>target</b>             | Menge aller Ziele                  | <b>targets</b>             |
| <b>state</b>              | Menge aller Zustände               | <i>Z</i>                   |
| <b>shortcut</b>           | Menge aller Überbrückungen         | <b>shortcuts</b>           |

Tabelle 3.8.: Typen und die Bezeichnung der zugehörigen Mengen

bestimmt. Weiterhin wird  $\Gamma$  dazu verwendet, Operatoren zur eindeutigen Identifizierung eine eindeutige Nummer zuzuordnen.

Nach der Anwendung eines Verbs auf ein Objekt durch eine Funktion besitzt das Objekt einen von dem Verb abhängigen Standardzustand. Die Funktion  $\rho : \mathcal{V} \mapsto \mathcal{Z}$  dient der Zuordnung eines Verbs zu einem Standardzustand.  $\rho$  ist injektiv, da ein Standardzustand nur von genau einem Verb erzeugt werden kann.

Jedem Element kann einer Kategorie zugeordnet werden, was durch die in Abschnitt 3.6.13 erläuterte Funktion `KAT` geschieht.

Zu jeder Assoziation lassen sich beliebig viele Bedingungen zuordnen. Mit Hilfe von Geschäftsvorfällen lassen sich die assoziierten Elemente filtern. Die zugehörigen Mengen **conditions** und **conditionfilters** sowie die Zuordnungsfunktion `COND` werden in Abschnitt 3.6.14 erläutert.

Jeder Knoten des Modellgraphen kann mit spezifischen Eigenschaften aus der Menge  $\textcircled{S}$  versehen werden. Diese Menge wird gemeinsam mit der Zuordnungsfunktion `S` im Abschnitt 3.6.15 beschrieben.

Neben der durch  $\iota$  zugeordneten formalisierten Beschriftungen der Elemente ist es möglich, jedes Element mit einer unformalisierten Beschreibung zu versehen. Die Zuordnungsfunktion von Elementen zu diesen Beschreibungen  $\iota_B$  wird in Abschnitt 3.6.16 erläutert.

#### 3.6.1. Varianten

Falls ein Element des Typs Gliederungselement, Geschäftsprozess, Funktion, Tätigkeit, Operator, Ereignis, Organisationseinheit, Projekt, Stelle oder Mitarbeiter mit dem Modell-element direkt oder indirekt durch Kanten verbunden ist, ist dieses Element eine Variante. Zu jeder Variante existiert ein Master-Element des selben Typs. Die Zuordnung des Masters zu einer Variante wird durch die Funktion  $M(v)$  vorgenommen.

Um in der Knotenmenge  $V$  eines Modells die Master von den Varianten unterscheiden zu können, ist in jedem Modell die Funktion `VARIANTS` definiert, die für eine Knotenmenge die darin enthaltenen Varianten zurück gibt.

Bei dem Mapping nach BPEL werden die Kanten des Masters bei der Variante übernommen. So stellt der Master in der Terminologie objektorientierter Programmierung eine Klasse und jede Variante eine Instanz dieser Klasse dar.

#### 3.6.2. Beschriftung

Jedes Element in einer erweiterten ereignisgesteuerten Prozesskette trägt eine Beschriftung. Grundlage der Beschriftung der Knoten bilden die Begriffe. Ein einzelner Begriff ist eine Zeichenkette, die ausschließlich aus Buchstaben besteht:

$$\mathcal{B} \subset L(G_1), |\mathcal{B}| < \infty$$

$L(G)$  ist die Funktion zur Bestimmung aller Wörter, die von der Grammatik  $G$  erzeugt werden. Die Grammatik  $G_1$  ist in der Abbildung B.1 definiert.

Grundsätzlich gilt für die Beschriftung die Eindeutigkeit innerhalb eines Typs. Es gibt folglich keine zwei Elemente des selben Typs mit der gleichen Beschriftung:

$$\forall v_1, v_2 \in V, \mathbb{T}(v_1) = \mathbb{T}(v_2), v_1 \neq v_2 : \iota(v_1) \neq \iota(v_2)$$

Da auch Begriffe Bestandteile des Modells sind, müssen auch sie beschriftet werden. Aus Konsistenzgründen ist jeder Begriff mit sich selbst beschriftet:

$$\begin{aligned} \iota_{[\mathcal{B}]} &\mapsto \mathcal{B} \\ \forall b \in \mathcal{B} : \iota(b) &= b \end{aligned}$$

Varianten eines Elements sind immer mit ihrem Master verbunden. Sie sind deshalb mit der Beschriftung des zugehörigen Masters beschriftet:

$$\forall v \in \text{VARIANTS}(V) : \iota(v) = \iota(\mathbb{M}(v))$$

Grundsätzlich werden alle Elemente mit einer Kombination aus Begriffen beschriftet. Die Ausnahme von dieser Regel bilden die Elemente der ereignisgesteuerten Prozesskette. Eine ereignisgesteuerte Prozesskette beschreibt, wie sich der Zustand von Objekten im Laufe der Verarbeitung verändern. Deshalb sind Funktionen und Ereignisse nicht mit Begriffen, sondern mit Objekten, Verben und Zuständen beschriftet. Da eine Funktion etwas mit einem Objekt vollzieht, ist jede Funktion mit einem Tupel aus Objekt und Verb versehen. Da weiterhin eine Tätigkeit eine Funktion näher beschreiben kann, ist auch jede Tätigkeit mit einem Tupel aus Objekt und Verb versehen.

$$\iota_{[\text{functions} \cup \text{activities}]}(v) \mapsto \mathcal{O} \times \mathcal{V}$$

Ein Ereignis beschreibt den Zustand eines Objekts, deshalb ist jedes Ereignis mit einem Tupel aus Objekt und Zustand beschriftet:

$$\iota_{[\text{events}]} \mapsto \mathcal{O} \times \mathcal{Z}$$

Jedes Objekt besitzt nach der Anwendung einer Funktion einen Standardzustand. Dies spiegelt sich in der Beschriftung der Trivialereignisse wieder. Sie ist von dem Objekt und dem Verb der zugehörigen Funktion abhängig. Das Objekt wird übernommen und der Zustand ist der zu dem Verb zugehörige Standardzustand:

$$\forall v \in \text{trivialevents}, f = \text{adj}_1^-(v) : \iota(v) = (\pi_1(\iota(f)), \rho(\pi_2(\iota(f))))$$

Die Operatoren werden mit ihrer Art beschriftet. Ein und-Operator erhält ein Symbol für „und“, ein oder-Operator ein Symbol für „oder“ und der entweder-oder-Operator ein Symbol für „entweder-oder“:

$$\forall v \in \text{operators} : \iota(v) = \begin{cases} \wedge & \text{falls } \mathbb{T}_{\text{OP}}(v) = \ominus \\ \vee & \text{falls } \mathbb{T}_{\text{OP}}(v) = \oplus \\ \times & \text{falls } \mathbb{T}_{\text{OP}}(v) = \otimes \end{cases}$$

Objekte, Verben und Zustände selbst werden mit einer Kombination von Begriffen beschriftet. Die Beschriftung der verbleibenden Elemente des Modellgraphen wird auch aus der Kombination von Begriffen gebildet. Deshalb kann die Beschriftung der Objekte, Verben und Zustände zusammen mit der Beschriftung der anderen Elemente wie folgt formalisiert werden:

$$\iota[V \setminus (\text{VARIANTS}(V) \cup \mathcal{B} \cup \text{functions} \cup \text{events} \cup \text{operators})](v) = b^*, b^* \in \mathcal{B}^*$$

$\mathcal{B}^*$  bezeichnet die Menge aller Kombinationen von Begriffen. Sie wird wie folgt definiert:

$$\mathcal{B}^* = \{(b_1, \dots, b_j) \mid b_i \in \mathcal{B}, 1 \leq i \leq j, 0 < j < \infty\}$$

#### 3.6.3. Kanten des Modellgraphen

In den vorhergehenden Abschnitten wurden die Elemente, Master und Varianten als Knoten des Modellgraphen eingeführt. In diesem Abschnitt werden die zugehörigen Kanten definiert.

Die Kanten des Modellgraphen drücken Beziehungen der Knoten zueinander aus. Diese Beziehung ist wechselseitig: So wird beispielsweise eine Funktion von einer Stelle durchgeführt und eine Stelle führt eine Funktion durch. Deshalb wird im Folgenden eine kompakte Darstellung  $G_k = (V_k, E_k, \iota_k)$  des Modellgraphen  $G$  verwendet. Die so ausgedrückte Beziehung wird auch Assoziation genannt.

Die Beschriftungen der Vor- und Rückwärtskanten werden in Tabellen angegeben. Diese haben die in Tabelle 3.9 dargestellte Form. Falls von einer Quell- in eine Zielmenge mehrere Beschriftungen möglich sind, wird die in Tabelle 3.10 dargestellte Form verwendet, um eine kompakte Darstellung zu ermöglichen.

| Von        | Nach      | Beschriftung   | Rückwärtsbeschriftung  |
|------------|-----------|--|--|
| Quellmenge | Zielmenge | Beschriftung der Kante von einem Element der Quellmenge zu einem Element der Zielmenge | Beschriftung der Kante von einem Element der Zielmenge zu einem Element der Quellmenge |

Tabelle 3.9.: Allgemeiner Aufbau der Beschriftungstabellen

#### 3.6.4. Kanten der ereignisgesteuerten Prozessketten

Ereignisgesteuerte Prozessketten bestehen aus Funktionen, Ereignissen und Operatoren. Jede Funktion ist genau einem Geschäftsprozess durch die Beziehung (beinhaltet, ist enthalten in) zugeordnet (vgl. Tabelle 3.11).

| Tupelelement       | Inhalt   |
|--------------------|--|
| von                | Quellmenge   |
| nach               | Zielmenge  |
| Beschriftungstupel | { (Beschriftung der Kante von einem Element der Quellmenge zu einem Element der Zielmenge, Beschriftung der Kante von einem Element der Zielmenge zu einem Element der Quellmenge) } |

Tabelle 3.10.: Kompakter Aufbau der Beschriftungstabellen

| Von                                      | Nach                             | Beschriftung | Rückwärtsbeschriftung |
|--|----------------------------------|--------------|-----------------------|
| VARIANTS<br>( <b>businessprocesses</b> ) | VARIANTS<br>( <b>functions</b> ) | beinhaltet   | ist enthalten<br>in   |

Tabelle 3.11.: Zugehörigkeit einer Funktion zu einem Geschäftsprozess

$$\forall f \in \text{VARIANTS}(\mathbf{functions}) : |\text{adj}^-(f) \cap \mathbf{businessprocesses}| = 1$$

Ereignisse und Operatoren werden in Geschäftsprozessen benutzt, besitzen jedoch keine direkte Zugehörigkeitskante. Die Verknüpfung wird durch Funktionen erreicht: Funktionen erzeugen Ereignisse, Ereignisse und Operatoren lösen Funktionen aus, Ereignisse und Operatoren sind Vorgänger von anderen Ereignissen und Operatoren. Die dafür benötigten Kanten werden in Tabelle 3.12 gezeigt.

| Von  | Nach   | Beschriftung         | Rückwärtsbeschriftung   |
|--|--|----------------------|-------------------------|
| VARIANTS<br>( <b>functions</b> )                 | VARIANTS<br>( <b>trivialevents</b> )             | erzeugt              | wird erzeugt<br>von     |
| VARIANTS<br>( <b>events</b> ∪ <b>operators</b> ) | VARIANTS<br>( <b>functions</b> )                 | löst aus             | wird ausgelöst<br>durch |
| VARIANTS<br>( <b>events</b> ∪ <b>operators</b> ) | VARIANTS<br>( <b>events</b> ∪ <b>operators</b> ) | ist Vorgänger<br>von | hat als<br>Vorgänger    |

Tabelle 3.12.: Kanten zwischen Knoten einer EPK

Da die in Tabelle 3.12 gezeigten Beschriftungen nicht für weitere Beziehungen verwendet

werden, ist die Menge aller Kanten aller ereignisgesteuerten Prozessketten  $E_{EPK}$  wie folgt definiert:

$$E_{EPK} = \left\{ e \mid e \in E_k, \pi_3(e) \in \left\{ \begin{array}{l} (\text{erzeugt, wird erzeugt von}), \\ (\text{löst aus, wird ausgelöst durch}), \\ (\text{ist Vorgänger von, hat als Vorgänger}) \end{array} \right\} \right\}$$

#### 3.6.5. Tätigkeiten

Tätigkeiten beschreiben Teilschritte einer Funktion. Die Beziehung (beinhaltet, ist enthalten in) verknüpft die Funktionen mit Tätigkeiten. Bei einem Funktions-Master können die Tätigkeiten angegeben werden, die für alle Varianten gelten. Deshalb sind in Tabelle 3.13 alle Funktionen mit Varianten von Tätigkeiten verbunden.

| Von              | Nach                     | Beschriftung | Rückwärtsbeschriftung |
|------------------|--------------------------|--------------|-----------------------|
| <b>functions</b> | VARIANTS<br>(activities) | beinhaltet   | ist enthalten<br>in   |

Tabelle 3.13.: Kanten zwischen Funktionen und Tätigkeiten

Die Reihenfolge, in der die Tätigkeiten aufgeführt werden sollen, wird durch die injektive Funktion  $\Gamma : \text{VARIANTS}(\text{activities}) \mapsto \mathbb{N}$  bestimmt. Die Ordnung der natürlichen Zahlen  $\mathbb{N}$  bestimmt die Reihenfolge der Tätigkeiten: Die Tätigkeit mit der kleinsten zugeordneten Zahl wird als erstes aufgeführt, dann die mit der zweitkleinsten zugeordneten Zahl usw.

#### 3.6.6. Organisation

Die Aufbauorganisation besteht aus den Elementen Organisationseinheit, Stelle und Mitarbeiter.

Das oberste Glied der Organisation wird von den Organisationseinheiten gebildet. In jedem Modell kann es eine oder mehrere obere Organisationseinheiten geben. Einer Organisationseinheit können beliebig viele Organisationseinheiten untergeordnet sein. Jeder Organisationseinheit sind beliebig viele Stellen und Mitarbeiter zugeordnet:

Jede Stelle kann aus einem oder mehreren Mitarbeitern bestehen:

Die Leitungsstruktur ist in Tabelle 3.16 dargestellt.

Neben der Leitungsstruktur gibt es die Stabsstruktur: Organisationen können Stabsstellen von Organisationen sein und Stellen können Stabsstellen von Stellen sein. Eine Stabsstellen-Beziehung zwischen Organisationseinheiten und Stellen ist nicht vorgesehen (vgl. Tabelle 3.17).

| Von  | Nach                                       | Beschriftung        | Rückwärtsbeschriftung |
|--|--|---------------------|-----------------------|
| VARIANTS<br>( <b>organizationalunits</b> ) | VARIANTS<br>( <b>organizationalunits</b> ) | ist<br>übergeordnet | ist<br>untergeordnet  |
| VARIANTS<br>( <b>organizationalunits</b> ) | VARIANTS<br>( <b>roles</b> )               | besteht aus         | gehört zu             |
| VARIANTS<br>( <b>organizationalunits</b> ) | VARIANTS<br>( <b>employees</b> )           | besteht aus         | gehört zu             |

Tabelle 3.14.: Aufbau der der Organisation

| Von                          | Nach                             | Beschriftung | Rückwärtsbeschriftung |
|------------------------------|----------------------------------|--------------|-----------------------|
| VARIANTS<br>( <b>roles</b> ) | VARIANTS<br>( <b>employees</b> ) | besteht aus  | ist                   |

Tabelle 3.15.: Verbindung von Stellen und Mitarbeitern

| Von  | Nach                             | Beschriftung            | Rückwärtsbeschriftung   |
|--|----------------------------------|-------------------------|-------------------------|
| VARIANTS<br>( <b>organizationalunits</b> ) | VARIANTS<br>( <b>roles</b> )     | wird geleitet<br>von    | leitet                  |
| VARIANTS<br>( <b>organizationalunits</b> ) | VARIANTS<br>( <b>employees</b> ) | wird geleitet<br>von    | leitet                  |
| VARIANTS<br>( <b>roles</b> )               | VARIANTS<br>( <b>roles</b> )     | ist Vorgesetzter<br>von | hat als<br>Vorgesetzten |
| VARIANTS<br>( <b>employees</b> )           | VARIANTS<br>( <b>employees</b> ) | ist Vorgesetzter<br>von | hat als<br>Vorgesetzten |

Tabelle 3.16.: Leitungsstruktur

| Von  | Nach                                       | Beschriftung          | Rückwärtsbeschriftung |
|--|--|-----------------------|-----------------------|
| VARIANTS<br>( <b>organizationalunits</b> ) | VARIANTS<br>( <b>organizationalunits</b> ) | hat als<br>Stabstelle | ist Stabstelle<br>von |
| VARIANTS<br>( <b>roles</b> )               | VARIANTS<br>( <b>roles</b> )               | hat als<br>Stabstelle | ist Stabstelle<br>von |

Tabelle 3.17.: Stabsstellen

### 3. Metamodell von Nautilus

---

Vertretungen von Mitarbeitern wird, wie in Tabelle 3.18 dargestellt, durch die Vertretungsassoziation modelliert:

| Von                              | Nach                             | Beschriftung | Rückwärtsbeschriftung |
|----------------------------------|----------------------------------|--------------|-----------------------|
| VARIANTS<br>( <b>employees</b> ) | VARIANTS<br>( <b>employees</b> ) | vertritt     | wird vertreten<br>von |

Tabelle 3.18.: Vertretungen

Bei der Modellierung der Projektorganisation ist nur die Zuordnung von Organisationseinheiten zu Projekten und die Gliederung in Teilprojekte vorgesehen, so dass Organisationseinheiten und Projekte in Projekten enthalten sein können (vgl. Tabelle 3.19).

| Tupelelement       | Inhalt   |
|--------------------|--|
| von                | VARIANTS( <b>projects</b> )                              |
| nach               | VARIANTS( <b>projects</b> ∪ <b>organizationalunits</b> ) |
| Beschriftungstupel | { (beinhaltet, ist enthalten) }                          |

Tabelle 3.19.: Projektorganisation

Organisationseinheiten, Stellen und Mitarbeiter verantworten und verwenden Arbeitsmittel (vgl. Tabelle 3.20).

| Tupelelement       | Inhalt  |
|--------------------|---|
| von                | <b>organizationalunits</b> ∪ <b>roles</b> ∪ <b>employees</b>                  |
| nach               | <b>tools</b>  |
| Beschriftungstupel | { (verantwortet, wird verantwortet von),<br>(verwendet, wird verwendet von) } |

Tabelle 3.20.: Verantwortung und Verwendung von Arbeitsmitteln

Organisationseinheiten, Stellen und Mitarbeiter verantworten, prüfen, assistieren oder wirken bei Gliederungselementen, Geschäftsprozessen, Funktionen, Tätigkeiten oder Werkzeugen mit (vgl. Tabelle 3.21).

Werkzeuge besitzen in dem Fall der Unterstützungen, Mitwirkungen, Prüfungen, Freigaben und Verantwortlichkeiten die selben Beziehungen wie Mitarbeiter mit der Ausnahme, dass es keine Beziehung zu anderen Werkzeugen gibt (vgl. Tabelle 3.22).

| Tupelelement       | Inhalt  |
|--------------------|---|
| von                | <b>organizationalunits</b> ∪ <b>roles</b> ∪ <b>employees</b>  |
| nach               | <b>structuralelements</b> ∪ <b>businessprocesses</b> ∪ <b>functions</b> ∪ <b>activities</b>   |
| Beschriftungstupel | { (assistiert bei, unter Assistenz von),<br>(wirkt mit an, unter Mitwirkung von),<br>(führt durch, wird durchgeführt von),<br>(prüft, wird geprüft von),<br>(gibt frei, wird freigegeben von),<br>(verantwortet, wird verantwortet von) } |

Tabelle 3.21.: Unterstützungen, Mitwirkungen, Durchführungen, Prüfungen, Freigaben und Verantwortlichkeiten

| Tupelelement       | Inhalt  |
|--------------------|---|
| von                | <b>tools</b>  |
| nach               | <b>structuralelements</b> ∪ <b>businessprocesses</b> ∪ <b>functions</b> ∪ <b>activities</b>   |
| Beschriftungstupel | { (assistiert bei, unter Assistenz von),<br>(wirkt mit an, unter Mitwirkung von),<br>(führt durch, wird durchgeführt von),<br>(prüft, wird geprüft von),<br>(gibt frei, wird freigegeben von),<br>(verantwortet, wird verantwortet von) } |

Tabelle 3.22.: Mögliche Automatisierungen durch Werkzeuge

### 3.6.7. Anmerkungen zum Modell

Falls die Dokumentation durch das Modell selbst nicht ausreicht, können durch die Elemente Dokumente und Verfahren externe Beschreibungen mit dem Modell verknüpft werden. Weiterhin erlauben die Elemente Frage, Potenzial und Ziel, Kommentare zum Modell im Modell selbst zu erfassen. Beides geschieht durch die Beziehung beschreibt (vgl. Tabelle 3.23).

| Tupelelement       | Inhalt  |
|--------------------|---|
| von                | <b>documents</b> ∪ <b>processes</b> ∪ <b>questions</b> ∪ <b>potentials</b> ∪ <b>targets</b> |
| nach               | <b>structuralelements</b> ∪ <b>businessprocesses</b> ∪ <b>functions</b> ∪ <b>activities</b> |
| Beschriftungstupel | { (beschreibt, wird beschrieben durch) }  |

Tabelle 3.23.: Mögliche Formen der Beschreibungen

Zu den Kommentaren zum Modell können, wie in Tabelle 3.24 dargestellt, Verantwortlichkeiten und Beteiligungen im Modell erfasst werden.

| Tupelelement       | Inhalt  |
|--------------------|---|
| von                | <b>organizationalunits</b> ∪ <b>roles</b> ∪ <b>employees</b>                            |
| nach               | <b>questions</b> ∪ <b>potentials</b> ∪ <b>targets</b>                                   |
| Beschriftungstupel | { (verantwortet, wird verantwortet von),<br>(ist beteiligt an, unter Beteiligung von) } |

Tabelle 3.24.: Beteiligungen und Verantwortlichkeiten an den Kommentaren

Weiterhin kann zu jeder Funktion und Tätigkeit angegeben werden, welche Waren und Werkzeuge sie bei ihrer Ausführung verwenden (vgl. Tabelle 3.25).

| Tupelelement       | Inhalt                               |
|--------------------|--------------------------------------|
| von                | <b>functions</b> ∪ <b>activities</b> |
| nach               | <b>products</b> ∪ <b>tools</b>       |
| Beschriftungstupel | { (verwendet, wird verwendet bei) }  |

Tabelle 3.25.: Nutzung von Waren und Werkzeugen durch Funktionen und Tätigkeiten

### 3.6.8. Modellelement und Gliederungselemente

Die direkten Nachbarn des Modellelements sind Gliederungselemente, Geschäftsprozesse, Organisationseinheiten und Projekte (vgl. Tabelle 3.26).

| Tupelement         | Inhalt   |
|--------------------|--|
| von                | <b>modelelements</b>   |
| nach               | <b>structuralelements</b> ∪ <b>businessprocesses</b> ∪<br><b>organizationalunits</b> ∪ <b>projects</b> |
| Beschriftungstupel | { (beinhaltet, ist enthalten in) }   |

Tabelle 3.26.: Kanten des Modellelements

Gliederungselemente dienen der Zusammenfassung von Geschäftsprozessen und weiteren Gliederungselementen (vgl. Tabelle 3.27).

| Tupelement         | Inhalt   |
|--------------------|--|
| von                | <b>structuralelements</b>                            |
| nach               | <b>structuralelements</b> ∪ <b>businessprocesses</b> |
| Beschriftungstupel | { (beinhaltet, ist enthalten in) }                   |

Tabelle 3.27.: Kanten von Gliederungselementen

### 3.6.9. Datenfluss

Generell kann zu jedem Element nur beschrieben werden, welche Daten es sendet und empfängt. Eine weitergehende Beschreibung, von welchem Element zu welchem Element die Daten fließen, kann nicht angegeben werden.

Die Daten eines Prozesses werden durch die Elemente Beleg, Dokument, Information und Ware beschrieben (vgl. Tabelle 3.4). Im folgenden bezeichnet  $V_{DF}$  die Menge aller Elemente, die Daten beschreiben:

$$V_{DF} = \mathbf{vouchers} \cup \mathbf{documents} \cup \mathbf{informations} \cup \mathbf{products}$$

Medien dienen dem Transport dieser Elemente:

| Tupelement         | Inhalt                                    |
|--------------------|---|
| von                | <b>datacarriers</b>                       |
| nach               | $V_{DF}$                                  |
| Beschriftungstupel | { (übermittelt, wird übermittelt durch) } |

Tabelle 3.28.: Übermittlung durch Medien

Daten werden, wie in Tabelle 3.29 dargestellt, zwischen Funktionen und Tätigkeiten ausgetauscht. Weiterhin können Funktionen und Tätigkeiten auf diese Elemente zugreifen,

### 3. Metamodell von Nautilus

---

ohne dass ein Austausch mit anderen Funktionen und Tätigkeiten stattfindet, was durch die Beschriftung „verwendet“ zum Ausdruck gebracht wird (vgl. Tabelle 3.30).

| Tupelelement       | Inhalt   |
|--------------------|--|
|                    | von <b>functions</b> $\cup$ <b>activities</b>  |
|                    | nach $V_{DF}$  |
| Beschriftungstupel | { (empfängt, wird empfangen von),<br>(sendet, wird gesendet von),<br>(erzeugt, wird erzeugt bei) } |

Tabelle 3.29.: Datenfluss bei Funktionen und Tätigkeiten

| Tupelelement       | Inhalt  |
|--------------------|---|
|                    | von <b>functions</b> $\cup$ <b>activities</b> |
|                    | nach $V_{DF}$                                 |
| Beschriftungstupel | { (verwendet, wird verwendet bei) }           |

Tabelle 3.30.: Externe Datennutzung (ohne Einbindung in den Datenfluss) durch Funktionen und Tätigkeiten

#### Datenfluss in der Aufbauorganisation

Analog zu den Funktionen und Tätigkeiten können Mitglieder der Aufbauorganisation (Organisationseinheiten, Stellen und Mitarbeiter) Daten senden, empfangen und verwenden. In dem Metamodell wurden für „senden“ und „empfangen“ die Begriffe „übermitteln“ und „erhalten“ gewählt (vgl. Tabelle 3.31).

| Tupelelement       | Inhalt  |
|--------------------|---|
|                    | von <b>organizationalunits</b> $\cup$ <b>roles</b> $\cup$ <b>employees</b>                                  |
|                    | nach $V_{DF}$   |
| Beschriftungstupel | { (übermittelt, wird übermittelt von),<br>(verwendet, wird verwendet von),<br>(erhält, wird erhalten von) } |

Tabelle 3.31.: Übermittlung, Verwendung und Empfang von Daten innerhalb der Organisation

#### 3.6.10. Überbrückungen

Überbrückungen werden in der EPK verwendet und beginnen dort bei einem Element und enden bei einem anderen. Das Anfangselement darf nur eine ausgehende EPK-Kante,

das Endelement nur eine eingehende EPK-Kante und die überbrückten Elemente immer genau eine ein- und eine ausgehende EPK-Kante besitzen.

$$\forall e \in \{e' \mid e' \in E, \mathbb{T}(\pi_1(e')) = \mathbf{shortcut}\}, v = \pi_2(e) :$$

$$\begin{cases} \delta_{[E_{EPK}]}^+(v) = 1 & \text{falls } \pi_3(e) = \text{„hat als erstes Element“} \\ \delta_{[E_{EPK}]}^-(v) = 1 & \text{falls } \pi_3(e) = \text{„hat als letztes Element“} \\ (\delta_{[E_{EPK}]}^-(v) = 1) \wedge (\delta_{[E_{EPK}]}^+(v) = 1) & \text{falls } \pi_3(e) = \text{„beinhaltet“} \end{cases}$$

| Tupelement         | Inhalt   |
|--------------------|--|
| von                | <b>shortcuts</b>   |
| nach               | VARIANTS( <b>functions</b> $\cup$ <b>events</b> $\cup$ <b>operators</b> )  |
| Beschriftungstupel | { (hat als erstes Element, ist erstes Element von),<br>(hat als letztes Element, ist letztes Element von),<br>(beinhaltet, ist enthalten in) } |

Tabelle 3.32.: Kanten von Überbrückungen

### 3.6.11. Einschränkung von Varianten

In einer Beziehung zu einem Element, das sowohl Master als auch Variante sein kann, darf maximal eine „Ausprägung“ eines Masters mit einem Element assoziiert sein. Mit „Ausprägung“ ist hier der Master selbst oder eine Variante des Masters gemeint.

Dieses stellt folgende Bedingung sicher: Falls eine Variante  $a$  eines Masters  $M$  mit einem Element  $l$  assoziiert ist, so gibt es keine weitere Variante  $a'$  von  $M$ , die über eine kompakte Kante mit dem selben Beschriftungstupel mit dem Element  $l$  verbunden ist.

$$\mathbb{M}'(v) = \begin{cases} \mathbb{M}(v) & \text{falls VARIANTS}(\{v\}) = \{v\} \\ v & \text{sonst} \end{cases}$$

$$\exists e = (v_1, v_2, l) \in E$$

$$\Rightarrow \nexists e = (v'_1, v'_2, l), (v'_1 \neq v_1) \wedge (v'_2 \neq v_2) \wedge (\mathbb{M}'(v_1) = \mathbb{M}'(v'_1)) \wedge (\mathbb{M}'(v_2) = \mathbb{M}'(v'_2))$$

### 3.6.12. Kontext der Varianten

Der Kontext einer Variante ist als kürzester Weg von dem Modellelement zu der Variante definiert. Hiervon sind Operatoren und Ereignisse ausgenommen, da sie wie in Abbildung 3.2 dargestellt, von zwei Geschäftsprozessen verwendet werden. Hier ist der kürzeste Weg von dem Modellknoten aus von der Struktur der EPK abhängig, in der der Operator oder das Ereignis verwendet wurden. Deshalb ist der Kontext der Varianten von Elementen der Typen Ereignis oder Operator immer der Modellknoten selbst.

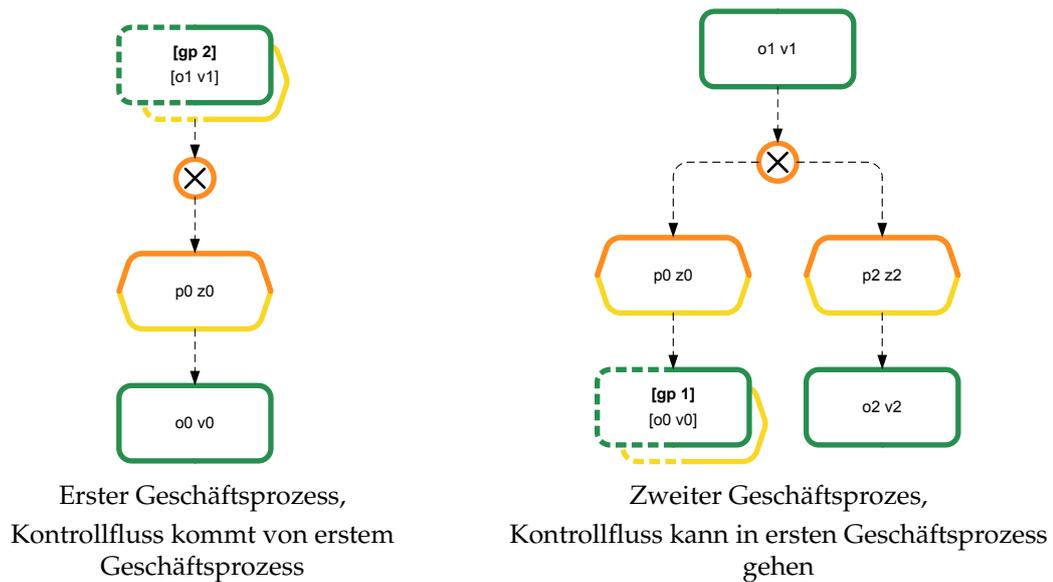


Abbildung 3.2.: Verwendung des selben Ereignisses und Operators in zwei Prozessen

Die Kontexte der Elemente des Kreditantrags ist in Abbildung 3.3 dargestellt. Die Verwendung der Elemente des Types Funktion, Geschäftsprozess und Gliederungselement ist in Abbildung 3.4 gezeigt.

#### 3.6.13. Kategorien

Zur Kategorisierung von Elementen kann mittels der Funktion  $KAT$  jedem Element eine Kategorie zugewiesen werden. Eine Kategorie besitzt einen Namen. Da keine weiteren Informationen in einer Kategorie enthalten sind, bildet die Funktion  $KAT$  in die Menge aller einzeiligen, nicht leeren Texte ab. Werden zwei Elemente auf den selben Text abgebildet, so gehören sie der selben Kategorie an.

$$KAT : V \mapsto L(G_2)$$

#### 3.6.14. Bedingungen und Geschäftsvorfälle

Geschäftsvorfälle stellen eine spezielle Sicht auf ein Modell her. Jede Assoziation kann mit beliebig vielen Bedingungen versehen werden. Widerspricht eine dieser Bedingungen einem Geschäftsvorfall nicht, so ist diese Assoziation in der Sicht auf das Modell enthalten. Der Umkehrschluss gilt nicht. Falls eine Bedingung einem Geschäftsvorfall widerspricht, so kann es immer noch eine andere Bedingung geben, die dem Geschäftsvorfall nicht widerspricht, wodurch die Assoziation in der Sicht des Modells enthalten ist.

Einem Ereignis sind ein Objekt und ein Zustand zugeordnet, den das Objekt besitzt (vgl. Abschnitt 3.6.2). So können Ereignisse als Teil einer Bedingung verwendet werden. Eine



Abbildung 3.3.: Funktionen, Geschäftsprozesse und Gliederungselemente mit Kontext



Abbildung 3.4.: Die Verbindung von Funktionen, Geschäftsprozessen und Gliederungselement im Kreditantragsprozess

Bedingung wird Gruppen aufgebaut. Jede Gruppe besteht aus einer Konjunktion von negierten und nicht-negierten Ereignissen. Die Gruppen werden durch den Oder-Operator verbunden und bilden so eine Bedingung:

$$E_{\text{cond}} = \left\{ E_i \mid \begin{array}{l} E_i = \bigwedge (\neg) e_i, e_i \in \mathbf{events}, \\ 1 \leq i \leq |\mathbf{events}|, e_j \neq e_k, 1 \leq j \leq i, 1 \leq k \leq i, j \neq k \end{array} \right\}$$

$$\forall b \in \mathbf{conditions} : b = \bigvee E'_{\text{cond}}, E'_{\text{cond}} \subseteq E_{\text{cond}}$$

$(\neg)e_i$  bedeutet, dass jedes  $e_i$  entweder negiert oder nicht negiert vorkommt. In nicht-negierter Form muss das zugehörige Objekt den dem Ereignis zugehörigen Zustand besitzen, in negierter Form darf das Objekt alle Zustände außer den dem Ereignis zugehörigen Zustand besitzen.

Die Zuordnung zu Assoziationen erfolgt über die Funktion COND:

$$\text{COND} : E \mapsto \wp(\mathbf{conditions})$$

Grundsätzlich kann jede Assoziation mit einer Bedingung versehen sein. Davon ausgenommen sind die EPK- und Aufbau-Organisationsassoziationen.

Jeder Geschäftsvorfall ist genauso wie eine Bedingung aufgebaut, besitzt jedoch zusätzlich einen eindeutigen Namen:

$$\forall gv \in \mathbf{conditionfilters} : gv = (n, c), n \in L(G_2), c = \bigvee E'_{\text{cond}}, E'_{\text{cond}} \subseteq E_{\text{cond}}$$

$$\forall gv_1, gv_2 \in \mathbf{conditionfilters}, gv_1 \neq gv_2 : \pi_1(gv_1) \neq \pi_1(gv_2)$$

#### 3.6.15. Spezifische Eigenschaften

Jedem Knoten kann eine spezifische Eigenschaft zugewiesen werden. Eine spezifische Eigenschaft ist ein Tripel aus Name, Typ und Wert (vgl. [Ge04, S. 42 f.]).

**Name** bezeichnet den Namen der spezifischen Eigenschaft. Der Name muss keiner besonderen Form genügen

**Typ** bestimmt den Typ des Wertes. Ein Typ kann aus der Menge {BOOLEAN, CURRENCY, DATE, FLOAT, TEXT, TIME} gewählt werden.

**Wert** beinhaltet den Wert der spezifischen Eigenschaft.

Die Tabelle 3.33 zeigt zu den Typen die zugehörigen Wertemengen.

Jedem Element werden durch die in Funktion S spezifische Eigenschaften zugeordnet:

$$S : V \cup T \mapsto \wp(\mathbb{S})$$

Die Menge  $\mathbb{S}$  aller spezifischen Eigenschaften eines Elements ist in Abbildung 3.5 dargestellt.

Die spezifischen Eigenschaften, die ein Element besitzt, werden durch seinen Elementtyp festgelegt. In der Festlegung kann ein Standardwert zugewiesen werden, der von einem Element ersetzt werden kann.

| Typ      | Wertemenge   |
|----------|--|
| BOOLEAN  | {TRUE, FALSE}  |
| CURRENCY | $\mathbb{Q} \times L(G_2)$   |
| DATE     | $\{1, \dots, 31\} \times \{1, \dots, 12\} \times \mathbb{Q}$       |
| FLOAT    | $\mathbb{R}$   |
| TEXT     | $L(G_3)$ (siehe Abbildung B.3)                                     |
| TIME     | $\{0, \dots, 23\} \times \{0, \dots, 59\} \times \{0, \dots, 59\}$ |

Tabelle 3.33.: Typen und zugehöriger Wertebereich

$$\textcircled{S} = \left\{ (n_i, t_i, v_i) \left| \begin{array}{l} n_i \in L(G_2), n_i \neq n_j, i \neq j, \\ t_i \in \{\text{BOOLEAN}, \text{CURRENCY}, \text{DATE}, \text{FLOAT}, \text{TEXT}, \text{TIME}\}, \\ v_i \in \begin{cases} \{\text{TRUE}, \text{FALSE}\} \cup \varepsilon & \text{falls } t_i = \text{BOOLEAN} \\ (\mathbb{Q} \times L(G_2)) \cup \varepsilon & \text{falls } t_i = \text{CURRENCY} \\ (\{1, \dots, 31\} \times \{1, \dots, 12\} \times \mathbb{Q}) \cup \varepsilon & \text{falls } t_i = \text{DATE} \\ \mathbb{R} \cup \varepsilon & \text{falls } t_i = \text{FLOAT} \\ L(G_3) \cup \varepsilon & \text{falls } t_i = \text{TEXT} \\ (\{0, \dots, 23\} \times \{0, \dots, 59\} \times \{0, \dots, 59\}) \cup \varepsilon & \text{falls } t_i = \text{TIME} \end{cases} \end{array} \right. \right\}$$

$\varepsilon$ : Zu dem Name existiert kein zugeordneter Wert.

$G_2$ : Grammatik zur Erzeugung von einzeiligem Text (vgl. Abbildung B.3)

Abbildung 3.5.: Die Menge  $\textcircled{S}$  aller spezifischen Attribute eines Elements

### 3.6.16. Beschreibungen

Die Knoten des Modellgraphen sind immer durch die in Abschnitt 3.6.2 beschriebene Funktion  $\iota_{[V]}$  mit einer Beschriftung versehen. Die Funktion  $\iota_B$  ordnet Elementen eine Beschreibung zu, die keinem besonderen Format genügen muss. Nicht jedes Element muss eine Beschreibung besitzen:

$$\iota_B : V \mapsto L(G_2) \cup \varepsilon$$

### 3.6.17. Nichtformalisierte Bestandteile

In der Software Nautilus können verschiedene Sichten auf Modelle erzeugt werden. Diese Sichten werden „Reports“ genannt und mittels des Elements „Report“ verwaltet. Weiterhin ist es möglich, Modelle zu HTML konvertieren. Hierfür muss der Ort der HTML-Dateien mittels des Elements „Ordner“ angegeben werden. Schließlich werden die Verknüpfungen zu erstellten Visio-Grafiken mittels des Elements „Grafik“ realisiert. Diese Elemente und deren Beziehungen wurden nicht in das Metamodell aufgenommen, da

sie Implementierungsspezifika darstellen und für das Mapping nach BPEL nicht relevant sind.

#### 3.6.18. Fehlende Beziehungen

Eine vollständige Modellierung des Datenflusses, wie er in Abschnitt 2.1 definiert wurde, ist durch Beziehungen nicht möglich. Falls beispielsweise eine Funktion  $f$  die Information  $i$  sendet und die Funktion  $g$  die Information  $i$  empfängt, ist nicht klar, ob es die selbe Information ist und ob sie von  $f$  zu  $g$  fließt. Es könnte sein, dass  $g$  die Information von einer anderen Funktion  $f'$ , die auch die Information  $i$  sendet, empfängt oder dass  $g$  die Information von außerhalb des Prozesses empfängt, wie beispielsweise von einer Datenbank erhält.

Eine Schachtelung der Elemente, die Daten beschreiben ( $V_{DF}$ , Belege, Dokumente, Informationen und Waren) ist nicht möglich. Durch Medien kann eine einstufige enthalten-Relation modelliert werden, jedoch gibt es keine Beziehung von Medien zu Funktionen oder Tätigkeiten. Eine Modellierung, dass die Information „Adresse“ die Informationen „Name“, „Straße“ und „Ort“ enthält, ist nicht durch Beziehungen möglich.

Bei der Projektorganisation ist die einzig mögliche Verbindung in die Aufbauorganisation die Zuordnung von Projekten zu Organisationseinheiten (vgl. Tabelle 3.19). Es fehlen Zuordnungen zu Stellen und Mitarbeitern, um langlaufende Projekte, bei denen immer die selben Stellen beteiligt sind, modellieren zu können.

Die vollständige Automatisierung durch Werkzeuge ist nicht modellierbar: Ein Werkzeug kann beispielsweise Geschäftsprozesse verantworten und durchführen, jedoch kein anderes Werkzeug verantworten, durchführen oder verwenden (vgl. Tabelle 3.22).

Es wäre möglich, diese Beziehungen durch die Verwendung von spezifischen Eigenschaften (siehe Abschnitt 3.6.15) zu modellieren. Diese Art der Modellierung wäre nicht erwartungskonform (vgl. [ISO9241-10]), da Beziehungen einerseits durch Beziehungen und andererseits durch spezifische Eigenschaften modelliert werden würden.

### 3.7. Extraktion der EPK-Graphen

In Abschnitt 3.4 wurde die Darstellung von ereignisgesteuerten Prozessketten als Graph eingeführt. In einem Modellgraphen  $\mathcal{M}$  sind die ereignisgesteuerten Prozessketten mittels kompakten Kanten abgelegt. Im Metamodell sind Funktionswegweiser und Trivialereignisse implizit definiert: Eine Funktion ist ein Funktionswegweiser, falls sie mit einer Funktion über Ereignisse oder Operatoren verbunden ist, jedoch nicht zu dem selben Prozess wie die verbundene Funktion gehört. Ein Ereignis ist genau dann ein Trivialereignis, falls es der Nachfolger einer Funktion ist. Für die Umsetzung von ereignisgesteuerten Prozessketten nach BPEL ist die in Abschnitt 3.4 eingeführte Darstellung besser geeignet. Weiterhin ist ein Modellgraph  $\mathcal{M}$  nur dann gültig, falls sich jeder enthaltene Geschäftsprozess in einen EPK-Graphen transformieren lässt.

Die Transformation selbst wird durch eine Tiefensuche vorgenommen. Sie wird von allen in einem Geschäftsprozess  $GP$  enthaltenen Funktionen gestartet.  $GP$  muss eine Variante sein, da das Element  $GP$  einen konkreten Geschäftsprozess und keinen Master für einen Geschäftsprozess darstellt. Sei  $GP \in \mathbf{businessprocesses}$  ein Geschäftsprozess eines Modellgraphen  $\mathcal{M} = (T, \mathbb{T}_{\mathcal{M}}, \mathbb{T}_{OP, \mathcal{M}}, V_{\mathcal{M}}, E_{\mathcal{M}}, \iota_{\mathcal{M}}, \mathbb{M}, \text{VARIANTS}, \Gamma, \rho, \dots, \iota_B)$ , dann wird der EPK-Graph  $G = (V, E, \iota, \mathbb{T}, \mathbb{T}_{OP})$  wie folgt gebildet:

$G$  ist zunächst undefiniert, weshalb  $V$  und  $E$  leere Mengen sind und  $\iota, \mathbb{T}$  sowie  $\mathbb{T}_{OP}$  sowie  $refEl$  undefiniert sind. Im Laufe der Tiefensuche werden sie um die Elemente der EPK ergänzt. Bei den Mengen  $V$  und  $E$  geschieht dies durch die Mengenoperationen und bei den Funktionen  $\iota, \mathbb{T}$  und  $\mathbb{T}_{OP}$  durch die Operation  $\leftarrow$ , die Funktionen an einer Stelle definieren. So bedeutet  $\mathbb{T}_{OP}(v) \leftarrow \mathbb{O}$ , dass  $v$  der Typ „oder-Operator“ zugewiesen wurde und somit  $\mathbb{T}_{OP}(v')$  für  $v' = v$  „ $\mathbb{O}$ “ zurück gibt.  $\leftarrow$  kann auch zur Definition von mehreren Elementen in einem Schritt verwendet werden. So bedeutet  $\mathbb{T}(V) \leftarrow \mathbf{function}$ , dass für alle Elemente aus der Menge  $V$  der von  $\mathbb{T}$  zugeordnete Typ „**function**“ ist. Schließlich kann  $\leftarrow$  auch als Zuweisungsoperator verwendet werden, so dass  $V \leftarrow \{v\}$  bedeutet, dass die Menge  $V$  aus dem Element  $v$  besteht.

Für die Beschreibung der Bedingungen wird eine Kurzform für die Ergebnisse verwendet: „ $\mathbb{T}(v) = (\mathbf{function} \vee \mathbf{function}\uparrow)$ “ steht für „ $(\mathbb{T}(v) = \mathbf{function}) \vee (\mathbb{T}(v) = \mathbf{function}\uparrow)$ “. Zuerst werden alle Funktionen, die in  $GP$  enthalten sind,  $V$  hinzugefügt. Von allen Knoten aus  $V$  wird nun eine Tiefensuche über die Kanten  $E_{EPK}$  der ereignisgesteuerten Prozesskette gestartet. Jeder gefundene Knoten wird nun in  $V$  eingefügt und die entsprechende Kante  $e$  hinzugefügt (vgl. Algorithmus 3.1). Nachdem der Graph aufgebaut ist, werden die Beschriftungen aus  $\mathcal{M}$  übernommen (vgl. Algorithmus 3.3) und die Funktionen  $\mathbb{T}$  sowie  $\mathbb{T}_{OP}$  definiert. Bei  $\mathbb{T}_{OP}$  werden die Typen aus  $\mathcal{M}$  übernommen. Die Definition von  $\mathbb{T}$  orientiert sich an  $\mathcal{M}$ , jedoch sind Ereignisse, die direkt nach Funktionen oder Funktionswegweisern folgen, Trivialereignisse (vgl. Algorithmus 3.2). Nach Anwendung der drei Algorithmen entsteht ein Graph  $G = (V, E, \iota, \mathbb{T}, \mathbb{T}_{OP})$ . Falls er die in Abschnitt 3.4 vorgestellten Bedingungen erfüllt, so ist die Konvertierung erfolgreich verlaufen und  $G$  stellt einen gültigen EPK-Graphen dar.

Der Algorithmus 3.1 fasst ereignisgesteuerte Prozessketten, die über mehrere Grafiken verteilt sind, zu einem EPK-Graphen zusammen. Folglich sind in  $G$  keine Funktionswegweiser enthalten. Ein Funktionswegweiser wird dadurch erkannt, dass von  $V'$  ausgehend Funktionen erreicht werden, die über die Kante „ist enthalten in“ mit einem anderen Geschäftsprozess  $GP' \neq GP$  verbunden ist. Falls Funktionswegweiser den Übergang in einen anderen Prozess darstellen, so darf der Algorithmus bei Erreichen eines Funktionswegweisers nicht weitersuchen. Weiterhin fügt der Algorithmus falsche Knoten hinzu. Falls ein Funktionswegweiser in den Prozess führt, so können nach ihm Knoten folgen, die auf dem Weg in den Prozess liegen und nicht zum eigentlichen Prozess gehören. Dies ist für den erweiterten Kreditantragsprozess der Bank AG aus Abbildung D.3 für alle Knoten von „Kredit Vertrag anlegen“ bis „Kredit Bedarf vorhanden“ der Fall. Hier würde die Tiefensuche auch den Knoten „Konto Abfrage ablehnen“ aus dem Kontoeröffnungsprozess aus Abbildung D.2 hinzufügen, der nicht zu dem Kreditantragsprozess gehört.

Um dies zu verhindern, wird die Tiefensuche in zwei Phasen aufgeteilt. Zuerst werden alle Funktionen, die in  $GP$  enthalten sind,  $V$  hinzugefügt. Von diesen wird eine Tiefensuche

---

**Algorithmus 3.1** Generierung der Mengen  $V$  und  $E$  aus einem Geschäftsprozess  $GP$

---

```

function GENERIERE GRAPH MIT FUNKTIONSWEGWEISERN ALS ÜBERGANG( $GP$ )
   $V \leftarrow \{v \mid v = \pi_2(e), e \in E_{\mathcal{M}}, \pi_1(e) = GP, \pi_3(e) = \text{beinhaltet}\}$ 
   $V' \leftarrow V$ 
   $E \leftarrow \emptyset$ 
   $converted \leftarrow \text{FALSE}$  ▷ Markiert bereits übernommene Knoten
  ▷  $converted : V \mapsto \{\text{TRUE}, \text{FALSE}\}$ , FALSE falls noch nicht definiert
  for all  $v \in V'$  do
    BESUCHE ELEMENT( $v$ )
  end for
end function

 ${}^l_{EPK,forward} = \{\text{erzeugt, löst aus, ist Vorgänger von}\}$ 
 ${}^l_{EPK,backward} = \{\text{wird erzeugt von, wird ausgelöst durch, hat als Vorgänger}\}$ 
function BESUCHE ELEMENT( $v$ )
  if  $converted(v) = \text{TRUE}$  then
    return  $v$ 
  end if
   $converted(v) = \text{TRUE}$ 
   $V \leftarrow V \cup \{v\}$ 
  ▷ Besuche alle Vorgänger und Nachfolger von  $v$ 
  for all  $e \in \{e' \mid e' \in E_{\mathcal{M}}, \pi_1(e) = v, \pi_3(e) \in {}^l_{EPK,forward}\}$  do
    BESUCHE ASSOZIATION( $e, v, \text{FALSE}$ )
  end for
  for all  $e \in \{e' \mid e' \in E_{\mathcal{M}}, \pi_1(e) = v, \pi_3(e) \in {}^l_{EPK,backward}\}$  do
    BESUCHE ASSOZIATION( $e, v, \text{TRUE}$ )
  end for
  for all  $e \in \{e' \mid e' \in E_{\mathcal{M}}, \pi_2(e) = v, \pi_3(e) \in {}^l_{EPK,forward}\}$  do
    BESUCHE ASSOZIATION( $e, v, \text{TRUE}$ )
  end for
  for all  $e \in \{e' \mid e' \in E_{\mathcal{M}}, \pi_2(e) = v, \pi_3(e) \in {}^l_{EPK,backward}\}$  do
    BESUCHE ASSOZIATION( $e, v, \text{FALSE}$ )
  end for
  return ( $V, E$ )
end function

function BESUCHE ASSOZIATION( $e, v, fv$ )
  ▷  $fv : \begin{cases} \text{TRUE} : & \text{erstes Tupelelement der Kante beinhaltet den neuen Knoten} \\ \text{FALSE} : & \text{zweites Tupelelement der Kante beinhaltet den neuen Knoten} \end{cases}$ 
  if  $fv = \text{TRUE}$  then
     $w \leftarrow \text{BESUCHE ELEMENT}(\pi_1(e))$ 
     $e' \leftarrow (w, v)$ 
  else
     $w \leftarrow \text{BESUCHE ELEMENT}(\pi_2(e))$ 
     $e' \leftarrow (v, w)$ 
  end if
   $E \leftarrow E \cup \{e'\}$ 
end function

```

---

**Algorithmus 3.2** Setzen des Knotentyps

---

```

function SETZE KNOTENTYPEN
  for all  $v \in V$  do
    if  $\mathbb{T}(v) \neq \perp$  then
      if  $\mathbb{T}_{\mathcal{M}}(v) = \text{function}$  then
         $\mathbb{T}(v) \leftarrow \text{function}$ 
      else if  $\mathbb{T}_{\mathcal{M}}(v) = \text{event}$  then
        if  $(|\delta^-(v)| = 1) \wedge (\mathbb{T}(\text{adj}_1^-(v)) = (\text{function} \vee \text{function}\uparrow))$  then
           $\triangleright$  Ein Ereignis mit einer Funktion als Vorgänger ist ein Trivialereignis
           $\mathbb{T}(v) \leftarrow \text{trivialevent}$ 
        else
           $\mathbb{T}(v) \leftarrow \text{event}$ 
        end if
      else if  $\mathbb{T}_{\mathcal{M}}(v) = \text{operator}$  then
         $\mathbb{T}(v) \leftarrow \text{operator}$ 
         $\mathbb{T}_{\text{OP}}(v) \leftarrow \mathbb{T}_{\text{OP},\mathcal{M}}(v)$ 
      end if
    end if
  end for
end function

```

---

**Algorithmus 3.3** Setzen des Beschriftung

---

```

function SETZE BESCHRIFUNG
  for all  $v \in V$  do
     $\iota(v) \leftarrow \iota_{\mathcal{M}}(v)$ 
  end for
end function

```

---

**Algorithmus 3.4** Generierung von  $G = (V, E, \iota, \mathbb{T}, \mathbb{T}_{\text{OP}})$  mit Funktionswegweisern als Verbindung zu einem anderen Prozess

---

```

function GENERIERE EPK-GRAPH MIT FUNKTIONSWEGWEISERN ALS ÜBERGANG( $GP$ )
  GENERIERE GRAPH MIT FUNKTIONSWEGWEISERN ALS ÜBERGANG( $GP$ )
  SETZE KNOTENTYPEN
  SETZE BESCHRIFUNG
  return  $(V, E, \iota, \mathbb{T}, \mathbb{T}_{\text{OP}})$ 
end function

```

---

in der Kontrollflussrichtung der EPK durchgeführt. Der rekursive Selbstaufwurf wird bei Erreichen eines Funktionswegweisers abgebrochen. Diesem Knoten wird an dieser Stelle der Typ Funktionswegweiser zugewiesen, damit der Algorithmus 3.2 zur Setzung des Knotentyps nicht modifiziert werden muss (vgl. Algorithmus 3.5). Nun sind alle Knoten gefunden worden, die zum eigentlichen Prozess gehören. Die Knoten, die in den Prozess führen, werden durch eine Rückwärtssuche von allen in  $GP$  enthaltenen Funktionen, hinzugefügt (vgl. Algorithmus 3.6). Nach dem Hinzufügen der Wege in den eigentlichen Prozess werden die Knotentypen, wie in Algorithmus 3.2 dargestellt, gesetzt.

Da Funktionswegweiser und die nachfolgenden Ereignisse den Weg in den Prozess darstellen, werden die Elemente von dem Funktionswegweiser bis zum ersten join-Operator zu einem Ereignis zusammengefasst. In der Zusammenfassung tritt jedes Ereignis als Bedingung auf. Die Bedingungen selbst sind durch „und“ verknüpft, da jede Bedingung erfüllt sein muss, um den Weg zum ersten join-Operator zu gehen. Für die Zusammenfassung wird die Funktion  $EvtCond(v)$  eingeführt, die für das neue Ereignis die zusammengefasste Bedingung zurück gibt. Um das neue Ereignis von anderen Ereignissen unterscheiden zu können, gibt die Funktion  $EvtCond(v)$  den Wert  $\perp$  zurück, falls  $v$  kein zusammengefasstes Ereignis darstellt.

Damit bei der Umsetzung nach BPEL der Datenfluss richtig umgesetzt wird, ist das entstandene Ereignis mit dem Funktionswegweiser versehen. So können beim Export die Datenflusskanten des Funktionswegweisers für die generierte Aktivität verwendet werden. Dazu wird die Funktion  $refEl : V \mapsto V_M$  eingeführt, die für ein zusammengefasstes Ereignis den passenden Funktionswegweiser zurückgibt. Da die beiden Funktionen  $EvtCond$  und  $refEl$  nicht die zentrale Komponente des Mappings darstellen, werden sie nicht in das Tupel des EPK-Graphen aufgenommen, sondern als global vorhanden angenommen.

---

**Algorithmus 3.5** Generierung der Mengen  $V$  und  $E$  aus einem Geschäftsprozess  $GP$  mit Funktionswegweisern als Prozessgrenze.

---

```
function GENERIERE GRAPH MIT FUNKTIONSWEGWEISERN ALS PROZESSGRENZE( $GP$ )  
   $V \leftarrow \{v \mid v = \pi_2(e), e \in E_M, \pi_1(e) = GP, \pi_3(e) = \text{beinhaltet}\}$   
end function
```

---

---

**Algorithmus 3.6** Generierung der Wege in den EPK-Graphen

---

```
function GENERIERE WEGE IN DEN EPK-GRAPHEN( $GP$ )  
   $V' \leftarrow \{v \mid v = \pi_2(e), e \in E_M, \pi_1(e) = GP, \pi_3(e) = \text{beinhaltet}\}$   
end function
```

---

---

**Algorithmus 3.7** Generierung von  $G = (V, E, \iota, \mathbb{T}, \mathbb{T}_{OP})$  mit Funktionswegweisern als Prozessgrenze

---

```
function GENERIERE EPK-GRAPH MIT FUNKTIONSWEGWEISERN ALS PROZESSGRENZE( $GP$ )  
    GENERIERE GRAPH MIT FUNKTIONSWEGWEISERN ALS PROZESSGRENZE( $GP$ )  
    GENERIERE WEGE IN DEN EPK-GRAPHEN( $GP$ )  
    SETZE KNOTENTYPEN  
    SETZE BESCHRIFTUNG  
    GENERIERE ZUSAMMENGEFASSTE ELEMENTE  
    return ( $V, E, \iota, \mathbb{T}, \mathbb{T}_{OP}$ )  
end function
```

---



# BUSINESS PROCESS EXECUTION LANGUAGE FOR WEB SERVICES

---

Die „Business Process Execution Language for Web Services“ (BPEL4WS) ist eine Sprache zur Beschreibung einer Zusammenschaltung von Web Services zu einem Prozess.

Ein Web Service ist ein von Maschinen angebotener Dienst, der über Operationen Nachrichten austauscht. Das Besondere an Web Services ist die Idee, die Schnittstellenbeschreibung eines Dienstes in zwei Teile zu teilen. Im ersten Teil wird der strukturelle Aufbau der Operationen und Nachrichten des Web Services beschrieben. Die konkrete Beschreibung, wie die Operationen aufzurufen sind und wie die Nachrichten versendet werden, erfolgt im zweiten Teil. Deshalb wird der erste Teil „abstrakte Definition“ und der zweite Teil „konkrete Beschreibung“ genannt. Die Web Service-Architektur stellt keine Anforderung an die zu verwendende Programmiersprache, Protokolle oder Binärformate.

Damit Web Services miteinander kommunizieren können, verwenden sie den „Service Bus“. Der „Service Bus“ übernimmt sowohl die Transformation der Daten als auch die Kommunikation: Jeder Web Service kommuniziert durch den Service Bus und dieser kommuniziert mit dem angesprochenen Web Service.

Im ARIS-Haus sind Web Services sowohl in der Daten-, in der Funktions- als auch in der Organisationssicht angesiedelt: Web Services definieren die Nachrichten, die ausgetauscht werden. Weiterhin werden von Web Services Operationen angegeben, die diese Daten verarbeiten können. Ein Web Service wird von einer bestimmten Maschine ausgeführt. Diese Maschine ist ein Teil der Aufbauorganisation. Durch die Aufteilung in die abstrakte Definition und die konkrete Beschreibung gehört ein Web Service im ARIS-Haus sowohl zu dem DV-Konzept als auch zur Implementierung (vgl. Abbildung 2.3).

Web Services kommunizieren nicht nur miteinander, sondern bieten auch Dienste an, die verwendet werden können. Die SCHUFA bietet beispielsweise einen Scoring-Service an. Dieser Service liefert für eine Person die Kreditwürdigkeit zurück (vgl. [SCHUFA]). Für eine automatisierte Kreditvergabe müsste dieser Dienst mit weiteren Web Services kombiniert werden. Eine solche Zusammenstellung wird auch „Orchestrierung von Web Services“ genannt. Die Business Process Execution Language for Web Services (BPEL4WS) ist eine Sprache, um eine Orchestrierung von Web Services vorzunehmen. Aktuell ist die

Version 1.1, welche von dem „OASIS Web Service Business Prozess Execution Language Technical Committee“ gepflegt und weiterentwickelt wird. Im folgenden soll „BPEL“ für die aktuelle Version 1.1 stehen.

Durch BPEL können sowohl abstrakte als auch ausführbare Prozesse beschrieben werden. Abstrakte Prozesse sind nicht ausführbar, sondern repräsentieren die Struktur eines Prozesses. Dies wird unter anderem zur Beschreibung von Protokollen zum Nachrichtenaustausch zwischen Geschäftspartnern verwendet. Durch die Möglichkeit, Prozesse sowohl abstrakt als auch ausführbar zu beschreiben, ist BPEL Bestandteil des DV-Konzepts und der Implementierung der Steuerungssicht im ARIS-Haus.

Zu einer vollständigen Beschreibung eines BPEL-Prozesses gehört auch die Beschreibung der verwendeten und angebotenen Dienste. Sowohl die verwendeten als auch die angebotenen Dienste sind Web Services, die durch WSDL spezifiziert werden. Bei dem Mapping einer ereignisgesteuerten Prozesskette wird sowohl eine BPEL-Prozessbeschreibung als auch eine WSDL-Beschreibung erzeugt. Zum Verständnis der Abbildung werden nun sowohl WSDL als auch BPEL kurz beschrieben. Eine ausführliche Erläuterung findet sich in den Spezifikationen von WSDL und BPEL ([WSDL], [BPEL4WS]).

### 4.1. WSDL

Die „Web Service Description Language“ (WSDL) ist ein XML-Format, durch das die Schnittstelle von Web Services spezifiziert wird. XML selbst ist in [XML] ausführlich beschrieben, weshalb an dieser Stelle nicht näher auf XML eingegangen wird.

Der prinzipielle Aufbau eines WSDL-Dokuments ist wie folgt:

```
<definitions name="nmtoken"? targetNamespace="uri"?>
  <import namespace="uri" location="uri"/>*
  <documentation.../>?
  <types>?
  <message name="nmtoken">*
  <portType name="nmtoken">*
  <binding name="nmtoken" type="qname">*
  <service name="nmtoken">*

  <!-- extensibility element -->*
</definitions>
```

In `types` werden XML-Typen durch XML-Schema und neue XML-Elemente definiert. `message`, `portType`, `binding` und `service` werden in den nächsten Abschnitten erläutert.

Ein WSDL-Dokument besteht aus der Beschreibung der ausgetauschten Nachrichten (`messages`), Schnittstellen (`portTypes`), der Bindungen an einen Service (`bindings`) und der Dienste, die Schnittstellen anbieten (`services`).

Nachrichten werden von Operationen gesendet und empfangen. Eine WSDL-Nachricht besteht aus beliebig vielen Teilen. Jeder Teil besitzt einen bestimmten Typ. Dieser kann direkt durch `type` angegeben werden oder indirekt durch die Angabe eines `element`. Die referenzierten Typen sind entweder unter dem `types`-Element oder in einer anderen XML-Datei definiert.

```
<message name="nmtoken">*
  <part name="nmtoken" element="qname"? type="qname"?/>*
</message>
```

Die Angabe von Nachrichten definiert den strukturellen Aufbau. Die Vorgehensweise für die Serialisierung der Daten wird durch das unten beschriebene Binding spezifiziert.

Ein `portType` beschreibt die Schnittstelle eines Web Services, d. h. welche abstrakten Operationen von dem Web Service angeboten und welche abstrakten Nachrichten verstanden werden. Eine Operation kann Nachrichten senden, empfangen oder einen Fehler werfen. Die Kommunikation mit dem Kommunikationspartner geschieht über die oben beschriebene Nachrichten.

```
<portType name="nmtoken">
  <operation name="nmtoken" parameterOrder="nmtokens"?>*
    <input name="nmtoken"? message="qname"/>?
    <output name="nmtoken"? message="qname"/>?
    <fault name="nmtoken"? message="qname"/>*
  </operation>
</portType>
```

Durch `parameterOrder` kann die Reihenfolge der Teile (`parts`) in den verwendeten Nachrichten angegeben werden. Dies kann wichtig werden, falls nach RPC gebunden wird.

Die konkreten Implementierungen von `portTypes` werden durch `ports` realisiert. Ein Service bietet `ports` an.

```
<service name="nmtoken">
  <port name="nmtoken" binding="qname">*
    <!-- extensibility element -->
  </port>
</service>
```

Im `extensibility element` des `ports` werden die für das implementierte binding erforderlichen Daten angegeben.

Das Binding stellt die Verbindung zwischen einem `portType` und einem `port` her: Es beschreibt, wie die Nachrichten kodiert sind und wie auf die Operationen zugegriffen wird.

```
<binding name="nmtoken" type="qname">
  <!-- extensibility element (1) -->*
  <operation name="nmtoken">*
    <!-- extensibility element (2) -->*
```

```
<input name="nmtoken"?>?
  <!-- extensibility element (3) -->*
</input>
<output name="nmtoken"?>?
  <!-- extensibility element (4) -->*
</output>
<fault name="nmtoken">*
  <!-- extensibility element (5) -->*
</fault>
</operation>
</binding>
```

Der Wert des `type`-Attributs gibt den `portType` an, dessen Nachrichten und Operationen gebunden werden. In den `extensibility elements` wird das konkrete Binding (wie beispielsweise ein Binding nach Java-RPC oder SOAP) angegeben. Die Bindings müssen in XML-Elementen angegeben sein. Ansonsten gibt es keine Vorschriften.

Ein BPEL-Prozess kommuniziert mit Partnern. Verbindungen zwischen den Partnern werden durch `partnerLinks` geschaffen. Jeder `partnerLinks` verbindet `portTypes` miteinander. Falls die Kommunikation nur in eine Richtung stattfindet, gehört nur ein `portType` zu einem `partnerLink`. Die `partnerLinks`, die ein BPEL-Prozess verwendet, werden im BPEL-Prozess selbst definiert. In einer WSDL-Datei werden die möglichen `partnerLinks` durch `partnerLinkTypes` spezifiziert (vgl. [BPEL4WS, S. 33 f.]). In jedem `partnerLinkType` werden Rollen angegeben, die von den Partnern eingenommen werden können.

```
<definitions name="ncname"
  targetNamespace="uri"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/">
  ...
  <plnk:partnerLinkType name="ncname">
    <plnk:role name="ncname">
      <plnk:portType name="qname"/>
    </plnk:role>
    <plnk:role name="ncname">?
      <plnk:portType name="qname"/>
    </plnk:role>
  </plnk:partnerLinkType>
  ...
</definitions>
```

`properties` dienen der Filterung eines Teils von Nachrichten. Sie werden von `Correlation Sets` referenziert, wodurch Nachrichten einer konkreten Prozessinstanz zugeordnet werden können (vgl. [BPEL4WS, S. 36]). Eine `property` besteht aus einem eindeutigen Namen und einem simplen XML-Schema-Datentyp.

```

<definitions name="ncname"
  xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/">
  ...
  <bpws:property name="ncname" type="qname"/>
  ...
</wsdl:definitions>

```

Durch einen `propertyAlias` wird eine spezielle Eigenschaft aus einer Nachrichten herausgefiltert werden.

```

<bpws:propertyAlias propertyName="qname"
  messageType="qname" part="ncname" query="queryString"/>

```

Durch `part` wird ein „part“ einer Nachricht angesprochen und durch `query` kann in diesem „part“ ein Element herausgefiltert werden. Die Syntax entspricht der von XQuery (vgl. [XQuery]). Falls von dem Prozess auf die durch `propertyName` referenzierte Property zugegriffen wird, wird das definierte Alias ausgewertet.

## 4.2. Die Säulen von BPEL

In der BPEL-Spezifikation wird ein Gerüst für die Beschreibung von Prozessen gelegt. Dieses Gerüst kann dazu verwendet werden, abstrakte und ausführbare Prozesse zu beschreiben. Wie die abstrakten und ausführbaren Prozesse genau definiert werden müssen, wird in der BPEL-Spezifikation beschrieben.

Eine Erweiterung für BPEL ist die „IBM Business Process Execution Language for Web Services Java™ Run Time“ (BPWS4J). Mit ihr ist es möglich, in Java geschriebene Aktivitäten, direkt in den Prozess einzubinden (vgl. [BPWS4J]). In dem BPEL4people-Whitepaper ist beschrieben, wie BPEL erweitert werden kann, so dass mit BPEL die Ausführung von Aktivitäten durch Menschen modelliert werden kann (vgl. [BPEL4people]). In der Abbildung 4.1 ist der Zusammenhang zwischen dem BPEL-Grundgerüst und den Erweiterungen grafisch dargestellt.

In dem im Kapitel 5 erläuterten Mapping werden nur abstrakte BPEL-Prozesse erstellt, weshalb im Folgenden BPEL4people und BPWS4J nicht näher ausgeführt werden.

## 4.3. Hauptbestandteile eines BPEL-Dokuments

Ein BPEL-Dokument enthält folgende Hauptbestandteile: Partner Links, Correlation Sets, Variablen, Handlers und Aktivitäten. Diese werden in den folgenden Abschnitten erläutert.

Durch einen `PartnerLink` werden die in einem `PartnerLinkType` vorgegebenen Rollen zugeordnet (vgl. [BPEL4WS, S. 34 f.]).

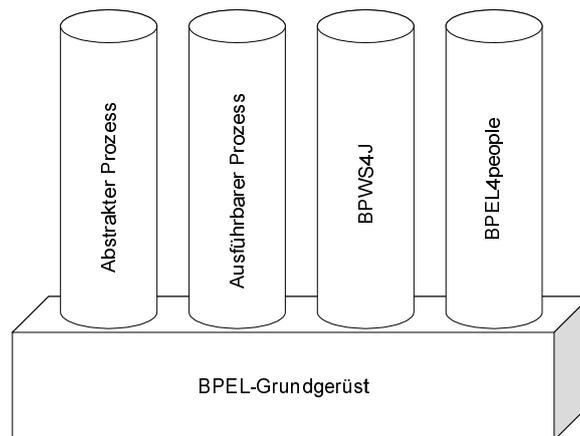


Abbildung 4.1.: Die Säulen von BPEL

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

Falls Aktivitäten Nachrichten senden oder empfangen, so wird dort immer ein Partner-Link angegeben. Dadurch ist klar, über welchen `portType` die angegebene Operation angesprochen wird: Bei einem `invoke` wird der der `partnerRole` zugehörige `portType` verwendet, bei einem `receive` hingegen der der `myRole` zugehörige `portType`.

In der aktuellen BPEL-Spezifikation BPEL4WS 1.1 muss zusätzlich bei jeder Verwendung einer Operation der `portType` angegeben werden. Dies wird jedoch in zukünftigen Versionen optional sein, da diese Information redundant ist (vgl. [BPELissues, Issue 44]).

Correlation Sets erlauben die Zuordnung von Nachrichten zu einer konkreten Prozessinstanz. Ein `correlationSet` besitzt einen eindeutigen Namen und referenziert ein oder mehrere in einem WSDL-Dokument definierte `properties`.

```
<correlationSets>?
  <correlationSet name="ncname" properties="qname-list"/>+
</correlationSets>
```

Anhand der angegebenen `Properties` werden eingehende Nachrichten den Prozessinstanzen zugeordnet. Bei jeder Aktivität, die Nachrichten empfängt und bei `reply` werden die zugehörigen Correlation Sets wie folgt angegeben:

```
<correlations>?
  <correlation set="ncname" initiate="yes|no"?
</correlations>
```

Falls `initiate="yes"`, wird das Correlation Set initialisiert, ansonsten wird es zur Zuordnung zu der passenden Prozessinstanz verwendet. Dies ist notwendig, um beispielsweise

von einer Zuordnung durch Kundennummern auf eine Zuordnung durch Auftragsnummern zu wechseln (vgl. [BPEL4WS, S. 46]). Bei `invoke` muss bei einem synchronen Aufruf zusätzlich noch ein `correlation-pattern` angegeben werden. Dieses Pattern gibt an, ob die Correlation bei der eingehenden, ausgehenden oder bei beiden Nachrichten angewandt werden soll.

```
<correlations>?
  <correlation set="ncname" initiate="yes|no"?
    pattern="in|out|out-in"/>+
</correlations>
```

Eine BPEL-Variable kann entweder eine WSDL-Nachricht oder eine beliebige XML-Struktur, die durch ein Schema definiert wird, enthalten. Die Deklaration erfolgt durch ein XML-Element:

```
<variables>
  <variable name="ncname" messageType="qname"? type="qname"?
    element="qname"? />+
</variables>
```

Mittels der Aktivität `assign` können Variablen Werte zugewiesen werden (vgl. [BPEL4WS, S. 42 ff.]). Diese wird hier nicht näher beschrieben, da sie in dem in Kapitel 5 angestrebten Mapping nicht benötigt wird.

Der Zugriff auf eine Variablen erfolgt mittels

```
{bpws:getVariableProperty('variableName', 'propertyName')}
```

oder mittels

```
bpws:getVariableData('variableName', 'partName?', 'locationpath?').
```

Im ersten Fall wird das dem Property-Namen `portyName` zugeordnete `propertyAlias` auf der angegebenen Nachricht ausgewertet. Im zweiten Fall gibt `partName` den Teil der Nachricht an, in dem `locationpath` ausgewertet werden soll. Eine Angabe ist nur dann zulässig, falls die Variable vom Typ einer Nachricht ist. `locationpath` ist dabei eine XQuery. XQuery ist eine Anfragesprache für XML-Dokumente. Näheres zu XQuery findet sich in [XQuery].

In BPEL gibt es zwei Typen von Aktivitäten: Basisaktivitäten und strukturierende Aktivitäten. Basisaktivitäten dienen beispielsweise dem Aufruf von Operationen oder dem Empfang von Nachrichten. Strukturierte Aktivitäten ermöglichen die Schachtelung von anderen Aktivitäten. Eine strukturierende Aktivität ist die `flow`-Aktivität. Sie ermöglicht die Graph-basierte Prozessmodellierung. Sie bildet das Grundgerüst des Mappings, weshalb sie in Abschnitt 4.6 ausführlich beschrieben wird. Im Folgenden werden die wichtigsten Eigenschaften der anderen Aktivitäten erläutert.

Jede Aktivität kann einen Namen besitzen. Falls die Aktivität in einer `flow`-Aktivität enthalten ist, kann eine `join-condition` und das Verhalten bei einem „`join failure`“ angegeben

werden. Die Semantik von join-conditions und join-failures ist im Abschnitt 4.6 beschrieben.

```
name="ncname"?
joinCondition="bool-expr"?
suppressJoinFailure="yes|no"?
```

Falls eine Aktivität innerhalb einer flow-Aktivität verwendet wird, kann sie ein- und ausgehende Kanten haben. Diese werden durch die Elemente `source` und `target` angegeben. Die flow-Aktivität unterstützt Übergangsbedingungen. Falls eine Kante eine Übergangsbedingung besitzt, so wird diese im `source`-Element als Attribut angegeben.

```
<source linkName="ncname" transitionCondition="bool-expr"?/>*
<target linkName="ncname"/>*
```

Von den Basisaktivitäten `receive`, `reply`, `invoke`, `wait` und `empty` werden vom im Kapitel 5 angestrebten Mapping `receive`, `invoke` und `empty` verwendet. Weshalb `reply` nicht verwendet werden kann, ist in Abschnitt 5.12 erläutert.

**receive** Durch die `receive`-Aktivität wird der Empfang eines Aufrufs einer Operation, die in der WSDL-Beschreibung des Prozesses definiert worden ist, realisiert. `variable` muss nur in einem ausführbaren Prozess angegeben werden. Die empfangene Nachricht wird in die durch `variable` angegebene Variable gespeichert. Das Attribut `createInstance` wird im Abschnitt 4.7 erläutert. Falls `createInstance` auf „yes“ gestellt ist, muss mindestens ein Correlation Set, das bei der Erzeugung einer neuen Prozessinstanz instanziiert wird, angegeben werden.

```
<receive partnerLink="ncname" portType="qname" operation="ncname"
  variable="ncname"? createInstance="yes|no"?
  standard-attributes>
  standard-elements
  <correlations>?
    <correlation set="ncname" initiate="yes|no"?>+
  </correlations>
</receive>
```

**reply** Eine `reply`-Aktivität sendet die Antwort zu einer durch `receive` empfangenen Nachricht. `partnerLink`, `portType` und `operation` spezifizieren, welche Operation bei welchem `portType` zu verwenden ist. Die zu sendende Nachricht ist in der durch das Attribut `variable` angegebenen Variable gespeichert und muss nur bei einem ausführbaren Prozess angegeben werden. Bei einem `reply` ist es möglich, statt einer Nachricht einen Fehler zu senden, der dann bei dem Partner behandelt werden muss.

```
<reply partnerLink="ncname" portType="qname" operation="ncname"
  variable="ncname"? faultName="qname"?
  standard-attributes>
  standard-elements
  <correlations>?
```

```

    <correlation set="ncname" initiate="yes|no"?>+
  </correlations>
</reply></receive>

```

**invoke** Ruft eine Operation, die in der WSDL-Beschreibung eines Web Services definiert worden ist, auf. Dieser Aufruf kann sowohl synchron als auch asynchron sein. Bei einem asynchronen Aufruf wird nur `inputVariable` angegeben. Diese Variable enthält die zu sendende Nachricht. Falls `outputVariable` angegeben ist, wird die Antwortnachricht der referenzierten operation darin gespeichert. In diesem Fall muss auch ein `correlation-pattern` angegeben werden. Dieses gibt an, ob die „Correlation“ nur bei der eingehenden, ausgehenden oder bei beiden Nachrichten angewandt werden soll. Weiterhin kann ein Fault- und Compensationhandler angegeben werden.

```

<invoke partnerLink="ncname" portType="qname" operation="ncname"
  inputVariable="ncname"? outputVariable="ncname"?
  standard-attributes>
  standard-elements
  <correlations>?
    <correlation set="ncname" initiate="yes|no"?
      pattern="in|out|out-in"/>+
  </correlations>
  <catch faultName="qname" faultVariable="ncname"?>*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
  <compensationHandler>?
    activity
  </compensationHandler>
</invoke>

```

**empty** Die `empty`-Aktivität führt gar nichts aus.

```

<empty standard-attributes>
  standard-elements
</empty>

```

Neben der `flow`-Aktivität dienen die Aktivitäten `sequence`, `while`, `switch` und `pick` der Strukturierung von Aktivitäten. Im Mapping werden neben der `flow`-Aktivität nur die `sequence` und die `while`-Aktivität verwendet. Wie auch bei der `reply`-Aktivität wird in Abschnitt 5.12 erläutert, weshalb `switch` und `pick` nicht verwendet werden.

**sequence** Die Aktivitäten in einer `sequence`-Aktivität werden sequenziell ausgeführt.

```

<sequence standard-attributes>
  standard-elements

```

```
    activity+
  </sequence>
```

**while** Die while-Aktivität ist die einzige Möglichkeit in BPEL Schleifen auszudrücken: Die enthaltene Aktivität wird so lange ausgeführt, solange die angegebene Bedingung erfüllt ist. Ist die Bedingung beim ersten Ausführen der while-Aktivität nicht wahr, wird die enthaltene Aktivität nicht ausgeführt und die while-Aktivität ist abgeschlossen vgl. [BPEL4WS, S. 60].

```
<while condition="bool-expr" standard-attributes>
  standard-elements
  activity
</while>
```

**switch** Ein switch führt Aktivitäten abhängig von Bedingungen aus. Trifft keine der angegebenen Bedingungen zu, so wird die in otherwise angegebene Aktivität ausgeführt.

```
<switch standard-attributes>
  standard-elements
  <case condition="bool-expr">+
    activity
  </case>
  <otherwise>?
    activity
  </otherwise>
</switch>
```

**pick** Mit einem pick lässt sich eine entweder-oder-Auswahl aus eingehenden Nachrichten realisieren. Sobald eine der angegebenen Nachrichten eingetroffen ist, werden die dazu angegebenen Aktivitäten ausgeführt. Danach ist die pick-Aktivität beendet.

Zudem kann bis zu einem bestimmten Zeitpunkt oder eine bestimmte Dauer gewartet werden, um beispielsweise Deadlines oder timeouts umzusetzen.

```
<pick createInstance="yes|no"? standard-attributes>
  standard-elements
  <onMessage partnerLink="ncname" portType="qname"
    operation="ncname" variable="ncname"?>+
    <correlations>?
      <correlation set="ncname" initiate="yes|no"?>+
    </correlations>
    activity
  </onMessage>
  <onAlarm (for="duration-expr" | until="deadline-expr")>+
    activity
  </onAlarm>
</pick>
```

## 4.4. Fehlerbehandlung und Kompensation

BPEL unterstützt Fehlerbehandlung und Kompensation (vgl. [BPEL4WS, S. 72 ff.]). Da in dem im Kapitel 5 vorgestellten Mapping nach BPEL keine Fehlerbehandlung und Kompensation benötigt werden, sind diese hier nicht erläutert.

## 4.5. Ereignisbehandlung

BPEL ermöglicht die Behandlung von Ereignissen parallel zur Ausführung von anderen Aktivitäten (vgl. [BPEL4WS, S. 80 ff.]). Ein Eventhandler reagiert auf eingehende Nachrichten, den Ablauf einer Zeitspanne oder das Erreichen eines Zeitpunkts.

```
<eventHandlers>?
  <!-- Es muss mindestens einen onMessage oder einen
        onAlarm handler geben -->
  <onMessage partnerLink="ncname" portType="qname"
            operation="ncname"variable="ncname"?>*
    <correlations>?
      <correlation set="ncname" initiate="yes|no">+
    </correlations>
    activity
  </onMessage>
  <onAlarm for="duration-expr"? until="deadline-expr"?>*
    activity
  </onAlarm>
</eventHandlers>
```

## 4.6. flow-Aktivität

Eine flow-Aktivität ermöglicht eine Graph-basierte Prozessmodellierung in BPEL. Jede Aktivität in einer flow-Aktivität ist ein Knoten in dem Graphen. Sie besitzen zwei den Fluss betreffende Attribute: `joinCondition` und `suppressJoinFailure`. In der `join-condition` wird eine Bedingung angegeben, die ausgewertet wird, sobald der Status aller eingehenden Kanten bestimmt wurde. Falls die `join-condition` wahr ist, wird die zugehörige Aktivität gestartet. Ist sie falsch, ist das Verhalten abhängig von dem `suppressJoinFailure`-Attribut. Falls dieses auf „no“ gesetzt, wird ein `bpws:joinFailure`-Fehler geworfen. Falls `suppressJoinFailure` auf „yes“ gesetzt, wird die im Abschnitt 4.6.1 erläuterte Dead Path Elimination durchgeführt (vgl. [BPEL4WS, S. 65]).

Die Kanten zwischen den Knoten werden wie folgt im XML-Element `links` angegeben:

```
this line should not appear
<links>
  <link name="ncname">+
</links>
```

Jede Aktivität gibt durch die XML-Elemente `source` und `target` an, von welchen Kanten sie Quelle und Ziel ist. Pro eingehende Kante wird ein `source`-Element angegeben und pro ausgehende Kante ein `target`-Element. Die Referenzierung der Kante erfolgt bei beiden Elementen durch das Attribut `linkName`. Bei `source` kann zusätzlich noch eine Übergangsbedingung durch das Attribut `transitionCondition` angegeben werden. Falls die Quellaktivität abgeschlossen ist, wird die ausgehende Kante nur dann auf „positiv“ gesetzt, falls die Übergangsbedingung zu `true` evaluiert wird.

Die Ausführung fängt bei den Aktivitäten ohne eingehende Kanten an. Sie werden parallel ausgeführt. Sobald eine von ihnen abgeschlossen ist, wird der Status der ausgehenden Kanten bestimmt und entsprechend fortgefahren. Näheres findet sich in [BPEL4WS, S. 63 ff.].

BPEL erlaubt innerhalb einer `flow`-Aktivität keine Zyklen, weshalb der durch die Links spezifizierte Graph azyklisch sein muss (vgl. [BPEL4WS, S. 64]).

### 4.6.1. Dead Path Elimination

In diesem Abschnitt wird ein kurzer Überblick über die Dead Path Elimination gegeben. Eine ausführliche Beschreibung ist in [BPEL4WS, S. 65 f.] und [LR99, S. 146 ff.] zu finden.

Mittels der Dead Path Elimination werden in einer `flow`-Aktivität nicht erreichbare Aktivitäten als nicht mehr ausführbar markiert und die Ausführung bei der nächsten ausführbaren Aktivität fortgesetzt. Sobald eine Aktivität beendet ist, wird der Status aller ausgehenden Links bestimmt. Ein Link erhält den Status „positiv“, falls die angegebene `transitionCondition` zu `true` evaluiert wird, und „negativ“ in allen anderen Fällen. Für jeden Nachfolger wird folgende Tiefensuche gestartet:

**Abbruchbedingung des rekursiven Selbstaufrufs** Es erfolgt ein Abbruch, falls eine der folgenden Bedingungen erfüllt ist:

1. Der Status mindestens eines eingehenden Links ist unbekannt.
2. Die Aktivität kann noch nicht ausgeführt werden. Dies ist der Fall, falls eine der folgenden Bedingungen erfüllt ist:
  - a) Die Aktivität ist Teil einer `sequence`, die `sequence` wird nicht ausgeführt oder die der Aktivität vorausgehende Aktivität wurde noch nicht ausgeführt.
  - b) Die Aktivität ist Teil eines anderen `flows`, der noch nicht gestartet wurde. Weiterhin besitzt sie keine eingehenden Kanten, die in dem `flow` spezifiziert wurden.

- c) Die Aktivität ist Teil eines Zweiges einer switch-Aktivität und dieser Zweig (noch) nicht gewählt.

**Bearbeitung des aktuellen Knoten** Falls die `joinCondition` zu „true“ evaluiert wird, wird die Aktivität gestartet. Falls die `joinCondition` zu „false“ evaluiert wird, wird überprüft, bei der Aktivität `suppressJoinFailure="no"` gilt. Falls ja, wird ein `bpws:joinFailure` geworfen. Falls der `join-failure` unterdrückt wird, wird der Status aller ausgehenden Links bestimmt und die Tiefensuche rekursiv mit jedem Nachfolger aufgerufen.

**Rekursiver Selbstaufwurf** Ein rekursiver Selbstaufwurf findet statt, falls die `joinCondition` zu „false“ evaluiert wird und `suppressJoinFailure="no"` gilt.

## 4.7. Prozesse in BPEL

In einem Prozess werden die im vorhergehenden Abschnitt vorgestellten Aktivitäten verwendet.

```
<process name="ncname"
  targetNamespace="uri"
  abstractProcess="yes|no"?
  <!-- default: no --->
  suppressJoinFailure="yes|no"?
  <!-- default: no --->
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/">
  ...
</process/>
```

Die Attribute haben folgende Bedeutung:

- `targetNamespace` stellt den Namespace, zu dem der Prozess gehört, dar.
- `abstractProcess` gibt an, ob der Prozess abstrakt oder ausführbar ist.
- `suppressJoinFailure` bestimmt global die grundsätzliche Einstellung des Attributs `suppressJoinFailure` jeder Aktivität. Es kann bei jeder Aktivität überschrieben werden. Den Einfluss auf die Ausführung des Prozesses wird im Abschnitt 4.6 beschrieben.

Der Fluss im Prozess wird durch die Schachtelung von Aktivitäten angegeben. Bei einer Instantiierung eines BPEL-Prozesses wird immer die äußerste Aktivität zuerst ausgeführt. Die Semantik der Ausführung dieser Aktivität bestimmt den weiteren Ablauf.

Eine neue Instanz eines BPEL-Prozesses wird durch die ausführende BPEL-Engine erzeugt sobald eine Nachricht eintrifft, die von keinem anderen Prozess verarbeitet werden kann und der Prozess eine Aktivität mit `createInstance="yes"` besitzt, die diese Nachricht annehmen kann (vgl. [BPEL4WS, S. 32] und [BPEL4WS, S. 55]). Daraus entsteht

#### 4. Business Process Execution Language for Web Services

---

die Forderung, dass ein BPEL-Prozess mindestens eine Aktivität enthalten muss, deren `createInstance`-Attribut auf „yes“ gesetzt ist.

Eine Prozessinstanz wird terminiert, sobald eine der drei folgenden Bedingungen zutrifft:

1. Die äußerste Aktivität eines Prozesses ist abgeschlossen
2. Ein `fault` erreicht den Scope des Prozesses
3. Explizite Benutzung von `terminate`

Eine genauere Erläuterung ist in [BPEL4WS, S. 31 f.] zu finden.

# MAPPING DES METAMODELLS NACH BPEL

---

Als Grundsatz für die Abbildung von ereignisgesteuerten Prozessketten nach BPEL gilt, dass die Struktur der ereignisgesteuerten Prozesskette in BPEL erkennbar sein soll. Deshalb wird gefordert, dass pro Knoten eines EPK-Graphen maximal eine BPEL-Aktivität erzeugt wird. Die `flow`-Aktivität wird als Umsetzung des Kontrollflusses verwendet, da sie, analog zur Verbindung von Funktionen durch Ereignisse und Operatoren, die verschiedenen Aktivitäten durch Kanten verbindet. Das in Abschnitt 2.3 beschriebene Problem der Nicht-Lokalität tritt durch die Unterdrückung von `join`-Fehlern in der `flow`-Aktivität nicht auf, da die im Abschnitt 4.6.1 beschriebene `Dead Path Elimination` das Problem der Nicht-Lokalität löst.

Für den erzeugten BPEL-Prozess werden die verwendeten Aktivitäten sowie die Abfolge der Aktivitäten benötigt. Die Aktivitäten werden von den Funktionen abgeleitet, die Abfolge aus der Verbindung der Funktionen durch Ereignisse und Operatoren. Für die Aktivitäten selbst werden Informationen über den Namen der Operation, die gesendete und empfangene Nachricht sowie der zugehörige `portType` benötigt. Der Name der Aktivität wird aus dem Namen der Funktion abgeleitet und die Information über die gesendeten und empfangenen Nachrichten aus der Datenflussinformation. Da ein Web Service eine Operation durchführt, wird als Information für den `portType` die Beziehung „führt durch“ verwendet. Die `partnerLinks` und `partnerLinkTypes` werden aus dem `portType` abgeleitet, da es kein Element gibt, das die Verbindung zu einem Geschäftspartner beschreibt.

Ein BPEL-Prozess besteht immer aus einem Hauptprozess und kann zusätzlich einen Eventhandler besitzen. Analog dazu muss ein EPK-Graph aus einer oder mehreren Zusammenhangskomponenten bestehen. Die erste Zusammenhangskomponente wird in eine `flow`-Aktivität abgebildet. Besteht der EPK-Graph aus mehreren Zusammenhangskomponenten, so müssen die weiteren Zusammenhangskomponenten genau ein Ereignis als Wurzelknoten besitzen. Die weiteren Zusammenhangskomponenten können dann als `onMessage`-Ereignis in einem Eventhandler umgesetzt werden. Falls die EPK diese Voraussetzungen nicht erfüllt, wird sie nicht in einen BPEL-Prozess abgebildet.

Funktionen werden immer auf Aktivitäten abgebildet. Abhängig von den gesendeten und empfangenen Daten wird grundsätzlich eine `receive`-, eine `invoke`- oder eine `empty`-Aktivität erzeugt. Operatoren werden grundsätzlich zu `empty`-Aktivitäten, die die `join`- und `fork`-Semantik des Operators umsetzen. Die Verbindung der Aktivitäten wird durch die Ereignisse erreicht. Die Ereignisse werden zu den Übergangsbedingungen zwischen den Funktionen und Operatoren. Ereignisse ohne eingehende Kante sind hiervon ausgenommen. Sie stellen ein externes Ereignis dar und werden in der ersten Zusammenhangskomponente als `receive`-Aktivität umgesetzt.

Da einerseits in ereignisgesteuerten Prozessketten beliebige Zyklen auftreten können und andererseits BPEL nur `while`-Schleifen erlaubt, müssen Zyklen der ereignisgesteuerten Prozesskette in `while`-Schleifen umgesetzt werden. Beliebige Zyklen können durch die Technik der Knotenaufspaltung in Zyklen mit genau einem Eingang transformiert werden. Durch die Knotenaufspaltung entstehen neue Knoten, so dass eine Funktion zur Generierung von mehreren BPEL-Aktivitäten führt. Dies verletzt den Grundsatz, dass jede Funktion immer auf eine Aktivität abgebildet wird. Deshalb werden ereignisgesteuerte Prozessketten, die ein Aufteilen von Funktionen erfordern, nicht nach BPEL exportiert.

Eine Einbindung der in Abschnitt 3.6.15 vorgestellten spezifischen Eigenschaften findet nicht statt, da diese nicht in jedem Modell enthalten sind und im Algorithmus gesondert behandelt werden müssten. Spezifische Eigenschaften würden zusätzliche Informationen, wie beispielsweise die Typen der `parts` von Nachrichten einbringen, jedoch würden sie nichts am Grundsatz der Abbildung ändern.

Jeder in einem Modell  $\mathcal{M}$  beschriebene Geschäftsprozess  $GP$  wird als ein BPEL-Prozess exportiert. Da Funktionswegweiser einerseits die Verbindung zu einem anderen Prozess und andererseits eine Prozessgrenze darstellen können, wird der Schalter `FUNCTIONPOINTERSAREPROCESSBORDER` eingeführt. Steht der Schalter auf `WAHR`, so stellen Funktionswegweiser die Prozessgrenze dar und es wird der Algorithmus 3.4 zur Generierung von  $G = (V, E, \iota, \mathbb{T}, \mathbb{T}_{OP})$  verwendet. Steht der Schalter `FUNCTIONPOINTERSAREPROCESSBORDER` auf `FALSCH`, so sind Funktionswegweiser die Verbindung in einen anderen Prozess und der Algorithmus 3.7 wird zur Generierung von  $G = (V, E, \iota, \mathbb{T}, \mathbb{T}_{OP})$  verwendet.

Im Metamodell ist es nicht möglich, eine Schachtelung von Elementen, die Daten beschreiben, durch Beziehungen zu modellieren (vgl. Abschnitt 3.6.18). Da es weiterhin nur die vier Datentypen `Beleg`, `Dokument`, `Information` und `Ware` gibt, ist es nicht möglich, Typen von WSDL-Nachrichten zu modellieren. Deshalb können die Elemente nicht zur Beschreibung von Variablen verwendet werden, was dazu führt, dass durch das Mapping keine Variablenbehandlung erzeugt wird. Weiterhin existiert keine Beziehung zwischen Objekten und den Elementen zur Beschreibung der Datenobjekte, so dass keine Extraktion von Daten aus einer WSDL-Nachricht in Variablen für die Weiterverarbeitung durch `Transitionconditions` statt findet. Aus diesen Gründen wird jeder Geschäftsprozess in einen abstrakten BPEL-Prozess abgebildet.

## 5.1. Grundsätzliche Umsetzung der Beschriftungen

Bei der Generierung des name-Attributs der XML-Elemente werden die Beschriftungen der Elemente verwendet. Da Leerzeichen und Doppelpunkte in Namen problematisch sind, werden sie durch Unterstriche ersetzt. Im Folgenden wird die Bezeichnung „BPEL-Name“ für das name-Attribut eines XML-Element in der BPEL- und WSDL-Datei verwendet.

Der BPEL-Name von Funktionen setzt sich aus dem Kontext der Funktion und ihrem Namen zusammen. Die Umsetzung des Kontexts erfolgt durch die Umsetzung des zugehörigen Wegs. Die Namen der Knoten auf dem Weg werden durch einen Punkt getrennt aneinandergereiht. Abschließend wird an diese Aneinanderreihung ein Punkt sowie der Name der Funktion angehängt. Dies ergibt den Namen der Aktivität, in die die Funktion umgesetzt wird.

Da es keinen eindeutigen Kontext von Ereignissen gibt (vgl. Abschnitt 3.6.12), ist der Name eines Ereignisses seine Beschriftung. Die Ausnahme bilden die zusammengefassten Ereignisse, deren BPEL-Namen durch *EvtCond* bestimmt wird.

Operatoren besitzen, wie Ereignisse, keinen eindeutigen Kontext, weshalb sich ihr Name aus dem Typ und der Nummer des Operators zusammen setzt. Der Name eines und-Operators ergibt sich zu AND<Nummer>, der eines oder-Operators zu OR<Nummer> und der eines entweder-oder-Operators zu XOR<Nummer>.

Alle anderen Elemente, insbesondere die Tätigkeiten, tragen als Namen ihre Beschriftung. Dieses Vorgehen verursacht keine Konflikte, da Tätigkeiten, die in einer Funktion enthalten sind, paarweise einen anderen Master und so auch eine andere Beschriftung besitzen (vgl. Abschnitt 3.6.1).

## 5.2. Gerüst des Prozesses

Das Gerüst jedes abgebildeten Prozesses bildet das in Abschnitt 4.7 vorgestellte process-Element. Das name Attribut wird auf den Namen des Prozesses gesetzt. Der BPEL-Prozess ist aus den in Abschnitt 5 aufgeführten Gründen abstrakt, weshalb das abstractProcess-Attribut auf yes gesetzt wird. In dem BPEL-Prozess ist es möglich, dass Aktivitäten nicht ausgeführt werden, weshalb suppressJoinFailure auf yes gesetzt wird, um die Dead Path Elimination zu aktivieren. Da die XML-Elemente von BPEL stammen, wird als Standard-Namespace der BPEL-Namespace gewählt. Um bei den joinConditions konsistent mit der BPEL-Spezifikation zu sein, wird bpws als URI für den BPEL-Namespace definiert. Schließlich wird tns als die URI des Targetnamespaces deklariert. Dies ist nötig, um die in der WSDL-Datei definierten Typen verwenden zu können. Listing 5.1 zeigt ein solches erstelltes process-Element.

```
<process name="Name" targetNamespace="uri" abstractProcess="yes"
  suppressJoinFailure="yes"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:tns="uri"
  xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/">
  ...
</process>
```

Listing 5.1: Bestandteile eines exportierten process-Elements

### 5.3. Abbildung der eEPK

Aus der erweiterten ereignisgesteuerten Prozesskette werden die Tätigkeiten, die Information über den Datenfluss sowie die ausführenden Elemente entnommen. Bei der Auswertung der Beziehungen kann ein Geschäftsvorfall als Filter verwendet werden. Ist ein solcher Filter aktiv werden nur die verknüpften Elemente verwendet, deren Bedingungen dem Geschäftsvorfall nicht widersprechen.

Eine Funktion kann entweder immer eine Interaktion mit einem Web Service darstellen oder eine Sammlung von Tätigkeiten realisieren. Falls eine Funktion immer eine Interaktion mit einem Web Service darstellt, werden die enthaltenen Tätigkeiten nicht exportiert. Falls Funktionen eine Sammlung von Tätigkeiten realisieren, stellt jede Tätigkeit eine Interaktion mit einem Web Service dar. Die Funktion dient dann nur als Container und wird als sequence-Aktivität abgebildet und die in der Funktion enthaltenen Tätigkeiten werden als Bestandteile der sequence-Aktivität exportiert. Die Reihenfolge der Tätigkeiten in der sequence-Aktivität wird durch von  $\Gamma$  vorgegebenen Ordnung bestimmt.

Um zu unterscheiden, ob Funktionen immer eine Interaktion mit einem Web Service darstellen oder ob die Tätigkeiten als Aktivität exportiert werden sollen, wird der Schalter USEACTIVITIES eingeführt. Tätigkeiten werden nur dann abgebildet, falls der Schalter USEACTIVITIES WAHR ist. Die genaue Aktivität wird in beiden Fällen durch die Datenflussinformation und die Beziehungen zu den Elementen zur Beschreibung von Datenobjekten bestimmt.

Bei der modellierten Information über den Datenfluss fehlt die Verbindung der Elemente (vgl. Abschnitt 3.6.18). So kann nicht entschieden werden, ob gesendete Daten an eine nachfolgende Funktion oder an eine externe Funktion gesendet werden. Deshalb gibt es zwei Arten der Interpretation der Datenflussinformation:

1. Daten fließen innerhalb des Prozesses
2. Daten fließen von und zu Web Services

Falls die Semantik von Funktionen in einer EPK die Darstellung von auszuführenden Aktivitäten ist, so fließen die Daten innerhalb des Prozesses. Die Angabe von durchführenden Werkzeugen oder Mitgliedern der Organisationseinheit (vgl. Tabellen 3.22 und 3.21) ändert an diesem Sachverhalt nichts, da diese Funktion von den assoziierten Werkzeugen

oder Mitgliedern durchgeführt wird und keine andere Funktion ausgeführt wird. So ist die Assoziation, dass die Funktion „Risiko schätzen“ die Informationen „Name“ und „Betrag“ sendet immer so zu interpretieren, dass diese Informationen von der Funktion an ihre Nachfolger gesendet werden.

Falls durch die EPKs BPEL-Prozesse modelliert werden, so sind durch Funktionen die Behandlung von Aufrufen von bereitgestellten Operatoren oder Aufrufe von Operationen von Web Services modelliert. Die angegebenen Daten fließen dann sowohl von und zu Web Services als auch innerhalb des Prozesses. Da an Assoziationen der Elemente des Datenflusses nicht vermerkt ist, wohin die Daten fließen (vgl. Abschnitt 3.6.18) kann die Assoziation, dass die Funktion „Risiko schätzen“ die Informationen „Name“ und „Betrag“ sendet, entweder so interpretiert werden, dass diese Informationen an einen Web Service gesendet werden oder an nachfolgende Funktionen gesendet und folglich von einem Web Service entgegengenommen werden.

Um die Semantik der in Tabelle 3.29 aufgeführten Beziehungen klar zu definieren, gibt es den Schalter DATAFLOWISINTERNAL. Ist DATAFLOWISINTERNAL WAHR, so bedeutet „senden“, dass die Daten an eine nachfolgende Funktion gesendet und „empfangen“, dass die Daten von einer vorhergehenden Funktion gesendet und von dieser Funktion empfangen werden. Falls DATAFLOWISINTERNAL auf FALSCH steht, so bedeutet „senden“, dass die Daten an einen Web Service gesendet und „empfangen“, dass die Daten von einem Web Service empfangen werden.

Falls DATAFLOWISINTERNAL auf FALSCH steht, wird die in Tabelle 5.1 aufgeführte Abbildung vorgenommen.

| Element durch<br>sendet verknüpft | Element durch<br>empfängt verknüpft | Resultierende<br>Aktivität |
|-----------------------------------|-------------------------------------|----------------------------|
| ✓                                 | ✓                                   | synchrones invoke          |
| ✓                                 | ✗                                   | asynchrones invoke         |
| ✗                                 | ✓                                   | receive                    |
| ✗                                 | ✗                                   | empty                      |

Tabelle 5.1.: Abbildung auf die Aktivitäten bei DATAFLOWISINTERNAL FALSCH

Steht DATAFLOWISINTERNAL auf WAHR, so werden grundsätzlich Funktionen ohne eingehende Kontrollflusskanten auf eine receive-Aktivität und Funktionen mit eingehenden Kontrollflusskanten auf eine invoke-Aktivität abgebildet. Ist USEACTIVITIES WAHR und besteht eine Funktion ohne eingehenden Kontrollflusskanten aus Tätigkeiten, so wird die erste Tätigkeit auf eine receive-Aktivität und alle weiteren Tätigkeiten auf eine invoke-Aktivität abgebildet. Bei der Abbildung auf eine receive-Aktivität werden die durch „sendet“ verknüpften Elemente zu einer Nachricht zusammengefasst und durch das receive empfangen, da das receive nur die Daten prozessintern weitergeben kann, die es empfangen hat. Bei der Abbildung auf ein invoke werden die durch sendet verknüpften Elemente als eingehende Nachricht und die durch empfängt als ausgehende Nachricht umgesetzt: Die prozessintern empfangenen Daten werden an die aufgerufene

Operation gesendet. Die Umsetzung der durch sendet und empfängt angegeben Elemente in Nachrichten wird in Abschnitt 5.3.1 beschrieben.

### 5.3.1. Umsetzung in gesendete und empfangene Nachrichten

In diesem Abschnitt wird die Umsetzung der gesendeten und empfangen Elemente von Funktionen und Tätigkeiten beschrieben. Die Erläuterung erfolgt anhand der sendet-Beziehung, die Umsetzung der empfängt-Beziehung folgt analog.

Falls eine Funktion oder Tätigkeit die Elemente  $E_1 \dots E_n$  sendet, so besteht die zugehörige Nachricht aus den  $n$  Teilen (parts), die den Namen der Elemente  $E_1$  bis  $E_n$  tragen. Der Typ der Teile ist `xsd:string`, da `xsd:any` in WSDL nicht erlaubt und im Modell keine Information über die Typen vorhanden ist. Der Name der Nachricht selbst setzt sich aus den Namen der enthaltenen Teile zusammen, welche durch Unterstriche miteinander verbunden sind.

Die Reihenfolge der Teile wird aus dem ersten Vorkommen von  $E_1 \dots E_n$  bestimmt. Sendet eine Funktion an einer Stelle „Name, Betrag“ und im Kontrollfluss nachfolgend „Betrag, Name“, so ist in beiden Fällen die zugehörige Nachricht `Name_Betrag`.

Es wird davon ausgegangen, dass die Datenbeschreibungselemente mit der gleichen Beschriftung ein Datum des selben Typs bezeichnen. Deshalb wird als Name der Variable, die in den Aktivitäten verwendet wird, der Name der Nachricht verwendet. Eine auf `invoke` abgebildete Funktion, die „Name, Betrag“ sendet, hat folglich als Attributwert von `inputVariable` „Name\_Betrag“. Ferner wird dadurch eine Variable `Name_Betrag` mit dem `messageType` `Name_Betrag` und eine Nachricht mit den Teilen `Name` und `Betrag` erzeugt.

### 5.3.2. Umsetzung der Durchführungen

Die Durchführungsinformation wird für die Generierung von `portTypes`, `partnerLinks` und `partnerLinkTypes` verwendet. Zuerst werden die `portTypes` generiert und davon dann die `partnerLinks` und `partnerLinkTypes` abgeleitet. Die Ableitung der `partnerLinkTypes` aus den `portTypes` wird vollzogen, da kein Element in dem Metamodell eine Verbindung zu einem Geschäftspartner darstellt.

Durch die Beziehung „führt durch“ können Organisationseinheiten, Stellen, Mitarbeiter und Werkzeuge mit Funktionen und Tätigkeiten verbunden werden (siehe Tabellen 3.21 und 3.22). Die Namen der Elemente, die mit einer Funktion oder Tätigkeit durch die Kante „wird durchgeführt von“ verbunden sind, werden, wie in der Umsetzung in Nachrichten, durch einen Unterstrich verbunden. Diese Verbindung ist der Name des Partners, der die Tätigkeit oder die Funktion ausführt. Falls keine Elemente durch die Kante „wird durchgeführt von“ verbunden sind, so ist der Name des Partners „default“.

Eine operation gehört immer zu einem portType, weshalb die PortTypes aus den generierten Operationen abgeleitet werden. Die generierten PortTypes sind in drei Klassen unterteilt:

1. PortTypes, die alle Operationen, die in receive-Aktivitäten oder onMessage-Elementen des Eventhandlers verwendet werden, enthalten.
2. PortTypes, die alle Operationen, die in invoke-Aktivitäten, die nur von einem Partner verwendet werden, enthalten.
3. PortTypes, die alle Operationen, die in invoke-Aktivitäten, die von mehr als einem Partner verwendet werden, enthalten.

In die erste Klasse fällt genau ein PortType. In ihm sind alle Operationen enthalten, die in receive-Aktivitäten oder bei onMessage-Elementen des Eventhandlers verwendet werden. Da die Operationen in diesem PortType immer den Empfang von Nachrichten durch den Prozess vornehmen, setzt sich der Name dieses PortTypes aus den Namen des Prozesses und dem Suffix `_PT` zusammen.

Die Namen der PortTypes der zweiten Klasse setzen sich aus dem Namen des Partners und des Suffixes `_PT` zusammen.

PortTypes der dritten Klasse entstehen, falls eine Funktion im selben Prozess mehrfach vorkommt und zusätzlich von verschiedenen Partnern durchgeführt wird. Diese Funktionen tragen die selbe Beschriftung. Falls sie die selben Nachrichten senden und empfangen, werden sie auf die selbe Operation abgebildet. Da sich die Partner unterscheiden, würden für jeden Partner jeweils ein PortType mit der gleichen Operation generiert werden. Um dies zu verhindern, werden diese Operationen dem PortType „globalPortType“ zugeordnet. Somit existiert in der dritten Klasse nur dieser PortType.

In einem Modell sind keine Informationen über Konversationen enthalten. Deshalb werden die `partnerLinks` und `partnerLinkTypes` als Stubs generiert, die während einer Überarbeitung der BPEL-Prozessbeschreibung geändert werden müssen.

PartnerLinks werden aus der Funktion abgeleitet. Sie werden bei der Aktivität auf die die Funktion abgebildet wurde, verwendet. Falls `USEACTIVITIES WAHR` ist, werden die in einer Funktion enthaltenen Tätigkeiten anstatt der Funktion selbst als Aktivitäten exportiert. Für die Tätigkeiten gelten die gleichen Überlegungen, weshalb die folgenden Erläuterung auch für Tätigkeiten gültig ist.

Ein PartnerLink stellt die Verbindung zwischen zwei Prozessen dar. Deshalb wird der Name des Partners mit dem Namen des Prozesses zusammengesetzt und als Name des PartnerLinks verwendet. Der zu dem PartnerLink gehörende PartnerLinkType besitzt den Namen des PartnerLinks, ergänzt durch das Suffix `_LinkType`. In dem PartnerLinkType ist ausschließlich der PortType der Operation enthalten. Dieser ist der Rolle `myRole` mit dem Namen „myRole“ zugeordnet, falls der PortType zu der ersten Klasse der PortTypes gehört. In allen anderen Fällen ist der PortType der Rolle `partnerRole` mit dem Namen „partnerRole“ zugeordnet. Diese Zuordnung wird bei der Deklaration der PartnerLinks übernommen: Falls eine Operation einen PortType der ersten Klasse verwendet, nimmt der PartnerLink die Rolle „myRole“ ein, ansonsten die Rolle „partnerRole“.

### 5.4. Umsetzung von Funktionswegweisern

Falls der Schalter `FUNCTIONPOINTERSAREPROCESSBORDER` auf `WAHR` steht, so stellen Funktionswegweiser den Aufruf durch einen anderen Prozess oder den Aufruf einer Funktion eines anderen Prozesses dar. Im Algorithmus 3.7 wird der Fall des Aufrufs des Prozesses durch einen Funktionswegweiser durch die Umsetzung in ein Ereignis behandelt.

Funktionswegweiser, die keine ausgehenden Kanten besitzen, stellen unabhängig von `DATAFLOWISINTERNAL` immer einen Aufruf einer Funktion eines anderen Prozesses dar. Diese Funktion wird von dem Prozess *GP* aufgerufen, weshalb dieser Aufruf als asynchrones `invoke` exportiert wird. Die von der Funktion empfangenen Daten müssen bei dem Aufruf von der Funktion übermittelt werden, weshalb die `invoke`-Aktivität als `inputVariable` die von der Funktion empfangenen Daten erhält.

### 5.5. Abbildung von Ereignissen

Ein Ereignis kann nach der Definition in Abschnitt 3.5 einerseits das Resultat einer Funktion sein und andererseits ein Ereignis, das außerhalb des betrachteten Prozesses auftritt. Da im Metamodell keine Unterscheidung zwischen diesen Ausprägungen getroffen wird, wird die Unterscheidung anhand der ein- und ausgehenden Kanten getroffen. Hat ein Ereignis *e* in *G* keine eingehenden Kanten, so ist das Ereignis ein externes Ereignis, das nicht als Resultat einer Funktion im Prozess auftritt. Hat *e* keine ausgehenden Kanten, so stellt *e* einen Zustand dar, der in Folge der Ausführung von Funktionen eingetreten ist. Falls das Ereignis *e* sowohl ein- als auch ausgehende Kanten besitzt, so stellt es eine Übergangsbedingung dar. Die gleichen Überlegungen gelten für Trivialereignisse – mit dem Unterschied, dass Trivialereignisse immer einen Vorgänger haben und so nie ein externes Ereignis darstellen. Trivialereignisse sind somit bei Vorhandensein ausgehender Kanten Übergangsbedingungen und ansonsten Zustände.

Zustände werden durch eine `empty`-Aktivität ausgedrückt. Diese dienen der reinen Dokumentation und besitzen keine weitere Semantik. Externe Ereignisse werden als `receive`-Aktivitäten umgesetzt. Da Ereignisse keine Verbindung zum Datenfluss besitzen, sind keine Informationen über den Datenfluss bekannt. Deshalb werden die `receive`-Aktivitäten ohne das `inputVariable`-Attribut erzeugt.

Aus dem Graphen werden alle Ereignisse, die eine Übergangsbedingung darstellen, entfernt. Das Ziel der eingehenden Kante wird auf das Ziel der ausgehenden Kante gesetzt und die ausgehende Kante daraufhin aus dem Graphen entfernt. Die Beschriftung des Ereignisses wird als Beschriftung der vormals eingehenden Kante übernommen. Da Objekte nicht mit einem Typ versehen sind, wird im Mapping nach BPEL der Zustand als Zeichenkette interpretiert und folglich die Beschriftung zu „Objekt = 'Zustand'“ umgesetzt.

Das gleiche Prinzip wird bei Trivialereignissen angewandt. Da Trivialereignisse immer nach der Ausführung einer Funktion eintreten, bringen sie nur dann zusätzliche Information, falls mit ihnen eine erfolgreiche Ausführung einer Funktion modelliert werden soll.

Deshalb wird der Schalter `USETRIVIALEVENTS` eingeführt. Steht er auf `WAHR`, so werden Trivialereignisse wie Ereignisse umgesetzt. Falls ein Trivialereignis einen Nachfolger besitzt, so wird es als Übergangsbedingung umgesetzt und zeigt an, dass nach einer Funktion nur dann weitergegangen werden darf, falls das Trivialereignis eintritt. Es wird keine Aussage darüber getroffen, was passieren soll, wenn die Funktion beendet wird, jedoch das Trivialereignis nicht eintritt. Mögliche Szenarien sind hier der Wurf eines Fehlers oder das Wiederholen der Ausführung der Funktion. Falls `USETRIVIALEVENTS` auf `FALSCH` steht, so werden die Trivialereignisse nicht umgesetzt: In dem umzusetzenden Graph werden die Trivialereignisse entfernt und die Kanten angepasst.

## 5.6. Schleifenerkennung

Die EPK schränkt die Möglichkeit von Sprüngen nicht ein (vgl. Abschnitt 3.4). Somit können durch Sprünge beliebige Zyklen erzeugt werden. Ein Zyklus ist nicht zwangsläufig eine Schleife. Der oder-Operator des Zyklus aus Abbildung 5.1 erlaubt die parallele Ausführung des Zyklus und der nachfolgenden Funktionen. Weiterhin wird durch den oder-Operator erlaubt, dass nach jeder Ausführung des Zyklus die nachfolgenden Funktionen nochmals ausgeführt werden. Dies kann nicht nach BPEL abgebildet werden, da eine mehrmalige Ausführung von Funktionen nur durch eine Schleife möglich ist und die Funktionen nach dem Zyklus nicht Bestandteil einer Schleife sind. Deshalb müssen für eine Abbildung nach BPEL sowohl der Schleifeneingangs- als auch der Schleifenausgangsknoten entweder-oder-Operatoren sein.

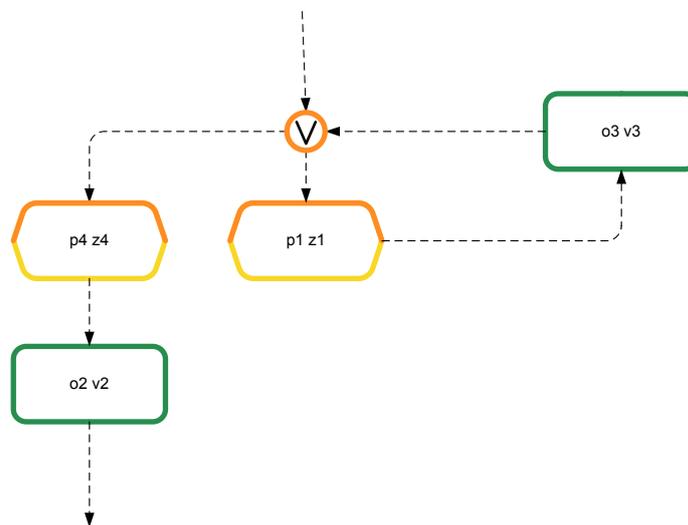


Abbildung 5.1.: Zyklus, der keine Schleife darstellt

Typischerweise kommen in Prozessmodellen folgende Arten von Schleifen vor. Der Schleifenrumpf steht symbolisch für eine beliebige Verknüpfung von Funktionen. Im Schleifenrumpf können so auch weitere Schleifen geschachtelt sein. Der Schleifenrumpf wird durch die selben Ideen wie die anderen Knoten in eine `flow`-Aktivität umgesetzt.

**Klassische while-Schleife** Diese Schleife wird so oft ausgeführt, wie die Schleifenbedingung wahr ist. Die Schleifenbedingung kann auch nie wahr sein, so dass der Schleifenrumpf nie ausgeführt wird. Vgl. Abbildung 5.2.

**Erweiterte while-Schleife** Im Unterschied zu der klassischen While-Schleife muss der Schleifenrumpf einmal ausgeführt worden sein, bevor die nachfolgenden Aktivitäten ausgeführt werden können. Vgl. Abbildung 5.3.

**Klassische repeat-until-Schleife** Bei der klassischen repeat-until-Schleife wird der Schleifenrumpf ausgeführt und dann die Schleifenbedingung überprüft. Ist die Schleifenbedingung wahr, so wird der Schleifenrumpf erneut ausgeführt. Dies wird so lange wiederholt, bis die Schleifenbedingung nicht mehr zutrifft. Vgl. Abbildung 5.4.

**Erweiterte repeat-until-Schleife** Die erweiterte repeat-until-Schleife unterscheidet sich von der klassischen repeat-until-Schleife bei den Funktionen, die nach dem Eintreten der Schleifenbedingung durchgeführt werden. Bei der klassischen repeat-until-Schleife wird der Schleifenrumpf erneut ausgeführt. Bei der erweiterten repeat-until-Schleife wird nach dem Eintreten der Schleifenbedingung und vor der Ausführung des Schleifenrumpfes weitere Funktionen ausgeführt. Vgl. Abbildung 5.5.

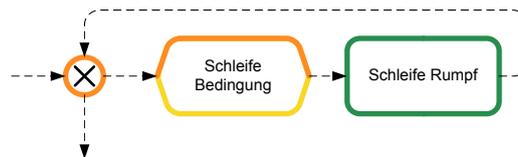


Abbildung 5.2.: Klassische Form der while-Schleife



Abbildung 5.3.: Erweiterte while-Schleife

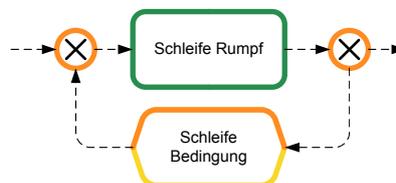


Abbildung 5.4.: Klassische repeat-until-Schleife

Jede dieser Schleifen besitzt einen Eingang und einen Ausgang. Bei der klassischen while-Schleife fallen Ein- und Ausgang zusammen. Dies kann bei repeat-until-Schleifen mit einem Ausgang nicht der Fall sein. Bei einer repeat-until-Schleife folgt dem Schleifeneingangsknoten eine Funktion. Da der Schleifeneingangsknoten ein oder-Operator ist und ihm eine Funktion folgt, darf er keine weiteren Ausgangskanten besitzen, da andernfalls

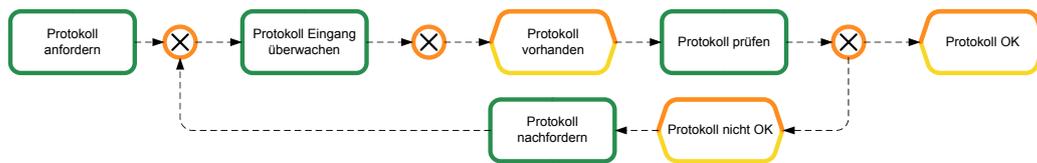


Abbildung 5.5.: Erweiterte repeat-until-Schleife mit einer Funktion, die ab dem zweiten Durchlauf ausgeführt wird

die Bedingung 6 für gültige ereignisgesteuerte Prozessketten aus Abschnitt 3.4 verletzt wäre.

Jeder der vier Schleifentypen kann in eine klassische while-Schleife transformiert werden. Bei der erweiterten while-Schleife wird der transitioncondition der Eingangskanten die Schleifenbedingung hinzugefügt und die Schleife als klassische while-Schleife behandelt. Somit ist gewährleistet, dass der Schleifenrumpf mindestens ein Mal ausgeführt wird, bevor die Ausführung bei den nachfolgenden Aktivitäten fortgesetzt wird.

Die klassische repeat-until-Schleife wird in eine while-Schleife umgesetzt, deren Bedingung die Schleifenbedingung ist. Die Schleifenbedingung muss nicht negiert werden, da genau dann der Schleifenrumpf nochmals ausgeführt wird, falls die Schleifenbedingung wahr ist. Damit die while-Schleife mindestens ein Mal ausgeführt wird, wird eine boolesche Hilfsvariable  $v$  eingeführt, mit der das Verhalten von repeat-until simuliert wird.  $v$  wird durch ein logisches oder mit der Schleifenbedingung verknüpft. Somit wird erreicht, dass die Schleife mindestens ein Mal ausgeführt wird. Vor Erreichen der Schleife wird die Variable  $v$  auf WAHR und als erste Aktivität in der Schleife auf FALSCH gesetzt. Die Ausgangskanten der while-Schleife müssen nicht angepasst werden, da sie erst nach dem Setzen der Variable  $v$  auf WAHR ausgeführt wird.

Die erweiterte repeat-until-Schleife unterscheidet sich von der klassischen repeat-until-Schleife in der Ausführung von Aktivitäten erst ab dem zweiten Durchlauf. Dieses Verhalten kann durch die Variable  $v$  emuliert werden. Falls  $v$  ganz am Anfang der Schleife nicht gilt, so werden die Funktionen ausgeführt, die erst ab dem zweiten Durchlauf ausgeführt werden sollen.

Neben Schleifen mit einem Ausgang kann es Schleifen mit keinem oder mehreren Ausgängen geben. Schleifen mit keinem Ausgang müssen nicht gesondert behandelt werden, da aus ihnen keine Kante herausführt. Eine Schleife mit mehreren Ausgängen ist in Abbildung 5.6 dargestellt.

Bei Schleifen mit mehreren Ausgängen können die Ausgänge zu einem Ausgang zusammengefasst werden. Der Schleifeneinstiegsknoten wird dann zusätzlich zum neuen Ausgangsknoten. Falls vor der Umwandlung der Schleifeneinstiegsknoten bereits ein Ausgangsknoten ist, so wird dieser Knoten nicht konvertiert. Bei allen anderen Ausgängen werden die Ausgangskanten in zwei Teile aufgeteilt: Der erste Teil wird in eine Aktivität innerhalb der Schleife umgesetzt und der zweite Teil als zusätzliche ausgehende Kante des Schleifeneinstiegsknoten. In der zusätzlichen Aktivität wird ein Flag gesetzt, dass die Kante benutzt wurde. Die zusätzliche Aktivität besitzt eine ausgehende Kante, deren Ziel der Schleifeneinstiegsknoten ist. Bei der zusätzlichen Kante des Schleifeneinstiegsknoten

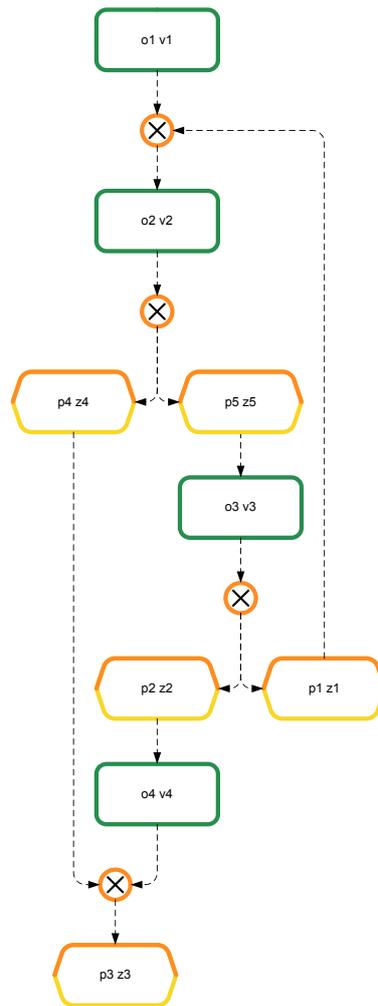


Abbildung 5.6.: EPK mit einer Schleife mit zwei Ausgängen

wird als Übergangsbedingung das Flag verwendet. Falls der Schleifeneinstiegsknoten bereits ausgehende Kanten hatte, wird bei deren Übergangsbedingungen zusätzlich gefordert, dass kein Flag gesetzt ist.

Schleifen mit mehreren Eingängen können in Schleifen mit einem Eingang umgewandelt werden. Dazu wird die Technik der „Knotenaufspaltung“ verwendet. Jeder Knoten  $n$ , der ein Eingang in die Schleife ist, wird in mehrere Knoten aufgeteilt. Für jeden Vorgänger wird ein neuer Knoten  $n_i$  erzeugt, der die gleichen Nachfolger hat. Die Nachfolger von  $n$  werden zu den Nachfolgern von jedem  $n_i$ , woraufhin  $n$  aus dem Graphen entfernt wird. Jetzt hat die Schleife entweder einen Eingang oder mehr als einen Eingang. Im zweiten Fall muss einer der Eingänge erneut aufgeteilt werden. Das Verfahren wird so lange wiederholt, bis die Schleife genau einen Eingang besitzt. In [He77] wird gezeigt, dass der entstandene Graph zu dem Ursprungsgraphen äquivalent ist. Im Falle der ereignisgesteuerten Prozessketten würden Funktionen automatisiert dupliziert werden und im abgebildeten BPEL-Prozess als zwei verschiedene Funktionen dargestellt werden. Diese Duplizierung

verletzt den Grundsatz, dass ein Knoten maximal in eine BPEL-Aktivität umgesetzt wird, so dass diese Transformation nicht automatisiert vorgenommen wird.

Die Schleife aus Abbildung 5.7 stellt einen Grenzfall für mehrere Eingänge dar. Die dargestellten Bedingungen könnten zu einer Bedingung verschmolzen werden, wodurch die Schleife nur einen Eingang besäße. Jedoch wird durch diese Verschmelzung der Grundsatz verletzt, dass ein Ereignis in eine Übergangsbedingung abgebildet wird, weshalb dieser Fall bei dem Export ausgeschlossen ist.

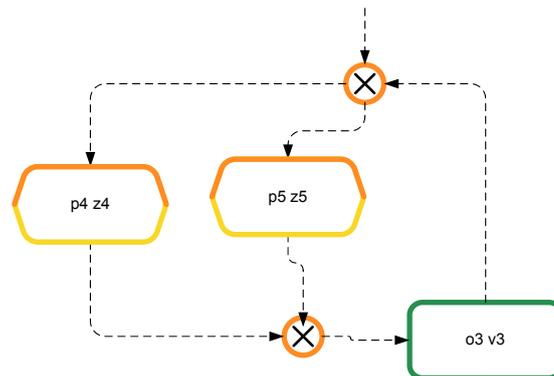


Abbildung 5.7.: Schleife mit einer geschachtelten Eingangsbedingung

In Abbildung 5.8 ist eine Schleife dargestellt, die ganz am Anfang des Prozesses ausgeführt wird. Es kann nur aus der Grafik abgelesen werden, an welchem Knoten die Ausführung der ereignisgesteuerten Prozesskette beginnen soll. Da es auch Modelle geben kann, in denen ein anderer Knoten der Startknoten ist, werden ereignisgesteuerte Prozessketten mit einer Schleife am Anfang nicht nach BPEL exportiert. Durch diesen Ausschluss besitzt jeder EPK-Graph mindestens einen Knoten ohne eingehende Kante, der auch Wurzelknoten genannt wird.

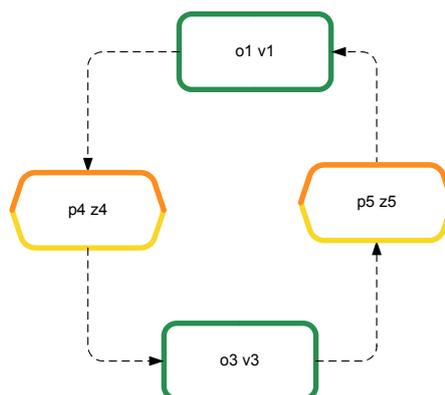


Abbildung 5.8.: EPK ohne Wurzelknoten

Im Compilerbau müssen Schleifen mit einem Eingang und beliebig vielen Ausgängen nicht gesondert behandelt werden. Deshalb wird für diese Art von Schleifen die Bezeich-

nung „natürliche Schleifen“ verwendet. Natürliche Schleifen können über die Dominanzrelation gefunden werden. Ein Knoten  $u$  dominiert einen Knoten  $v$ , wenn alle Wege von der Wurzel des Graphen zu  $v$  über  $u$  gehen. Bei einer natürlichen Schleife dominiert der Schleifeneingang alle Knoten in der Schleife. Dazu gehören insbesondere die Kanten, deren Ziel der Schleifeneingang ist. Hier dominiert das Ziel der Kante deren Quelle. In [Mu97] wird eine solche Kante als Rückwärtskante bezeichnet. Zu einer Schleife gehören alle Knoten, die bei einer Rückwärtssuche von einer Rückwärtskante bis zu dem Schleifeneingang gefunden werden. Da es in einer Schleife mehrere Rückwärtskanten geben kann (vgl. Abbildung 5.9), muss diese Rückwärtssuche von allen Rückwärtskanten gestartet werden.

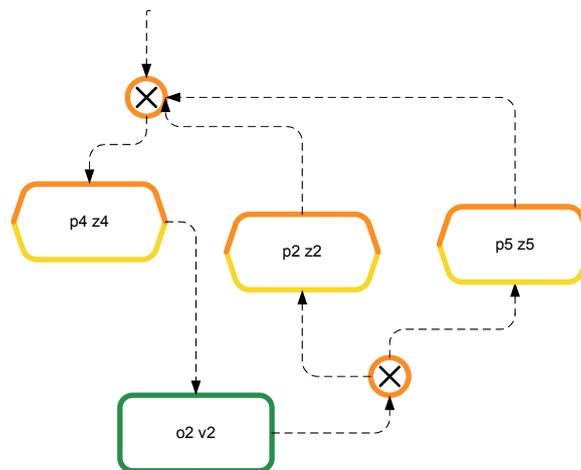


Abbildung 5.9.: Schleife mit mehreren Rückwärtskanten

Der EPK-Ausschnitt aus Abbildung 5.10 enthält zwei Rückwärtskanten zu dem „entweder-oder“-Operator. Im Unterschied zu der EPK aus Abbildung 5.9 werden durch die Rückwärtskanten zwei Schleifen dargestellt. Eine weitere Möglichkeit der Mehrfachverwendung eines entweder-oder-Operators ist in Abbildung 5.11 dargestellt. Dort sind zwei repeat-until-Schleifen ineinander geschachtelt.

Die Unterscheidung der beiden Fälle kann über die Ausgangsknoten vorgenommen werden. Das Ziel der ausgehenden Kanten des Ausgangsknotens der äußeren Schleife ist in keiner bei der Rückwärtssuche bestimmten Knotenmenge enthalten. Somit ist der Ausgangsknoten bestimmt und ist der äußeren Schleife zugeordnet. In der Schleife aus Abbildung 5.9 ist das Ziel jeder ausgehenden Kante des Operators in den selben Knotenmengen enthalten. Deshalb ist dieser Operator kein Ausgangsknoten. Diese Fälle können auch gemischt vorkommen. So kann ein Operator drei Ausgangskanten besitzen, wovon zwei als Ziel Knoten in der Schleife und eine als Ziel ein Knoten außerhalb der Schleife besitzen.

Durch die Anwendung dieser Fallunterscheidungen kann ein entweder-oder-Operator mehrere Schleifeneingänge darstellen. Dies widerspricht dem Grundsatz, dass ein Knoten des EPK-Graphen in maximal eine BPEL-Aktivität abgebildet wird. Deshalb wird gefordert, dass von dem Modellierer geschachtelte Schleifen durch separate Operatoren mo-

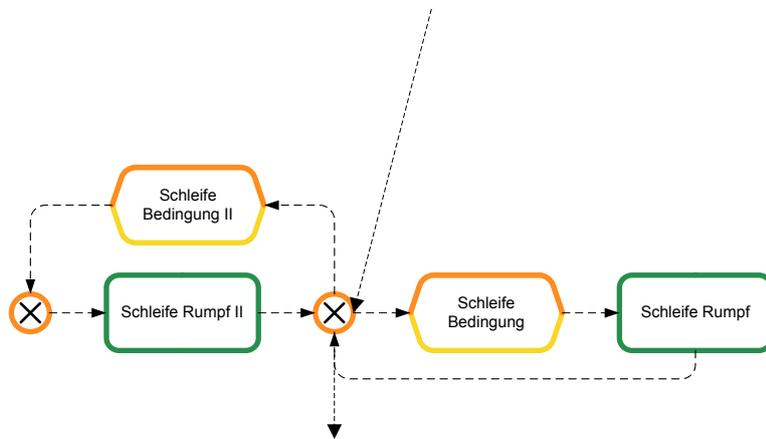


Abbildung 5.10.: Zwei Schleifen mit gleichem Eingangsknoten

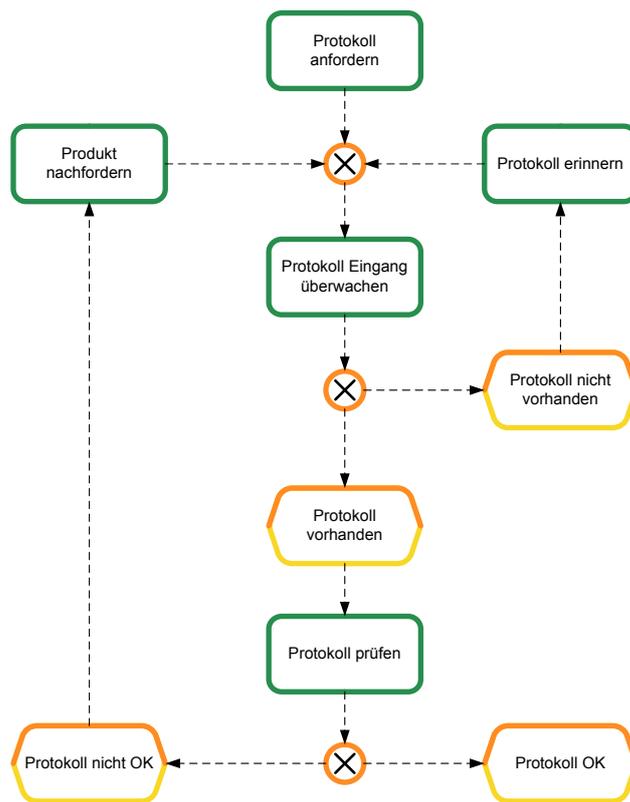


Abbildung 5.11.: Geschachtelte Schleifen

delliert werden. Für die geschachtelten Schleifen aus Abbildung 5.11 ist dies in wie in Abbildung 5.12 vollzogen.

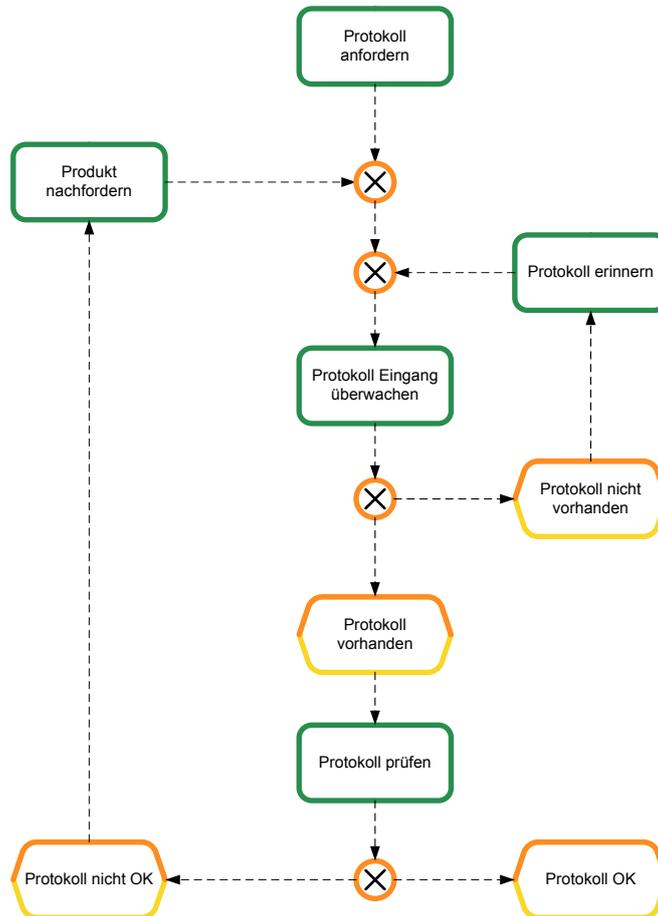


Abbildung 5.12.: Geschachtelte Schleifen

In Abbildung 5.13 sind zwei Schleifen dargestellt, die keine gemeinsamen Eingangsknoten besitzen. Der Schleifenausgangsknoten der ersten Schleife ist der Schleifeneingangsknoten der zweiten Schleife. Somit wird zuerst die erste Schleife ausgeführt und daraufhin so lange eine der beiden Schleifen bis keine der beiden Schleifenbedingungen zutrifft. Da BPEL als Schleifenkonstrukt nur die while-Schleife kennt, müssen beide Schleifen jeweils eine while-Schleife umgesetzt werden. Dadurch dass die Schleifen nicht nacheinander ausgeführt werden, sondern zuerst die erste Schleife und dann entweder die erste oder die zweite, müssen die beiden Schleifen in ein sie umgebendes while-Konstrukt eingebettet

werden. Dieses Vorgehen verstößt gegen den Grundsatz der Erhaltung der Struktur der ereignisgesteuerten Prozesskette, weshalb diese Abbildung nicht umgesetzt wird.

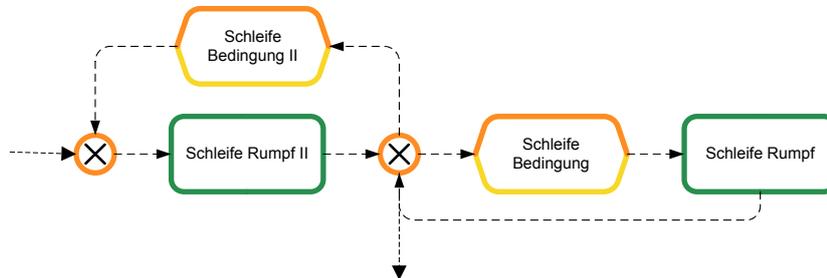


Abbildung 5.13.: Zwei Schleifen mit unterschiedlichem Eingangsknoten

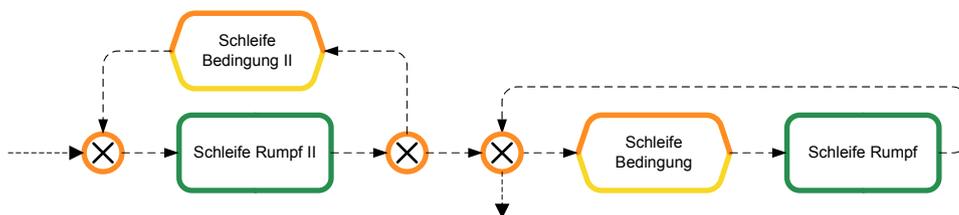


Abbildung 5.14.: Zwei Schleifen mit unterschiedlichem Eingangsknoten

Zusammengefasst werden Schleifen mit einem Eingangsknoten und maximal einem Ausgangsknoten abgebildet. Jede Schleife muss weiterhin genau eine Schleifenbedingung besitzen. Dies ermöglicht die Klassifizierung der Schleifen in die vier am Anfang vorgestellten Schleifentypen. Falls bei der Schleife der Eingangs- mit dem Ausgangsknoten zusammenfällt, handelt es sich um eine klassische while-Schleife. Ansonsten wird überprüft, ob der erste Knoten nach dem Schleifeneingangsknoten ein Ereignis ist. Falls ja, ist es eine while-Schleife. Falls nicht, muss der Knoten nach dem Schleifenausgangsknoten, der in der Schleife liegt, ein Ereignis sein. Ansonsten stellt die Schleife keine gültige repeat-until-Schleife dar. Falls nach dem Ereignis direkt der Schleifeneinsteigsknoten folgt, ist die Schleife eine klassische repeat-until-Schleife, ansonsten eine erweiterte repeat-until-Schleife.

Die Datenstruktur zur Speicherung einer Schleife ist ein Tupel aus dem Typ der Schleife, Eingangsknoten, dem Ausgangsknoten, der Schleifenbedingung, den Ausgangskanten, dem ersten Knoten der Schleife, dem ersten Knoten der Aktivitäten der zweiten Ausführung und der Schachtelungstiefe. Mit dem „ersten Knoten der Aktivitäten der zweiten Ausführung“ ist der Knoten gemeint, der bei der erweiterten repeat-until-Schleife der Schleifenbedingung folgt. Dieses Tuppelelement ist mit  $\perp$  belegt, falls es sich um keine repeat-until-Schleife handelt. Der gespeicherte Typ ist entweder „while“ oder „repeat-until“. Die Unterscheidung zwischen der klassischen und der erweiterten Ausprägung wird bei der while-Schleife durch den Ausgangsknoten und bei der repeat-until-Schleife durch das letzte Tuppelelement vollzogen. Die Ausgangskanten werden für das Schreiben der Kanten der flow-Aktivität benötigt, in der die Schleife enthalten ist.

Da ein Operator von maximal einer Schleife der Eingangs- oder Ausgangsknoten sein

kann, ist es möglich, die Schachtelungstiefe der Schleifen durch eine Tiefensuche von den Wurzelknoten der Zusammenhangskomponenten aus zu bestimmen. Die Tiefensuche wird mit der Schachtelungstiefe 0 gestartet. Trifft sie auf einen Schleifeneingangsknoten, so wird die Schachtelungstiefe um eins erhöht und dieser Wert der zugehörigen Schleife zugeordnet. Die Tiefensuche wird dann bei der ersten Aktivität der Schleife fortgesetzt. Sobald die Tiefensuche auf einen Schleifenausgangsknoten stößt, wird die Schachtelungstiefe um eins verringert und die Tiefensuche fortgesetzt. Die Tiefensuche besucht bei fork-Operatoren, die keinen Schleifeneingang darstellen, jeden Ausgang. Bei jeder Schleife, die einen Ein- und einen Ausgang besitzt, führt mindestens ein Weg vom Eingang zum Ausgang. Da die Schleife nur über den Ausgang verlassen werden kann und gefordert wurde, dass die Schleife nur einen Ausgang besitzt, wurden nach Abschluss der Tiefensuche wurden alle Schleifen mit der zugehörigen Schachtelungstiefe versehen.

Bei dem Export der Schleife wird als BPEL-Name der Name des Schleifeneingangsknoten benutzt. Dieser ist durch seine Nummer eindeutig und wird als Ziel der Kanten der umgebenen flow-Aktivität verwendet.

### 5.7. Abbildung von Operatoren

In Abschnitt wurde bereits beschrieben, in welchen Fällen entweder-oder-Operatoren zu einer while-Aktivität werden. Falls ein Operator nicht in eine Schleife umgesetzt wird, wird er gestrichen oder zu einer empty-Aktivität.

In einer EPK beschreiben join-Operatoren durch ihren Typ, wie viele eingehende Kanten aktiv sein müssen, damit der Operator aktiv wird und die Ausführung an ihm fortgesetzt wird. Das BPEL-Äquivalent ist die „join-condition“. Bei einem „oder“-Operator wird keine join-condition geschrieben, da die Standardbelegung der join-condition die logische Veroderung des Status der eingehenden Kanten ist (vgl. [BPEL4WS, S. 31]). Im Falle eines „und“-Operators wird eine join-condition, in der der Status der eingehenden Kanten durch ein logisches „und“ verknüpft sind, erzeugt. BPEL unterstützt die logische Verknüpfung „entweder-oder“ nicht. Deshalb kann ein „entweder-oder“-Operator nicht direkt nach BPEL umgesetzt werden, sondern muss mittels der logischen Verknüpfungen „und“ sowie der logischen Negation ausgedrückt werden. Falls  $e_1$  und  $e_2$  die eingehenden Kanten eines „entweder-oder“-Operators sind, so ist die join-condition:

```
((bpws:getLinkStatus(e1)) AND NOT (bpws:getLinkStatus(e2))) OR  
(NOT (bpws:getLinkStatus(e1)) AND (bpws:getLinkStatus(e2)))
```

Für mehr als zwei Operatoren ist die Umsetzung entsprechend. Da in einer ereignisgesteuerten Prozesskette oft „entweder-oder“-Operatoren verwendet werden und die exakte Umsetzung nicht zur Übersicht beiträgt, kann durch den Schalter `TOGGLEJOINCONDITIONS` bestimmt werden, ob join-conditions erzeugt werden sollen. Steht der Schalter auf `WAHR`, so werden die join-conditions wie oben beschrieben erzeugt. Steht er auf `FALSCH`, so werden keine join-conditions geschrieben.

In einer EPK gibt es neben join-Operatoren auch fork-Operatoren, die durch ihren Typ bestimmen, wie viele der ausgehenden Kanten aktiv werden können. Das BPEL-Äquivalent wäre ein „fork-condition“-Attribut. Dieses Attribut wird von BPEL nicht unterstützt, weshalb es weder erzeugt noch emuliert wird.

In dem EPK-Graphen werden die Operatoren gestrichen, die im BPEL-Prozess keine zusätzliche Semantik bringen. Dies sind Operatoren, die genau eine eingehende Kante und ausgehende Kanten besitzen und bei denen entweder die eingehende Kante oder alle ausgehenden Kanten keine Übergangsbedingung besitzen. Falls dies der Fall ist, wird geprüft, ob vom Vorgängerknoten eine Kante zu einem der Nachfolger des Operators führt. Falls nicht, kann der Operator entfernt werden. Falls die Kante vom Vorgänger eine Übergangsbedingung besaß, wird sie bei ausgehenden Kante gesetzt und danach aus dem Graphen entfernt. Andernfalls besaß die Kante vom Vorgänger keine Übergangsbedingung und kann aus dem Graphen ohne Vorkehrungen entfernt werden. Bei den ausgehenden Kanten wird als Quelle der Vorgänger des Operators eingetragen und anschließend der Operator aus dem Graphen entfernt.

Analog dazu können die Operatoren gestrichen werden, die genau eine ausgehende Kante und eingehende Kanten besitzen und bei denen entweder die ausgehende Kante oder alle eingehenden Kanten keine Übergangsbedingung besitzen. Das weitere Vorgehen erfolgt analog zu der Streichung von fork-Operatoren. Im Unterschied zu fork-Operatoren wird der Typ eines join-Operators in eine join-Condition umgesetzt. Diese muss bei der Streichung des Operators an den Nachfolger weitergegeben werden. Da ein Operator mit einer eingehenden Kante keine join-Condition darstellt, wird ihm als join-Typ der Typ gestrichenen Operators zugewiesen. Somit können auch Funktionen und Ereignisse join-Typen besitzen, die bei der Abbildung in eine BPEL-Aktivität berücksichtigt werden müssen. Falls einem Operator auf diesem Wege ein join-Typ zugewiesen wird, muss dieser nicht zwangsläufig der gleiche wie der Operatortyp sein. In Abbildung 5.15 wird dem AND-fork nach der Streichung des entweder-oder-Operators als join-Typ „entweder-oder“ zugewiesen. Falls ein Operator, dem bereits einen join-Typ zugewiesen wurde, gestrichen wird, so wird dieser join-Typ an seinen Nachfolger weitergereicht.

## 5.8. Generierung der Kanten

Die Generierung der Kanten einer flow-Aktivität erfolgt über eine Tiefensuche. Eine solche flow-Aktivität kann entweder eine Zusammenhangskomponente des zu exportierenden EPK-Graphen  $G$  sein, der Schleifenrumpf einer Schleife oder die Funktionen, die bei dem Eintritt eines Ereignisses ausgeführt werden sollen. Sie startet bei den Wurzelknoten der flow-Aktivität für die die Kanten geschrieben werden sollen. Sobald ein weiterer Knoten besucht wird, wird die Kante vom zuletzt besuchten Knoten zum aktuellen Knoten dem links-Element der flow-Aktivität als link hinzugefügt. Der Name der Kante setzt sich aus dem Namen des Quell- und Zielknotens zusammen. Diese beiden Namen werden durch „-to-“ verbunden, um Quelle und Ziel der Kante anzuzeigen. Trifft die Tiefensuche auf einen Schleifeneingangsknoten, so wird die Tiefensuche bei den Schleifenausgangskanten fortgesetzt. So werden die Kanten, die in der Schleife enthalten sind, nicht geschrieben.

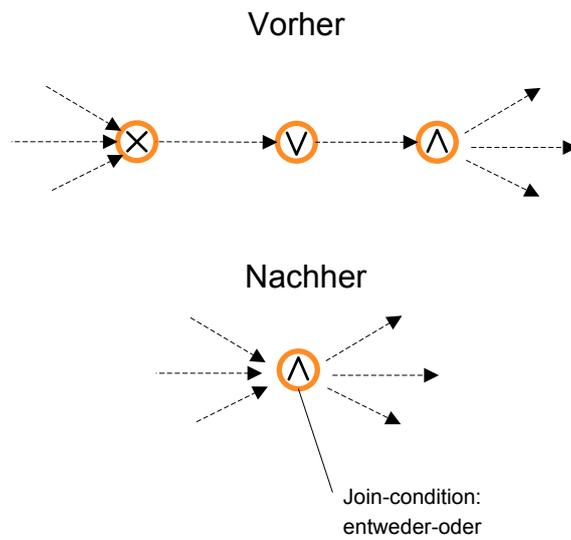


Abbildung 5.15.: Veranschaulichung der Übernahme der join-Condition

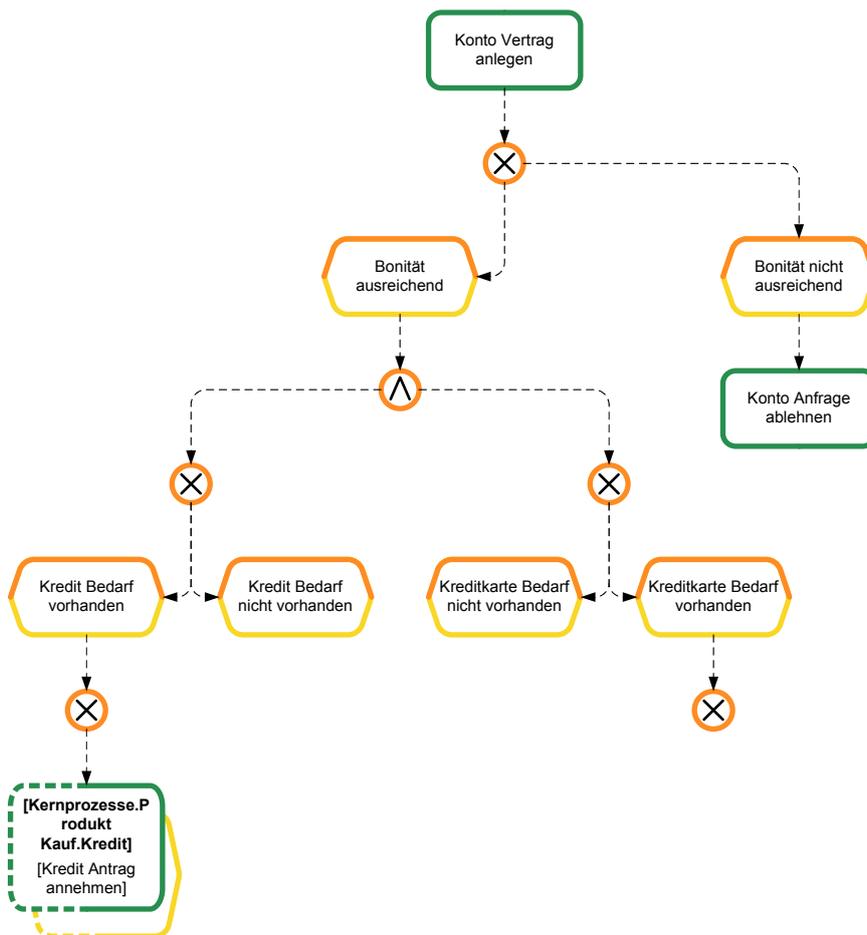


Abbildung 5.16.: Prozess vor der Streichung der Operatoren

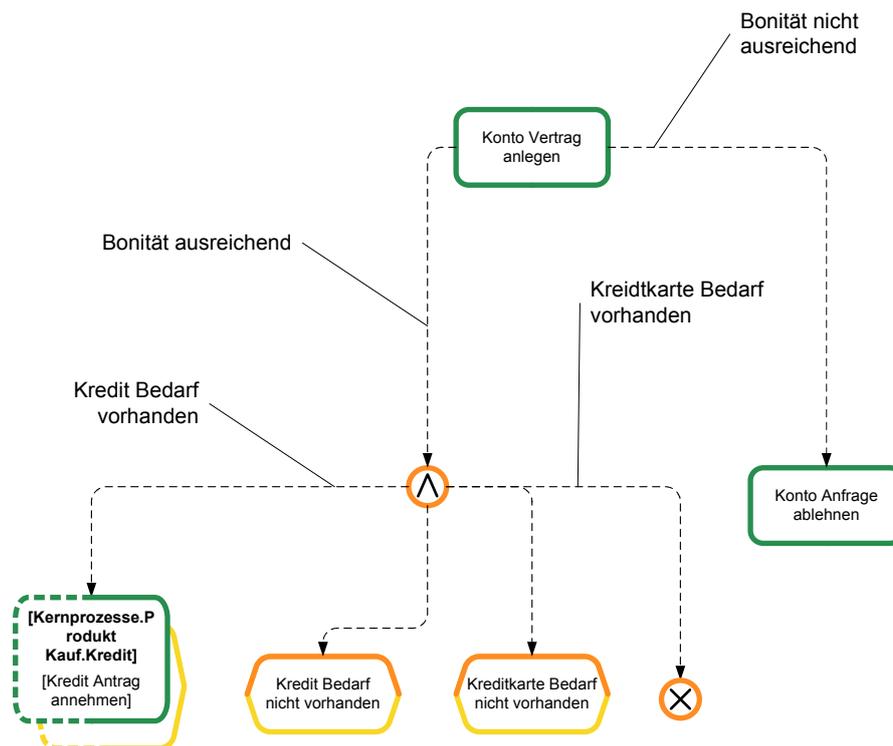


Abbildung 5.17.: Prozess nach der Streichung von Operatoren

Falls die Schachtelungstiefe der Ziele der Schleifenausgangskanten niedriger ist als die Schachtelungstiefe der flow-Aktivität, so wird die Tiefensuche abgebrochen. Dies ist der Fall, falls entweder die Kanten einer klassischen while-Schleife oder die Kanten der Aktivitäten der erweiterten repeat-until-Schleife geschrieben werden.

Falls der Schleifenausgang nicht mit dem Schleifeneingang zusammenfällt, kann die Tiefensuche auf einen Schleifenausgangsknoten treffen. Da alle Schleifen maximal einen Schleifeneingang besitzen, muss der besuchte Schleifenausgangsknoten der Knoten der Schleife sein, in dem die flow-Aktivität eingebunden ist. Die Kante zum Schleifenausgangsknoten wird bereits als Schleifenbedingung der zugehörigen Schleife verwendet. Deshalb wird an dieser Stelle keine Kante geschrieben und die Tiefensuche abgebrochen.

## 5.9. Generierung des Eventhandlers

Ein Eventhandler führt nach dem Empfang einer Nachricht eine Aktivität aus. Die dazu analoge Darstellung ist eine Zusammenhangskomponente mit genau einem Wurzelknoten des Typs „Ereignis“. Dieses Ereignis stellt das onMessage-Ereignis dar und die nachfolgenden Funktionen werden zu einer flow-Aktivität, die bei Eintreffen der Nachricht ausgeführt wird. Jede Zusammenhangskomponente, die genau einen Wurzelknoten mit dem Typ Ereignis besitzt, wird als onMessage-Ereignis in einen globalen Eventhandler

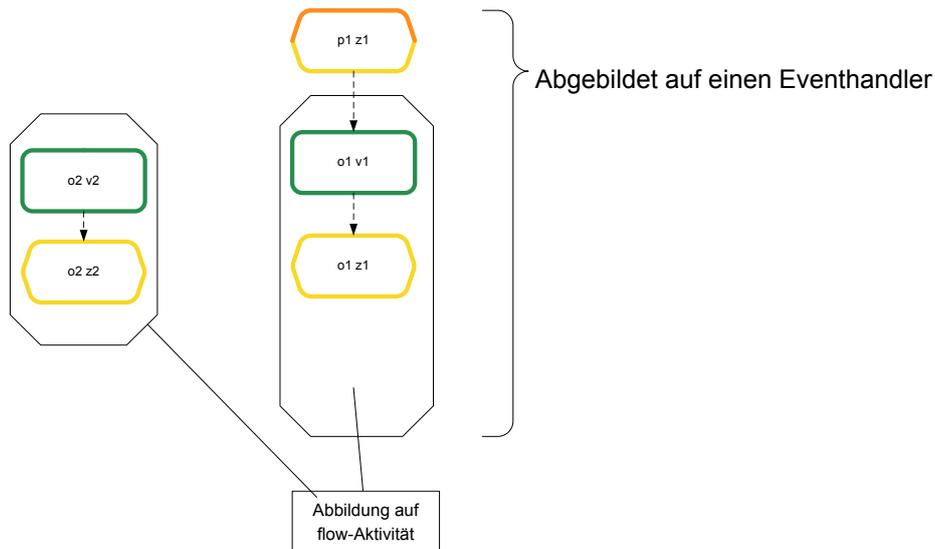


Abbildung 5.18.: Prinzip der Abbildung

exportiert. Alle anderen Zusammenhangskomponenten werden als flow-Aktivität exportiert.

In Abbildung 5.18 ist eine EPK dargestellt, die alle obigen Bedingungen erfüllt. In der Abbildung 5.19 werden EPK-Strukturen dargestellt, die die Voraussetzungen nicht erfüllen.

### 5.10. Umsetzung in Correlation Sets

Anhand der Beziehungen kann nicht sicher entschieden werden, welche Elemente zur Identifikation der konkreten Prozessinstanz verwendet werden. Deshalb können Correlation Sets nicht aus dem Modell heraus generiert werden. Da bei Aktivitäten, die eine Instanz erzeugen, auch bei einem abstrakten BPEL-Prozess ein Correlation Set angegeben werden muss, wird ein Stub-Correlation Set mit dem Namen „Dummy“ erzeugt:

```
<correlationSets>
  <correlationSet name="dummy" properties="tns:dummyProperty" />
</correlationSets>
```

Die Definition der verwendeten Property „tns:dummyProperty“ ist im Listing 5.2 angegeben.

Falls eine instanzerzeugende Aktivität eine Nachricht empfängt, so muss das zugeordnete Correlationset zu der Nachricht passen. Deshalb wird für jede Nachricht, die von einer instanzerzeugenden Aktivität empfangen wird, ein propertyAlias erzeugt. Da jeder Teil der empfangenen Nachricht vom Typ string ist, wird als propertyalias der erste Teil der Nachricht verwendet.

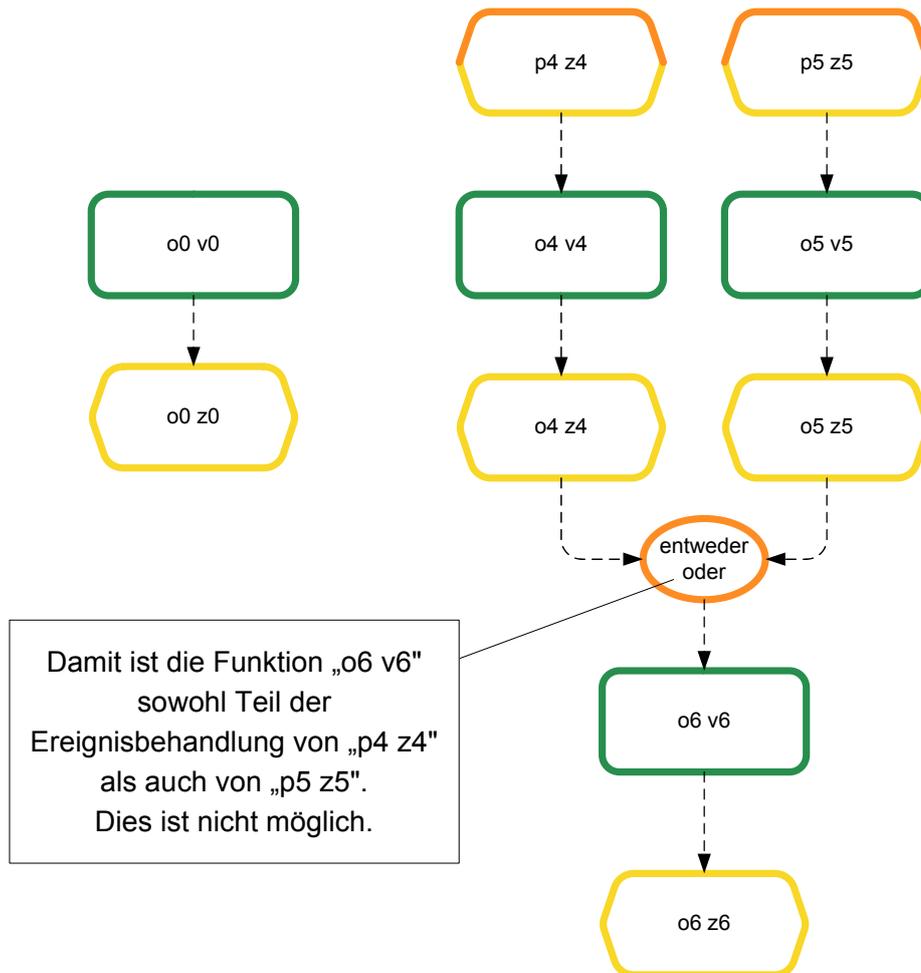


Abbildung 5.19.: Illustration von nicht erlaubten EPK-Formen

```
<message name="dummy">
  <part name="dummy" type="xsd:string" />
</message>
<bpws:property name="dummyProperty" type="xsd:string" />
<bpws:propertyAlias propertyName="tns:dummyProperty"
  messageType="tns:dummy" part="dummy" query="/dummy" />
```

Listing 5.2: Definition der Dummy-Property

### 5.11. Generierung der WSDL-Datei

Beim Export wird die BPEL-Prozessdefinition und die zugehörige WSDL-Datei geschrieben. Diese WSDL-Datei dient auch als Spezifikation der zu erstellenden Services. In die WSDL-Datei werden alle Nachrichtentypen, PortTypes, PartnerLinkTypes und das in Abschnitt 5.10 beschriebene Dummy-Correlationset geschrieben.

### 5.12. Nicht erzeugte Konstrukte

Da es für die folgenden Konstrukte keine äquivalente Darstellung durch Beziehungen gibt, können diese nicht erzeugt werden:

- Compensation- und Faulthandler, throw
- scope
- Variablen, assign
- wait
- terminate
- reply

Ein reply wird in BPEL dann verwendet, falls eine Antwort auf eine Anfrage von der selben operation gesendet wird: Die operation hat eine input- und eine output Nachricht. Die input-Nachricht wird vom Prozess empfangen und die output-Nachricht wird von einer reply versandt. Eine solche Situation könnte wie in Abbildung 5.20 dargestellt modelliert werden.

Eine Umsetzung wie sie im Abschnitt 5.3.2 beschrieben wurde, generiert zwei operations, da „o1 v1“ und „o2 v2“ zwei verschiedene Funktionen sind. Die Umsetzung kann so modifiziert werden, dass ein solcher Fall immer auf ein reply abgebildet wird. Bei einer solchen Modifikation müssen Sonderfälle beachtet werden. Falls nach „o2 v2“ eine weitere Funktion folgt, die „Information“ an den „Server“ sendet, so darf „o2 v2“ nicht mit „o1 v1“ zu einer operation verschmolzen werden. Analog dazu darf keine Verschmelzung erfolgen, falls es eine Funktion gibt, die wie „o1 v1“ „Information“ vom „Server“ empfängt. Weiterhin kann es sein, dass explizit ein asynchroner Aufruf modellieren werden sollte, so

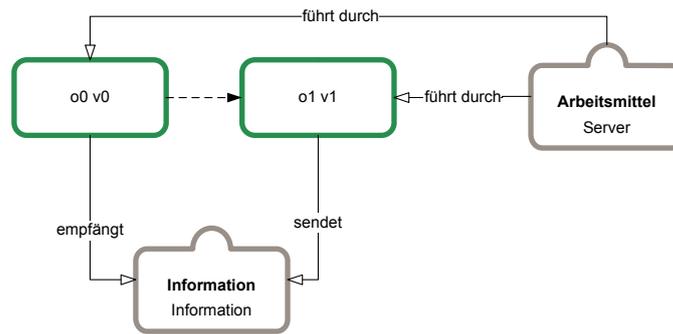


Abbildung 5.20.: Mögliche Modellierung von reply

dass dieses Mapping auf reply falsch wäre. Deshalb wird auf das Mapping auf reply verzichtet.

Die Abbildung 5.21 zeigt eine EPK-Struktur, die die Semantik einer pick-Aktivität haben könnte. Falls der entweder-oder-Knoten keine eingehenden Kanten besitzt, so wird diese Struktur auf einen Eventhandler abgebildet (vgl. Abschnitt 5.9). Falls der entweder-oder-Knoten eingehende Kanten hat, so werden die Ereignisse zu transitionConditions (vgl. Abschnitt 5.5). Falls es möglich wäre, Ereignisse explizit als externe Ereignisse zu kennzeichnen, könnte die gezeigte EPK-Struktur auf ein pick abgebildet werden, falls jedes Ereignis als externes Ereignis gekennzeichnet ist.

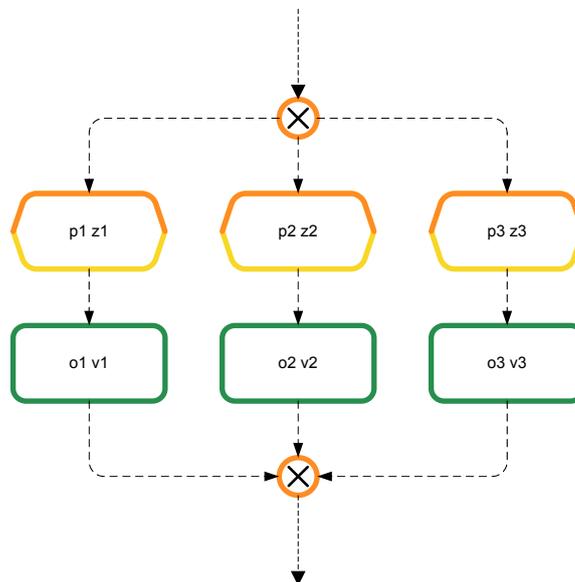


Abbildung 5.21.: Struktur eines picks oder eines switches in einer EPK

Falls der entweder-oder-Knoten aus der Struktur in Abbildung 5.21 eingehende Kanten hat, könnte diese Struktur auf ein switch abgebildet werden. Da bei dieser Struktur nicht zwingend ein switch verwendet werden muss, sondern auch die Abbildung

in ein flow-Konstrukt möglich ist, wird in diesem Fall die Abbildung nach flow gewählt. Die switch-Aktivität erzwingt, dass genau ein Zweig gewählt wird, die Abbildung auf flow hingegen nicht. Falls diese 1 aus n-Auswahl gewünscht ist, muss die BPEL-Prozessbeschreibung nachbearbeitet werden und der betreffende Teil der flow-Aktivität in eine switch-Aktivität geändert werden.

Die EPK-Struktur aus Abbildung 5.22 kann sowohl auf einen flow mit Kanten ohne transitionConditions zwischen „o1 v1“ ... „o3 v3“ als auch auf eine sequence abgebildet werden. Da beide Abbildungen in diesem Fall semantisch äquivalent sind, wird eine Abbildung auf flow gewählt, um keine zusätzlichen Typen von Aktivitäten, die nicht wirklich notwendig sind, zu verwenden.

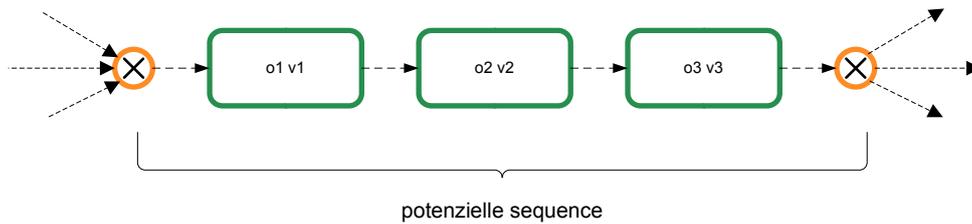


Abbildung 5.22.: Struktur einer Sequence in einer EPK

## BPEL-EXPORT

---

Der im vorangegangenen Kapitel vorgestellte Algorithmus zur Abbildung eines Geschäftsprozesses nach BPEL wurde in der Prozessmodellierungssoftware Nautilus als Export-Modul eingebunden. Die Schnittstelle zum Exportmodul bildet der in Abbildung 6.1 dargestellte Dialog. Der Benutzer erreicht den Dialog durch die Auswahl von „BPEL...“ im Datei-Menü unter „Export“.

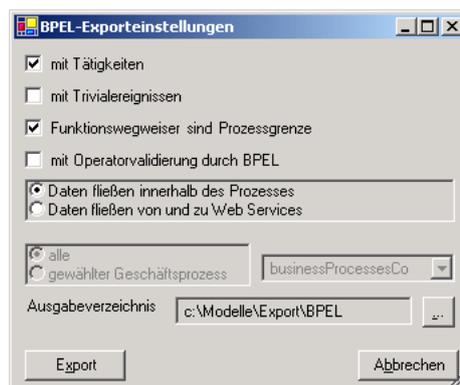


Abbildung 6.1.: Fenster zum Steuern des Exports

In diesem Dialogfenster sind alle im letzten Kapitel aufgeführten Schalter enthalten. Zur Verdeutlichung des Schalters `DATAFLOWISINTERNAL` wurde die Auswahl zwischen „Daten fließen innerhalb des Prozesses“ und „Daten fließen von und zu Web Services“ eingeführt. Es werden zwei Export-Varianten angeboten: Zum Einen der Export aller im Modell enthaltenen Geschäftsprozesse und zum anderen der Export eines bestimmten Geschäftsprozesses. Für jeden exportierten Geschäftsprozess werden im angegebenen Verzeichnis zwei Dateien mit dem Namen des Geschäftsprozesses und der Endung `.bpe1` für die BPEL-Datei und `.wsdl` für die WSDL-Datei erzeugt.

Sobald der Benutzer auf „Export“ klickt, wird der Export gestartet. Zunächst wird für jeden Geschäftsprozess überprüft, ob er nach BPEL exportiert werden kann. Ist dies nicht der Fall, erscheint eine Fehlermeldung, die den Grund des Scheiterns angibt. Dies kann beispielsweise eine Kante sein, die einen Zyklus erzeugt, der nicht nach BPEL exportiert

werden kann. Falls der Prozess exportiert werden kann, werden die beiden Dateien entsprechend des im vorigen Kapitel vorgestellten Algorithmus erzeugt.

Nautilus bietet bereits Exportfunktionalität an. Unter anderem kann ein Modell nach XML exportiert werden. Die Daten eines Modells werden in einer Datenbank gespeichert, auf die über Hilfsklassen zugegriffen werden kann. Folglich gibt es folgende Möglichkeiten zur Implementierung des BPEL-Exports:

1. XML-Transformation des XML-Exports in eine BPEL-Datei durch XSLT
2. Direktzugriff auf die Datenbank
3. Zugriff über bereits implementierte Hilfsklassen

Eine XML-Transformation der XML-Datei würde nicht aus der reinen Umbenennung der Elemente, sondern komplexe Transformationen enthalten, da unter anderem die Schleifenerkennung durchgeführt werden muss. Somit würde eine Umsetzung zu komplexen, programmähnlichen XSLT-Dateien führen. Weiterhin stellt die XML-Datei ein Datenbank-Abbild dar. Da sich die Software in ständiger Weiterentwicklung befindet, kann sich das Format der XML-Datei von Version zu Version ändern. Dadurch müsste der Code des Export-Moduls bei jeder Änderung der Datenbank oder des XML-Formats angepasst werden. Diese Anpassungen können durch die Nutzung der vorhandenen Zugriffsklassen vermieden werden. Gedilan setzt Microsoft Visual Studio.Net als Entwicklungswerkzeug und C# als Programmiersprache ein. Somit kann durch den Einsatz von C# eine konsistente Entwicklung stattfinden und so die Wartung des Export-Moduls begünstigt werden.

C# ist eine objektorientierte Programmiersprache, deren Syntax sich an C++ orientiert. Ein C#-Programm besteht aus Namespaces, in denen Klassen enthalten sind. In diesen Klassen ist die Funktionalität eines Programms in Form von Methoden enthalten. Näheres ist in C#-Lehrbüchern, wie [Li05] beschrieben.

Wichtige Namespaces von Nautilus sind:

- `Gedilan.Nautilus.Gui`,
- `Gedilan.Nautilus.DataAccess`,
- `Gedilan.Nautilus.Modelling` und
- `Gedilan.Nautilus.Export`.

In `Gedilan.Nautilus.Gui` ist das Hauptprogramm enthalten, das unter anderem für die Darstellung des „Graphical User Interface“ (GUI) zuständig ist. Das Hauptfenster von Nautilus ist im Anhang C erläutert, die weiteren Bestandteile werden in [Ge04] erläutert.

Diese Namespaces finden sich in der in Abbildung 6.2 dargestellten Architektur von Nautilus wieder. Nautilus speichert alle Bestandteile eines Modells in einer Datenbank und realisiert den Zugriff durch Klassen aus dem Namespace „`Gedilan.Nautilus.DataAccess`“. Auf den Datenbankzugriff setzen die Klassen des Namespaces „`Gedilan.Nautilus.Modelling`“ auf. Mit ihnen können Modellierungsaktionen wie die Bildung einer neuen Variante oder die Abfrage der Nachbarn vorgenommen werden. Die Datenbank- und die Modellierungssicht werden von allen darüberliegenden Schichten verwendet.

Im Exportmodul ist der XML-Export realisiert, in der GUI und in Visio findet die Modellierung durch den Benutzer statt. Dabei kann der Benutzer ganz auf die Unterstützung durch Visio verzichten und nur mittels der GUI modellieren (vgl. [Ge04, S. 113]).

|                       |                                 |
|-----------------------|---------------------------------|
|                       | Gedilan.Nautilus.VisualModeller |
| GUI - Visio-Anbindung | Gedilan.Nautilus.GUI            |
| Exportmodule          | Gedilan.Nautilus.Export         |
| Modelling             | Gedilan.Nautilus.Modelling      |
| Data-access           | Gedilan.Nautilus.DataAccess     |
| Entities              | LLBLGen Pro                     |
| Datenbank             |                                 |

Abbildung 6.2.: Architektur von Nautilus

## 6.1. Die Datenschicht

Die Klassen zur Datenhaltung sind im Namespace „Gedilan.Nautilus.DataAccess“ enthalten. Nautilus verwendet zum Speichern von Modellen eine Datenbank, in der jeder Bestandteil des Modells in Form einer „Entity“ abgelegt ist. Die Klassen zum Zugriff auf die Datenbank werden durch das Programm „LLBLGen Pro“ erzeugt (vgl. [LLBLGen]). Jeder Bestandteil eines Modells ist durch Klasse realisiert. So stellt beispielsweise jede Instanz der Klasse „Association“ eine Beziehung und jede Instanz der Klasse „Function“ eine Funktion dar. In Abbildung 6.3 ist die Vererbungshierarchie für ausgewählte Elemente dargestellt.

Die Klasse „EntityBase2“ dient der Datenhaltung und ist eine Klasse aus dem LLBLGen Pro-Paket. Ein Kind dieser Klasse ist die Klasse „ElementEntity“ die zur Speicherung aller Modell-Elemente verwendet wird. Alle anderen abgeleiteten Klassen dienen der persistenten Datenhaltung. So müssen beispielsweise Bedingungen und Geschäftsvorfälle sowie die Assoziationen persistent gespeichert werden, damit sie nicht erneut eingepflegt werden müssen. Für jedes persistent gespeicherte Objekt wird von LLBLGen Pro eine zugehörige „entity“-Klasse angelegt, die der Verwaltung des persistenten Objekts dient. Die zur Speicherung eines Objekts der Klasse „Association“ generierte Klasse erhält so den Namen „AssociationEntity“. Die Klassen zur Verwaltung der Objektdaten werden von den „entity“-Klassen abgeleitet. Somit besteht die Hierarchie ab der Klasse EntityBase2 aus drei Stufen: „EntityBase2“, die zu einer Klasse gehörende „Entity“-Klasse, sowie die Klasse selbst.

Bei Modell-Elementen besteht die Hierarchie aus mehr als drei Stufen. In der Klasse „Element“ werden alle gemeinsamen Eigenschaften von Modell-Elementen verwaltet. Die

verschiedenen Arten der Beschriftung spiegeln sich in der Vererbungshierarchie wieder. Die Elemente der ereignisgesteuerten Prozesskette und die Tätigkeiten werden nicht mit einer Kombination von Begriffen, sondern entweder mit anderen Elementen oder einem Typ beschriftet. Deshalb sind sie Nachfahren der Klasse „ComplexElement“. Die Klasse „ComplexElement“ selbst ist ein Kind der Klasse „Element“. Alle anderen Elemente sind Kinder der Klasse „Element“.

Kinder der Klasse „ComplexElement“ sind die Klassen „AtomicFunction“, „Function“ und „Event“. Mittels der Klasse „AtomicFunction“ werden Tätigkeiten und durch die Klasse „Function“ werden Funktionen realisiert. Die Klasse „Event“ ist die Oberklasse der Klassen „EventOperator“ und „AtomicEvent“. Durch die Klasse „EventOperator“ werden Operatoren und durch die Klasse „AtomicEvent“ werden Ereignisse umgesetzt. Diese Hierarchie wurde aus der Idee heraus erstellt, dass Funktionen mehrere Ereignisse auslösen können und so beispielsweise nicht nur das Trivialereignis auslösen sondern auch den nachfolgenden Operator. In der Datenbank selbst wird eine solche Beziehung nicht realisiert, so dass nur die im Kapitel 3 beschriebenen Beziehungen in der Datenbank gespeichert sind.

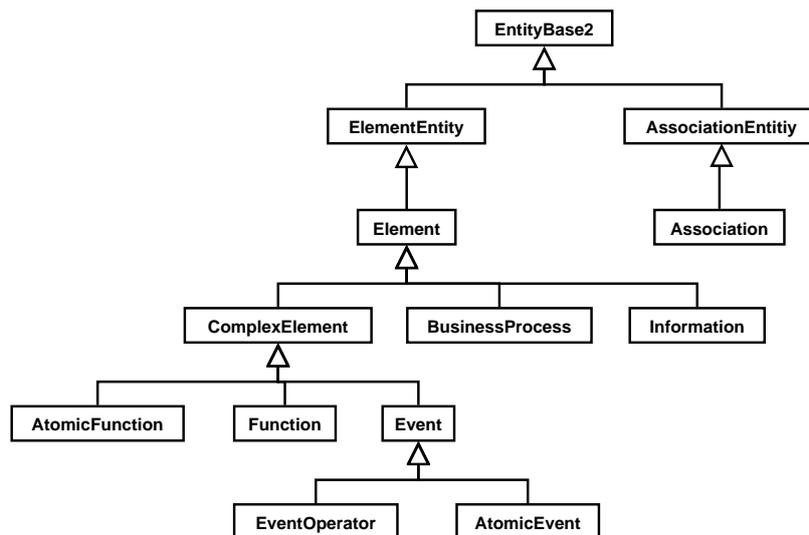


Abbildung 6.3.: Vererbung der Klassen zum Datenbankzugriff

Um die Datenhaltung von der Verwendung in Modellen zu trennen, erfolgt die Verwendung durch Interfaces. Das Interface für eine Funktion ist beispielsweise „IFunction“ und das Interface einer Assoziation ist „IAssociation“. Jedes Interface ist ein Nachfahre von „IUniqueObject“ (vgl. Abbildung 6.4). Durch die Implementierung des Interfaces „IUniqueObject“ sichert jede Klasse zu, dass jede Instanz global eindeutig identifizierbar ist.

Das Interface „IFunction“ bietet die Methoden „GetObject“ und „GetVerb“ an, mit denen das zugehörige Objekt und das zugehörige Verb erfragt werden kann. Das erzeugte Trivialereignis kann nicht durch Methoden des Interfaces „IFunction“ erfragt werden.

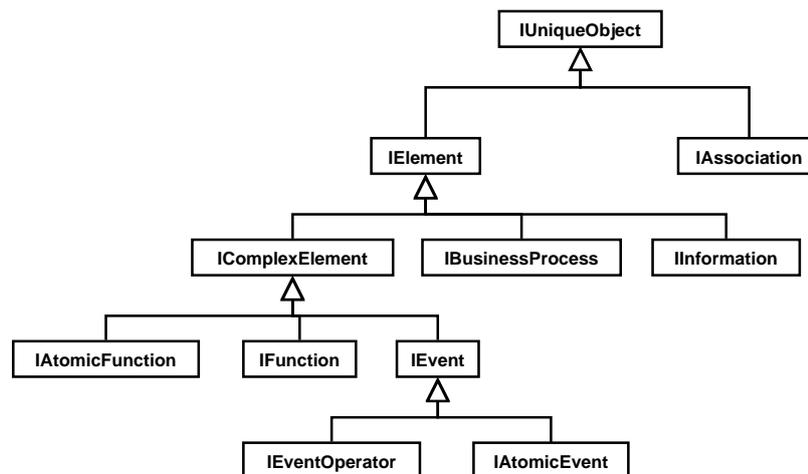


Abbildung 6.4.: Vererbung der Interfaces der Modellelemente

Dafür sind die „Business Components“ in dem Namespace „Gedilan.Nautils.Modelling“ zuständig.

## 6.2. Die Modellierungsschicht

Eigenschaften, die nicht nur einen Bestandteil eines Modells selbst, sondern die Verbindung zu anderen Bestandteilen betreffen, werden grundsätzlich durch Klassen in der Modellierungsschicht realisiert. Jede Klasse in der Modellierungsschicht wird „Business Component“ genannt, da sie Geschäftslogik enthalten. Der Zugriff auf Business Components erfolgt analog zur Datenbankschicht über Interfaces. Für jedes Interface der Datenbankschicht existiert ein Interface in der Modellierungsschicht. Im Gegensatz zur Datenbankschicht sind die Interfaces nicht vererbt (vgl. Abbildung 6.6). Die Klassen, die die Interfaces implementieren, sind analog zu der Datenbankschicht vererbt (vgl. Abbildung 6.5). Somit wird eine implizite Vererbungshierarchie der Interfaces geschaffen.

Die „Business Components“ sind als Singleton (siehe [GHJ97]) realisiert. Durch die „Instance“-Methode jeder „Business Component“ wird die aktuelle Instanz zurückgegeben. Eine Methode der „Business Component“ für Funktionen ist „GetAtomicEventOfFunction“. Diese Funktion gibt das zugeordnete Trivialereignis zurück.

## 6.3. Speicherung des Metamodells

Im Kapitel 3 wurde ein Modellgraph eingeführt, der ein Modell einer Firma realisiert. Die Bestandteile des Modellgraphen sind Knoten, Kanten, sowie Funktionen und Mengen, die die Eigenschaften der Knoten und Kanten näher bestimmen. Diese Aufteilung ist in der Klassenstruktur von Nautilus wiedergespiegelt. Die Knoten des Modellgraphen werden als „Element“ verwaltet. Die Kanten werden als kompakte Kante durch

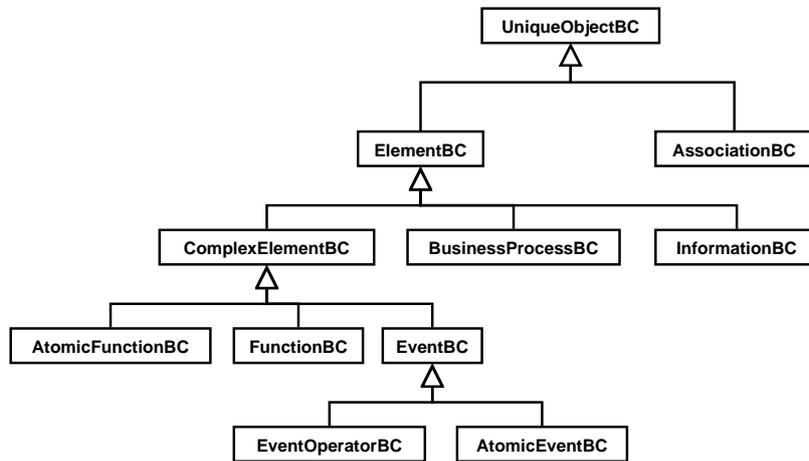


Abbildung 6.5.: Vererbung der Modellierungs-Klassen

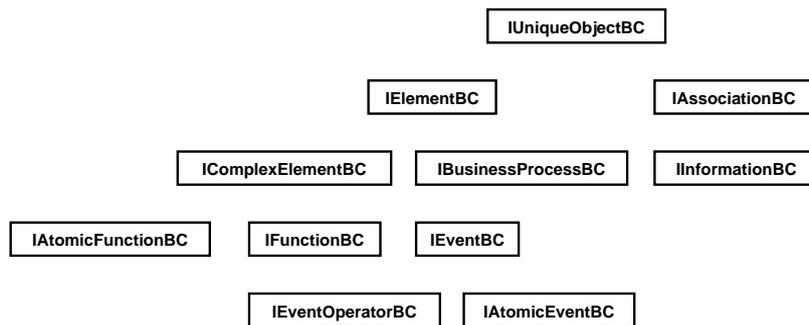


Abbildung 6.6.: Die Modellierungs-Interfaces

die Klasse „Association“ realisiert. Die Beschriftung wird durch Assoziationstypen umgesetzt. Jeder „Association“ ist ein „AssociationType“ zugeordnet, der die Beschriftung realisiert. Jedem Assoziationstyp ist eine Nummer zugeordnet, aus der die Beschriftung abgeleitet wird. Die kompakte Kante, die von Funktionen zu Informationen führt und mit dem Beschriftungstupel (sendet, wird gesendet von) versehen ist, wird von dem Assoziationstyp mit der Nummer 108 realisiert. Die Konstanten dafür sind in der Klasse „AssociationTypes“ definiert. Die Zahl 108 ist der Wert der Konstante „FUNCTION\_INFORMATION\_SEND“.

Die Funktionen und Mengen sind in Datenbanktabellen gespeichert. Die Funktionen sind grundsätzlich als Eigenschaften von Klassen umgesetzt. So wird die Beschriftung grundsätzlich mit der Methode „GetName()“ umgesetzt. Die Beschriftung der Kanten erfolgt über die Klasse „AssociationTypes“. In ihr sind alle Beschriftungen als Konstante gespeichert. Das Interface „IElement“ bietet die Methode „GetAssociations()“ an, mit dem alle Assoziationen eines Elements abgefragt werden können. Zusätzlich gibt es die Möglichkeit, über die „Business Component“ „AssociationBC“ die Assoziationen eines Elements zu erfragen. Das Pendant zu „IElement.GetAssociations()“ in „AssociationBC“ ist „GetAssociations()“. Der Unterschied zwischen dem Zugriff über „IElement“ und dem Zugriff über die „Business Component“ besteht in der Verwendung eines Datenbankkontexts. In Nautilus muss jede Interaktion mit der Datenbank durch einen Kontext geschehen. Ein Bestandteil des Modells kann nur innerhalb eines Kontexts bestehen. Jedem Bestandteil ist deshalb ein Kontext zugeordnet. Falls eine „Business Component“ benutzt wird, muss ein Kontext übergeben werden, der den zurückgegebenen Elementen zugeordnet wird. Ein Kontext wird durch „UniqueObjectBC.Instance.NewContext()“ erzeugt. Falls die Änderungen innerhalb eines Kontexts übernommen werden sollen, muss am Ende der Benutzung die Methode „Save()“ verwendet werden. Falls die Methode „Save()“ nicht verwendet wird, werden Änderungen nicht in die Datenbank übernommen.

## 6.4. Umsetzung des Exports

Der im vorigen Kapitel beschriebene Algorithmus wurde als Modul in der Prozessmodellierungssoftware Nautilus umgesetzt. Dafür wurde der Namespace „Gedilan.Nautilus.Export.BPEL“ eingeführt. In diesem Namespace sind alle Klassen zur Konvertierung enthalten. Die Ausnahme bildet das in Abbildung 6.1 dargestellte Fenster zur Festlegung der Parameter für den Export, welches im Namespace der GUI angesiedelt ist.

Die Komponenten des Namespaces „Gedilan.Nautilus.Export.BPEL“ sind die Datenstrukturen zur Speicherung des EPK-Graphen, die Klassen, die sich ausschließlich mit der Ausgabe befassen und die Klassen, die den Algorithmus umsetzen, zusammengefasst. In der nun folgenden Erläuterung ist das Konzept der Aufteilung der Klassen sowie deren beschrieben.

In Abbildung 6.7 sind die Klassen mit den wichtigsten Attributen gezeigt. Jeder EPK-Graph wird durch eine Instanz der Klasse „Graph“ verwaltet. Jeder Knoten ist durch eine Instanz der Klasse „Node“ und jede Kante durch eine Instanz der Klasse „Edge“ realisiert. Jeder Knoten wird bei der Generierung des EPK-Graphen aus einem Element erzeugt.

Damit bei dem Export auf die Assoziationen des Ursprungselements zugegriffen werden kann, ist in der Klasse „Node“ eine Referenz auf das Ursprungselement abgelegt. Analog dazu wird bei jeder Kante die zugehörige Assoziation referenziert, durch die die Kante generiert wurde. Jede Kante kann mit einer Übergangsbedingung versehen werden, weshalb jede Kante das Attribut „transitionCondition“ besitzt. Falls dieses den Wert „null“ enthält, so besitzt die Kante keine Übergangsbedingung. Die Knoten werden als Aktivität des entstehenden BPEL-Prozesses exportiert. Jede BPEL-Aktivität kann einen Namen besitzen. Deshalb besitzt jeder Knoten das Attribut „ExportName“, das den Namen des Knotens für den BPEL-Export enthält. Die Kanten werden zu links in einer flow-Aktivität. Jeder Link in einer flow-Aktivität ist mit einem eindeutigen Namen versehen. Dieser Name wird von der Klasse „Edge“ bestimmt und ist in dem „ExportName“-Attribut enthalten. Falls eine Kante eine Übergangsbedingung besitzt, so wird diese mittels des Attributes „ExportName“ der Klasse „TransitionCondition“ umgesetzt. Eine Übergangsbedingung wurde aus einem Ereignis generiert. Weiterhin ist jedem Ereignis ein Objekt und ein Zustand zugeordnet. Deshalb ist im Attribut „ExportName“ der Klasse „TransitionCondition“ die Zeichenkette „<Objekt>= '<Zustand>'“ enthalten. „<Objekt>“ steht hier für den BPEL-Namen des zugehörigen Objekts und „<Zustand>“ für den BPEL-Namen des zugehörigen Zustands.

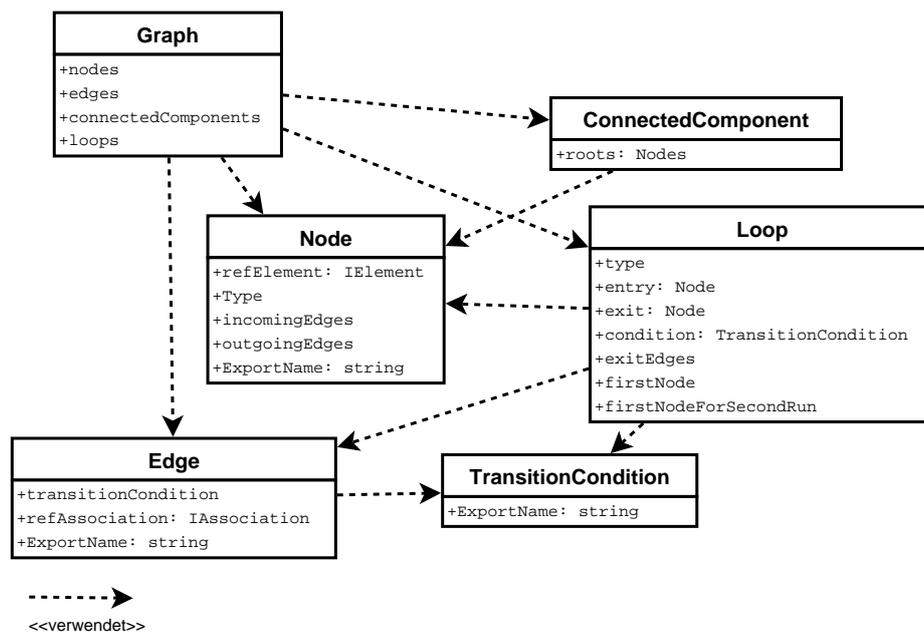


Abbildung 6.7.: Datenstrukturen zur Speicherung der EPK-Grafik

In der Abbildung 6.8 werden die Klassen zum Schreiben der BPEL- und WSDL-Dateien gezeigt. Die Klasse „BPELwriter“ übernimmt die Steuerung des Exports. In ihr wird der EPK-Graph traversiert und jeder Bestandteil exportiert. Je nach der Stellung des Schalters DATAFLOWISINTERNAL und der ein- und ausgehenden Nachrichten wird in „ExportFunction“ entschieden, ob ein „receive“, „invoke“ oder eine „empty“-Aktivität generiert wird. Je nach Entscheidung wird in der Klasse „XmlBPELwriter“ die Methode „WriteStartInvoke“ oder „WriteStartReceive“ verwendet.

Zu jeder „Start“-Methode gehört eine „End“-Methode. Weiterhin müssen bei jeder Aktivität die links durch die Methoden „WriteLinkSource“ und „WriteLinkTarget“ ein- und ausgehenden Kanten geschrieben werden. Da bei jeder invoke-Aktivität unterschiedliche Varianten geschrieben werden können, sind in der Klasse „XmlBPELwriter“ die Methoden WriteStartInvoke und WriteEndInvoke vorhanden. Eine empty-Aktivität besitzt ausschließlich ein- und ausgehende Kanten, weshalb eine empty-Aktivität immer auf die selbe Weise geschrieben wird. Deshalb wird das Schreiben der empty-Aktivität in der Klasse „BPELwriter“ durch die „WriteEmptyActivity“-Methode realisiert und so müssen zum Schreiben einer empty-Aktivität nicht mehrere Methoden aufgerufen werden.

Diese Einschränkung der Funktionalität der Methoden von „XmlBPELwriter“ ist mit dem Design begründet: „XmlBPELwriter“ wurde in Anlehnung an die .net-Klasse „XmlWriter“ entworfen. Die „XmlWriter“-Klasse bietet ausschließlich rudimentäre Methoden zur Generation von XML-Dateien an.

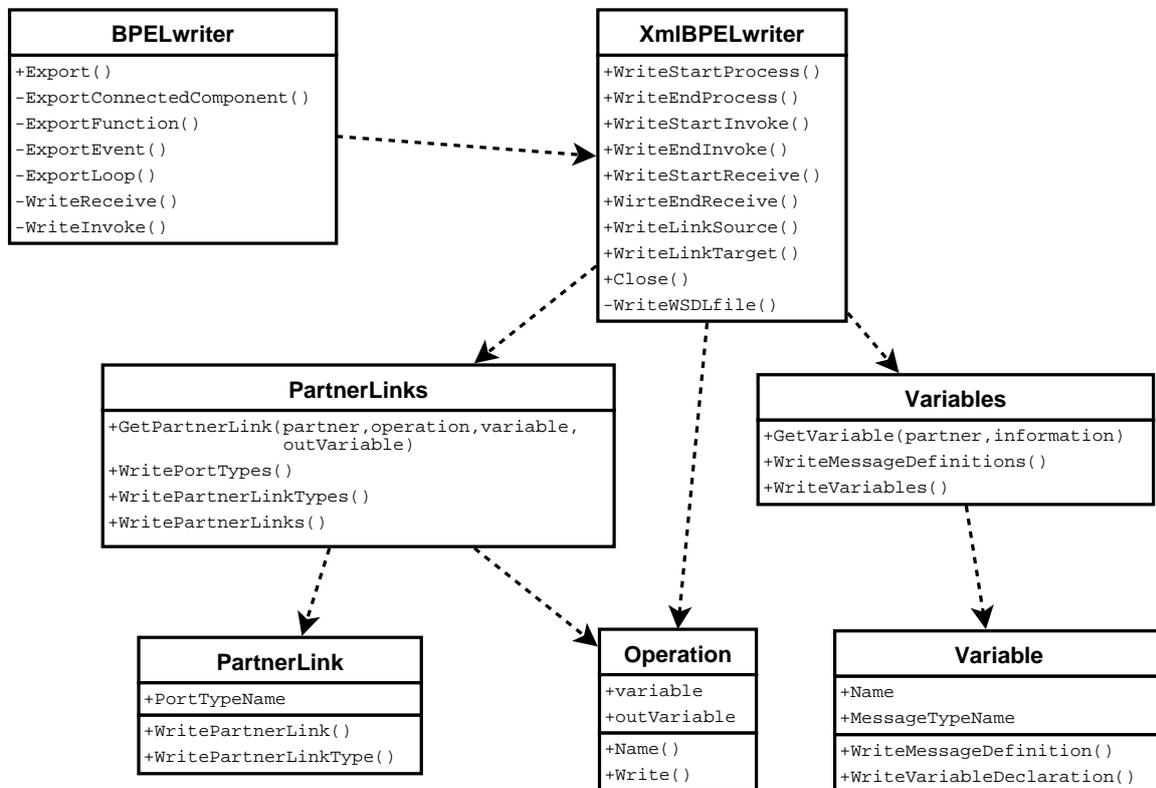


Abbildung 6.8.: Klassen zur Generierung der BPEL- und WSDL-Dateien

Der Export wird von der Klasse „BPEL\_Export“ gesteuert. Die Klasse „BPEL\_Export-PreferencesForm“ instanziiert „BPEL\_Export“ und ruft die Methode „Export“ auf, die dann den Export durchführt. Diese verwendet dann die Klassen „BPELwriter“ und „XmlBPELwriter“ zum Schreiben auf (vgl. Abbildung 6.9). Für den Export selbst werden neben den Klassen zur Datenhaltung und den Klassen zum Erzeugen der BPEL- und WSDL-Dateien die Klassen „NameConversion“, „CheckGraphConditions“ und

„GraphGeneration“ benötigt. Die Klasse „GraphGeneration“ setzt den im Abschnitt 3.7 vorgestellten Algorithmus zur Extraktion des EPK-Graphen aus dem Metamodell um. Die Klasse „CheckGraphConditions“ überprüft, ob der Graph in einen BPEL-Prozess abbildbar ist. Dies umfasst die Überprüfung auf Zyklen, die keine Schleifen darstellen und die Überprüfung der Zusammenhangskomponenten, ob sie als flow-Aktivität oder als Eventhandler exportiert werden können. Die Klasse NameConversion setzt die im Abschnitt 5.1 beschriebene Erzeugung von BPEL-Namen um. Falls ein Ereignis eine Übergangsbedingung darstellt, wird die transitionCondition in der flow-Aktivität durch die Methode „ExportNameForEventAsCondition“ erzeugt.

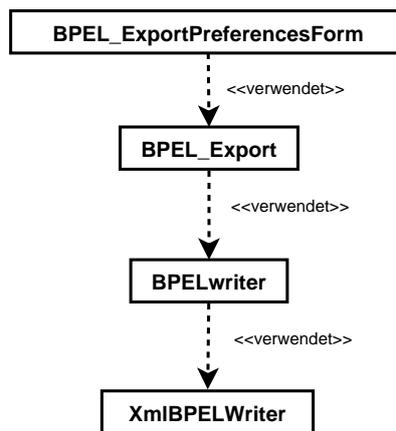


Abbildung 6.9.: Aufrufgraph von Export-Klassen

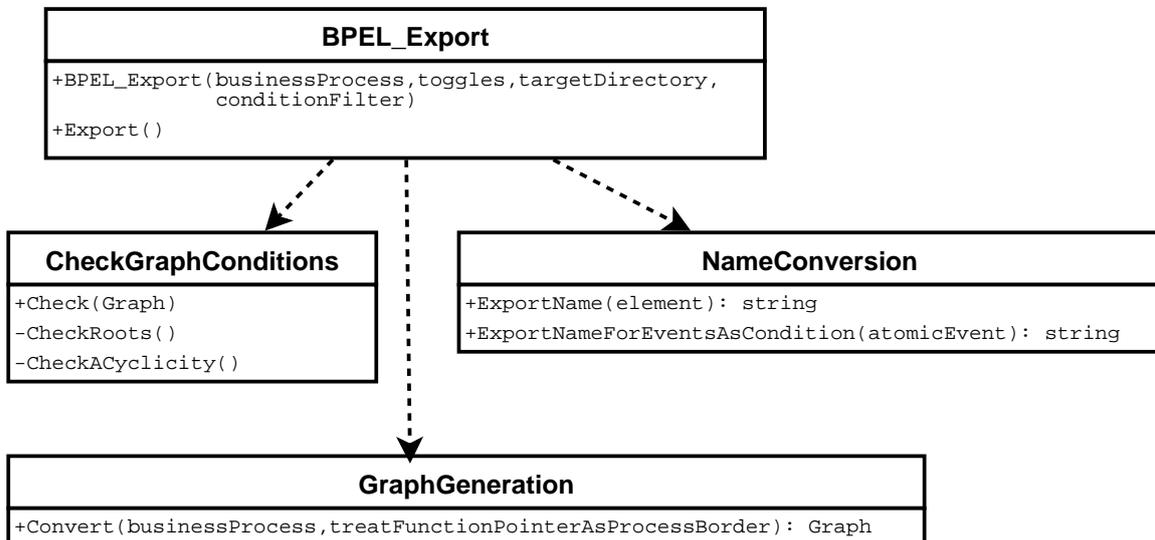


Abbildung 6.10.: Klassen zur Steuerung des Exports

Bei jeder receive- und invoke-Aktivität muss ein portType und ein partnerLink angegeben werden. Diese werden wie im Abschnitt 5.3.2 beschrieben aus der Beziehung

„führt durch“ abgeleitet. Die Klasse „PartnerLinks“ übernimmt die Generierung der `partnerLinks`. Sie sammelt die Information, von wie vielen Partnern die angegebene Operation verwendet wird. Weiterhin stellt sie sicher, dass eine Operation immer die selben Nachrichten sendet und empfängt. Dies wird durch die Vergabe von eindeutigen Suffixen für den Namen der Operation erreicht.

Der Graph wird mittels einer Breitensuche exportiert. Bei der ersten Zusammenhangskomponente wird die Warteschlange der Breitensuche mit dem Wurzelknoten der ersten Zusammenhangskomponente initialisiert. Jeder Knoten der Warteschlange selbst wird exportiert und alle seine Nachfolger in die Warteschlange gelegt. Dieses Verfahren wird so lange wiederholt, bis die Warteschlange leer ist.

Die Ausnahme von diesem Vorgehen bildet der Export von Schleifen. Die Aktivitäten in einer Schleife werden eigenständige `flow`-Aktivität exportiert. Deshalb wird bei Erreichen einer Schleife die Warteschlange geleert und so alle Knoten, die vor der Schleife in die Warteschlange gelegt wurden, exportiert. Daraufhin wird der Kopf der Schleife und der Kopf der `flow`-Aktivität geschrieben. Anschließend werden die Kanten, die die Funktionen in der Schleife enthalten, geschrieben. Jetzt kann der Rumpf der Schleife exportiert werden. Dazu wird der erste Knoten des Schleifenrumpfs in die Warteschlange gelegt und die Warteschlange nach dem oben beschriebenen Verfahren geleert. Jetzt sind alle Funktionen, die in der `while`-Schleife enthalten sind, exportiert und die Nachfolger der Schleife können in die Warteschlange zur weiteren Bearbeitung gelegt werden.

Zu jedem Zeitpunkt, an dem eine `flow`-Aktivität erzeugt wird, müssen die Kanten, die die darin enthaltenen Aktivitäten verbinden, geschrieben werden. Dies geschieht durch den im Abschnitt 5.8 beschriebenen Algorithmus.

Nach dem vollständigen Export der ersten Zusammenhangskomponente werden die weiteren Zusammenhangskomponenten exportiert. Dazu wird zunächst das Grundgerüst des Eventhandlers geschrieben. Daraufhin wird, wie im Abschnitt 5.9 beschrieben, jede weitere Zusammenhangskomponente als `onMessage`-Aktivität, die eine `flow`-Aktivität enthält, exportiert.

Nachdem alle Zusammenhangskomponenten exportiert wurden, sind alle exportierten Operationen bekannt. Somit kann die zugehörige WSDL-Datei durch die Methode `writeWSDLfile()`, wie in Abschnitt 5.11 beschrieben, exportiert werden. Jetzt muss die erzeugte BPEL-Datei mittels `writeEndProcess` abgeschlossen werden und der Ausgabestrom mittels `close` geschlossen werden. Damit ist der Prozess vollständig exportiert worden.

## 6.5. Onlineprüfung

Es ist wichtig, während der Modellierung ein Feedback zu erhalten, ob die gerade modellierte ereignisgesteuerte Prozesskette nach BPEL exportiert werden kann. Der Aufbau einer ereignisgesteuerten Prozesskette ändert sich genau dann, wenn eine neue Kante hinzugefügt wurde. In Nautilus wird von „Business Component“ „AssociationBC“

die Methode „PerformAssociationCheck“ angeboten, die vor dem Einfügen einer neuen Kante in die Datenbank aufgerufen wird. Die Methode „PerformAssociationCheck“ überprüft, ob die hinzuzufügende Assoziation das Modell ungültig machen würde. Eine Verwendung der Klasse „CheckGraphConditions“ ist an dieser Stelle nicht möglich, da somit zyklische Abhängigkeiten erzeugt werden würden. Die Klassen zur Graphgenerierung verwenden Klassen aus dem Namespace „Gedilan.Nautilus.Modelling“. Da „AssociationBC“ zu dem Namespace „Gedilan.Nautilus.Modelling“ gehört, kann es somit keine Klassen von „Gedilan.Nautilus.Export.BPEL“ verwenden.

Eine Alternative ist die Umstrukturierung der Software Nautilus, so dass es einen separaten Namespace für die Überprüfung von Assoziationen gibt. Da eine solche Umstrukturierung einen tiefen Eingriff in die Architektur von Nautilus darstellt, wurde sie nicht vollzogen. Die Eliminierung des Namespaces „Gedilan.Nautilus.Export.BPEL“ und Aufnahme der BPEL-Klassen in den Namespace „Gedilan.Nautilus.Modelling“ ist ebenso ausgeschlossen, da somit die Trennung der Modellingschicht von der Exportschicht aufgewichtet wird.

Deshalb wurde die Onlineprüfung in den „VisualModellerAdapter“ eingefügt. In dieser Klasse wird die Methode „OnVMControlFlowAdded“ dann aufgerufen, wenn in der EPK eine Kontrollflusskante hinzugefügt wurde. Sobald die Kontrollflusskante in der Datenbank gespeichert ist, wird der EPK-Graph erzeugt und dieser auf Gültigkeit überprüft. Falls er nicht die Voraussetzungen für den BPEL-Export nicht erfüllt, wird die Kante in Visio als ungültig markiert und sie wieder aus der Datenbank entfernt. Eine Alternative zu diesem Vorgehen ist der Aufgabe des EPK-Graphen parallel zur Modellierung. Dazu müsste bei dem Öffnen der EPK-Graphik in Visio der EPK-Graph erzeugt werden und bei jeder Änderung des Kontrollflusses der Graph angepasst werden. Eine solche Anpassung besteht bei der Erzeugung einer Schleifen nicht nur aus dem Hinzufügen einer Kante, sondern dem Anpassen der gespeicherten Schleifen. Dazu muss bei jedem Hinzufügen von Kanten erneut eine Dominanzanalyse und die Gültigkeit der Schleifen überprüft werden. Bei der vollständigen Generierung des EPK-Graphens werden prinzipiell die gleichen Schritte durchlaufen. Weiterhin kann die Onlineprüfung wahlweise aktiviert oder deaktiviert werden. Deshalb wird der Code zur Generierung des EPK-Graphen aus der Datenbank weiterhin benötigt und der EPK-Graph würde an zwei Stellen im Programmcode generiert werden. Da sowohl bei der vollständigen Generierung als auch bei dem Hinzufügen oder Entfernen einer Kante eine Dominanzanalyse durchgeführt werden muss, erzeugen beide Algorithmen den gleichen Aufwand. Deshalb wird auf die Anpassung des EPK-Graphens während der Modellierung verzichtet und der Graph bei jeder Änderung der Kanten neu aufgebaut. Die Überprüfung wird dabei von der Klasse „BPEL\_Check“ vorgenommen. Die Klasse „BPEL\_Check“ verwendet dabei die Klassen „GraphGeneration“ und „CheckGraphConditions“ (vgl. Abbildung 6.11).

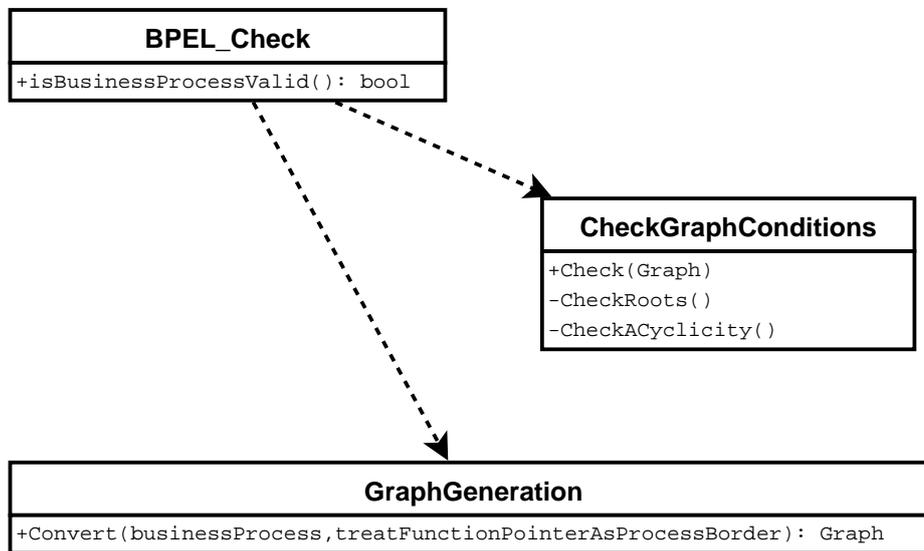


Abbildung 6.11.: Klassen zur Überprüfung der Gültigkeit eines EPK-Graphs



# ZUSAMMENFASSUNG UND AUSBLICK

---

In dieser Arbeit wurde die Welt der ereignisgesteuerten Prozessketten mit der Welt der BPEL-Prozesse verbunden: Ereignisgesteuerte Prozessketten wurden ohne Erweiterungen der Syntax in BPEL-Prozesse abgebildet. Hierfür war es zunächst notwendig, das Metamodel von Nautilus formal zu definieren, da dies bisher nicht geschah. Es folgte die Erarbeitung und Beschreibung eines Algorithmus, der die Abbildung realisiert. Zuletzt wurde die Abbildung als BPEL-Exportmodul in der Prozessmodellierungssoftware Nautilus realisiert.

Der exportierte BPEL-Prozess kann als Vorlage für einen ausführbaren BPEL-Prozess dienen, so dass mittels der vorgestellten Abbildung eine komfortable grafische Programmierung von Geschäftsprozessen vorgenommen werden kann. Mit einer Abbildung von BPEL in ereignisgesteuerte Prozessketten wäre in Zukunft ein inkrementeller Entwicklungszyklus realisierbar, in dem Entscheider und Entwickler über Schnittstellen einander zuarbeiten könnten.

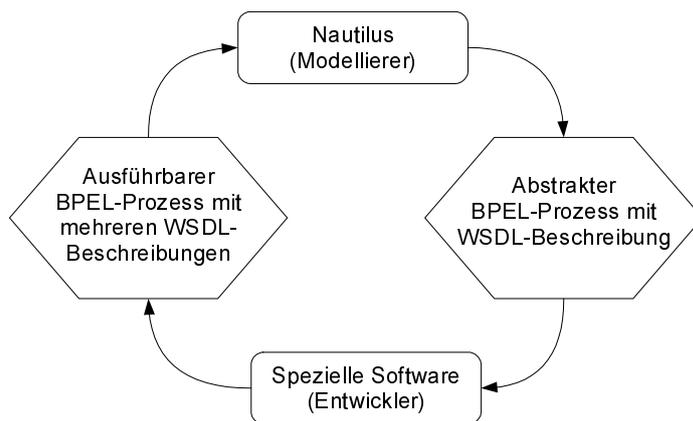


Abbildung 7.1.: Möglicher zukünftiger Modellierungszyklus



## ANHANG A

# TABELLEN

---

In Tabelle A.2 sind die verwendeten mathematische Operatoren aufgeführt. Die verwendeten Symbole finden sich in Tabelle A.4. Die Tabelle A.6 enthält alle verwendeten Mengen, die nicht zu einem Modell gehören, auf. Die Tabelle A.8 fasst alle in der Ausarbeitung verwendeten Funktionen zusammen.

| Bezeichnung     | Kurzerzläuterung   |
|-----------------|--|
| $\in$           | Enthalten in Menge   |
| $\notin$        | Nicht enthalten in Menge   |
| $\neg$          | Logische Negierung   |
| $\vee$          | Logische Voroderung  |
| $\wedge$        | Logische Verundung   |
| $\cup$          | Vereinigung von Mengen   |
| $\dot{\cup}$    | Disjunkte Vereinigung von Mengen. Drückt die Voraussetzung aus, dass die zu vereinigenden Mengen paarweise disjunkt sein müssen. |
| $\cap$          | Schnitt von Mengen   |
| $\setminus$     | Differenz von Mengen   |
| $\blacklozenge$ | Auswahl eines beliebigen Elements aus einer Menge. Siehe Abschnitt 3.1   |
| $\times$        | Kreuzprodukt   |
| $\mapsto$       | Abbildung  |
| $\wp$           | Potenzmenge  |

Tabelle A.2.: Übersicht über die verwendeten Operatoren

| Bezeichnung   | Siehe<br>Abschnitt | Kurzerläuterung   |
|---------------|--------------------|---|
| $\perp$       | —                  | Unbelegtes Tupелеlement   |
| $\leftarrow$  | 3.7 (S. 49)        | Zuweisungsoperator. Die rechte Seite wird der linken Seite zugewiesen |
| $e$           | 3.1 (S. 17 ff.)    | Eine Kante in einem Graphen   |
| $\mathcal{M}$ | 3.6 (S. 30)        | Ein Modellgraph   |

Tabelle A.4.: Übersicht über die verwendeten Symbole

| Bezeichnung          | Siehe<br>Abschnitt | Kurzerläuterung  |
|----------------------|--------------------|--|
| $V$                  | 3.1 (S. 17 ff.)    | Menge aller Knoten in einem Graphen  |
| $V_{DF}$             | 3.6.9 (S. 41)      | Menge aller Datenbeschreibungselemente                                     |
| $E$                  | 3.1 (S. 17 ff.)    | Menge aller Kanten in einem Graphen  |
| <b>trivialevents</b> | 3.4 (S. 23)        | Die Menge aller trivialen Ereignisse                                       |
| $\mathbb{N}$         | –                  | Menge der natürlichen Zahlen   |
| $\mathbb{Q}$         | –                  | Menge der gebrochen rationalen Zahlen                                      |
| $\mathbb{R}$         | –                  | Menge der reellen Zahlen   |
| $\textcircled{S}$    | 3.6.15 (S. 46)     | Die Menge aller spezifischen Eigenschaften eines Elements oder Elementtyps |
| $\mathcal{B}^*$      | 3.6.2 (S. 34)      | Die Menge aller möglichen Kombinationen von Begriffen                      |

Tabelle A.6.: Übersicht über die verwendeten Mengen

---

| Bezeichnung        | Siehe<br>Abschnitt | Kurzerläuterung   |
|--------------------|--------------------|---|
| $[M]$              | 3.1 (S. 19)        | Einschränkung auf eine Untermenge $M$ .                                       |
| COND               | 3.6.14 (S. 46)     | Zuordnung von Bedingungen zu Assoziationen                                    |
| $\Gamma$           | 3.6.5 (S. 36)      | Globale Ordnung der Tätigkeiten   |
| $\iota$            | 3.1 (S. 21)        | Labelfunktion von Graphen   |
| $\iota_B$          | 3.6.16 (S. 47)     | Zuordnung von Beschreibungen zu Elementen                                     |
| $EvtCond(v)$       | 3.7 (S. 52)        | Zuordnung der zusammengefassten Bedingung zu einem neu erzeugten Ereignis $v$ |
| $\mathbb{K}$       | 3.6.12 (S. 43)     | Kontext einer Variante  |
| KAT                | 3.6.13 (S. 44)     | Kategorie eines Elements  |
| $L(G)$             | —                  | Menge der erzeugten Wörter einer Grammatik $G$                                |
| $\mathbb{M}$       | 3.6.1 (S. 32)      | Zuordnung des zugehörigen Masters von einer Variante                          |
| $\pi_i$            | 3.1 (S. 18)        | Projektion auf die $i$ te Komponente eines Tupels                             |
| $\rho$             | 2.3 (S. 13)        | Zuordnung eines Standardzustands  |
| $\mathbb{S}$       | 3.6.15 (S. 46)     | Zuordnung einer spezifischen Eigenschaft                                      |
| $\mathbb{T}$       | 3.4 (S. 23)        | Bestimmung des Typs eines Elements  |
| $\mathbb{T}_{OP}$  | 3.4 (S. 24)        | Bestimmung des Operatortyps   |
| DATAFLOWISINTERNAL | 5.3 (S. 73)        | Bestimmung der Semantik der Datenflussbeziehungen                             |
| VARIANTS           | 3.6.1 (S. 32)      | Bestimmung der Varianten einer Knotenmenge                                    |
| $ways(x, y)$       | 3.1 (S. 21)        | Bestimmung aller Wege von $x$ nach $y$  |
| $way(x, y)$        | 3.1 (S. 21)        | Bestimmung eines Wegs von $x$ nach $y$  |
| $minways(x, y)$    | 3.1 (S. 21)        | Bestimmung der kürzesten Wege von $x$ nach $y$                                |
| $way_s(x, y)$      | 3.1 (S. 21)        | Bestimmung eines kürzesten Wegs von $x$ nach $y$                              |

---

Tabelle A.8.: Übersicht über die definierten Funktionen



## ANHANG B

# GRAMMATIKEN

---

$$\begin{aligned}G_1 &= (N, T, P, S) \\N &= \{S, B\} \\T &= \{a, \dots, z, A, \dots, Z\} \\P &= \{B \rightarrow a | \dots | z | A | \dots | Z, S \rightarrow SS | B\}\end{aligned}$$

Abbildung B.1.: Grammatik  $G_1$  zur Erzeugung von Zeichenketten ohne Leerzeichen

$$\begin{aligned}G_2 &= (N, T, P, S) \\N &= \{S, B\} \\T &= \{a, \dots, z, A, \dots, Z, \sqcup\} \\P &= \{ \\&\quad B \rightarrow a | \dots | z | A | \dots | Z | \sqcup, \\&\quad S \rightarrow SS | B \\&\quad \}\end{aligned}$$

Abbildung B.2.: Grammatik  $G_2$  zur Erzeugung von einzeiligem, nicht leerem Text

$$\begin{aligned}G_3 &= (N, T, P, S) \\N &= \{S, B\} \\T &= \{a, \dots, z, A, \dots, Z, \sqcup, \mathbb{F}\} \\P &= \{ \\&\quad B \rightarrow a | \dots | z | A | \dots | Z | \sqcup | \mathbb{F}, \\&\quad S \rightarrow SS | B \\&\quad \}\end{aligned}$$

Abbildung B.3.: Grammatik  $G_3$  zur Erzeugung von Fließtext



## ANHANG C

# OBERFLÄCHE VON NAUTILUS

Nautilus besteht im Prinzip aus einem Hauptfenster, in dem die Modellierung stattfindet. Im Element-Explorer sind die Elemente aus denen das Modell besteht, dargestellt. Die ereignisgesteuerten Prozessketten werden durch den Ablaufexplorer verwaltet. Bei einem Rechtsklick auf einen Prozess öffnet sich ein Kontextmenü. In diesem kann Microsoft Visio zur grafischen Modellierung gestartet werden. Eine genauere Erläuterung ist im Benutzerhandbuch von Nautilus ([Ge04]) zu finden.

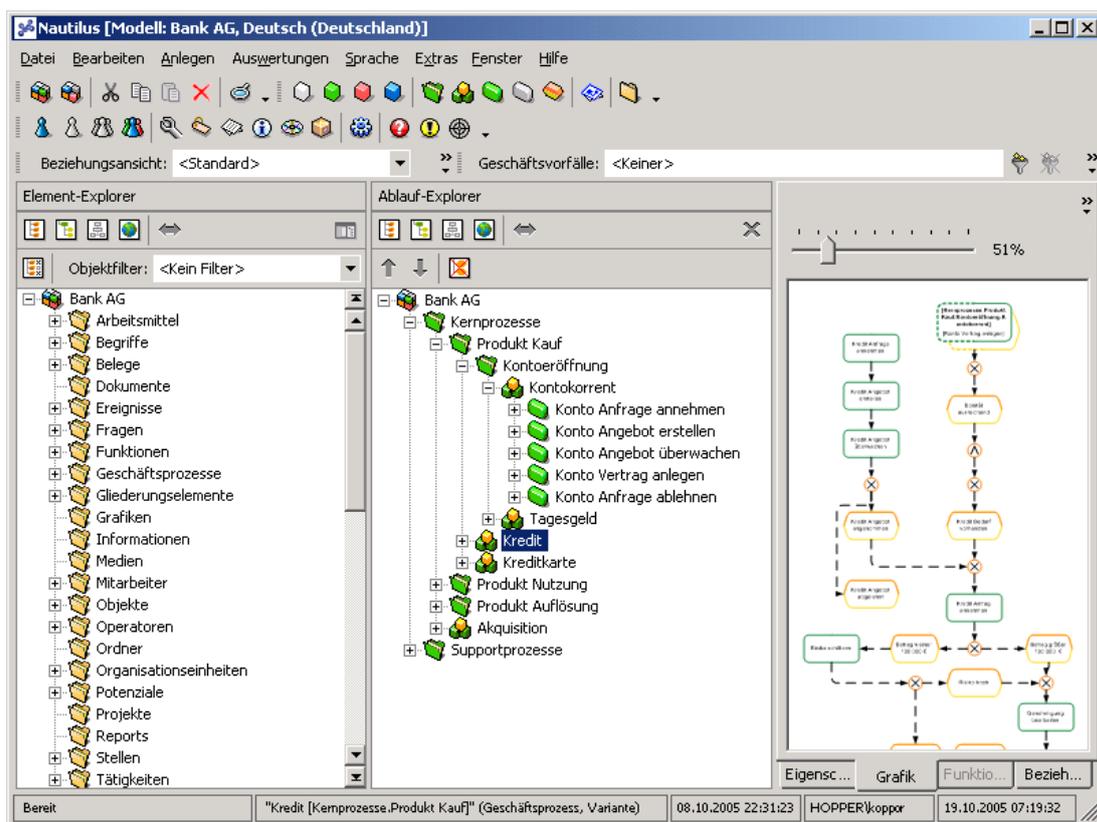


Abbildung C.1.: Das Hauptfenster von Nautilus



# PROZESSE DER BANK AG

---

Die folgenden Prozesse sind den Gedilan-Schulungsunterlagen entnommen. Sie dienen der Illustration der Verwendung von ereignisgesteuerten Prozessketten und werden im Text an geeigneter Stelle referenziert.

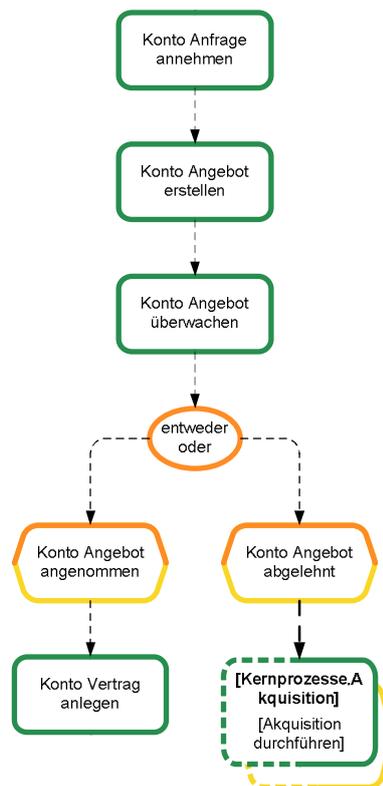


Abbildung D.1.: Kontoeröffnung: Tagesgeld

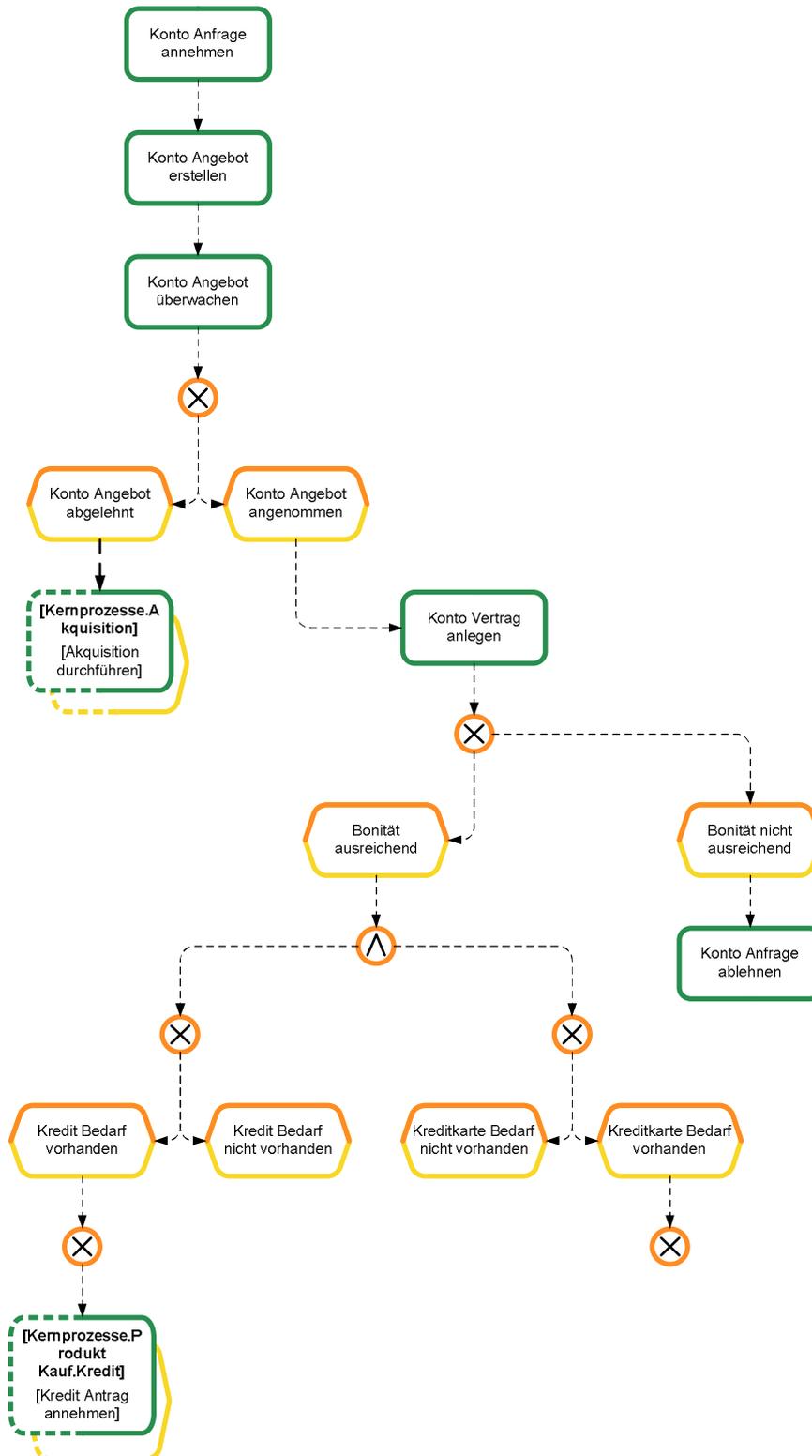


Abbildung D.2.: Kontoeröffnung: Kontokorrent

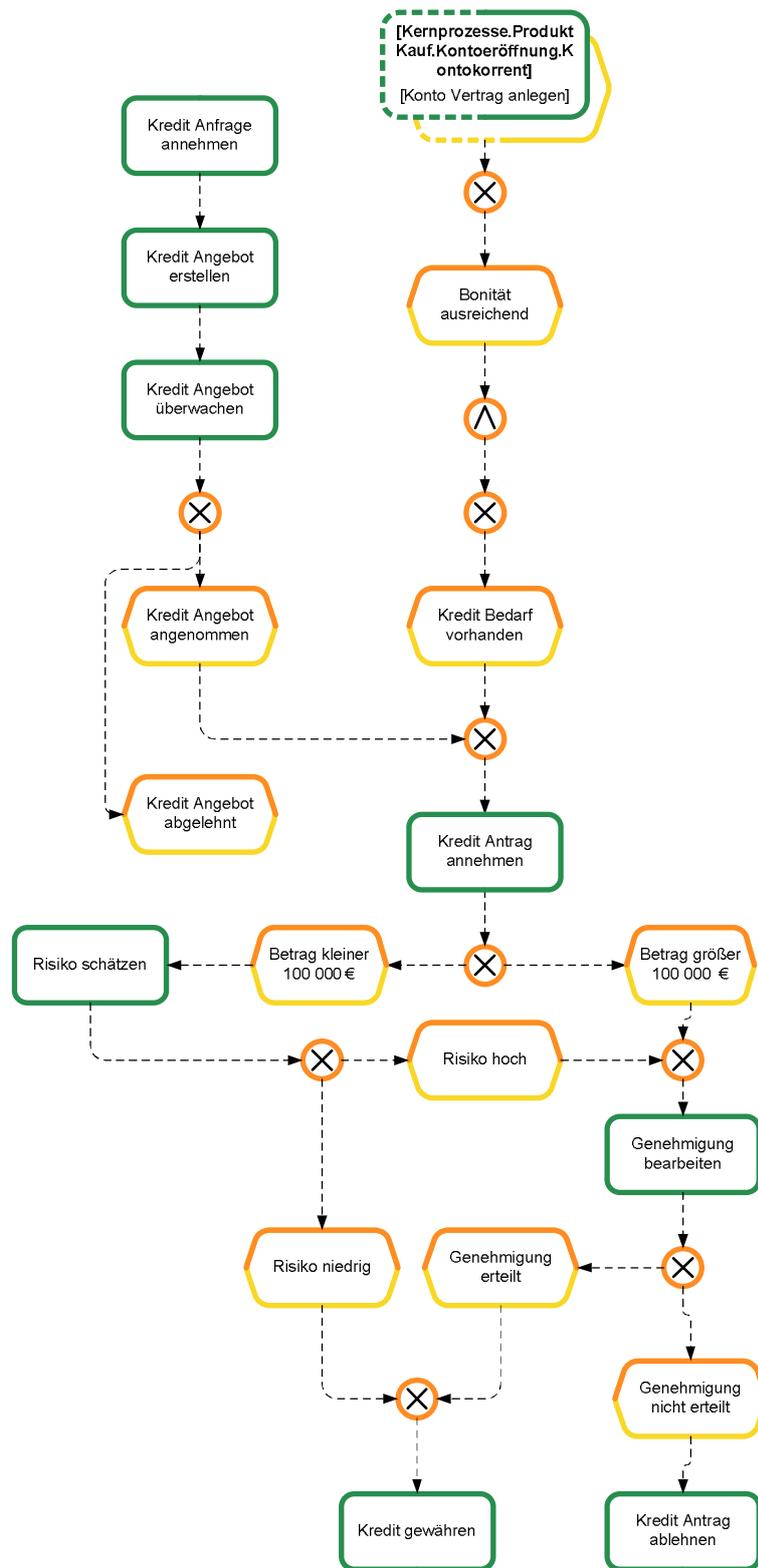


Abbildung D.3.: Kreditvergabe

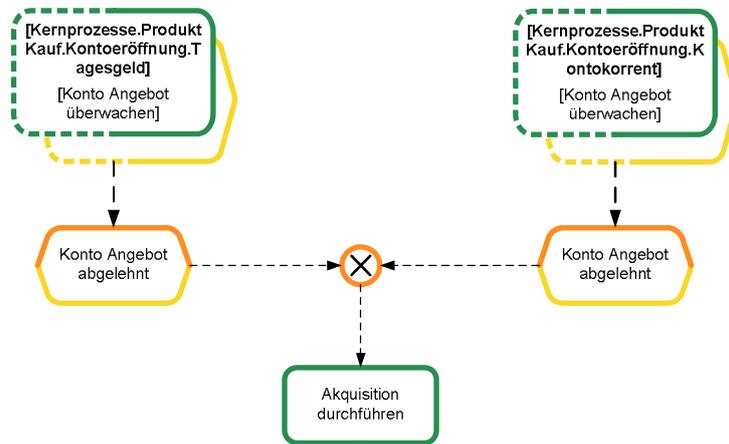


Abbildung D.4.: Akquisition

# UMSETZUNG DES KREDITANTRAGSPROZESSES

---

In diesem Kapitel wird die Umsetzung des in Abbildung 2.3.1 gezeigten Kreditantragsprozesses nach BPEL dargestellt. Der Schalter DATAFLOWISINTERNAL wurde hierfür auf WAHR gestellt, da keine Interaktion mit Web Services, sondern ein prozessinterner Datenfluss modelliert wurde.

## E.1. BPEL-Prozess

```
<process name="Kredit_Antrag_Bearbeitung" targetNamespace="uri"
  abstractProcess="yes" suppressJoinFailure="yes"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:tns="uri"
  xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/">
  <partnerLinks>
    <partnerLink name="Sachbearbeiter_Kredit_Antrag_Bearbeitung"
      partnerLinkType="tns:Sachbearbeiter_Kredit_Antrag_Bearbeitung_LinkType"
      myRole="myRole" />
    <partnerLink name="Abteilungsleiter_Kredit_Antrag_Bearbeitung"
      partnerLinkType="tns:Abteilungsleiter_Kredit_Antrag_Bearbeitung_LinkType"
      partnerRole="partnerRole" />
  </partnerLinks>
  <correlationSets>
    <correlationSet name="dummy" properties="tns:dummyProperty" />
  </correlationSets>
  <variables>
    <variable name="Risiko" messageType="tns:Risiko" />
    <variable name="Status_der_Genehmigung"
      messageType="tns:Status_der_Genehmigung" />
    <variable name="Name" messageType="tns:Name" />
  </variables>
</process>
```

```

    <variable name="Name_Betrag" messageType="tns:Name_Betrag" />
  </variables>
<flow>
  <links>
    <link name="Kredit_Antrag_entgegennehmen-to-Risiko_schätzen" />
    <link name="Risiko_schätzen-to-Kredit_gewähren" />
    <link name="Risiko_schätzen-to-Genehmigung_bearbeiten" />
    <link name="Genehmigung_bearbeiten-to-Kredit_gewähren" />
    <link name="Genehmigung_bearbeiten-to-Kredit_Antrag_ablehnen" />
    <link
      name="Kredit_Antrag_entgegennehmen-to-Genehmigung_bearbeiten" />
  </links>
  <receive name="Kredit_Antrag_entgegennehmen"
    partnerLink="Sachbearbeiter_Kredit_Antrag_Bearbeitung"
    portType="tns:Kredit_Antrag_Bearbeitung_PT"
    operation="Kredit_Antrag_entgegennehmen" variable="Name_Betrag"
    createInstance="yes">
    <source linkName="Kredit_Antrag_entgegennehmen-to-Risiko_schätzen"
      transitionCondition="Betrag = '<=100_000_EUR'" />
    <source
      linkName="Kredit_Antrag_entgegennehmen-to-Genehmigung_bearbeiten"
      transitionCondition="Betrag = '>100_000_EUR'" />
    <correlations>
      <correlation set="dummy" inititate="yes" />
    </correlations>
  </receive>
  <invoke name="Risiko_schätzen"
    partnerLink="Sachbearbeiter_Kredit_Antrag_Bearbeitung"
    portType="tns:Sachbearbeiter_PT" operation="Risiko_schätzen"
    inputVariable="Name_Betrag" outputVariable="Risiko">
    <target linkName="Kredit_Antrag_entgegennehmen-to-Risiko_schätzen"
      />
    <source linkName="Risiko_schätzen-to-Kredit_gewähren"
      transitionCondition="Risiko = 'gering'" />
    <source linkName="Risiko_schätzen-to-Genehmigung_bearbeiten"
      transitionCondition="Risiko = 'hoch'" />
  </invoke>
  <invoke name="Genehmigung_bearbeiten"
    partnerLink="Abteilungsleiter_Kredit_Antrag_Bearbeitung"
    portType="tns:Abteilungsleiter_PT"
    operation="Genehmigung_bearbeiten" inputVariable="Name_Betrag"
    outputVariable="Status_der_Genehmigung">
    <target
      linkName="Kredit_Antrag_entgegennehmen-to-Genehmigung_bearbeiten"
      />
    <target linkName="Risiko_schätzen-to-Genehmigung_bearbeiten" />
  </invoke>
</flow>

```

```

    <source linkName="Genehmigung_bearbeiten-to-Kredit_gewähren"
      transitionCondition="Genehmigung = 'erteilt'" />
    <source
      linkName="Genehmigung_bearbeiten-to-Kredit_Antrag_ablehnen"
      transitionCondition="Genehmigung = 'nicht_erteilt'" />
  </invoke>
  <invoke name="Kredit_gewähren"
    partnerLink="Sachbearbeiter_Kredit_Antrag_Bearbeitung"
    portType="tns:Sachbearbeiter_PT" operation="Kredit_gewähren"
    inputVariable="Name">
    <target linkName="Risiko_schätzen-to-Kredit_gewähren" />
    <target linkName="Genehmigung_bearbeiten-to-Kredit_gewähren" />
  </invoke>
  <invoke name="Kredit_Antrag_ablehnen"
    partnerLink="Sachbearbeiter_Kredit_Antrag_Bearbeitung"
    portType="tns:Sachbearbeiter_PT"
    operation="Kredit_Antrag_ablehnen" inputVariable="Name">
    <target
      linkName="Genehmigung_bearbeiten-to-Kredit_Antrag_ablehnen" />
  </invoke>
</flow>

</process>

```

## E.2. WSDL-Datei

```

<definitions targetNamespace="uri"
  xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="uri"
  xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <message name="Risiko">
    <part name="Risiko" type="xsd:string" />
  </message>
  <message name="Status_der_Genehmigung">
    <part name="Status_der_Genehmigung" type="xsd:string" />
  </message>
  <message name="Name">
    <part name="Name" type="xsd:string" />
  </message>
  <message name="Name_Betrag">
    <part name="Name" type="xsd:string" />
    <part name="Betrag" type="xsd:string" />
  </message>
  <bpws:property name="dummyProperty" type="xsd:string" />

```

```

<bpws:propertyAlias propertyName="tns:dummyProperty"
  messageType="tns:Name_Betrag" part="Name" query="/Name" />
<!--PortTypes, die durch receives generiert wurden-->
<portType name="Kredit_Antrag_Bearbeitung_PT">
  <operation name="Kredit_Antrag_entgegennehmen">
    <input message="tns:Name_Betrag" />
  </operation>
</portType>
<!--PortTypes, die bei genau einem Partner verwendet werden-->
<portType name="Abteilungsleiter_PT">
  <operation name="Genehmigung_bearbeiten">
    <input message="tns:Name_Betrag" />
    <output message="tns:Status_der_Genehmigung" />
  </operation>
</portType>
<portType name="Sachbearbeiter_PT">
  <operation name="Kredit_gewahren">
    <input message="tns:Name" />
  </operation>
  <operation name="Risiko_schätzen">
    <input message="tns:Name_Betrag" />
    <output message="tns:Risiko" />
  </operation>
  <operation name="Kredit_Antrag_ablehnen">
    <input message="tns:Name" />
  </operation>
</portType>
<plnk:partnerLinkType name="Kredit_Antrag_Bearbeitung_LinkType">
  <plnk:role name="myRole">
    <plnk:portType name="tns:Kredit_Antrag_Bearbeitung_PT" />
  </plnk:role>
</plnk:partnerLinkType>
<plnk:partnerLinkType
  name="Sachbearbeiter_Kredit_Antrag_Bearbeitung_LinkType">
  <plnk:role name="partnerRole">
    <plnk:portType name="tns:Sachbearbeiter_PT" />
  </plnk:role>
</plnk:partnerLinkType>
<plnk:partnerLinkType
  name="Abteilungsleiter_Kredit_Antrag_Bearbeitung_LinkType">
  <plnk:role name="partnerRole">
    <plnk:portType name="tns:Abteilungsleiter_PT" />
  </plnk:role>
</plnk:partnerLinkType>
<service name="Kredit_Antrag_Bearbeitung">
  <documentation />

```

```
</service>  
</definitions>
```

### E.3. Darstellung in ActiveWebflow Professional Designer

Der ActiveWebflow Professional Designer ist ein Eclipse-basiertes Modellierungswerkzeug zur Modellierung von BPEL-Prozessen. Der exportierte Kreditantragsprozess wurde in das Werkzeug importiert, woraufhin der Kreditantragsprozess grafisch dargestellt wurde (vgl. Abbildung E.1).

BPEL-Designer. Der Kreditantragsprozess wurde

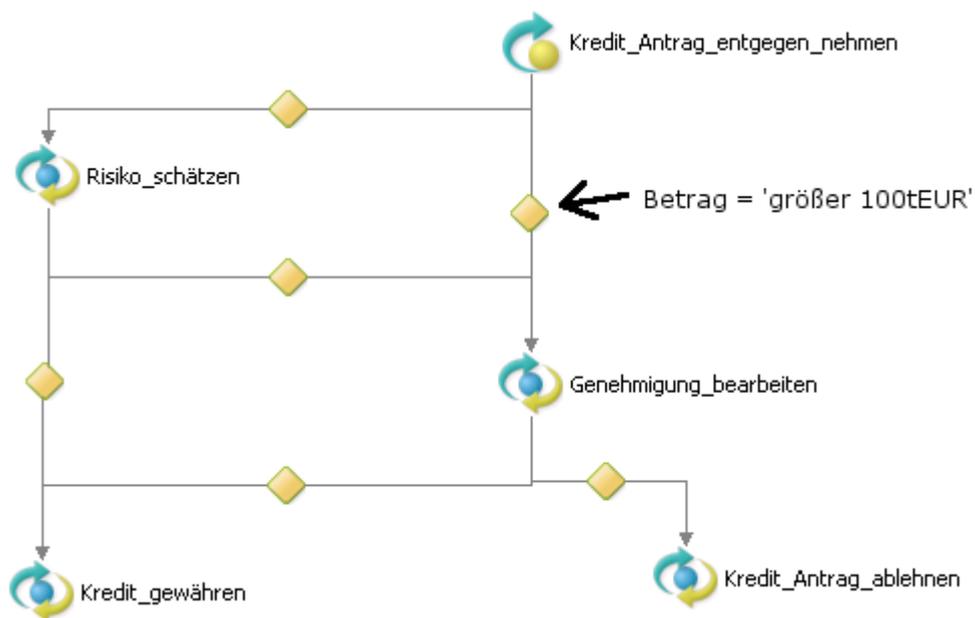


Abbildung E.1.: Der Kreditantragsprozess im ActiveWebflow Professional Designer



# LITERATURVERZEICHNIS

---

- [ADK02] Wil van der Aalst, Jörg Desel und Ekkhart Kindler. On the semantics of EPCs: A vicious circle. <http://wwwcs.uni-paderborn.de/cs/kindler/Publikationen/copies/epk02.pdf>
- [BPWS4J] IBM. BPWS4J – A platform for creating and executing BPEL4WS processes. <http://www.alphaworks.ibm.com/tech/bpws4j>
- [BPELissues] WS BPEL issues list. [http://www.choreology.com/external/WS\\_BPEL\\_issues\\_list.html](http://www.choreology.com/external/WS_BPEL_issues_list.html)
- [BPEL4people] WS-BPEL Extension for People – BPEL4people. <http://www-128.ibm.com/developerworks/webservices/library/specification/ws-bpel4people/>
- [BPEL4WS] Business Process Execution Language for Web Services, version 1.1. <ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf>
- [Cun04] Nicolas Cuntz. Über die effiziente Simulation von Ereignisgesteuerten Prozessketten. Diplomarbeit. Universität Paderborn. <http://wwwcs.uni-paderborn.de/cs/kindler/Forschung/EPCTools/downloads/DA-Cuntz.pdf>
- [Ge04] Nautilus Benutzerhandbuch. 2004. Gedilan-Consulting GmbH. <http://www.gedilan.de>
- [Ge05] Nautilus Schulung. Gedilan-Consulting GmbH. <http://www.gedilan.de>
- [GHJ97] Erich Gamma, Richard Helm, Ralph E. Johnson. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. ISBN 0201633612.
- [Ha99] Claudius Haasis. SAP Business Navigator und SAP R/3 Referenzmodell. Release 4.0b. Arbeitsbereich STS. TU-Hamburg-Harburg. <http://www.sts.tu-harburg.de/slides/1999/04-99-Haas-0S-R3.pdf>

- [He77] M. S. Hecht. Flow Analysis of Computer Programs. Programming Language Series. Elsevier North Holland, Amsterdam. Wie in [JH97] zitiert.
- [ISO9241-10] International Organisation for Standardization. Ergonomic requirements for office work with visual display terminals (VDTs) – Part 10: Dialogue principles. <http://www.iso.org>
- [JH97] Johan Janssen und Henk Corporaal. Making Graphs Reducible with Controlled Node Splitting. ACM Transactions on Programming Languages and Systems, Vol. 19, No. 6, November 1997, Seiten 1031-1052. [http://scholar.google.com/url?sa=U&q=http://portal.acm.org/ft\\_gateway.cfm%3Fid%3D269971%26type%3Dpdf%26d1%3DGUIDE%26d1%3DACM%26CFID%3D11111111%26CFTOKEN%3D2222222](http://scholar.google.com/url?sa=U&q=http://portal.acm.org/ft_gateway.cfm%3Fid%3D269971%26type%3Dpdf%26d1%3DGUIDE%26d1%3DACM%26CFID%3D11111111%26CFTOKEN%3D2222222)
- [KNS92] G. Keller, M. Nüttgens und A.-W. Scheer. Semantische Prozeßmodellierung auf der Grundlage „Ereignisgesteuerter Prozessketten (EPK)“. Technischer Bericht, Institut für Wirtschaftsinformatik, Universität des Saarlandes, Saarbrücken, Januar 1992. Wie in [Rum99] zitiert.
- [Li05] Jesse Liberty. Programming C#. Fourth Edition. O'Reilly. ISBN 0-596-00699-3.
- [LLBLGen] LLBLGen Pro. THE n-tier generator and O/R mapper. <http://www.llblgen.com/>
- [LR99] Frank Leymann und Dieter Roller. Production Workflow – Concepts and Techniques. Prentice Hall PTR, 1999.
- [Lu98] Jochen Ludwig. Software Engineering. Skript zur Vorlesung Software Engineering in der Fakultät Informatik der Universität Stuttgart.
- [Mu97] Steven S. Muchnick. Advanced Compiler Design and Implementation. Academic Press. ISBN 1-55860-320-4.
- [NR02] Markus Nüttgens und Frank J. Rump. Syntax und Semantik Ereignisgesteuerter Prozessketten (EPK). Promise 2002 - Prozessorientierte Methoden und Werkzeuge für die Entwicklung von Informationssystemen, Proceedings des GI-Workshops und Fachgruppentreffens, Potsdam, LNI Vol. P-21, Seiten 64-77, Oktober 2002. [http://epk.et-inf.fho-empden.de/literatur/2002/Promise2002\\_Nuettgens\\_Rump.pdf](http://epk.et-inf.fho-empden.de/literatur/2002/Promise2002_Nuettgens_Rump.pdf)
- [Obe96] A. Oberweis. Modellierung und Ausführung von Workflows mit Petri-Netzen. Teubner-Reihe Wirtschaftsinformatik. Teubner, Stuttgart, Leipzig, 1996. Wie in [Rum99] zitiert.
- [Pe99] Holger Petersen. Vorlesung Graphentheorie. Universität Stuttgart.

- [Rum99] Frank J. Rump. Geschäftsprozeßmanagement auf der Basis ereignisgesteuerter Prozessketten. B.G.Teubner Stuttgart Leipzig. ISBN 3-519-00295-7.
- [Sch91] A.-W. Scheer. Architektur integrierter Informationssysteme: Grundlagen der Unternehmensmodellierung. Springer, Berlin, 1991. Wie in [Rum99] zitiert.
- [Sch95] A.-W. Scheer. Wirtschaftsinformatik – Referenzmodelle für industrielle Geschäftsprozesse. Springer-Verlag, 6. Auflage, 1995. Wie in [Rum99] zitiert.
- [SCHUFA] SCHUFA BusinessLine Scores.  
[http://www.schufa-businessline.de/scoring\\_services.html](http://www.schufa-businessline.de/scoring_services.html)
- [ScSe04] Hermann J. Schmelzer und Wolfgang Sesselmann. Geschäftsprozessmanagement in der Praxis. 4. Auflage. Hanser Verlag. München, Wien 2004. ISBN 3-446-22876-4. Wie in [Wik] zitiert.
- [Se92] Robert Sedgewick. Algorithmen in C. Addison-Wesley. ISBN 3-89319-376-6.
- [SNZ95] A.-W. Scheer, M. Nüttgens und V. Zimmermann. Rahmenkonzept für ein integriertes Geschäftsprozessmanagement. Wirtschaftsinformatik, 37(5):426-434, 1995. Wie in [Rum99] zitiert.
- [URI] Uniform Resource Identifiers (URI): Generic Syntax, RFC 2396.  
<http://www.ietf.org/rfc/rfc2396.txt>
- [WCL05] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann u. a. Web Service Platform Architecture – SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More. Prentice Hall, 2005. ISBN 0-13-148874-0.
- [Wik] Wikipedia. Online-Enzyklopädie. <http://www.wikipedia.de>
- [WSDL] Web Service Description Language (WSDL) 1.1.  
<http://www.w3.org/TR/wsd1>
- [XML] World Wide Web Consortium. Extensible Markup Specification (XML) 1.0 (Third Edition). <http://www.w3.org/TR/2004/REC-xml-20040204/>
- [XML-Names] World Wide Web Consortium. Namespaces in XML.  
<http://www.w3.org/TR/REC-xml-names/>
- [XML-Schema] World Wide Web Consortium. XML Schema.  
<http://www.w3.org/XML/Schema#dev>
- [XPath] XML Path Language (XPath) Version 1.0.  
<http://www.w3.org/TR/1999/REC-xpath-19991116>
- [XQuery] XQuery 1.0: An XML Query Language. W3C working draft.  
<http://www.w3.org/TR/2004/WD-xquery-20040723/>

- [ZM05] J. Ziemann und J. Mendling. EPC-Based modelling of BPEL processes: A pragmatic transformation approach. <http://wi.wu-wien.ac.at/~mendling/publications/05-MITIP.pdf>

# TABELLENVERZEICHNIS

---

|   |    |
|---|----|
| 2.1. Semantik von fork-Operatoren . . . . .   | 12 |
| 2.2. Semantik von Operatoren als join-Knoten . . . . .  | 12 |
| 2.3. Symbole der Operatoren . . . . .   | 13 |
| 3.1. Grundlegende Elemente . . . . .  | 26 |
| 3.2. Elemente der ereignisgesteuerten Prozesskette . . . . .  | 27 |
| 3.3. Das Element Tätigkeit . . . . .  | 27 |
| 3.4. Elemente des Datenflusses . . . . .  | 28 |
| 3.5. Elemente zur Beschreibung anderer Elemente . . . . .   | 28 |
| 3.6. Elemente zur Strukturierung des Modells . . . . .  | 29 |
| 3.7. Bestandteile der Aufbauorganisation . . . . .  | 29 |
| 3.8. Typen und die Bezeichnung der zugehörigen Mengen . . . . .   | 31 |
| 3.9. Allgemeiner Aufbau der Beschriftungstabellen . . . . .   | 34 |
| 3.10. Kompakter Aufbau der Beschriftungstabellen . . . . .  | 35 |
| 3.11. Zugehörigkeit einer Funktion zu einem Geschäftsprozess . . . . .  | 35 |
| 3.12. Kanten zwischen Knoten einer EPK . . . . .  | 35 |
| 3.13. Kanten zwischen Funktionen und Tätigkeiten . . . . .  | 36 |
| 3.14. Aufbau der der Organisation . . . . .   | 37 |
| 3.15. Verbindung von Stellen und Mitarbeitern . . . . .   | 37 |
| 3.16. Leitungsstruktur . . . . .  | 37 |
| 3.17. Stabsstellen . . . . .  | 37 |
| 3.18. Vertretungen . . . . .  | 38 |
| 3.19. Projektorganisation . . . . .   | 38 |
| 3.20. Verantwortung und Verwendung von Arbeitsmitteln . . . . .   | 38 |
| 3.21. Unterstützungen, Mitwirkungen, Durchführungen, Prüfungen, Freigaben<br>und Verantwortlichkeiten . . . . . | 39 |
| 3.22. Mögliche Automatisierungen durch Werkzeuge . . . . .  | 39 |
| 3.23. Mögliche Formen der Beschreibungen . . . . .  | 40 |
| 3.24. Beteiligungen und Verantwortlichkeiten an den Kommentaren . . . . .                                       | 40 |
| 3.25. Nutzung von Waren und Werkzeugen durch Funktionen und Tätigkeiten . . . . .                               | 40 |
| 3.26. Kanten des Modellelements . . . . .   | 41 |
| 3.27. Kanten von Gliederungselementen . . . . .   | 41 |
| 3.28. Übermittlung durch Medien . . . . .   | 41 |
| 3.29. Datenfluss bei Funktionen und Tätigkeiten . . . . .   | 42 |

|   |     |
|---|-----|
| 3.30. Externe Datennutzung (ohne Einbindung in den Datenfluss) durch Funktionen und Tätigkeiten . . . . . | 42  |
| 3.31. Übermittlung, Verwendung und Empfang von Daten innerhalb der Organisation . . . . .                 | 42  |
| 3.32. Kanten von Überbrückungen . . . . .   | 43  |
| 3.33. Typen und zugehöriger Wertebereich . . . . .  | 47  |
| 5.1. Abbildung auf die Aktivitäten bei DATAFLOWISINTERNAL FALSCH . . . . .                                | 73  |
| A.2. Übersicht über die verwendeten Operatoren . . . . .  | 111 |
| A.4. Übersicht über die verwendeten Symbole . . . . .   | 112 |
| A.6. Übersicht über die verwendeten Mengen . . . . .  | 112 |
| A.8. Übersicht über die definierten Funktionen . . . . .  | 113 |

# ABBILDUNGSVERZEICHNIS

---

|  |    |
|--|----|
| 2.1. Illustrationsprozess Kreditantrag nach [LR99, S. 33 ff.] . . . . .  | 8  |
| 2.2. Der Datenfluss des Illustrationsprozesses Kreditantrag . . . . .  | 8  |
| 2.3. Aris-Haus . . . . .   | 10 |
| 2.4. EPK-Grafik eines Geschäftsprozesses zur Bearbeitung eines Kreditantrags . . . . .                           | 11 |
| 2.5. Kompakte Darstellung der EPK aus Abbildung 2.4 . . . . .  | 14 |
| 2.6. Illustration eines Funktionswegweisers in einen anderen Prozess . . . . .                                   | 15 |
| 2.7. Illustration des Auslöseimpulses für eine Funktion durch einen Funktionswegweiser . . . . .                 | 16 |
| 2.8. Die eEPK des Kreditantragsprozesses (Abbildungen 2.1 und 2.2) . . . . .                                     | 16 |
| 3.1. Eine geschachtelte Bedingung . . . . .  | 25 |
| 3.2. Verwendung des selben Ereignisses und Operators in zwei Prozessen . . . . .                                 | 44 |
| 3.3. Funktionen, Geschäftsprozesse und Gliederungselemente mit Kontext . . . . .                                 | 45 |
| 3.4. Die Verbindung von Funktionen, Geschäftsprozessen und Gliederungselement im Kreditantragsprozess . . . . .  | 45 |
| 3.5. Die Menge © aller spezifischen Attribute eines Elements . . . . .   | 47 |
| 4.1. Die Säulen von BPEL . . . . .   | 60 |
| 5.1. Zyklus, der keine Schleife darstellt . . . . .  | 77 |
| 5.2. Klassische Form der while-Schleife . . . . .  | 78 |
| 5.3. Erweiterte while-Schleife . . . . .   | 78 |
| 5.4. Klassische repeat-until-Schleife . . . . .  | 78 |
| 5.5. Erweiterte repeat-until-Schleife mit einer Funktion, die ab dem zweiten Durchlauf ausgeführt wird . . . . . | 79 |
| 5.6. EPK mit einer Schleife mit zwei Ausgängen . . . . .   | 80 |
| 5.7. Schleife mit einer geschachtelten Eingangsbedingung . . . . .   | 81 |
| 5.8. EPK ohne Wurzelknoten . . . . .   | 81 |
| 5.9. Schleife mit mehreren Rückwärtskanten . . . . .   | 82 |
| 5.10. Zwei Schleifen mit gleichem Eingangsknoten . . . . .   | 83 |
| 5.11. Geschachtelte Schleifen . . . . .  | 83 |
| 5.12. Geschachtelte Schleifen . . . . .  | 84 |
| 5.13. Zwei Schleifen mit unterschiedlichem Eingangsknoten . . . . .  | 85 |
| 5.14. Zwei Schleifen mit unterschiedlichem Eingangsknoten . . . . .  | 85 |

---

|   |     |
|---|-----|
| 5.15. Veranschaulichung der Übernahme der join-Condition . . . . .              | 88  |
| 5.16. Prozess vor der Streichung der Operatoren . . . . .                       | 88  |
| 5.17. Prozess nach der Streichung von Operatoren . . . . .                      | 89  |
| 5.18. Prinzip der Abbildung . . . . .   | 90  |
| 5.19. Illustration von nicht erlaubten EPK-Formen . . . . .                     | 91  |
| 5.20. Mögliche Modellierung von reply . . . . .                                 | 93  |
| 5.21. Struktur eines picks oder eines switches in einer EPK . . . . .           | 93  |
| 5.22. Struktur einer Sequence in einer EPK . . . . .                            | 94  |
| 6.1. Fenster zum Steuern des Exports . . . . .                                  | 95  |
| 6.2. Architektur von Nautilus . . . . .   | 97  |
| 6.3. Vererbung der Klassen zum Datenbankzugriff . . . . .                       | 98  |
| 6.4. Vererbung der Interfaces der Modellelemente . . . . .                      | 99  |
| 6.5. Vererbung der Modellierungs-Klassen . . . . .                              | 100 |
| 6.6. Die Modellierungs-Interfaces . . . . .                                     | 100 |
| 6.7. Datenstrukturen zur Speicherung der EPK-Grafik . . . . .                   | 102 |
| 6.8. Klassen zur Generierung der BPEL- und WSDL-Dateien . . . . .               | 103 |
| 6.9. Aufrufgraph von Export-Klassen . . . . .                                   | 104 |
| 6.10. Klassen zur Steuerung des Exports . . . . .                               | 104 |
| 6.11. Klassen zur Überprüfung der Gültigkeit eines EPK-Graphs . . . . .         | 107 |
| 7.1. Möglicher zukünftiger Modellierungszyklus . . . . .                        | 109 |
| B.1. Grammatik $G_1$ zur Erzeugung von Zeichenketten ohne Leerzeichen . . . . . | 115 |
| B.2. Grammatik $G_2$ zur Erzeugung von einzeiligem, nicht leerem Text . . . . . | 115 |
| B.3. Grammatik $G_3$ zur Erzeugung von Fließtext . . . . .                      | 115 |
| C.1. Das Hauptfenster von Nautilus . . . . .                                    | 117 |
| D.1. Kontoeröffnung: Tagesgeld . . . . .  | 119 |
| D.2. Kontoeröffnung: Kontokorrent . . . . .                                     | 120 |
| D.3. Kreditvergabe . . . . .  | 121 |
| D.4. Akquisition . . . . .  | 122 |
| E.1. Der Kreditantragsprozess im ActiveWebflow Professional Designer . . . . .  | 127 |

# VERZEICHNIS DER LISTINGS

---

|   |     |
|---|-----|
| 5.1. Bestandteile eines exportierten process-Elements . . . . . | 72  |
| 5.2. Definition der Dummy-Property . . . . .                    | 92  |
| Kredit Antrag Bearbeitung.bpel . . . . .                        | 123 |
| Kredit Antrag Bearbeitung.wdsl . . . . .                        | 125 |



# VERZEICHNIS DER ALGORITHMEN

---

|   |    |
|---|----|
| 3.1. Generierung der Mengen $V$ und $E$ aus einem Geschäftsprozess $GP$ . . . . .   | 50 |
| 3.2. Setzen des Knotentyps . . . . .  | 51 |
| 3.3. Setzen der Beschriftung . . . . .  | 51 |
| 3.4. Generierung von $G = (V, E, \iota, \mathbb{T}, \mathbb{T}_{OP})$ mit Funktionswegweisern als Verbindung zu einem anderen Prozess . . . . . | 51 |
| 3.5. Generierung der Mengen $V$ und $E$ aus einem Geschäftsprozess $GP$ mit Funktionswegweisern als Prozessgrenze. . . . .                      | 52 |
| 3.6. Generierung der Wege in den EPK-Graphen . . . . .  | 52 |
| 3.7. Generierung von $G = (V, E, \iota, \mathbb{T}, \mathbb{T}_{OP})$ mit Funktionswegweisern als Prozessgrenze . . . . .                       | 53 |



## **Erklärung**

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

---

(Oliver Kopp)

