Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Konzept zur Verknüpfung von Use Cases mit ereignisgesteuerten Prozessketten

Masterarbeit

im Studiengang Informatik

von

Dimitri Diegel

Erstprüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Prof. Dr. Michael H. Breitner
Betreuer: Dipl. Wi.-Inf. Daniel Lübke

Hannover, 20 Oktober 2006

Erklärung

Hiermit versichere ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Dimitri Diegel, Hannover, den 20.10.2006

Für die hervorragende Betreuung meiner Arbeit, die hilfreichen und anregenden Ratschläge danke ich Herrn Daniel Lübke.

Inhaltsverzeichniss

I. Einführung

I.1 Motivation

Um konkurrenzfähig zu bleiben und die Kosten zu senken sind die Unternehmen ständig bemüht ihre geschäftlichen Abläufe zu optimieren. Damit die Geschäftsprozesse eines Unternehmens analysiert, bewertet und schließlich optimiert werden können, müssen sie erst einmal dokumentiert werden. Für die Modellierung und damit auch die Dokumentation von Geschäftsprozessen hat sich heutzutage das auf ereignisgesteuerten Prozessketten (EPK) basierte ARIS[1]- Konzept als internationaler Standard durchgesetzt [AWScheer2001].

Die Optimierung von Geschäftsprozessen erfolgt nicht selten dadurch, dass Teile dieser Prozesse durch Softwaresysteme automatisiert werden. Die Erhebung und Fixierung von funktionalen Anforderungen ist der erste Schritt im Entwicklungszyklus jedes Softwaresystems. Requirements Engineering hat sich inzwischen zu einer eigenen Disziplin innerhalb des Software Engineering entwickelt, sie befasst sich mit den Techniken und Methoden der Aufnahme, Analyse, Dokumentation und Management von Anforderungen. Die Use Cases (Anwendungsfälle) ist eine weit[2] verbreitete Technik auf dem Gebiet des Requirements Engineering zur Dokumentation von funktionalen Anforderungen.

Die mit den EPKs modellierten Geschäftsprozesse sind also oft Teil eines Softwareentwicklungsprojektes. Die betriebswirtschaftlichen Abläufe und Zusammenhänge stehen jedoch im Vordergrund der Geschäftsprozessmodellierung, unabhängig davon wie sie mit Soft- oder Hardware unterstützt werden. Die mit den EPKs definierten Anforderungen sind deshalb für die Entwickler von Softwaresystemen nicht detailliert genug und müssen oft mit den Methoden des Requirements Engineering präzisiert werden [GeUML2003]. Die Beschleunigung dieses Prozesses durch computergestützte Tools sowie die explizite Zuordnung zwischen EPKs und entsprechenden Use Cases anhand von offenen Standards wäre daher äußerst wünschenswert.

I.2 Aufgabenstellung

Das Ziel dieser Arbeit ist die Entwicklung und Implementierung eines Konzepts mit dem eine Zuordnung zwischen den Anforderungen in Form von Use Cases und den entsprechenden Prozessen in Form von EPKs vorgenommen werden kann. Dieses Konzept soll es also ermöglichen sowohl aus Use Cases die entsprechende EPK- Darstellung zu erzeugen als auch umgekehrt aus den EPKs die entsprechenden Use Cases zu generieren. Um die Integration in möglichst viele computergestützte Werkzeuge zu erleichtern oder überhaupt zu ermöglichen, soll die Implementierung auf einem offenem, allgemein anerkanntem Datenaustauschformat basieren, also auf XML[3]. Die Beschreibung der notwendigen Transformation in XML setzt voraus, dass sowohl die EPKs als auch die Use Cases in einem XML- Format vorliegen. Ein XML- Format für die EPKs wurde bereits in [EPK2002 Seite 81] vorgeschlagen, ein XML-

[1] Architektur Integrierter Systeme
[2] Die Use Cases sind fester Bestandteil vieler heutzutage gebräuchlichen Softwareentwicklungsmodelle: Wasserfallmodell, Rational Unified Process, Exreme Programming
[3] Extensible Markup Language

Format für die Use Case liegt noch nicht vor und soll deshalb in dieser Arbeit ebenfalls entwickelt werden.

Die Use Cases und die EPKs haben zwar Gemeinsamkeiten, sind aber dennoch nicht vollkommen äquivalent, wie sich im Laufe dieser Arbeit herausstellen wird. Eine vollautomatische Übersetzung der EPKs in entsprechende Use Cases und umgekehrt wird also ohne gewisse Benutzereingabe kaum möglich sein. Sowohl die Ausarbeitung welche Informationen durch den Menschen eingegeben werden müssen, als auch die Entwicklung eines Prototypen mit dem die Eingabe dieser Informationen vorgenommen werden kann ist ein weiteres Ziel der vorliegende Arbeit.

I.3 Aufbau

Im ersten Kapitel wird das Konzept der Use Cases und der EPKs vorgestellt und näher erläutert. Die Use Cases und die EPKs bilden die Grundlage für diese Arbeit.

Im zweiten Kapitel werden die Ausdrucksmöglichkeiten der Use Cases und EPKs miteinander verglichen und auf die Unterschiede und Gemeinsamkeiten dieser Konzepte eingegangen. Ausgehend von diesem Vergleich werden Abbildungsvorschriften vorgeschlagen, die eine Zuordnung zwischen den Elementen dieser beiden Modelle ermöglichen.

Im dritten Kapitel werden die XML Formate für Use Cases und EPKs vorgestellt, EPML[1] wurde von [MendlingNüttgens] veröffentlicht, und stellt ein Konzept zur Abspeicherung der EPKs in Form einer XML Datei dar, da für Use Cases noch kein XML Format vorliegt, ist die Ausarbeitung eines Use Case Formats ein weiterer Meilenstein dieses Kapitels.

XSLT[2] ist eine Scriptsprache, die entwickelt wurde um Transformationen von XML Dokumenten zu beschreiben, da sowohl für die Use Cases als auch für die EPKs ein XML-Schema (das im 3 Kapitel vorgestellt worden ist) vorliegt, wird XSLT verwendet um die in Kapitel 2 definierten Transformationsvorschriften umzusetzen. Kapitel 4 enthält eine kurze Einführung in XSLT sowie die Dokumentierung der Implementierung dieser Vorschriften.

Die XML Beschreibungen von Use Cases und besonders von EPKs können von den Menschen nur mit viel Mühe gelesen und verstanden werden. Im Rahmen dieser Arbeit wird deshalb ein Prototyp entwickelt mit dem EPKs und Use Cases erstellt und in einander überführt werden können, die Benutzung dieses Programms sowie die Vorzüge des in dieser Arbeit entwickelten Konzepts werden anhand eines Anwendungsszenario im fünften Kapitel demonstriert.

Das sechste Kapitel bietet sowohl eine Zusammenfassung der Ergebnisse, die in dieser Arbeit erzielt wurden als auch einen Ausblick auf weitere Arbeiten, die in dieser Richtung bereits betrieben wurden und in Zukunft noch werden könnten.

[1] Eventdriven Processchain Modelling Language
[2] eXtensible Stylesheet Transformation Language

1. Grundbegriffe

1.1 Use Case

Missstände beim Anforderungsmanagement ist der häufigste Grund für das Scheitern von Softwareprojekten [Standishgroup]. Die Erhebung und Fixierung der Anforderungen ist daher ein unverzichtbarer Bestandteil jedes Softwareentwicklungsprozesses. Die in den 60er Jahren von Ivar Jacobson entwickelte Technik der Use Cases ist eine weit verbreitete, allgemein anerkannte Vorgehensweise auf diesem Gebiet.

In einem Use Case (Anwendungsfall) wird beschrieben wie ein Benutzer durch eine Folge von Interaktionen mit dem System eines seiner Ziele erreichen kann. Der zweithäufigste Grund für das Scheitern von Softwareprojekten ist die unzureichende Einbeziehung des Kunden in die Entwicklung [Standishgroup], daher ist es wichtig, dass die Use Cases nicht nur von den Entwicklern sondern auch von den Auftraggebern gelesen und verstanden werden. Die Verfassung der Use Cases erfolgt daher in einer natürlichen Sprache, ohne jegliche informatikspezifische Notationen oder Diagramme.

Die Use Cases können nicht nur als Vertrag zwischen Auftraggebern und Entwicklern dienen, sondern sie enthüllen auch den Umfang und Zweck des zu erstellenden Systems und werden daher für die Entwicklung des Designs benutzt. Ein weiterer Verwendungszweck der Use Cases ist z.B. die Abschätzung der Komplexität und der Kosten des zu erstellenden Systems, dieses Konzept wurde in [Clemmons2006] ausführlich beschrieben.

Folgendes Beispiel demonstriert wie eine Anforderung an die Funktionalität eines Geldautomaten in Form eines Use Cases notiert wird. Die Anforderung ist stark vereinfacht, damit der Use Case übersichtlich bleibt.

Titel	Geld am Automaten abheben
Primärakteur	Ein Kunde
Umfang	Geldautomat
Ebene	Anwenderziel
Standardszenario	1. Kunde steckt seine Karte in den Geldautomaten 2. Kunde gibt seine PIN ein 3. Kunde gibt den Geldbetrag ein 4. Geldautomat gibt das Geld aus
Erweiterungen	1.1 Karte ist falsch rum eingesteckt 1.1.1 Automat informiert den Kunden 1.1.2 Kunde dreht seine Karte um 1.1.3 Weiter mit Schritt 2 2.1 Die PIN passt nicht zur eingesteckten Karte 2.1.1 Automat fordert eine andere PIN 2.1.2 Weiter mit Schritt 2

Abbildung 1-1

Nach dem Buch „Use Cases effektiv erstellen“ [AC2003] von Alistair Cockburn, der heutzutage als Fachmann in dem Verfassen von Use Cases gilt, sollte ein Use Case mindestens folgende Komponenten aufweisen:

- **Titel:** Das Ziel des Primärakteurs wird als Titel des Use Cases formuliert. Im oberen Beispiel besteht dieses Ziel daran an einem Automaten einen Geldbetrag abzuheben.

- **Primärakteur:** Jemand oder etwas (z.B. eine Person, eine Organisation oder ein Computersystem) dessen Ziele mit Hilfe des zu modellierenden Systems erreicht werden sollen. Der Primärakteur in dem oberen Beispiel ist ein Kunde der Bank, die den Geldautomaten aufgestellt hat.

- **Umfang:** Umfasst die Hardware- und Softwareteile, die zum zu modellierenden System gehören. Das System besteht in diesem Fall aus dem Geldautomaten und der darauf laufenden Software.

- **Ebene:** Die Aktionen eines Szenarios können fast immer in feinere Aktionen zerlegt werden. In [AC2003] werden drei Ebenen definiert: Überblicksebene, Anwenderebene und Subfunktionsebene. Ein Use Case kann auf mehreren dieser Ebenen formuliert werden. Falls ein Use Case auf der Überblicksebene formuliert ist, dann beschreibt er nur grob die Aktionen des Akteurs, seine Teil- Use Cases können auf einer beliebigen Ebene liegen. Falls ein Use Case auf der Anwenderebene spezifiziert ist, dann enthält er diejenigen Aktionen, die für ein unmittelbares Ziel des Primärakteurs notwendig sind. Die Schritte eines Use Case auf Ebene der Subfunktionen befriedigen ein Teilziel eines Use Case der Anwenderebene. Die Use Cases auf dieser Ebene werden nur dann geschrieben, wenn sie von mehreren anderen Use Cases referenziert werden.

- **Standardszenario:** Das Standardszenario enthält eine Sequenz von Interaktionen des Primärakteurs mit dem System und endet mit dem Erreichen des Ziels des Primärakteurs. Im oberen Bespiel ist das Ziel einen Geldbetrag abzuheben nach der letzen Aktion „Automat gibt das Geld aus“ erreicht. Jede Interaktion des Akteurs ist entweder eine einfache Aktion oder kann aus mehreren Aktionen bestehen, die einen eigenständigen Use Case bilden.

- **Erweiterungen:** Während der Interaktion mit dem System kann es passieren, dass etwas nicht so läuft, wie der Primärakteur es vorsieht. Die Aktionen, die vom Standardszenario abweichen, weil z.B. ein Fehler passiert ist, werden im Erweiterungsteil des Use Cases festgehalten. Es kann z.B. passieren, dass der Kunde seine Karte falsch rum eingesteckt hat, dann verläuft die Interaktion so weiter, wie es in dem entsprechenden Erweiterungsteil spezifiziert ist (Aktionen 1.1.1- 1.1.3).

Folgende Use Case Komponenten werden von [AC2003] als optional angesehen:

- **Vorbedingungen:** Hier wird angegeben welche Bedingungen vor dem Ablauf des Use Case Szenario erfüllt sein müssen. Eine Vorbedingung zeigt an, dass ein anderer Use Case abgelaufen ist, der sie eingerichtet hat. Die Vorbedingung „Der Geldautomat ist betriebsbereit“ wäre für das Beispiel aus der Abbildung 1-1 eventuell sinnvoll.

- **Invarianten:** Hier werden die Mindestgarantien des Systems an den Primärakteur festgehalten, also diejenigen Bedingungen die sichergestellt sind, egal ob der Akteur sein Ziel erreicht hat oder nicht.

- **Trigger:** Als Trigger wird dasjenige Ereignis bezeichnet, welches die Ausführung des Use Case Szenario auslöst. Die erste Aktion des Standardablaufs und der Trigger sind oft ein und dasselbe. „Der Kunde steckt seine Karte in den Automaten“ kann für das Beispiel aus der Abbildung 1-1 auch als Trigger angesehen werden.

- **Nachbedingungen:** Die Nachbedingungen benennen die Ziele des Primärakteurs, die nach dem Ablauf des Standardszenario oder eines anderen Erfolgspfades erfüllt wurden. Für das Beispiel aus der Abbildung 1-1 wäre „Der Kunde hat den gewünschten Geldbetrag erhalten“ z.B. so eine Nachbedingung.

Die Nachbedingung eines Use Case kann als Vorbedingung in einem anderen auftauchen, auf diese Weise können unterschiedliche Use Cases miteinander verbunden werden. Wie schon erwähnt kann eine Aktion eines Use Case als Abkürzung für einen anderen Use Case stehen, in diesem Fall wird diese Aktion unterstrichen. Folgendes Beispiel demonstriert wie Use Cases durch Referenzen sowie Nach- und Vorbedingungen miteinander verknüpft werden können.

Titel	Geldautomaten in Betrieb nehmen
…	…
Standardszenario	1. Der Techniker baut den Automaten zusammen 2. Der Techniker schließt den Automaten an das Netz an
Nachbedingungen	1. Der Geldautomat ist betriebsbereit

Abbildung 1-2

Bevor der Geldautomat in Betrieb genommen werden kann, muss er zusammengebaut und an das Computernetzwerk der Bank angeschlossen werden. Der erste Schritt des Standardszenarios referenziert den Use Case „Geldautomat zusammenbauen“, der zweite Schritt referenziert „Geldautomat anschließen“, beide Use Cases sind unten dargestellt.

Titel	Geldautomaten zusammenbauen
…	…
Standard-Szenario	1. Teil X mit Teil Y montier 2. Teil X mit Teil Z montier …
Nach-Bedingungen	1. Der Automat ist zusammengebaut

Titel	Geldautomaten anschließen
…	…
Vor-Bedingungen	1. Der Automat ist zusammengebaut
Standard-Szenario	1. Stecker X in die Steckdose Y 2. Schalter Z umlegen …
Nach-Bedingungen	1. Der Automat ist angeschlossen

Abbildung 1-3

Die Reihenfolge in der diese beiden Use Cases abgearbeitet werden ist einerseits durch die Abfolge der Aktionen im Use Case aus der Abbildung 1-2 festgelegt, andererseits dadurch, dass

die Nachbedingung des Use Case „Geldautomat zusammenbauen“ als Vorbedingung im Use Case „Geldautomat anschließen“ auftaucht.

Zum Schluss dieses Kapitels ist in der folgenden Abbildung die Use Case Elemente und ihre Beziehungen zueinander in Form eines UML Diagramms noch mal zusammengestellt. Zusätzlich zu den oben genannten Elementen wurde noch ein Topic Element zum Modell hinzugefügt, das dazu dient die Use Cases nach Themengebieten zu ordnen.

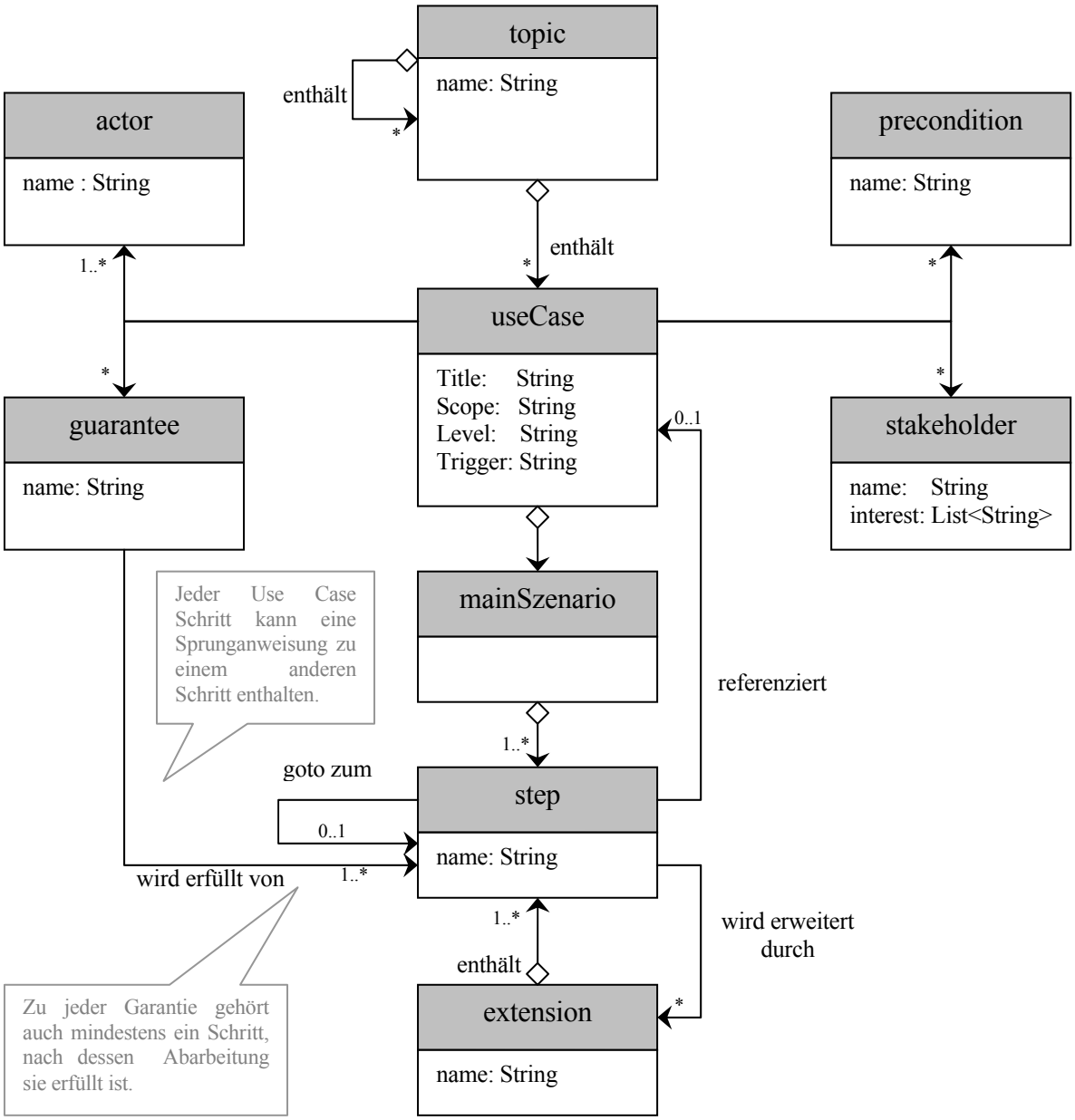


Abbildung 1-4

In diesem Abschnitt wurde das Konzept der Use Cases nach dem Buch von [AC2003] vorgestellt. Die Use Cases werden also erstens als Mittel der Kommunikation zwischen Entwicklern und Stakeholdern des zu entwickelnden Systems verwendet und zweitens erfolgt auf ihrer Grundlade die Abschätzung der Kosten, der Komplexität und Zeitpläne für das Projekt. Die Use Cases sind heutzutage gut erforscht und ihre Vorzüge sind durch Verwendung in der Praxis bestätigt. Ein Entwickler, der Use Cases zur Fixierung von Anforderungen

verwendet, kann auf eine umfangreiche Sammlung von Praktiken zur Erstellung, Verfeinerung und Analyse von Use Cases zugreifen.

Die Use Cases haben allerdings auch einen Nachteil. Da sie die Interaktionen der Benutzer mit dem System beschreiben und die Anzahl der Benutzer und ihrer möglichen Interaktionen im Allgemeinen recht hoch ist, kann es passieren, dass die Anzahl der Use Cases ebenfalls recht hoch ausfällt. Da die Use Cases in Tabellarischer- oder Textform vorliegen wird es in der Regel nicht einfach sein die Übersicht über die vielen Use Cases und ihre Beziehungen zueinander zu behalten. Zum Repertoire von UML gehören zwar Use Case Diagramme mit denen die Use Cases strukturiert werden können, eine Übersicht über den globalen Kontrollfluss kann mit diesen Diagrammen jedoch nicht gewonnen werden.

1.2 Ereignisgesteuerte Prozesskette

Ein Geschäftsprozess ist eine zusammengehörende Abfolge von Unternehmungsverrichtungen zum Zweck einer Leistungserstellung [AWScheer2001]. Die Optimierung der Geschäftsprozesse in Bezug auf Kosten, Zeit oder Komplexität erfordert, dass diese Prozesse in einer dazugeignetten Art und Weise dokumentiert werden. Das von Keller, Nüttgens und Scheer[1] vorgeschlagene EPK[2]- Konzept zur Modellierung und Dokumentation von Geschäftsprozessen ist nicht nur in der Praxis weit verbreitet, sondern auch ein populäres Forschungsobjekt auf dem Gebiet der (Wirtschafts-) Informatik.

Die wesentlichen Bestandteile des EPK- Modells sind Funktionen und Ereignisse. Ereignisse lösen Funktionen aus und sind deren Ergebnis. Diese Beziehung wird dadurch dargestellt, dass Funktionen und Ereignisse mit Kontrollflusskanten verbunden werden. Eine Funktion kann durch mehrere Ereignisse ausgelöst werden, oder mehre Ereignisse als Ergebnis liefern. Um diesen Sachverhalt darzustellen werden Verknüpfungsoperatoren verwendet, dabei wird zwischen konjunktiven, disjunktiven und adjunktiven Verknüpfungen unterschieden. Durch Prozesswegweiser können unterschiedliche EPK- Diagrammen miteinander verbunden werden. In der Abbildung 1-5 ist die Symbolik der Basiselemente des EPK- Modells zusammengestellt.

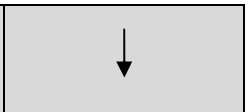
				
Ein Ereignis wird durch ein Sechseck dargestellt.	Durch ein Rechteck mit abgerundeten Ecken wird eine Funktion dargestellt	Ein Prozessweg-weiser ist eine Abkürzung für eine andere EPK	XOR- AND- OR- Operatoren. (von links nach rechts)	Kontroll-flusskante, dient dem Verbinden der links aufgeführten Elemente

Abbildung 1-5

Ein EPK- Diagramm ist also einen gerichteter, zusammenhängender Graph. Die Knoten dieses Graphen sind Ereignisse, Funktionen, Prozesswegweiser und Verknüpfungsoperatoren, die durch Kontrollflusskanten nach folgenden Regeln miteinander verbunden werden[3]:

[1] Semantische Prozessmodellierung auf der Grundlage Ereignisgesteuerter Prozessketten [KNS1992]
[2] Ereignisgesteuerte Prozesskette
[3] Entnommen aus „Geschäftsprozessmodellierung mit der Ereignisgesteuerten Prozesskette“ [ScheerThomas2005]

- Es gibt keine Kanten mit gleichen Anfangs und Endknoten. Zwischen je zwei Knoten gibt es höchstens eine Kante.
- Funktionen und Ereignisse besitzen genau eine eingehende und eine ausgehende Kante. Die Ausnahme dieser Regel sind die Start- und Endereignisse sie besitzen nur eine ausgehende oder eingehende Kante. In jeder EPK existieren mindestens ein Start und ein Endereignis.
- Prozesswegweiser besitzen entweder eine eingehende oder eine ausgehende Kante
- Verknüpfungsoperatoren haben entweder eine eingehende und mehrere ausgehende oder mehrere eingehende und eine ausgehende Kante.
- Ereignisse sind nur mit Funktionen oder Prozesswegweisern (eventuell über Verknüpfungsoperatoren) verbunden.
- Funktionen und Prozesswegweiser sind nur mit Ereignissen (eventuell über Verknüpfungsoperatoren) verbunden.
- Es gibt keinen gerichteten Kreis, der nur aus Verknüpfungsoperatoren besteht. Nach Ereignissen folgen kein XOR und kein OR- Split.

In der Abbildung 1-6 sind alle Verknüpfungsmöglichkeiten der EPK- Grundelemente (ausgenommen Prozesswegweiser) aufgeführt, die grau markierten Konstrukte sind nicht erlaubt.[1]

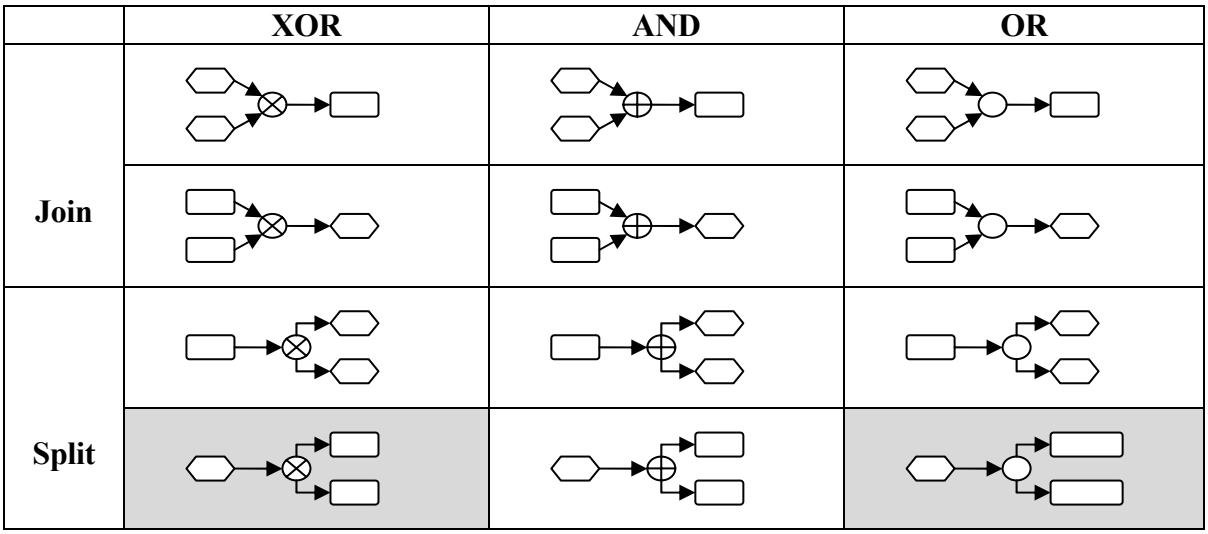
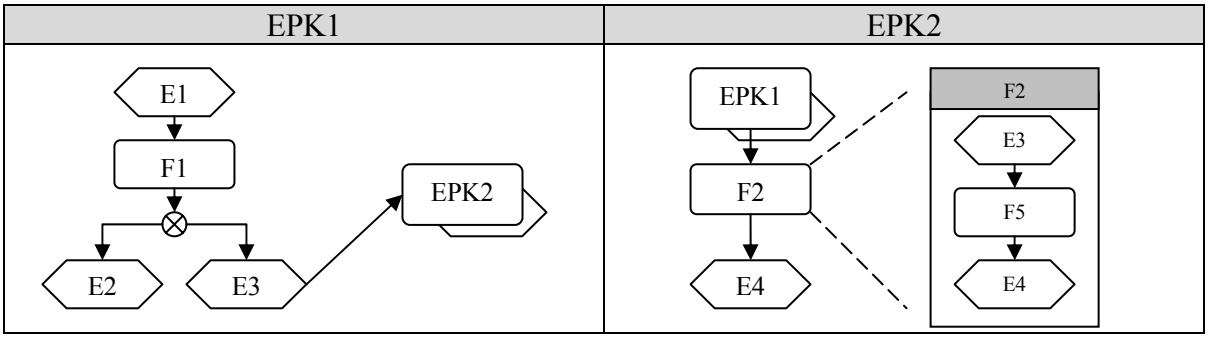


Abbildung 1-6

Die Funktionsweise der Prozesswegweiser, sowie das Konzept der Funktionsverfeinerung werden durch die untere Abbildung demonstriert



[1] Entnommen aus [KNS1992]

Das linke und das rechte Teilmodell werden durch die beiden Prozesswegweiser *EPK2* und *EPK1* verknüpft. Die Bezeichnung eines Prozesswegweisers wird so gewählt, dass er namentlich auf das entsprechende Teilmodel verweist. Zusätzlich zum Prozesswegweiser existiert noch das Konzept der Funktionsverfeinerung, eine Funktion kann damit in weitere Funktionen zerlegt werden wie rechts im Bild zu sehen ist. Die Prozesswegweiser werden verwendet um die EPKs horizontal und die hierarchische Funktionen um die EPKs vertikal zu partitionieren.

Zur Veranschaulichung ist in der Abbildung 1-7 die Anforderung an die Funktionalität eines Geldautomaten in Form einer EPK dargestellt. Dieselbe Anforderung ist in der Abbildung 1-1 auch als Use Case dargestellt.

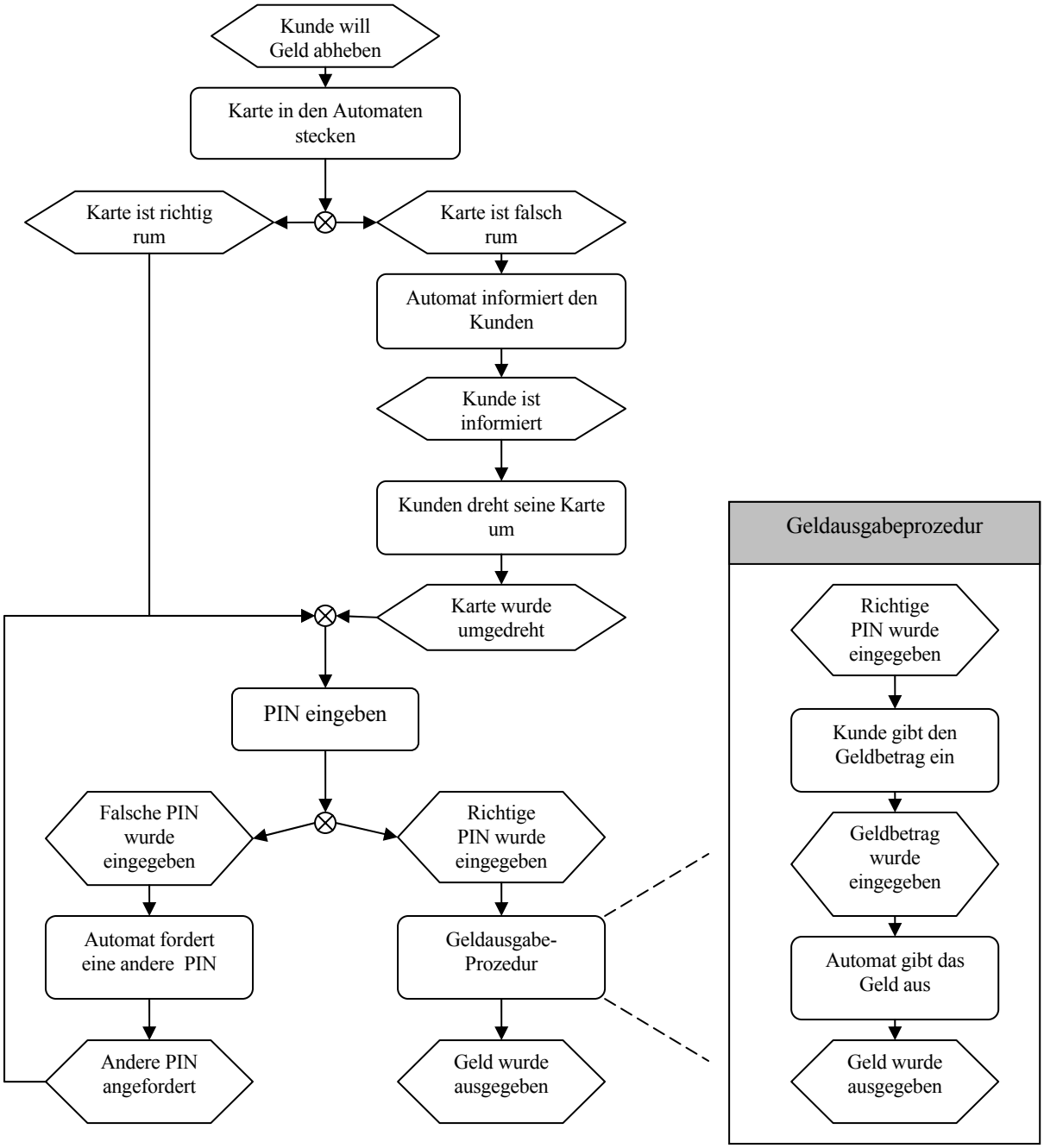


Abbildung 1-7

1.3 Zusammenfassung

In diesem Kapitel wurden die Use Cases und die EPKs vorgestellt. Die eine Methodik wird verwendet um die Anforderungen an Softwaresysteme zu dokumentieren, die andere um Geschäftsprozesse darzustellen, die Use Cases haben eine tabellarische Form, die EPKs eine graphische. Beide Konzepte haben aber auch eine Gemeinsamkeit, beide beschreiben Abläufe, im Falle der Use Cases ist es der Ablauf einer Interaktion eines Akteurs mit dem System, im Falle der EPKs kann das der Ablauf von beliebigen geschäftlichen Aktionen sein. Wie im Laufe dieses Kapitels demonstriert wurde, können beide Konzepte verwendet werden um ein und denselben Sachverhalt auf zwei unterschiedliche Weisen zu beschreiben. Es stellt sich deshalb die Frage ob es dann auch möglich ist die Use Cases ins EPK Modell und die EPKs ins Use Case Modell zu übersetzen. Sollte es tatsächlich möglich sein, so möchte man eventuell wissen inwieweit diese Übersetzung automatisiert werden kann und welche Vorteile sich daraus ergeben, alles Fragen, die im weiteren Verlauf dieser Arbeit beantwortet werden.

2. Definition der Abbildungen

In diesem Kapitel werden die Sprachkonstrukte der Use Cases und EPKs miteinander verglichen und ihre Gemeinsamkeiten herausgearbeitet, ausgehend von diesen Gemeinsamkeiten wird schrittweise dargestellt, wie Use Cases auf semantisch äquivalente EPKs und umgekehrt abgebildet werden können.

2.1 Abbildung eines Use Case auf eine EPK

Als erstes soll untersucht werden wie das Standardszenario und die dazugehörenden Erweiterungen eines Use Case ins EPK- Modell übertragen werden können. Abbildung 2-1 zeigt, wie eine Sequenz von zwei Aktionen eines Use Case auf eine EPK abgebildet wird. Die einzige, sinnvolle Entsprechung einer Aktion im Use Case- Modell ist offensichtlich eine Funktion im EPK- Modell.

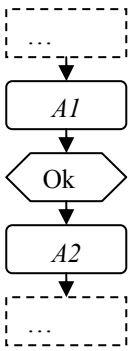
Use Case	EPK
… Aktion *A1* Aktion *A2* …	

Abbildung 2-1

Unabhängig davon ob die Aktionen *A1* und *A2* im Standardszenario oder im Erweiterungsteil des Use Case stehen, die Bedeutung ist dieselbe: nach der Aktion *A1* wird *A2* ausgeführt. Da im EPK- Modell Funktionen nicht mit Funktionen durch Kontrollflusskanten verbunden werden können, wird ein zusätzliches Ereignis (Ok- Ereignis) benötigt um das sequenzielle Ausführen von *A1* und *A2* im EPK- Modell auszudrücken. Das Ok- Ereignis signalisiert die Beendigung der Funktion *A1* und den Start der Funktion *A2* bei der EPK, beim Use Case ist dieses Ereignis nicht explizit vorhanden. Bei der Übersetzung eines Use Case in eine EPK wird also zu jeder Aktion eine EPK- Funktion sowie ein Ereignis erzeugt, das die erfolgreiche Ausführung dieser Funktion signalisiert.

Folgende Abbildung 2-2 zeigt wie eine Aktion *A1* eines Use Case mit spezifizierten Fehlerereignissen *F1* bis *Fm* durch eine EPK dargestellt wird. Die Semantik von *A1* mit ihren Fehlerereignissen sei wie folgt festgelegt:

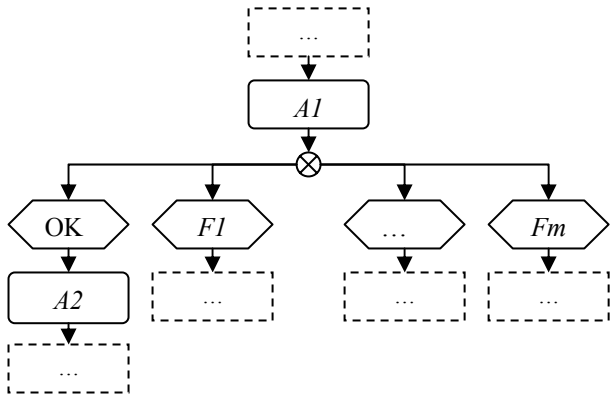
Use Case	EPK
… Aktion *A1* Fehlerereignis *F1* Fehlerbehandlung von *F1* Fehlerereignis *F2* Fehlerbehandlung von *F2* … Fehlerereignis *Fm* Fehlerbehandlung von *Fm* Aktion *A2* …	

Abbildung 2-2

Entweder geht *A1* gut, dann wird mit der Ausführung der nächsten Aktion, hier der Aktion *A2* weitergemacht oder es treten einer oder mehrere der Fehler *F1* bis *Fm* ein, in diesem Fall wird genau eine der spezifizierten Fehlerbehandlungen durchgeführt, nämlich diejenige, die weiter oben steht. (diese Festlegung entspricht der Semantik von try- catch Blöcken in Java oder C++). Egal wie viele der spezifizierten Fehlerereignisse aufgetreten sind, es wird also immer nur die Fehlerbehandlung eines einzelnen Fehlerfalls durchgeführt. Das Eintreffen mehrerer Fehlerfälle mit anschließender Ausführung der Fehlersequenz eines Fehlerfalls kann also auch als das Eintreffen nur dieses Fehlerfalls betrachtet werden, ohne das dabei relevante Information verloren geht. Es wird also davon ausgegangen, dass höchstens einer der spezifizierten Fehlerereignisse einer Use Case Aktion auftreten kann. Die Ausdrucksmöglichkeiten der Use Cases werden durch diese Festlegung keinesfalls eingeschränkt, soll auf das Eintreffen mehrerer Fehlerereignisse reagiert werden können, so können diese Ereignisse durch eine oder-Verknüpfung in Form eines Ereignisses dargestellt werden:

```
…
Aktion A1
        Fehlerereignis F1 oder F2 oder …
            Fehlerbehandlung …
…
```

Nach dem nun die Semantik festgelegt ist kann der in der Abbildung 2-2 links stehende Use Case in Form einer EPK formuliert werden, das Ergebnis ist rechts in derselben Abbildung dargestellt. Das Äquivalent eines Fehlerereignisses im Use Case ist ein Ereignis der EPK. Falls im Use Case zu einer Aktion *A1* Fehlerereignisse angegeben sind, dann wird also entweder nach der Ausführung von *A1* eins dieser Ereignisse eintreffen oder nicht. Dieser Sachverhalt wird in einer EPK dadurch ausgedrückt, dass die Funktion *A1* durch den XOR- Konnektor sowohl mit ihren Fehlerereignissen als auch einem Ereignis (OK- Ereignis), welches das Nichteintreffen jeglichen Fehlers repräsentiert, verbunden wird.

Als nächstes wird dargestellt, wie Sprünge im Kontrollfluss eines Use Case in einer EPK modelliert werden. In der Abbildung 2-3 ist ein Teil eines Use Case mit einer Sprunganweisung dargestellt: in Folge eines Fehlers während der Aktion *A1*, wird also die Aktion *A1.1* durchgeführt, danach soll es mit der Abarbeitung der Aktion *A2* weitergehen. Derselbe Sachverhalt ist auch durch die EPK in der unteren Abbildung rechts dargestellt.

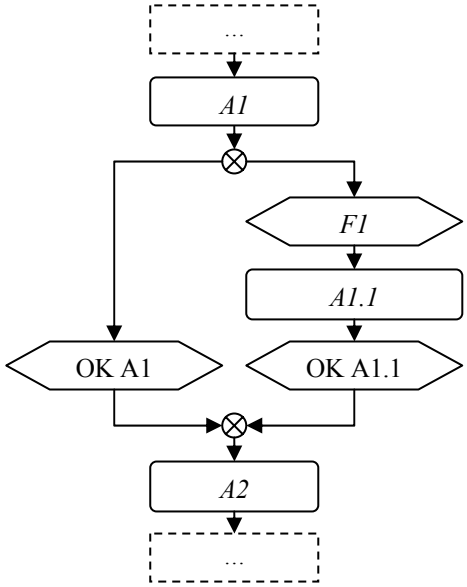
Use Case	EPK
… Aktion *A1* Fehler *F1* Aktion *A1.1* Weiter mit Aktion *A2* Aktion *A2* …	

Abbildung 2-3

Eine Sprunganweisung von einem Ausführungspfad zu einem anderen im Use Case- Modell entspricht daher dem Join dieser beiden Pfade im EPK- Modell. Da die Pfade eines Use Case nicht parallel ausgeführt werden (festgelegt auf Seite 11) und sich deshalb gegenseitig ausschließen, werden die entsprechenden EPK Pfade mit dem XOR- Konnektor getrennt, die Zusammenführung muss deshalb ebenfalls mit dem XOR- Konnektor gemacht werden.

Bis jetzt wurde gezeigt, wie Aktionen mit ihren Fehlerereignissen und Sprunganweisungen im EPK- Modell dargestellt werden können. Im Folgenden wird die Übersetzung der Trigger sowie die der Vor- und Nachbedingungen ins EPK- Modell behandelt.

Bevor der erste Schritt eines Use Case ausgeführt wird, müssen die spezifizierten Vorbedingungen zutreffen sowie der Trigger, der im Grunde nichts anderes ist als eine weitere Vorbedingung. Es liegt daher nahe den Startereignis (oder -Ereignisse) einer EPK aus dem Trigger und Vorbedingungen des Use Case zusammenzustellen. In der unteren Abbildung ist rechts ein Use Case mit den Vorbedingungen *b1* bis *bn* und dem Trigger *T* dargestellt, im rechten Teil der Abbildung ist die dazu semantisch äquivalente EPK zusehen.

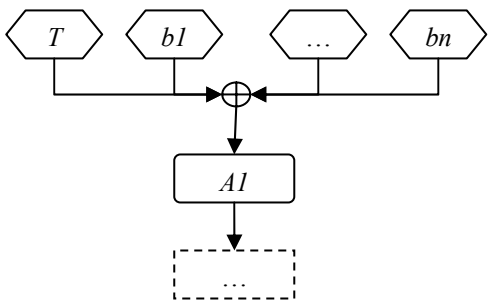
Use Case	EPK
… Vorbedingungen: *1.* b_1 … *2.* b_n Trigger: *T* Standardszenario Aktion *A1* …	

Abbildung 2-4

Die Funktion *A1* wird erst dann ausgeführt, wenn die Ereignisse *T* und *b1* bis *bn* eingetroffen sind. Falls kein Trigger im Use Case angegeben ist so löst die Konjunktion der Vorbedingungen den Start der ersten Aktion eines Use Case, sollten keine Vorbedingungen angegeben sein, so kann das Startereignis nur durch den Trigger beschrieben werden, falls weder der Trigger noch Vorbedingungen angegeben sind dann wird ein Ereignis mit dem Namen „Start“ erzeugt welches als Startereignis der übersetzten EPK dient. (jede EPK muss mindestens ein Startereignis aufweisen).

Die Übersetzung der Nachbedingungen (Successguarantee) erfolgt ganz analog, falls durch ein Use Case Schritt eine Nachbedingung erfüllt wird, so wird zu dieser Nachbedingung ein namensgleiches Ereignis erzeugt und mit der diesem Schritt entsprechenden Funktion durch eine Kontrollflusskante verbunden, wie die folgende Graphik demonstriert.

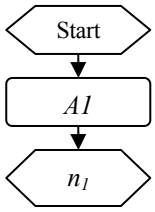
Use Case	EPK
… Nachbedingungen: 1. n_1 erfüllt durch *A1* Standardszenario Aktion *A1*	

Abbildung 2-5

Sollte eine Nachbedingung durch mehrere Schritte erfüllt werden, so werden die EPK Funktionen dieser Schritte in einem XOR- Join zusammengefasst und eine Kontrollflusskante von dem XOR Konnektor zu dem dieser Nachbedingung entsprechenden Ereignis erstellt.

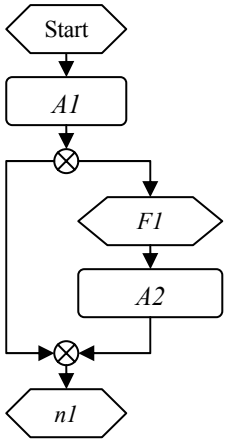
Use Case	EPK
… Nachbedingungen: 1. n_1 erfüllt durch *A1*, *A2* Standardszenario 1. Aktion *A1* Erweiterungen 1. *F1* 1.1 Aktion *A2*	

Abbildung 2-6

Die Nachbedingung *n1* im Use Case der oberen Abbildung wird sowohl von der Aktion *A1* als auch von der Aktion *A2* erfüllt, in der äquivalenten EPK rechts sind deshalb die Funktionen *A1* und *A2* mit dem Ereingis *n1* über einen XOR- Join zusammengeführt

Falls ein Use Case Schritt mehrere Nachbedingungen erfüllt, so wird dieser Zusammenhang mit einem AND Konnektor im EPK Modell nachgebildet.

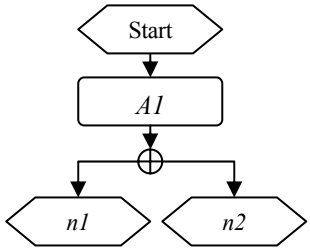
Use Case	EPK
… Nachbedingungen: 1. n_1 erfüllt durch *A1* 2. n_2 erfüllt durch *A1* Standardszenario Aktion *A1*	

Abbildung 2-7

Wie in Abschnitt 1 dargestellt, können die Nachbedingungen mancher Use Cases in anderen Use Cases als Vorbedingungen wieder auftauchen, um diesen Zusammenhang im EPK- Modell graphisch darzustellen werden Prozesswegweiser verwendet. Zur Demonstration seien die folgenden 2 Use Cases gegeben.

Titel	Anmelden
…	…
Akteur	Kunde
Nach-Bedingungen	Kunde ist angemeldet
Szenario	1. Usernamen eingeben 2. …

Titel	Bestellung aufgeben
Trigger	Kunde will bestellen
Vor-Bedingungen	Kunde ist angemeldet
Akteur	Kunde
Szenario	1. Produkt wählen 2. …

Abbildung 2-8

Ein Kunde kann eine Bestellung nur dann aufgeben, wenn er sich zuvor im System angemeldet hat. Die Nachbedingung „Kunde ist angemeldet“ des linken Use Case ist Vorbedingung des rechten, diese beiden Use Cases werden deshalb so in eine einzige EPK übertragen, wie in der unteren Graphik verdeutlicht.

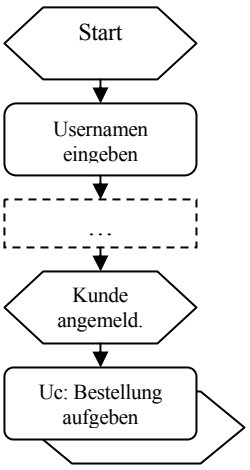
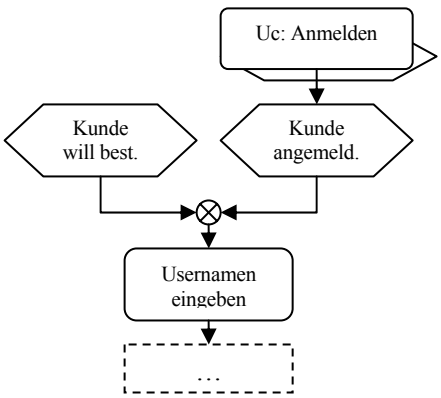
EPK anmelden	EPK Bestellung aufgeben
	

Abbildung 2-9

Falls also eine Nachbedingung *n* eines Use Cases *u1* in einem anderen (und nur in einem) Use Case *u2* als Vorbedingung oder Trigger wieder auftaucht, wird das Ereignis welches mit dieser Bedingung assoziiert ist von den zu diesen Use Cases gehörenden EPKs geteilt. Zusätzlich wird in der EPK von *u1* ein Prozesswegweiser erzeugt der auf die EPK von *u2* verweist, und in der EPK von *u2* wird ein Prozesswegweiser erzeugt, der auf die EPK von *u1* verweist. Die beiden Prozesswegweiser werden jeweils in den entsprechenden EPKs mit *n1* durch eine Kontrollflusskante verbunden (siehe Abbildung 2-9).

Falls eine Nachbedingung *n* eines Use Case v_1 in mehreren Use Cases $u_1 - u_n$ als Vorbedingung oder Trigger vorhanden ist, so werden zu jedem Use Case $u_1 - u_n$ je ein Prozesswegweiser erzeugt und in der EPK von v_1 mit n_1 durch einen AND Konnektor verknüpft. In jeder EPK zu $u_1 - u_n$ wird ein Prozesswegweiser erzeugt der auf die EPK von v_1 verweist und der mit *n* durch eine Kontrollflusskante entsprechend der Abbildung 2-10 verknüpft ist.

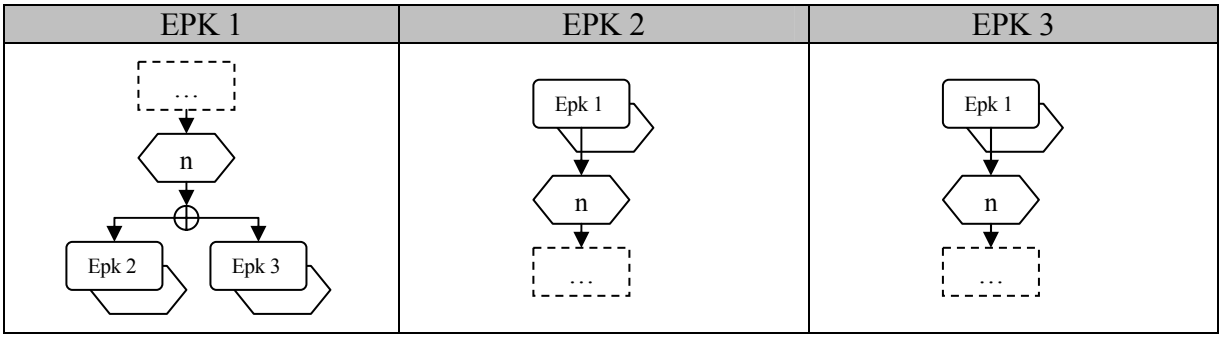


Abbildung 2-10

Falls eine Vorbedingung oder Trigger *n* eines Use Case *v* in mehreren Use Cases $u_1 - u_n$ als Nachbedingung vorhanden ist, so wird zu jedem Use Case $u_1 - u_n$ ein Prozesswegweiser erzeugt und in der EPK von *v* durch einen ODER- Join mit dem Ereignis von *n* verknüpft. In den EPKs von $u_1 - u_n$ wird jeweils der Prozesswegweiser zu *v* mit *n* durch eine Kontrollflusskante verbunden. (vergleiche die Abbildung 2-11)

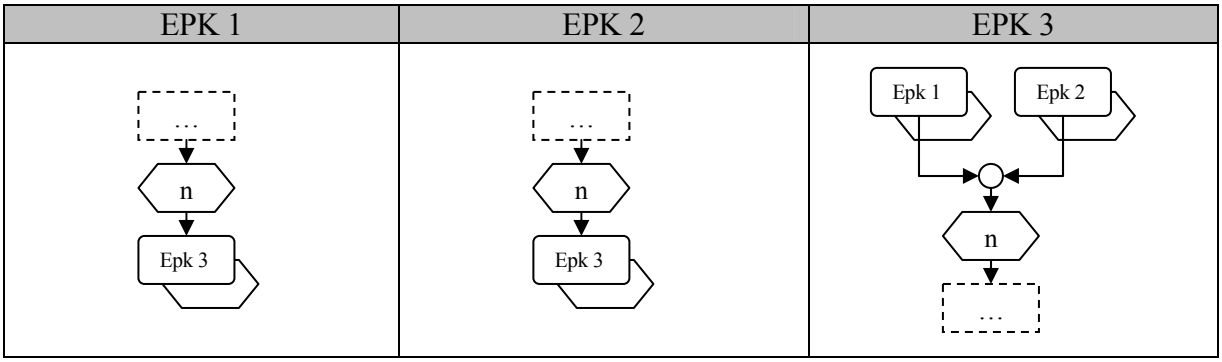


Abbildung 2-11

In diesem Abschnitt wurde ein Konzept vorgestellt, wie Use Cases auf EPKs abgebildet werden können, dabei wurde festgestellt, dass folgende Use Case Elemente eine Entsprechung im EPK Modell besitzen:

1. Trigger und Vorbedingungen entsprechen Startereignissen einer EPK, die mit einem AND Konnektor gejoint werden, dieser AND Konnektor wird mit dem ersten Schritt

der Standardszenarios der als eine Funktion ins EPK Modell übernommen wird, verbunden.

2. Die Schritte der Use Cases entsprechen EPK- Funktionen, die durch „OK“ Events mit ihren Nachfolgern verbunden werden.

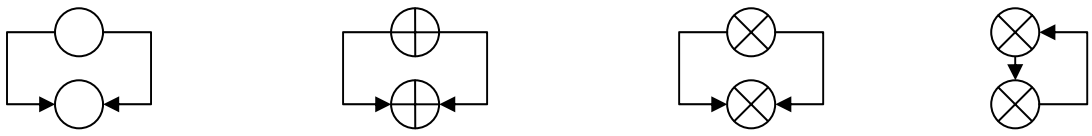
3. Die Erweiterungen eines Use Case Schritts entsprechen inneren Ereignissen, die über den XOR Konnektor mit der vorhergehenden Funktion, verbunden werden.

4. Nachbedingungen entsprechen End Ereignissen, die durch einen AND Split mit der Funktion, von der sie erfüllt werden, verbunden werden.

5. Jeder Use Cases entspricht einer EPK. Falls die Use Cases gemeinsame Vor- bzw. Nachbedingung haben, werden die entsprechenden EPKs über Prozesswegweiser verknüpft.

2.2 Abbildung einer EPK auf ein Use Case

Bevor die Übersetzung des EPK- Modells auf das Use Case- Modell diskutiert wird, sollen im Folgenden einige Stilregeln der Modellierung mit den EPKs vorgestellt werden. Die EPKs wurden ursprünglich eingeführt und benutzt ohne ihre Semantik formal zu definieren. Später als die Computersimulation der EPKs eine genaue Semantikdefinition erforderte, stieß man auf einige Anomalien des ursprünglichen Modells, die durch die Definition der wohlstrukturierten EPKs behoben wurden[1].



a) OR- Konstrukt

b) AND Konstrukt, parallele Ausführung

c) XOR- Konstrukt, Alternative

d) Iteration

Abbildung 2-12

Das Hauptmerkmal einer wohlstrukturierten EPK ist die Tatsache, dass zu jedem Join-Konnektor ein Split- Konnektor des gleichen Typs gehört. Die in Abbildung 2-12 gezeigten EPKs sind wohlstrukturiert, die Funktionen und Ereignisse wurden weg abstrahiert, sie werden weiterhin genau so verwendet, wie in Abschnitt 1.2 beschrieben. Wird in eine Kante einer wohlstrukturierten EPK eine wohlstrukturierte EPK eingesetzt, so ist das Ergebnis wieder eine wohlstrukturierte EPK, bei den Konstrukten a) bis c) der Abbildung 2-12 sind auch mehr als zwei Kanten möglich. Bei der in dieser Arbeit diskutierten Abbildung der EPKs auf Use Cases werden nur wohlstrukturierte EPKs betrachtet.

Folgende Abbildung 2-13 zeigt die Übersetzung einer Sequenz von zwei Funktionen einer EPK in die entsprechenden Elemente eines Use Case. Wie schon früher festgestellt entspricht eine EPK- Funktion einem Use Case Schritt. Falls zwei Funktionen nur durch ein Ereignis

[1] Die wohlstrukturierten EPKs wurden in [GruhnLaue] definiert, auch einige Anomalien des ursprünglichen EPK-Modells sind in diesem Dokument beschrieben.

miteinander verbunden sind, wie in der Abbildung 2-13 mit *A1* und *A2* der Fall ist, dann wird dieses Ereignis bei der Übersetzung auf ein Use Case weggelassen.

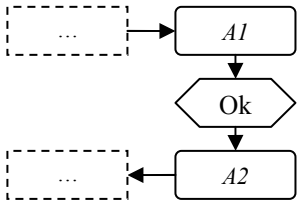
EPK	Use Case
	… *A1* *A2* …

Abbildung 2-13

In Abschnitt 2.1 wurde festgestellt, dass eine Aktion mit angegebenen Fehlerereignissen im Use Case- Modell einem XOR- Split im EPK- Modell entspricht, umgekehrt verhält es sich natürlich genau so.

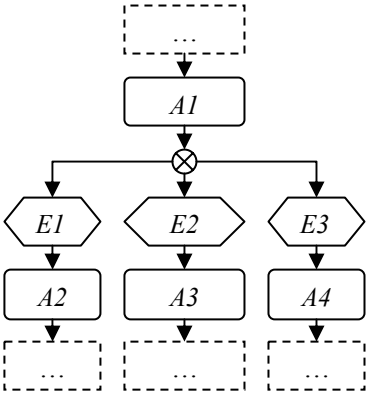
EPK	Use Case a
	… Aktion *A1* Erweiterung *E2*: Aktion *A3* … Erweiterung *E3*: Aktion *A4* … Aktion *A2* …

Abbildung 2-14

Allerdings enthalten die EPK Diagramme keine Information darüber, welche Pfade einer XOR Verzweigung Fehlerpfade sind, und welcher Pfad kein Fehlerpfad ist. Bei der Betrachtung der EPK aus der Abbildung 2-14 ist also nicht klar, dass die Pfade, die mit *E2* und *E3* beginnen Fehlerpfade sind und deshalb als Erweiterungen der Funktion *A1* ins Use Case Modell übertragen werden sollen, wie es z.B. beim Use Case a geschehen ist. Die Use Cases b und c stellen zwei weitere Möglichkeiten dar die oben dargestellte EPK zu übersetzen.

Use Case b	Use Case c
… Aktion *A1* Erweiterung *E1*: Aktion *A2* … Erweiterung *E3*: Aktion *A4* … Aktion *A3* …	… Aktion *A1* Erweiterung *E1*: Aktion *A2* … Erweiterung *E2*: Aktion *A3* … Aktion *A4* …

Abbildung 2-15

Umgekehrt bei der Abbildung der drei Use Cases a-c ist klar, dass in jedem Fall die EPK aus Abbildung 2-14 rauskommt. Bei einer Abbildung eines Use Case auf ein EPK- Diagramm geht also gewisse Information verloren, die bei der umgekehrten Abbildung wieder hinzugefügt werden muss (es handelt sich um die Information welche XOR Split Pfade als Erweiterungen übersetz werden sollen). Die Ausdrucksmöglichkeiten von Use Cases und EPKs sind deshalb nicht äquivalent, und eine vollautomatische Übersetzung deshalb nicht möglich.

Als nächstes soll die Übersetzung eines XOR- Joins vorgestellt werden, dazu betrachte man die EPK der folgenden Abbildung. Seien dabei die mit *E2* und *E3* beginnenden Pfade die Fehlerpfade, die als Erweiterungen übersetzt werden. Im Abschnitt 2.1 wurde schon festgestellt, dass eine Sprunganweisung innerhalb eines Use Case einem XOR- Join entspricht, deshalb ist es nahe liegend XOR- Joins als Sprunganweisungen im Use Case- Modell zu interpretieren. In der folgenden Abbildung sind zwei Übersetzungsmöglichkeiten in Form der beiden Use Cases a-b angegeben.

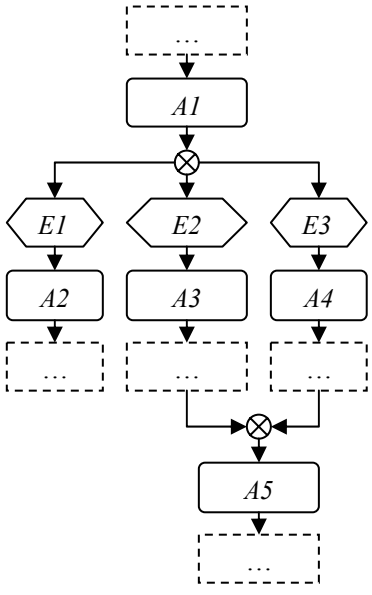
EPK	Use Case a	Use Case b
	… Aktion *A1* Erweiterung *E2*: Aktion *A3* … Aktion *A5* … Erweiterung *E3*: Aktion *A4* … weiter mit *A5* Aktion *A2* …	… Aktion *A1* Erweiterung *E2*: Aktion *A3* … weiter mit *A5* Erweiterung *E3*: Aktion *A4* … Aktion *A5* … Aktion *A2* …

Abbildung 2-16

Auch beim XOR Join bestehen also mehrere Übersetzungsmöglichkeiten, so kann z.B. der mit *A5* beginnende Pfad als Fortsetzung des mit *E2* beginnenden Pfades betrachtet werden und der Pfad, der mit *E3* beginnt, enthält dann in der Übersetzung eine Sprunganweisung zu *A5* (Use Case a). Die Alternative ist den mit *A5* beginnenden Pfad als die Fortsetzung des mit *E3* beginnenden Pfades zu betrachten und in den mit *E2* beginnendem Pfad eine entsprechende Sprunganweisung einzufügen (Use Case b).

Sowohl bei einem XOR- Split als auch einem XOR- Join müssen für eine eindeutige Übersetzung zusätzliche Informationen durch den Nutzer angegeben werden. Bei einem XOR Split muss angegeben werden, welche Pfade als Erweiterungen übersetzt werden, bei einem XOR Join muss angegeben werden, in welche Use Case Erweiterungen Sprunganweisungen eingefügt werden sollen.

Im EPK- Modell sind zwei Typen von XOR- Joins zugelassen, mehrere Ereignisse können über einen XOR- Konnektor mit einer Funktion verbunden werden oder mehrere Funktionen können über einen XOR- Konnektor in einem Ereignis verbunden werden. In der Abbildung 2-17 sind diese beiden XOR- Joins dargestellt.

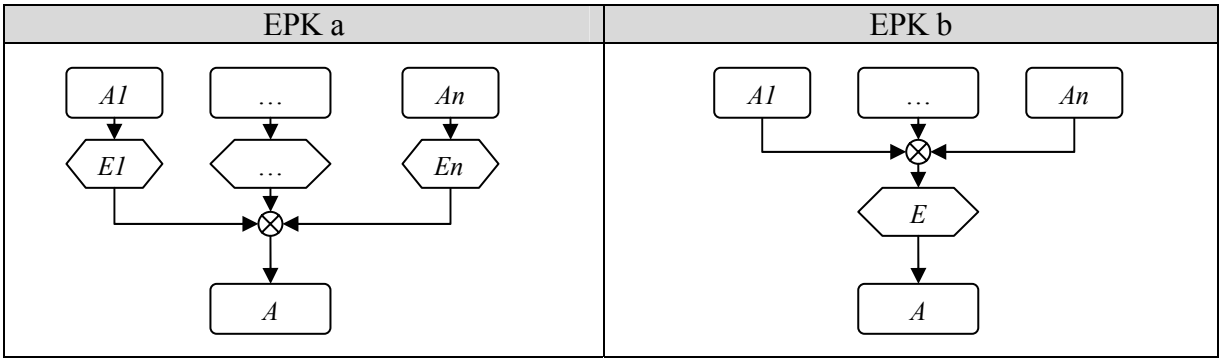


Abbildung 2-17

Die beiden EPKs aus der oberen Abbildung sind zwar syntaktisch unterschiedlich, semantisch stellen sie jedoch dengleichen Kontrollfluss dar: Der Ausführung einer der Funktionen *A1* bis *An* folgt die Ausführung von *A*. Da sowohl die Beutung der EPK *a* als auch der EPK *b* dieselbe ist, liegt es auf der Hand diese beiden EPKs auf die gleiche Weise auf Use Cases abzubilden. Die Abbildung von Ereignis XOR- Joins und Funktionen XOR- Joins erfolgt also nach dem Schema aus Abbildung 2-16.

Bis jetzt wurden alle Vorschriften angegeben, die notwendig sind um EPKs, die nur XOR-Konnektoren enthalten auf Use Cases abzubilden. Folgende Abbildung gibt einen Überblick über die EPK- Konstrukte die noch nicht ins Use Case- Modell übersetzt werden können. Für hellgrau markierten Konstrukte wurden bereits Abbildungsvorschriften angegeben, dunkelgrau markierten sind im EPK- Modell nicht erlaubt, es bleiben also noch die unmarkierten EPK-Konstrukte, die im Folgenden betrachtet werden sollen.

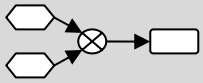
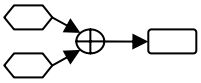
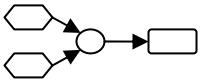
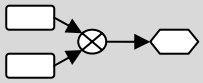
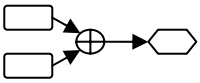
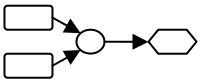
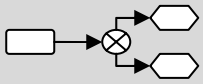
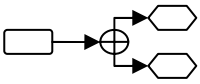
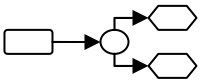
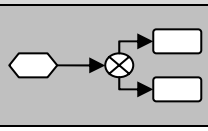
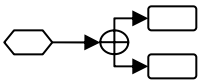
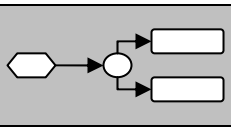
	XOR	UND	ODER
Join			
			
Split			
			

Abbildung 2-18

In Abschnitt 2.1 wurde festgestellt, dass für die vollständige Abbildung der Semantik, die allein durch die Form und nicht durch den Inhalt der Use Cases gegeben ist, weder AND- noch OR Konnektoren (abgesehen von den Vor- und Nachbedingungen, die aber nur Start oder Endereignissen entsprechen) des EPK- Modells benötigt werden. Abbildung 2-19 veranschaulicht diesen Sachverhalt.

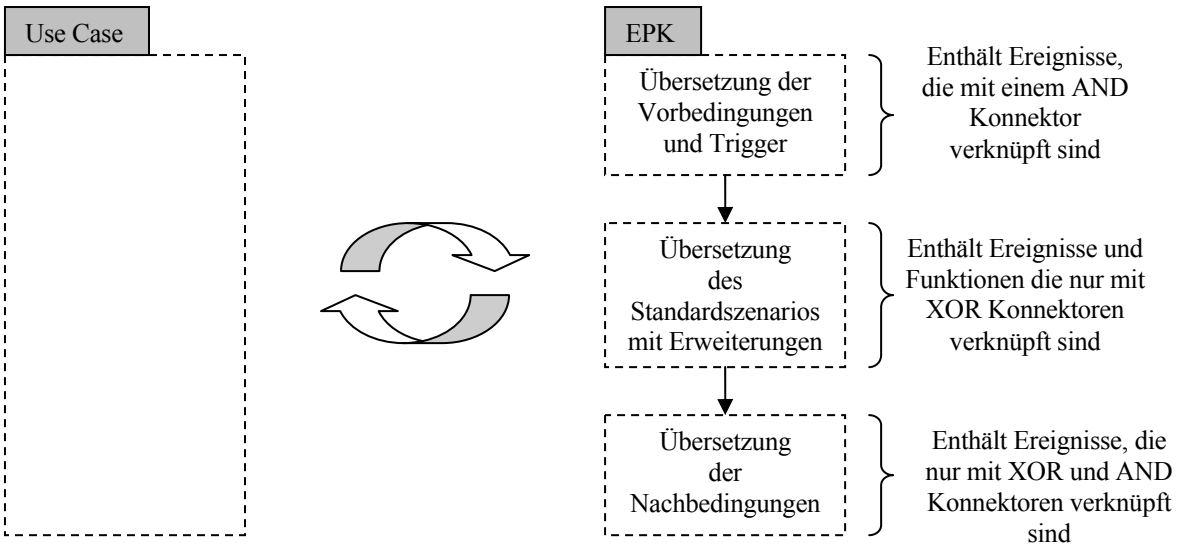


Abbildung 2-19

Nach dem momentanen Stand können also nur diejenigen EPKs abgebildet werden, die nach dem Schema aus der oberen Abbildung aufgebaut sind (das sind solche EPKs, die keine AND und OR Konnektoren in Ihrem Inneren aufweisen). Die durch die Form der Use Cases gegebene Struktur reicht also nicht aus um AND- und OR Konstrukte der EPKs beschreiben zu können, folglich müssen diese Konstrukte durch den Inhalt und nicht durch die Form der Use Cases ausgedrückt werden.

Bis jetzt wurde ein Konzept vorgeschlagen mit dem jeder Use Case in eine EPK transformiert werden kann, aber nur bestimmte EPKs in Use Cases überführt werden können. Um alle EPKs abbilden zu können reicht es deshalb aus, eine Methode anzugeben, eine EPK so zu

transformieren, dass sie nur XOR- Konnektoren enthält (und ihre ursprüngliche Bedeutung nicht verliert). Im restlichen Teil dieses Abschnitts wird so eine Methode vorgestellt werden.

Die Kernidee besteht darin die UND- und ODER Konnektoren aus den EPKs in die Bezeichnungen der Ereignisse und Funktionen zu verschieben, wie es in der folgenden Abbildung 2-20 beispielhaft mit einem UND- Operator durchgeführt wurde.

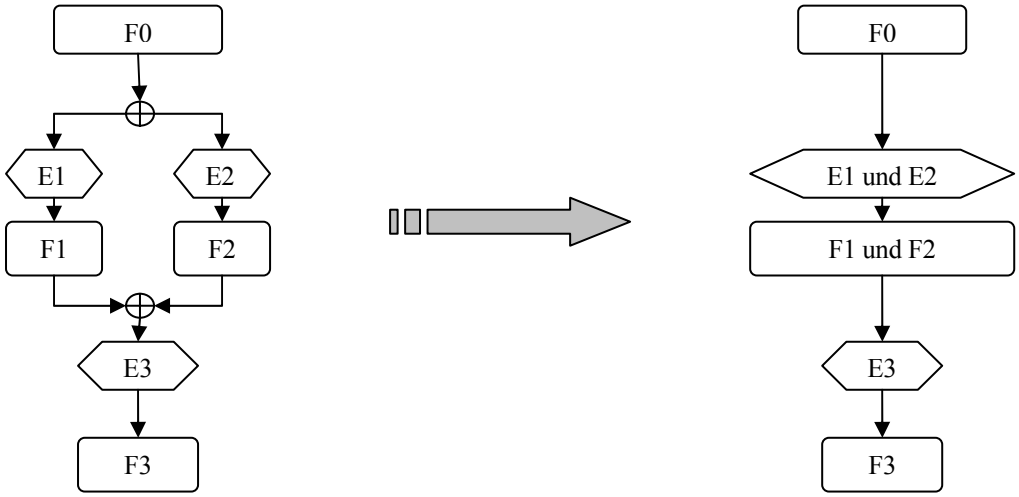


Abbildung 2-20

Die Verschiebung der Operatoren kann als Anwendung bestimmter Operationen auf die Äste einer EPK angesehen werden. Da die AND- und OR Konnektoren aus den EPKs entfernt werden sollen, werden dementsprechend AND- und OR Operationen auf den Ästen einer EPK definiert, dabei reicht es aus diejenigen EPK Elemente zu betrachteten, die:

1. nach einem AND und OR Split sich gegenüber stehen können
2. vor einem AND und OR Join sich gegenüber stehen können

Die Tabelle aus Abbildung 2-21 enthält diejenigen Operationen, die notwendig sind um AND Splits auflösen zu können.

Operand 1	Operand 2	⊕
E1 → *K1*	*E2* → *K2*	*E1 & E2* → *K1 ⊕ K2*
E1 → *K1*	*E2* → *K2*	*E1 & E2* → *K1 ⊕ K2*

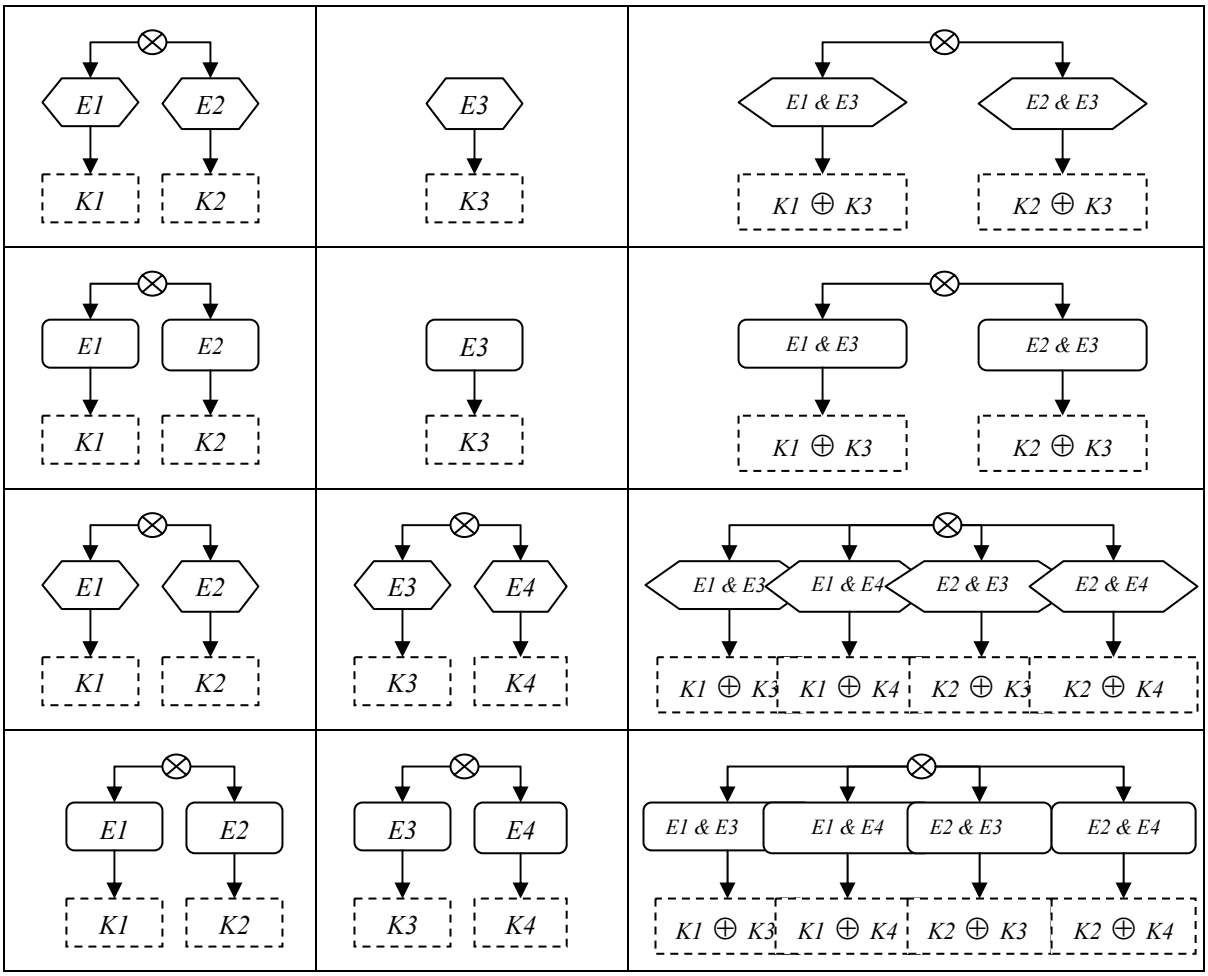


Abbildung 2-21

Das Ergebnis der Anwendung einer UND- Operation auf zwei jeweils mit dem Ereignis *E1* beziehungsweise *E2* beginnenden EPK- Äste, ist ein EPK- Ast der mit dem Ereignis *E1 & E2* beginnt, und als Rest die Anwendung der UND- Operation auf die Reste der beiden anderen Äste hat (Erste Zeile der oberen Tabelle).

Zur Veranschaulichung ist am Ende dieses Kapitels eine Transformation mit den angegebenen Operationen am einen konkreten Beispiel durchgeführt worden.

Die Grundlage der in der oberen Tabelle und in den folgenden Tabellen zusammengestellten Operationen bilden die Distributivgesetze der Prädikatenlogik:

- $(E1 \otimes E2) \wedge E3 \Leftrightarrow (E1 \wedge E3) \otimes (E2 \wedge E3)$
  [Zeilen 3 und 4 der Tabelle aus Abbildung 2-21]
- $(E1 \otimes E2) \wedge (E3 \otimes E4) \Leftrightarrow (E1 \wedge E3) \otimes (E1 \wedge E4) \otimes (E2 \wedge E3) \otimes (E2 \wedge E4)$
  [Zeilen 5 und 6 der Tabelle aus Abbildung 2-21]

Diejenigen Operationen, die notwendig sind um AND Joins auflösen zu können, sind in der unteren Tabelle zusammengestellt.

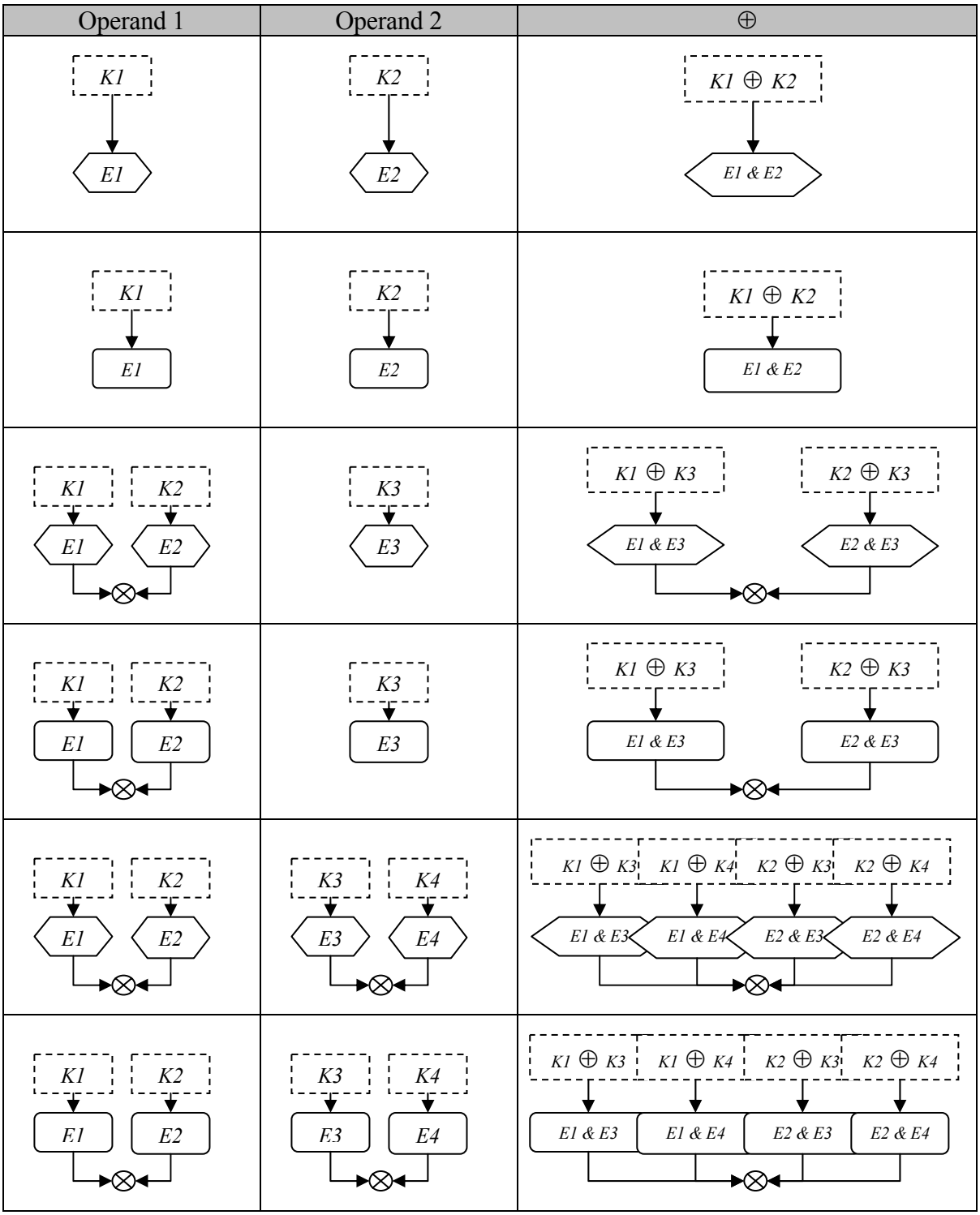


Abbildung 2-22

Folgende Tabelle enthält Operationen für die Auflösung der OR Splits. Die Anzahl der Operationen ist halb so groß, wie in der letzten Tabelle. Das liegt daran, dass OR Splits nur nach Funktionen auftreten dürfen, AND Splits dürfen dagegen sowohl nach Ereignissen als auch nach Funktionen auftreten, d.h. ein EPK Ast nach einem AND Split kann sowohl mit einem Ereignis als auch mit einer Funktion beginnen, ein EPK Ast nach einem OR Split beginnt immer nur mit einem Ereignis, wodurch die Anzahl der möglichen Operationen gegenüber einem AND Split um die hälfte reduziert wird.

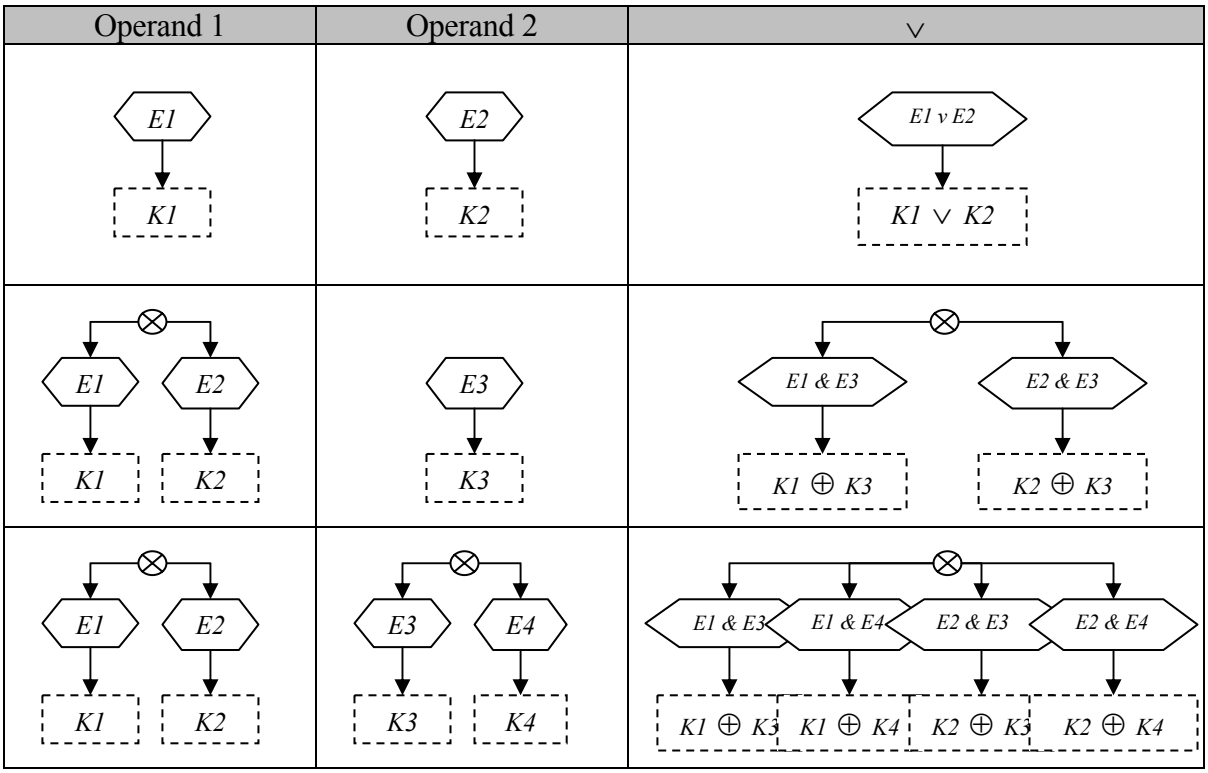
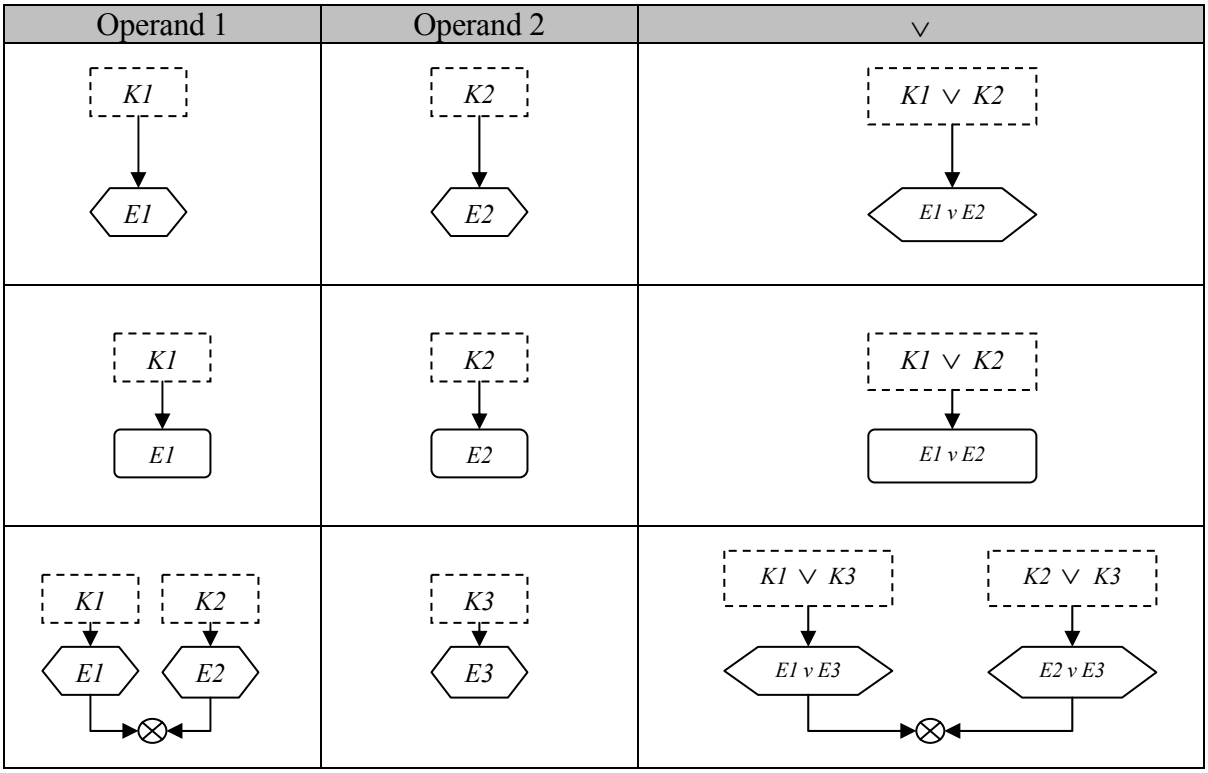


Abbildung 2-23

In der nächsten Tabelle sind die Operationen zusammengestellt, die notwendig sind um OR Joins auflösen zu können.



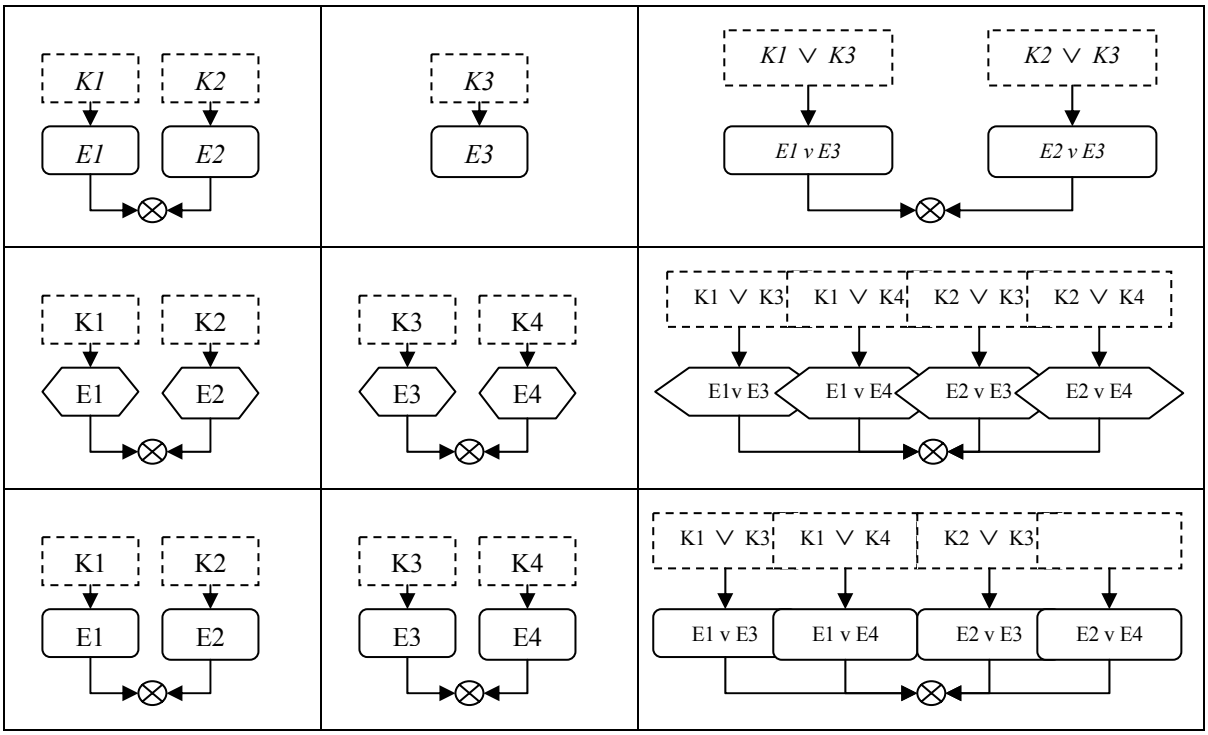


Abbildung 2-24

Die Entfernung der UND- und ODER Konnektoren aus den EPKs erfolgt nun dadurch, dass die mit diesen Konnektoren verbundene EPK- Äste mit den oben definierten Operationen ausgewertet und anschließend durch das Ergebnis dieser Auswertung ersetzt werden, wie in der folgenden Abbildung demonstriert wird.

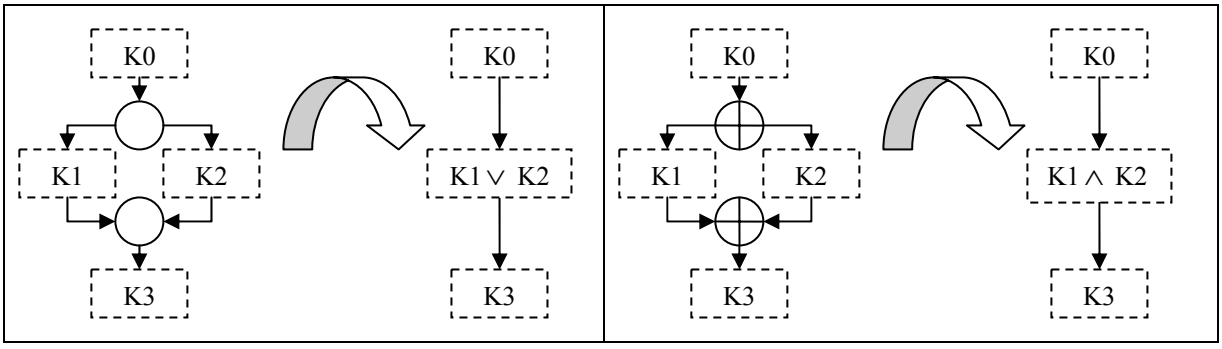


Abbildung 2-25

Falls durch einen Konnektor mehr als zwei Äste verbunden sind, erfolgt die Auswertung dadurch, dass zunächst zwei Äste ausgewertet werden, dann das Ergebnis dieser Auswertung mit dem nächsten Ast ausgewertet wird u.s.w. Damit zwei EPK- Äste mit den oben angegebenen Operationen ausgewertet werden können, müssen sie gleichlang sein, sollte jedoch ein Ast kleiner sein als der andere, so kann er mit „leeren“ Funktionen und Ereignissen auf die passende Größe aufgefühlt werden, wie in folgender Abbildung dargestellt.

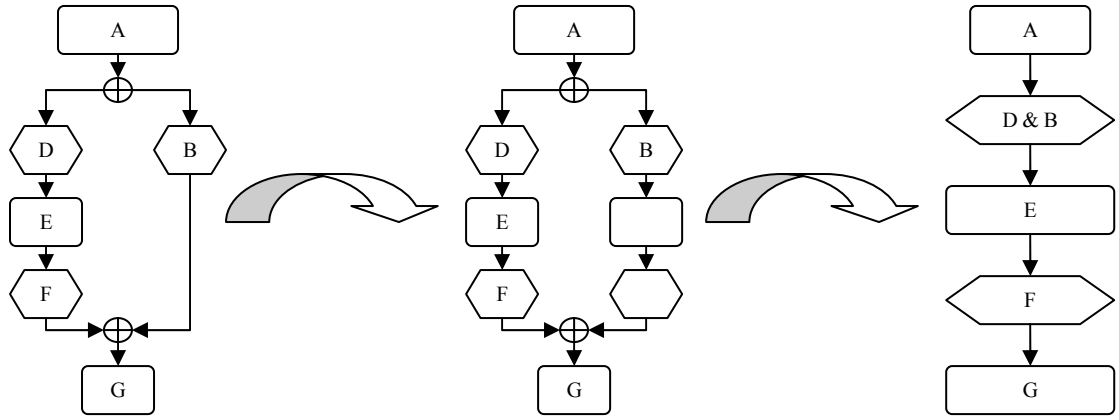


Abbildung 2-26

Im restlichen Teil dieses Kapitels wird die Auswertung einer AND Operation anhand eines konkreten Beispiels mit den aufgestellten Transformationsregeln durchgeführt. Sei die folgende (sehr vereinfachte) EPK zur Bestellungsabwicklung gegeben.

Textuelle Notation:

> *Nach dem Eintreffen einer Kundenbestellung, wird die Ware verpackt und der entsprechende Geldbetrag vom Konto des Kunden abgebucht, sobald das geschehen ist wird die Ware versendet. Falls das Konto des Bestellers nicht genug Geld aufweist, wird die Bestellung zurückgewiesen.*

Graphische Notation:

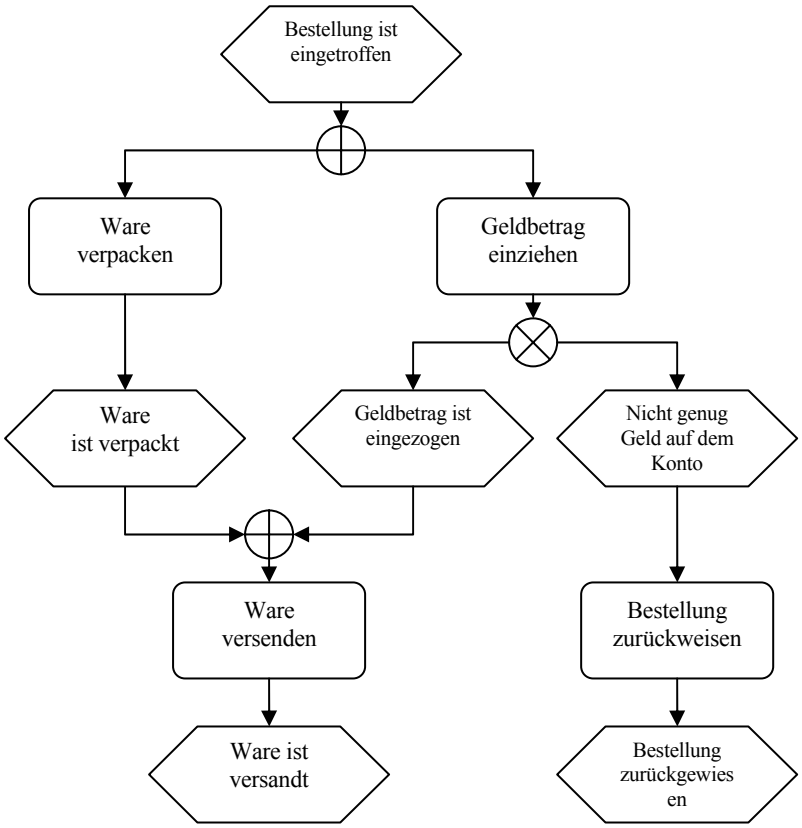
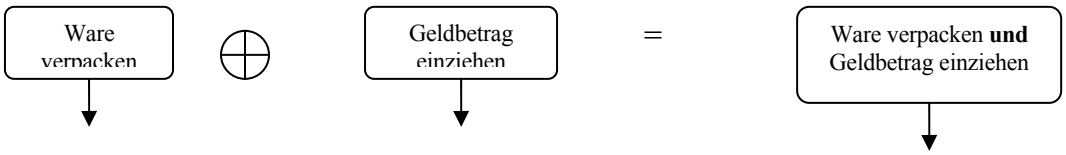
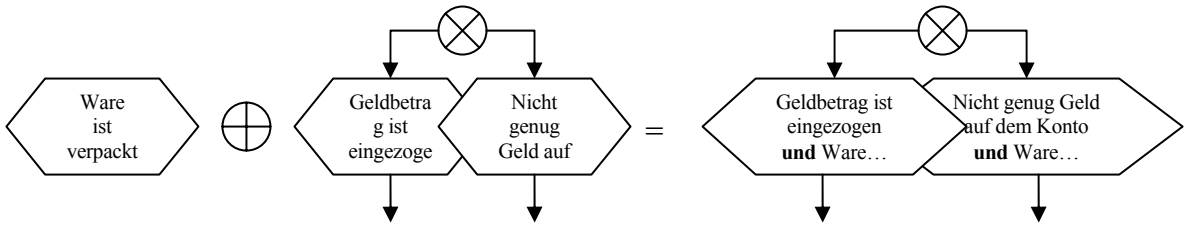


Abbildung 2-27

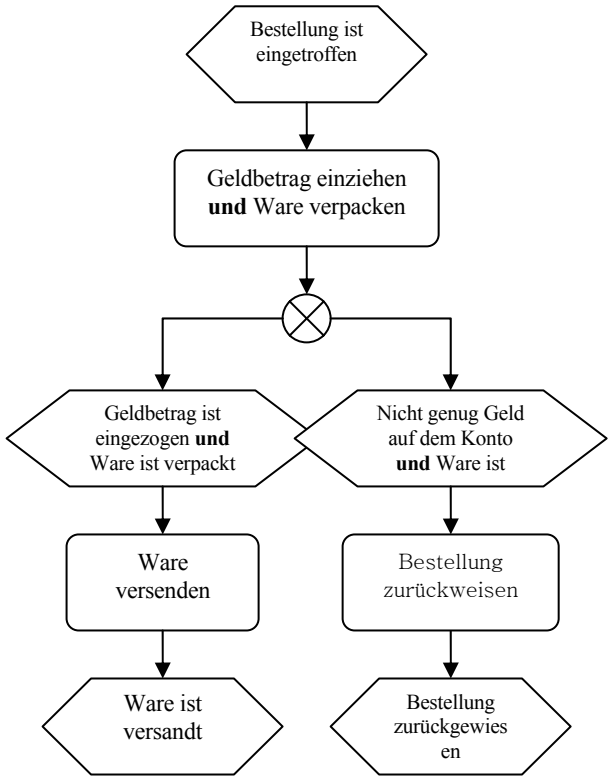
Auf die beiden Äste, die mit den Funktionen „Ware verpacken“ und „Geldbetrag einziehen“, muss also die AND Operation angewendet werden. Die Auswertung beginnt mit den beiden oben genannten Funktionen, die Tabelle der Abbildung 2-21 enthält in der zweiten Zeile die Definition der hierzu passenden Operation, es gilt:



Als nächstes müssen die Reste der beiden oberen Äste ausgewertet werden, die Tabelle aus der Abbildung 2-21 (Zeile 3) liefert folgendes Ergebnis.



Da nach dem Ereignis „Ware ist verpackt“ ein AND Join kommt, muss als nächstes in der Tabelle aus der Abbildung 2-22 (Auflösung von AND Joins) nachgesehen werden. Zeile 1 enthält die gesuchte Operation und die besagt, dass die Auswertung fertig ist und nicht auf die Nachfolgenden EPK Elemente ausgeweitet werden soll. Die Ersetzung der oben ausgewerteten Elemente durch die Ergebnisse dieser Auswertung führt schließlich zur der in der Abbildung 2-28 dargestellten EPK, die wie man leicht feststellen kann immer noch denselben Sachverhalt beschreibt.



Titel	Bestellungsannahme
Trigger	Bestellung ist Eingetroffen
Szenario	**1.** Geldbetrag einziehen und Ware verpacken **2.** Ware versenden
Erweiterungen	**1.1** Nicht genug Geld auf dem Konto und Ware ist verpackt **1.1.1** Bestellung zurückweisen

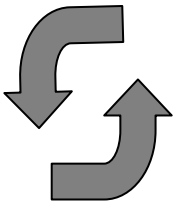


Abbildung 2-28

Jedoch fällt das Ereignis „Nicht genug Geld und Ware ist verpackt“ besonders auf, günstiger wäre es vielleicht erst sicherzustellen, dass das Konto des Kunden genug Geld aufweist und erst dann die Ware verpacken und versenden. Dieser Entwurfsfehler ist natürlich auch in der ursprünglichen EPK aus Abbildung 2-27 enthalten, fällt aber weniger auf. Die vorgeschlagene Transformation hat also auch einen positiven Nebeneffekt: Entwurfsfehler lassen sich nach ihr leichter entdecken.

Die transformierte EPK kann nun mit den am Anfang dieses Kapitels definierten Regeln in ein Use Case und wieder zurück konvertiert werden, das Ergebnis, dieser Konvertierung ist in der Abbildung 2-28 dargestellt.

Eine Alternative zur oben beschrieben Strategie die AND und OR Konnektoren aufzulösen, besteht darin die EPK an den AND Konnektoren in mehrere Use Cases aufzuteilen und die Ereignisse die mit dem AND Split verbunden sind als Vorbedingungen bzw. Erfolgsgarantien der Use Cases zu interpretieren, wie in der folgenden Abbildung dargestellt ist.

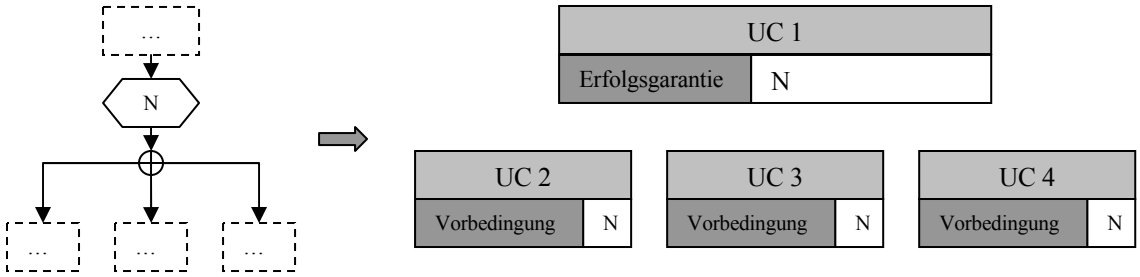


Abbildung 2-29

Die Übersetzung der EPK aus der Abbildung 2-27 wären demnach die folgenden drei Use Cases.

Titel	…
Trigger	Bestellung ist Eingetroffen
Szenario	**1.** Ware verpacken
Erfolgsgarantie	Ware verpackt

Titel	…
Vorbedingungen	**1.** Ware verpackt **2.** Geldbetrag ist eingezogen
Szenario	**1.** Ware versenden
Erfolgsgarantie	Ware ist versandt

Titel	…
Trigger	Bestellung ist Eingetroffen
Szenario	**1.** Geldbetrag einziehen
Erfolgsgarantie	Geldbetrag ist eingezogen
Erweiterungen	**1.1** Nicht genug Geld auf dem Konto **1.1.1** Bestellung zurückweisen

Abbildung 2-30

Diese Strategie funktioniert allerdings nur für AND Joins oder Splits und zwar auch nur für solche, wo dem AND Konnektor ein Ereignis vorangeht. Folgende EPK Konstrukte wären durch diese Strategie nicht abgedeckt.

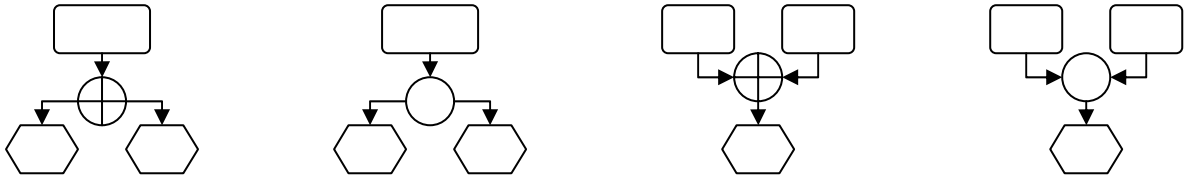


Abbildung 2-31

Jede EPK, die eins dieser Konstrukte hätte, könnte mit dieser Strategie nicht mehr in ein Use Case konvertiert werden, wogegen die Strategie mit der Auswertung und Ersetzung der EPK Konnektoren mit all diesen Konstrukten keine Probleme hat.

Eine mögliche Lösung wäre, wann immer möglich die letzt genannte Strategie zu benutzen und falls es nicht mehr geht, weil die zu übersetzende EPK eins der oben dargestellten Konstrukte aufweist, die erst genannte Strategie zu benutzen. Aus Zeitgründen war es allerdings nicht möglich beide Strategien zu implementieren. Die Entscheidung wurde deshalb zu Gunsten der universellären, erstgenannten Strategie getroffen.

Zum Schluss dieses Abschnitts sollte noch erwähnt werden, dass es auch Fälle gibt, wo die Verschmelzungsstrategie zwar möglich ist, aber dennoch gewisse Nachteile mit sich bringt.

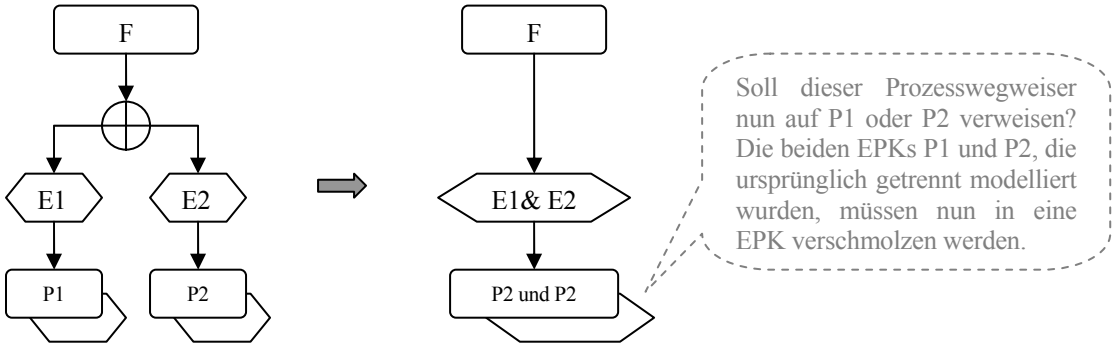


Abbildung 2-32

Wenn z.B. Prozesswegweiser oder hierarchische Funktionen verschmolzen werden müssen, so muss die Verschmelzung sich auch auf die EPKs ausbreiten, die diese Elemente referenzieren. Die Informationen über die Partitionierung der EPKs wären in den so erzeugten Use Cases nicht mehr enthalten und könnten aus diesen Use Cases nicht mehr rekonstruiert werden.

2.3 Zusammenfassung

In diesem Kapitel wurde ein Konzept vorgestellt, wie Use Cases und EPKs aufeinander abgebildet werden können. Dabei wurde festgestellt, dass beliebige Use Cases vollautomatisch in die EPK Darstellung überführt werden können.

Die vollautomatische Übersetzung der EPKs in Use Cases funktioniert leider nicht, weil die Eingabe bestimmter Informationen, die in den Use Cases enthalten sind, aber in den EPKs nicht dargestellt werden, unvermeidlich ist.

Weiterhin wurde festgestellt, dass die EPKs, die durch die Konvertierung der Use Cases entstehen, alle nach einem bestimmten Schema aufgebaut sind, um auch die EPKs, die diesem Schema nicht entsprechen, abbilden zu können, wurden Transformationsvorschriften zur Überführung auf das oben genannte Schema angegeben, die die anfängliche Semantik der EPKs nicht verändern.

Die in diesem Kapitel vorgestellten Transformationen wurden mit der XSLT Technologie implementiert und können mit einer in Java entwickelten Benutzeroberfläche, welche auf der beiliegenden CD zu finden ist, ausprobiert werden. Eine Benutzeranleitung in Form eines Anwendungsszenarios ist in Kapitel 5 enthalten.

3 Formate

Bevor die Implementierung der Abbildungen aus dem letzten Kapitel diskutiert wird, sollen in diesem Kapitel die Formate der Use Cases und EPKs vorgestellt werden. Die Kenntnis dieser Formate ist vor allem für das Verständnis der im nächsten Kapitel vorgestellten Umsetzung der Transformationen relevant.

3.1 XML und XML Schema

XML ist ein Standard zur Modellierung von Daten in Form einer Baumstruktur, dieser Standard beinhaltet einige wenige und sehr einfache Regeldefinitionen [Wikipedia]: für den Aufbau von Dokumenten, die Daten enthalten

- Das Dokument besitzt ein Wurzelelement
- Alle Elemente eines Dokuments haben ein Begin- und ein Endtag (<name>…</name>)
- Die Begin- und die End- Tags sind korrekt verschachtelt

XML selbst ist also eine Metasprache, sie regelt nur, wie die Dokumente aussehen müssen, nicht welche Elemente sie enthalten. Soll XML für den Datenaustausch verwendet werden, so sollte ein Format der Daten festgelegt werden, was nichts anderes ist, als eine Beschreibung welche Elemente es gibt und wie sie aufgebaut sind. Zur Festlegung des Formats wird XML-Schema verwendet, sie umfasst die Spezifikation neuer XML Elemente und deren Attribute, der Typ der Elemente kann dabei einfach oder komplex sein. Folgendes Statement definiert z.B. einen Name- Element, das den einfachen Typ string hat.

```
<xs:element name=name" type="xs:string"/>
```

Wie in herkömmlichen Programmiersprachen werden auch in XML Schema die komplexen Datentypen aus einfachen (integer, string, boolean, usw.) zusammengesetzt, und können durch andere Typen erweitert oder eingeschränkt werden. In der folgenden Anweisung wird ein Stakeholder Typ definiert, der aus genau einem Name- Element und beliebig vielen interest-Elementen zusammengesetzt ist und einen Attribut id vom Typ integer besitzt.

```
<xs:complexType name="stakeholder">
        <xs:sequence>
                <xs:element name="name" type="xs:string"/>
                <xs:element name="interest" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:integer"/>
</xs:complexType>
```

Ein mögliches Element vom Typ stakeholder könnte wie folgt aussehen:

```
<stakeholder id="1"><name>BankKunde</name><interest>Geld Abheben<interest></stakeholder>
```

Wie beim Entwurf von Datenbanksystemen können auch in XML- Schema, Schlüssel- und Fremdschlüssel- Beziehungen definiert werden. So wird in der folgenden Anweisung festgelegt, dass das id- Attribut der Stakeholder Elemente eine Primärschlüsseleigenschaft besitzt.

```
<xs:key name="pKey">
        <xs:selector xpath="stakeholder"/><xs:field xpath="@id"/>
</xs:key>
```

Folgendes Statement besagt, das Attribut *stakeholderID* des *guarantee*- Elements ein Fremdschlüssel ist und den oben definierten Schlüssel *pKey* referenziert.

```
<xs:keyref name="fKey" refer="pKey">
        <xs:selector xpath="guarantee"/>
        <xs:field xpath="@stakeholderID"/>
</xs:keyref>
```

Leider ist es nicht möglich in dieser Arbeit, alle möglichen Sprachkonstrukte von XML-Schema darzulegen, da dies den Rahmen sprengen würde. Doch die oben vorgestellten dürften bereits ausreichen um die Beschreibung des Use Case Modells, das im folgenden Abschnitt präsentiert wird, nachzuvollziehen. Eine ausführlichere Beschreibung von XML Schema ist z.B. bei [W3C] zu finden.

3.2 Definition eines Formats für Use Cases

Das Konzept der Use Cases wurde in Abschnitt 1.1 dargestellt, in diesem Abschnitt soll das XML- Format, in dem Use Cases abgespeichert werden können, vorgestellt werde. Im Folgenden ist die Implementierung des Use Case Modells aus Abbildung 1-4 als XML Schema dargestellt.

```
<? xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="unqualified"
        attributeFormDefault="unqualified">
```

Das Topic Element dient dazu die Use Cases nach Themengebieten zu ordnen, es enthält beliebig viele Use Cases und andere Topic Elemente. Identifiziert wird jedes Topic Element durch seinen Namen.

```
<xs:element name="topic">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="useCase" minOccurs="0" maxOccurs="unbounded"/>
                        <xs:element ref="topic" minOccurs="0" maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="name" type="xs:string" use="required"/>
        </xs:complexType>
```

Die Use Cases werden durch ihre id- Attribute identifiziert, das id- Attribut jeden Use Case ist daher ein Schlüssel.

```
        <xs:key name="useCaseIDKey">
                <xs:selector xpath=".//useCase"/>
                <xs:field xpath="@id"/>
        </xs:key>
```

Ein Use Case Schritt kann auf einen anderen Use Case verweisen, diese Beziehung wird durch einen Fremdschlüsselconstraint nachgebildet, das useCaseID- Attribut des Step Elements referenziert das id- Attribut eines Use Case

```
        <xs:keyref name="useCaseKeyRefAction" refer="useCaseIDKey">
                <xs:selector xpath=".//step"/>
                <xs:field xpath="@useCaseID"/>
        </xs:keyref>
</xs:element>
```

Ein Use Case besitzt einen Titel-, Umfang- und Level Element. Mindestens ein Akteur und ein Trigger muss angegeben werden. Ein Use Case hat beliebig viele Vor- und Nachbedingungen sowie ein Mainszenario, das aus Schritten besteht. Es können auch beliebig viele Stakeholder definiert werden.

```
<xs:element name="useCase">
<xs:complexType>
        <xs:sequence>
        <xs:element name="titel" type="xs:string"/>
        <xs:element name="scope" type="xs:string"/>
        <xs:element name="level" type="xs:string"/>
        <xs:element name="actor" type="xs:string" maxOccurs="unbounded"/>
        <xs:element name="trigger" type="xs:string"/>
        <xs:element name="precondition" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="guarantee" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="minimumguarantee" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="stakeholder" type="stakeholder" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="mainScenario">
                <xs:complexType>
                        <xs:sequence>
                        <xs:element name="step" type="step" maxOccurs="unbounded"/>
                        </xs:sequence>
                </xs:complexType>
                <xs:unique name="ID">
                        <xs:selector xpath=".//*"/>
                        <xs:field xpath="@id"/>
                </xs:unique>
        </xs:element>
        </xs:sequence>
        <xs:attribute name="id" type="xs:positiveInteger" use="required"/>
        </xs:complexType>
```

Innerhalb jeden Use Case sind die id's der Step- Elemente unique.

```
        <xs:key name="stepIDKey">
                <xs:selector xpath=".//step"/>
                <xs:field xpath="@id"/>
        </xs:key>
```

Jeder Step- Element kann eine Sprunganweisung zu einem anderen Step- Element desgleichen Use Case enthalten, das goto Attribut referenziert deshalb mit einem Fremdschlüsselconstraint die id eines anderen Step Elements.

```
        <xs:keyref name="stepKeyRefGoto" refer="stepIDKey">
                <xs:selector xpath=".//step"/>
                <xs:field xpath="@goto"/>
        </xs:keyref>
```

Eine Garantie wird von mindestens einem Use Case Schritt erfüllt. Das value-Attribut ist daher ein Fremdschlüssel auf den Schlüssel Attribut des Step-Elements von dem die Garantie erfüllt wird.

```
        <xs:keyref name="stepKeyRefGuarantee" refer="stepIDKey">
                <xs:selector xpath="./guarantee/stepID"/>
                <xs:field xpath="@value"/>
        </xs:keyref>
</xs:element>
```

Ein Use Case Schritt hat eine id, einen Namen und eine Beschreibung. Er kann beliebig oft erweitert werden und kann eine Sprunganweisung zu einem anderen Schritt oder eine Referenz zu einem Use Case enthalten.

```
<xs:complexType name="step">
        <xs:sequence>
                <xs:element name="description" minOccurs="0"/>
                <xs:element name="extension" type="extension" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:positiveInteger" use="required"/>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="goto" type="xs:positiveInteger" use="optional"/>
        <xs:attribute name="useCaseID" type="xs:positiveInteger" use="optional"/>
</xs:complexType>
```

Ein Stakeholder hat einen Namen und beliebig viele Interessen.

```
<xs:complexType name="stakeholder">
        <xs:sequence>
                <xs:element name="name" type="xs:string"/>
                <xs:element name="interest" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
</xs:complexType>
```

Eine Erweiterung hat eine ID, eine Beschreibung, und einen Namen. Sie enthält auch beliebig viele Schritte.

```
<xs:complexType name="extension">
        <xs:sequence>
                <xs:element name="step" type="step" maxOccurs="unbounded"/>
                <xs:element name="description" type="xs:string" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:positiveInteger" use="required"/>
        <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>
```

Eine Garantie (hier ist Successguarantee gemeint) hat einen Namen, und Verweise auf die Use Case Schritte von denen Sie erfüllt wird.

```
<xs:element name="guarantee">
        <xs:complexType>
                <xs:sequence>
                        <xs:element name="name" type="xs:string"/>
```

Jede Garantie wird durch mindestens ein Use Case Schritt erfüllt. Die Fremdschlüsselbeziehung stepID/value → step/id ist innerhalb des Elements Use Case definiert.

```
                        <xs:element name="stepID" maxOccurs="unbounded">
                                <xs:complexType>
                                        <xs:attribute name="value" type="xs:positiveInteger"/>
                                </xs:complexType>
                        </xs:element>
                </xs:sequence>
        </xs:complexType>
        <xs:unique name="stepRefS">
                <xs:selector xpath="stepID"/>
                <xs:field xpath="@value"/>
        </xs:unique>
</xs:element>
</xs:schema>
```

Zum Schluss dieses Kapitels ist in der folgenden Abbildung der Geldautomat- Use Case aus Abbildung 1-1 nach dem oben dargestellten Schema notiert.

Titel	Geld abheben
Akteur	Ein Kunde
Umfang	Geldautomat
Ebene	Anwenderziel
Szenario	1. Kunde steckt seine Geldkarte rein 2. Kunde gibt seine PIN ein 3. Kunde gibt den Geldbetrag ein 4. Automat gibt das Geld aus
Erweiterungen	1.1 Karte ist falsch rum eingesteckt 1.1.1 Automat informiert den Kunden 1.1.2 Kunde dreht seine Karte um 1.1.3 Weiter mit Schritt 2 1.2 Die PIN passt nicht zur Geldkarte 1.2.1 Automat fordert eine andere PIN 1.2.2 Weiter mit Schritt 2

```
<? xml version="1.0" encoding="UTF-8"?>
<topic name="Bankautomat" xmlns:xsi="http
<useCase id="1">
        <titel> Geld abheben </titel>
        <scope> Geldautomat </scope>
        <level> Anwenderziel </level>
        <actor> Ein Kunde </actor>
        <trigger/>
        <mainScenario>
                <step id="1" name="Kunde steckt seine Geldkarte rein">
                        <extension id="11" name="Karte ist falsch rum eingesteckt ">
                                <step id="111" name="Automat informiert den Kunden"/>
                                <step id="112" name="Kunde dreht seine Karte um" goto="2"/>
                        </extension>
                </step>
                <step id="2" name="Kunde gibt seine PIN ein">
                        <extension id="21" name="Die PIN passt nicht zur Geldkarte">
                                <step id="211" name="Automat fordert eine andere PIN" goto="2"/>
                        </extension>
                </step>
                <step id="3" name="Kunde gibt den Geldbetrag ein"/>
                <step id="4" name="Geldautomat gibt das Geld aus"/>
        </mainScenario>
</useCase>
</topic>
```

Abbildung 3-1

3.3 EPML das Format für EPKs

Event-Driven-Process-Chain-Markup-Language[1] (EPML) ist eine auf XML basierte Beschreibungssprache für EPKs. EPML wurde entwickelt um für die Programme, die auf dem Gebiet der EPKs arbeiten einen einheitlichen, werkzeugunabhängigen Austauschformat bereitzustellen. Wie das Format der Use Cases wurde auch das Format für Ereignisgesteuerte Prozessketten mit einer XML- Schema definiert, diese Schema Definition liegt momentan in der Version 1.2 vor und ist unter http://wi.wu-wien.ac.at/home/mending/EPML/EPML_12.xsd verfügbar. Wegen des enormen Umfangs dieses Schemas wird in diesem Kapitel nur die grundlegende Struktur, die zum Verständnis der Abbildungen notwendig ist, vorgestellt.

.

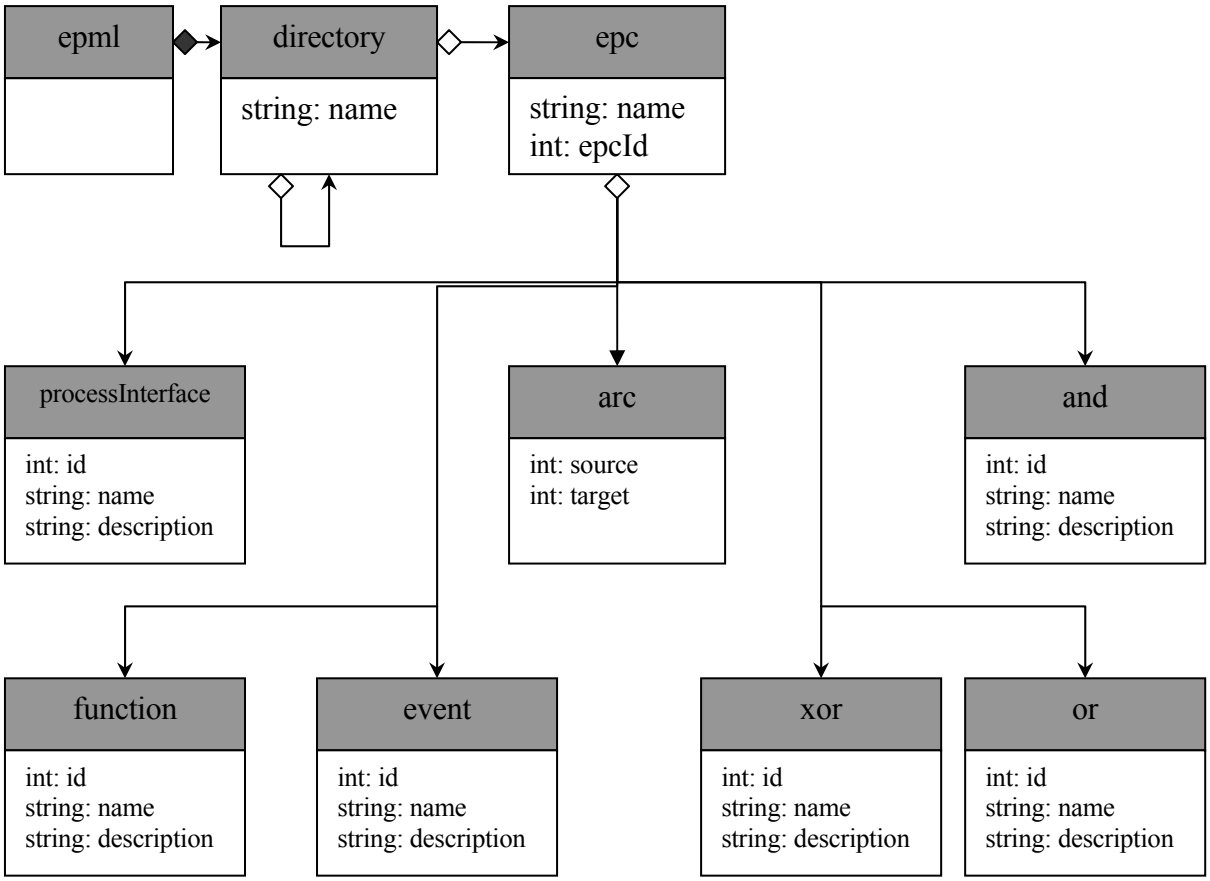


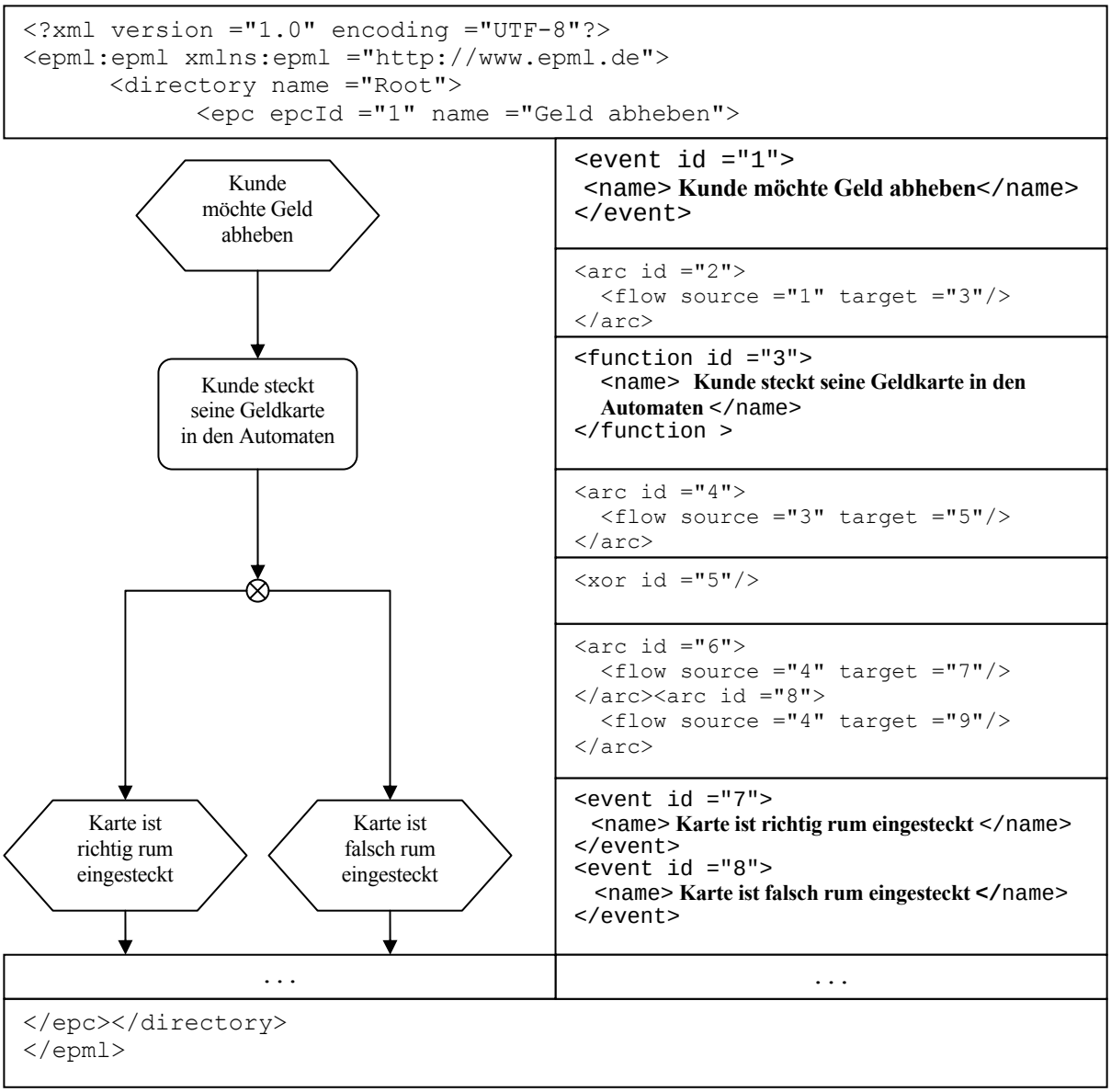
Abbildung 3-2

In der Abbildung 3-2 ist eine Übersicht über einige EPML Elemente zusammengestellt, das Schema enthält viele weitere Elementdefinitionen, die aber für diese Arbeit keine Rolle spielen und deshalb weggelassen wurden.

Jedes EPML Dokument wird durch ein *epml* Element repräsentiert. Das *epml* Element besteht aus mindestens einem *directory*, das wiederum beliebig viele *directory*- Elemente und beliebig viele *epc* Elemente enthalten kann. Eine EPK wird mit dem *epc* Element modelliert, dieses Element enthält daher eine beliebige Anzahl von *processInterface*, *function*, *event*, *and*, *or*, *xor* und *arc* Elemente. Jedes der letzt genanten Elemente hat eine in der EPK eindeutige id, das *arc* Element stellt eine Kante dar, das *source* Attribut dieses Elements verweist dabei auf die id des Quellknotens, das *target* Attribut auf die id des Zielknotens.

[1] EPML (Eventdriven Process Markup Language) wurde in [MendlingNüttgens] veröffentlicht.

Im Folgenden sind zur Veranschaulichung Teile der EPK, die die Anforderung an die Funktionalität eines Geldautomaten beschreibt in EPML notiert.




```
<?xml version ="1.0" encoding ="UTF-8"?>
<epml:epml xmlns:epml ="http://www.epml.de">
      <directory name ="Root">
            <epc epcId ="1" name ="Geld abheben">

<event id ="1">
 <name> Kunde möchte Geld abheben</name>
</event>

<arc id ="2">
  <flow source ="1" target ="3"/>
</arc>

<function id ="3">
  <name> Kunde steckt seine Geldkarte in den
  Automaten </name>
</function >

<arc id ="4">
  <flow source ="3" target ="5"/>
</arc>

<xor id ="5"/>

<arc id ="6">
  <flow source ="4" target ="7"/>
</arc><arc id ="8">
  <flow source ="4" target ="9"/>
</arc>

<event id ="7">
  <name> Karte ist richtig rum eingesteckt </name>
</event>
<event id ="8">
  <name> Karte ist falsch rum eingesteckt </name>
</event>

...                                    ...

</epc></directory>
</epml>
```

Abbildung 3-3

3.4 Zusammenfassung

XML ist heutzutage das Standarddatenaustauschformat, XML Schema dient der Beschreibung der Dokumentstruktur und ist ebenfalls sehr populär. Das Format für die EPKs wurde unter Verwendung dieser Technologien definiert. Dieselben Technologien wurden deshalb auch für die Definition des Use Cases Formats verwendet. Das Use Case Format, sowie einige relevante EPML Elemente wurden in diesem Kapitel vorgestellt. Im nächsten Kapitel wird die Implementierung der Abbildungen präsentiert, dabei wird davon ausgegangen, dass die Use Cases und die EPKs in diesen beiden Formaten vorliegen. Da dabei auch teilweise Code Schnipsel vorgestellt werden ist zumindest ein Überblick über diese Formate empfehlenswert.

4 Implementierung der Abbildungen

In Kapitel 2 wurde ein Konzept vorgeschlagen, wie Use Cases und Ereignisgesteuerte Prozessketten aufeinander abgebildet werden können, in Kapitel 3 wurden die auf XML basierenden Use Case- und EPK Formate vorgestellt. In diesem Kapitel wird nun die Implementierung des in Kapitel 2 vorgeschlagenen Konzepts in Grundzügen dargestellt. Da sowohl die Use Cases als auch die EPKs in XML vorliegen, werden die Abbildungen ebenfalls in einer auf XML basierenden Sprache (also XSLT) formuliert. Im folgenden Abschnitt wird deshalb kurz auf die Funktionsweise von XSLT eingegangen. Die beiden darauf folgenden Abschnitte befassen sich mit der Implementierung der Abbildungen der Use Cases beziehungsweise der EPKs.

4.1 XSLT

XSLT ist eine Scriptsprache, die entwickelt wurde um Transformationen von beliebigen XML Dokumenten zu beschreiben. Da XSLT auf XML basiert ist diese Sprache vollkommen plattformunabhängig, jeder Internetbrowser kann heutzutage XSLT verarbeiten[1], beinahe jede Programmiersprache[2] besitzt einen XSLT Prozessor, der es ermöglicht Transformationen mit relativ wenig Aufwand auszuführen. Da XSLT sehr verbreitet ist und die Scripte deshalb leicht in jedes Computerprogramm integriert werden können, wurden die Abbildungen der Use Cases und EPKs zum größten Teil[3] in XSLT implementiert. Folgende Abbildung veranschaulicht die Funktionsweise von XSLT:

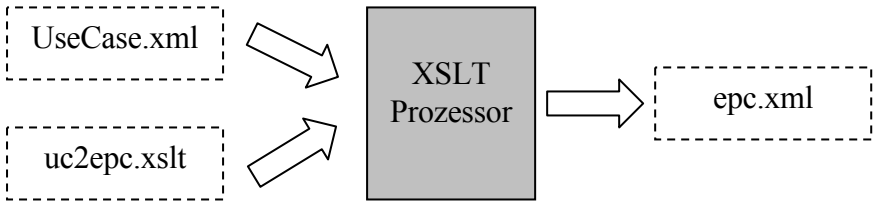


Abbildung 4-1

Ein XSLT Prozessor bekommt als Eingabe eine Quelldatei (UseCase.xml) sowie eine Datei mit den Transformationsregeln (uc2epc.xslt) und liefert als Ergebnis eine Zieldatei (epc.xml), die aus der Quelldatei durch die Anwendung dieser Regeln hervorgeht.

Die XSLT- Transformationsregeln werden als so genannte Templates (Schablonen) spezifiziert. In einem Template wird festgelegt zu welchen XML Knoten welche Ausgabe erzeugt werden soll. Diese Ausgabe kann sowohl andere XML Knoten, als auch beliebiger Text sein. Abbildung 4-2 demonstriert diesen Sachverhalt: der Kasten links unten enthält ein Template wonach zu jedem XML Knoten mit dem Namen „Hallo“ die Ausgabe „Hallo XSLT!“ produziert wird. Ein XSLT Template entspricht also einer Funktion einer herkömmlichen Programmiersprache, wie Funktionen können die Templates benannt werden und Parameterdeklarationen enthalten, anders als die Funktionen können in den Templates jedoch keine Rückgabewerte deklariert werden.

[1] Internet Explorer, Opera, Firefox, Mozilla
[2] Java, c/c++, .net und viele weitere
[3] Für die Benutzereingaben wurde eine Java Benutzeroberfläche entwickelt (siehe Kapitel 5)

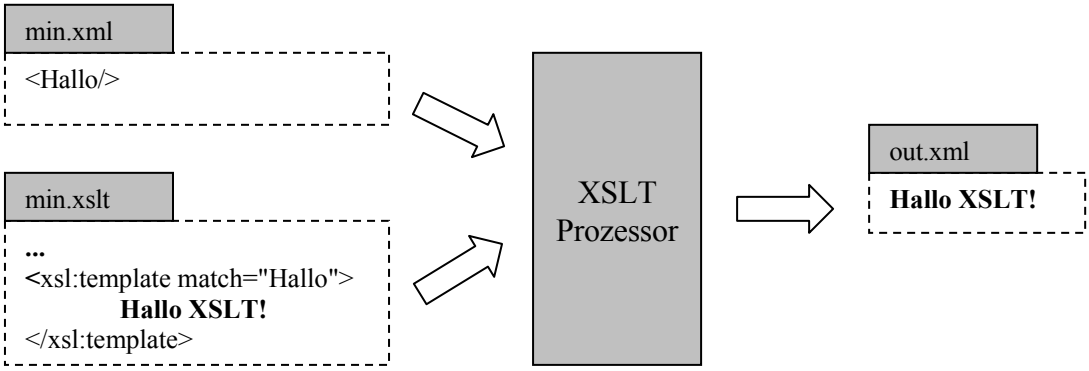
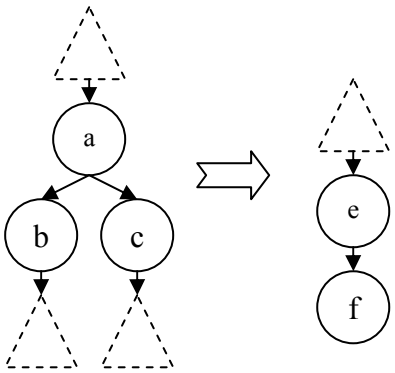


Abbildung 4-2

Ein XSLT Prozessor liest also die zu transformierende XML Datei, sucht die Regeln, die zu dem gelesenen Knoten passen, wenn eine Regel gefunden wird, werden die Anweisungen des entsprechenden Template ausgeführt, falls nicht wird dieser Knoten so in die Zieldatei übernommen, wie er in der Quelldatei steht. Die Transformationsregeln in Form von Templates lassen sich deshalb gut auch graphisch darstellen:

Aus einen *a* Knoten, der ein *b* und ein *c* Knoten als direkte Nachfolger hat wird ein *e* Knoten und ein *f* Knoten der ein direkter Nachfolger von *e* ist, erzeugt. Dasselbe als ein XSLT Template:

```
<xsl:template match="a[./b and ./c]">
        <e><f/></e>
</xsl:template>
```



Der in rechteckigen Klammern eingeschlossene Ausdruck stellt eine Bedingung dar, in dem oberen Template lautet diese Bedingung: der aktuelle Knoten hat einen *b* und einen *c* Knoten als direkte Nachfolger. Der punkt repräsentiert den aktuellen Knoten, da nur *a* Knoten gematcht werden, ist der aktuelle Knoten immer also ein *a* Knoten, mit dem Schrägstrich wird die Menge der Nachfolgerknoten angesprochen. Außer der Möglichkeit die Nachfolgerknoten eines aktuellen Knoten anzusprechen besteht auch die Möglichkeit die Vorgänger- und die Nachbar- und eine menge weiter Knoten anzusprechen. In der folgenden Tabelle sind die gebräuchlichsten der so genannten Axen- Anweisungen aufgelistet:

self::	Der aktuelle Knoten (Abkürzung ist.)
child::	Die Menge der direkten Nachfolger (Abkürzung ist /)
descendant::	Die Menge aller Nachfolger (Abkürzung ist //)
parent::	Der direkte Vorgänger
ancestor::	Die Menge aller Vorgänger
preceding-sibling::	Die Menge der vorangehenden Nachbarsknoten
following-sibling::	Die Menge der nachfolgenden Nachbarsknoten

Weitere wichtige Anweisungen, die im weiteren Verlauf dieses Abschnitts häufig verwendet werden sind in der folgenden Tabelle zusammengestellt, eine vollständige Auflistung aller XSLT- Funktionen und Sprachkonstrukte ist bei [W3C] zu finden.

<xsl:apply-templates>	Auf die selektierte Knotenmenge werden die definierten Regeln angewendet, diese Anweisung wird dazu verwendet, eine bestimmte Reihenfolge der Anwendung der Transformationsregeln zu erreichen.
<xsl:for-each>	Stellt eine Iteration über einer Menge selektierter Knoten dar, entspricht einer for- Schleife in herkömmlichen Programmiersprachen.
<xsl:attribute>	Dient dazu ein Attribut zu erzeugen. Mit @ wird ein Attribut selektiert.
<xsl:call-template>	Wird dazu verwendet, ein Template explizit aufzurufen, ohne das ein Knoten mit der passenden Eigenschaft gematcht wird.
<xsl:variable>	Dient der Deklaration einer Konstanten.
<xsl:if>	Entspricht dem *if* in einer herkömmlichen Programmiersprache
<xsl:choose> *<xsl:when>* *<xsl:otherwise>* *</xsl:choose>*	Entspricht dem *if ... then ... else* Konstrukt in herkömmlichen Programmiersprachen

Die beiden nächsten Abschnitte befassen sich mit der Implementierung der Transformationen der Use Cases bzw. der EPKs, die verwendeten Transformationsregeln werden dabei sowohl graphisch als auch in Form von XSLT- Code Schnipsel präsentiert.

4.2 Transformation der Use Cases

Dieser Abschnitt dokumentiert die Vorgehensweise, bei der Implementierung der Transformationen der Use Cases. Zum besseren Verständnis der Sachverhalte, die hier präsentiert werden, sind XSLT Kenntnisse sowie die Kenntnisse der Formate der Use Cases und EPKs, die im letzten Kapitel vorgestellt wurden notwendig.

In Kapitel 2 wurde festgestellt, dass folgende Elemente des Use Case Formats bei der Abbildung auf eine EPK relevant sind:

1. das Topic Element entspricht einem directory Element im EPK Modell
2. das Use Case Element selbst entspricht einer EPK
3. der Trigger, die Vorbedingungen und die Nachbedingungen entsprechen den Ereignissen
4. die Schritte eines Use Case entsprechen Funktionen
5. die Erweiterungen eines Schritts entsprechen Ereignissen, die durch einen XOR Konnektor mit der dem Schritt entsprechenden Funktion verknüpft sind

Folgende Syntaxgraphen bieten eine Übersicht über die Funktionalität des XSLT Transformationsscripts zur Abbildung der Use Cases:

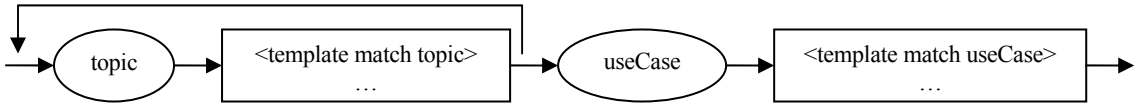


Abbildung 4-3

Falls ein *topic* Element vom XSLT Prozessor gelesen wird, werden die Regeln ausgeführt, die im entsprechenden Template (im Graphen dargestellt durch ein Rechteck) angegeben sind, falls anschließend noch ein *topic* Element gelesen wird, so werden die Anweisungen dieses Templates noch einmal ausgeführt, falls ein *useCase* Element gelesen wird, so werden diejenigen Anweisungen ausgeführt, die im Template zur Transformation eines useCase Elements formuliert sind. Eine Ellipse repräsentiert also einen XML Knoten und ein Rechteck eine Funktion oder hier ein Template, dessen Anweisungen beim Erkennen dieses Knotens ausgeführt werden.

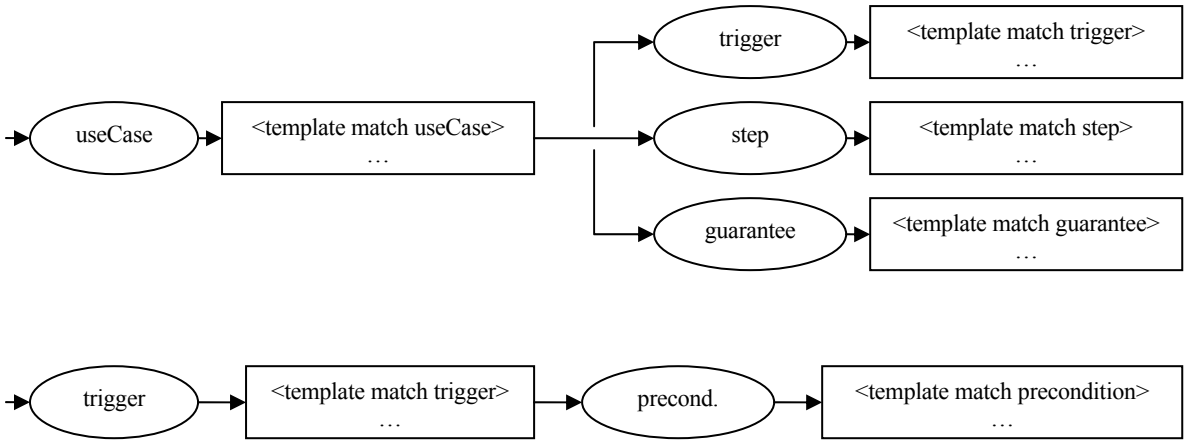


Abbildung 4-4

Der Trigger und die Vorbedingungen haben in einem Use Case keinerlei Verbindung zu einander, bei der Übersetzung in eine EPK braucht das Template zur Transformierung der Vorbedingungen jedoch bestimmte Parameter die erst nach der Transformation des Triggers vorhanden sind, deshalb ruft das Trigger- Template das Vorbedingungs- Template auf und übergibt ihm die erforderlichen Parameter (im weiteren Teil dieses Abschnitts werden die Templates genauer vorgestellt). Der Informationsfluss zwischen Templates kann in XSLT nur durch Parameterübergabe stattfinden, da die Variablen ihren anfangs zugewiesenen Wert nicht mehr ändern können.

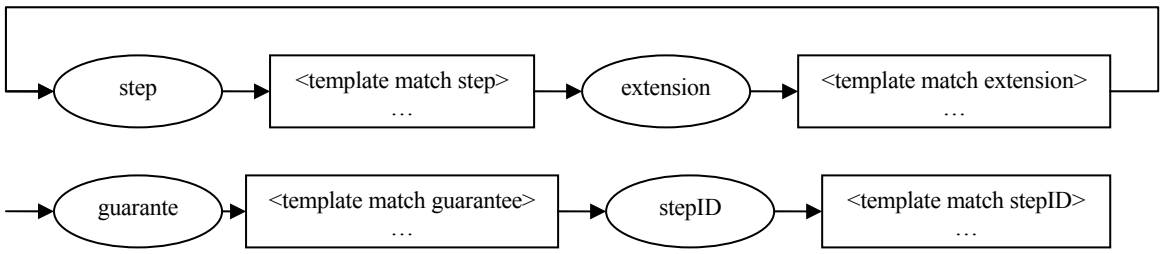


Abbildung 4-5

Wie in Kapitel 3 dargestellt, kann ein *topic* Element beliebig viele *useCase* Elemente und andere *topic* Elemente enthalten. Ein *directory* Element nimmt beliebig viele *epc* Elemente auf und kann außerdem auch beliebig viele andere *directory* Elemente enthalten. Ein *topic* Element entspricht daher genau einem *directory* Element im EPK Modell. Trifft also der XSLT-Prozessor auf einen *topic* Knoten mit dem Namen *n*, der kein Wurzel Knoten ist, so werden folgende Operationen ausgeführt:

1. Es wird ein *directory* Element *d* mit demselben Namen erzeugt
2. Alle direkten *useCase* Nachfolger werden in *epc* Elemente umgewandelt und unter *d* angehängt
3. Alle direkten *topic* Nachfolger werden in *directory* Elemente umgewandelt und unter *d* angehängt

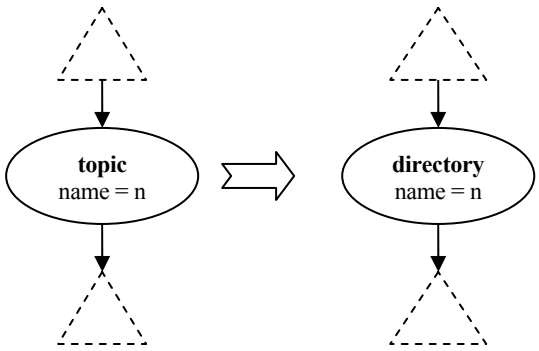


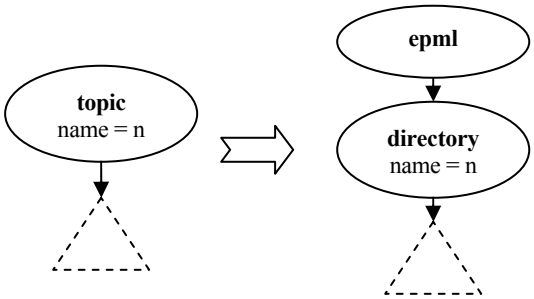
Abbildung 4-6

Rechts in der oberen Abbildung ist die Transformation graphisch dargestellt, das gestrichelte Dreieck repräsentiert dabei einen beliebigen XML Baum. Die gleiche Transformation ist unten als Template beschrieben, die Nummern, die in dem Template als Kommentare auftauchen, verweisen auf die Nummern der entsprechenden Operationen, die rechts in Abbildung 4-6 aufgeführt sind.

```
<! -- Falls ein topic Knoten erkannt wird und dieser Knoten hat  irgendwelche Vorgänger, d.h. es ist kein Wurzel
     Knoten -->
<xsl:template match="topic[ancestor::*]" >
        <!-- 1.  -->
        <directory>
                <xsl:attribute name="name"select="./@name"/>
                 <!-- 2. -->
                <xsl:apply-templates select="useCase"/>
                 <!-- 3. -->
                 <xsl:apply-templates select="topic"/>
        </directory>
</xsl:template>
```

Sollte der *topic* Knoten der Wurzel Knoten des XML Dokuments mit den Use Cases sein, so muss auch der entsprechende EPK Wurzelknoten erzeugt werden, dieser Wurzelknoten ist nach dem EPML Format immer ein *epml* Element:

1. ein *empl* Element *e* erzeugen
2. ein *directory* Element *d* erzeugen und unter *e* anhängen
3. alle direkten *useCase* Nachfolger in *epc* Elemente umwandeln und unter *d* anhängen
4. Alle direkten *topic* Nachfolger in *directory* Elemente umwandeln und unter *d* anhängen




```
<! -- Falls ein topic Knoten gematcht wird und dieser Knoten hat keine Vorgänger, d.h. es ist ein  Wurzel
     Knoten -->
<xsl:template match="topic[not(ancestor::*)]">
        <epml:epml>
                <! -- die gleichen Anweisungen wie im letzten Template -->
                <directory>
                        <xsl:attribute name="name"select="./@name"/>
                         <xsl:apply-templates select="useCase"/>
                        <xsl:apply-templates select="topic"/>
                </directory>
        </epml:epml>
</xsl:template>
```

Einem *useCase* Element entspricht ein *epc* Element. Ein *epc* Element hat eine id und einen Namen als Attribute, diesen Attributen entsprechen die id und der Titel des *useCase* Elements. Erkennt der XSLT Prozessor einen *useCase* Knoten so werden deshalb folgende Anweisungen ausgeführt:

1. Es wird ein *epc* Knoten *e* erzeugt und die entsprechenden Attributwerte gesetzt
2. Trigger und Vorbedingungen des *useCase* werden übersetzt und an *e* drangehängt
3. Schritte des *useCase* Elements werden übersetzt und an *e* drangehängt
4. Nachbedingungen des *useCase* Elements werden übersetzt und an *e* drangehängt

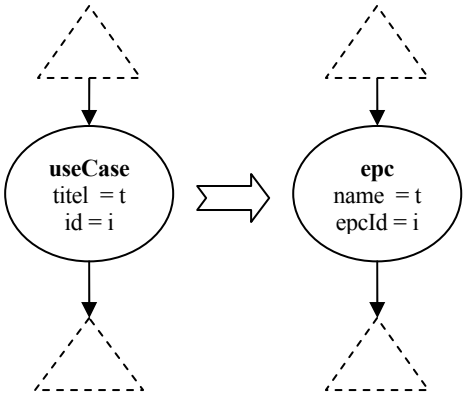


Abbildung 4-7

Folgendes Template beschreibt diese Aktionen, man vergleiche dazu auch den Syntaxgraphen aus Abbildung 4-4.

```
<xsl:template match="useCase">
        <!-- 1 -->
        <epc>
                <xsl:attribute name="epcId" select="./@id"/>
                <xsl:attribute name="name" select="./titel"/>
                <!-- 2 -->
                <xsl:apply-templates select="trigger"/>
                <!-- 3 -->
                <xsl:apply-templates select="*/step" mode="create"/>
                <!-- 4 -->
                <xsl:apply-templates select="guarantee"/>
        </epc>
</xsl:template>
```

Einem Trigger entspricht ein Startereignis einer EPK, wie dieses Startereignis mit den anderen Elementen der EPK verbunden wird hängt davon ab, ob im Use Case Vorbedingungen angegeben sind oder nicht. Falls keine Vorbedingungen angegeben sind:

1. wird ein Ereignis *t* erzeugt der denselben Namen wie der Trigger hat
2. *t* wird mit der Übersetzung des ersten Schritt des Use Case durch eine Kontrollflusskante *a* verbunden
3. Falls Use Cases existieren, die mit t eine namensgleiche Nachbedingung haben, wird über Prozesswegweiser zu diesen Use Cases eine Verbindung aufgebaut.

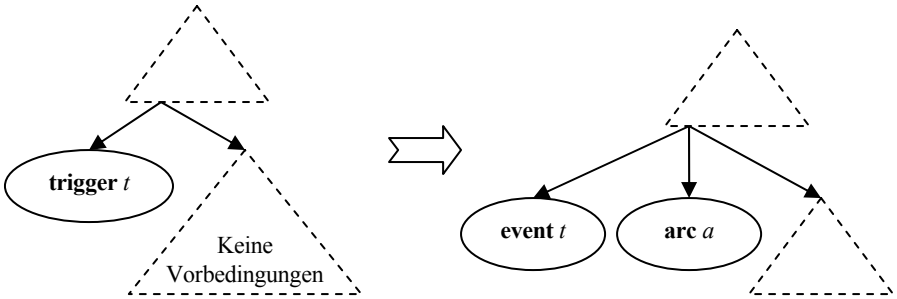


Abbildung 4-8

Man beachte, dass die Use Cases in Form eines XML Baums vorliegen, das Ergebnis der Transformation dieses Baumes ist ein XML Baum der den EPK Graphen repräsentiert. Zur besseren Übersichtlichkeit werden im Folgenden nicht die XML Teilbäume sondern gleich die EPK Teilgraphen, die von diesen Bäumen repräsentiert werden dargestellt. Die Transformationsregel der Abbildung 4-8 ist in der unteren Abbildung auf diese Weise dargestellt.

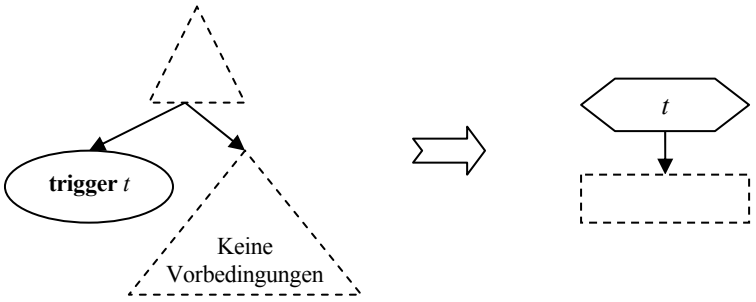


Abbildung 4-9

Das gestrichelte Rechteck repräsentiert einen beliebigen Teilgraphen, in der oberen Abbildung ist es die Übersetzung des mit „Keine Vorbedingungen“ beschrifteten Teilbaums.

```
<!-- Falls ein trigger Element gelesen wird, der Teil eines  useCase Elements ohne Vorbedingungen ist-->
<xsl:template match="trigger[not(ancestor::useCase/precondition)]">
        <xsl:variable name="triggerEventID" .../>
        <xsl:variable name="firstStepID" .../>
        <xsl:variable name="arcID" .../>
         <!-- 1. -->
         <xsl:call-template name="genTriggerEvent">
                <xsl:with-param name="name" select="."/>
                <xsl:with-param name="id" select="$triggerEventID"/>
         </xsl:call-template>

         <!-- 2.-->
          <xsl:call-template name="connectToNextStep">
                  <xsl:with-param name="sourceID" select="$triggerEventID"/>
                  <xsl:with-param name="targetID" select="ancestor::useCase/mainScenario/step[1]/@id"/>
                  <xsl:with-param name="arcID" select="$arcID"/>
          </xsl:call-template>
          <!-- 3. -->
          <xsl:call-template name="genProcessInterfacesToPrecondition">
                <xsl:with-param name="preconditionName" select="." as="xs:string"/>
                <xsl:with-param name="preconditionEventID" select="$triggerEventID" as="xs:double"/>
         </xsl:call-template>
</xsl:template>
```

Das Template *genTriggerEvent* macht nichts anderes als ein event Element mit dem übergebenen Namen und der übergebenen id zu erzeugen. Das Template *connectToNextStep* ist dafür zuständig eine Verbindung von einem Element mit *sourceID* zu einem Element mit *targetID* herzustellen und für diese Verbindung diejenige id zu verwenden, die als Parameter *arcID* übergeben wurde, dieses Template wird im Weiteren noch öfters verwendet, eine genaure Erklärung zur Funktionsweise dieses Templates folgt später. Mit dem Aufruf des Templates *genProcessToPrecondition* werden die notwendigen Prozesswegweiser erzeugt und mit *t*

verbunden. Dieses Template realisiert die Abbildungsvorschriften, die in Kapitel 2 auf Seite 15 dargestellt sind:

```
<xsl:template name="genProcessInterfacesToPrecondition">
        <xsl:param name="preconditionName" as="xs:string"/>
        <xsl:param name="preconditionEventID" as="xs:double"/>
        <xsl:choose><xsl:when test="fn:count(root()//useCase[guarantee/name=$preconditionName]) gt 1">
          <!-- Falls diese Vorbedingung in mehreren Use Cases die Nachbedingung ist-->
                <!--  Es wird ein OR Konnektor o erzeugt -->
                <!--  Zu jedem Use Case der eine Nachbedienung mit dem Namen $preconditionName hat,
                      wird ein Prozesswegweiser erzeugt und mit o verbunden -->
              <!-- o wird mit dem Ereignis der Vorbedingung, dessen id als preconditionEventID übergeben
                      wurde verbunden-->
        </xsl:when><xsl:when test="fn:count(root()//useCase[guarantee/name=$preconditionName]) = 1">
          <!-- Falls diese Vorbedingung in genau einem Use Cases die Nachbedingung ist-->
                <!-- Zu diesem Use Case wird ein Prozesswegweiser erzeugt und mit dem Ereignis der
                     Vorbedingung verbunden, die id dieses Ereignisses wurde als $preconditionEventID
                     übergeben -->
        </xsl:when></xsl:choose>
</xsl:template>
```

Falls ein useCase Element neben einem Trigger auch noch Vorbedingungen enthält, so werden folgende Schritte ausgeführt:

1. es wird ein Ereignis *t* erzeugt
2. es wird ein AND Konnektor *a* erzeugt
3. *t* wird mit *a* verbunden
4. alle Vorbedingungen werden übersetzt, diese Übersetzungen werden mit *a* verbunden
5. *a* wird mit der Übersetzung des ersten Schritts des Use Case verbunden
6. es werden Prozesswegweiser erzeugt und mit *t* verbunden

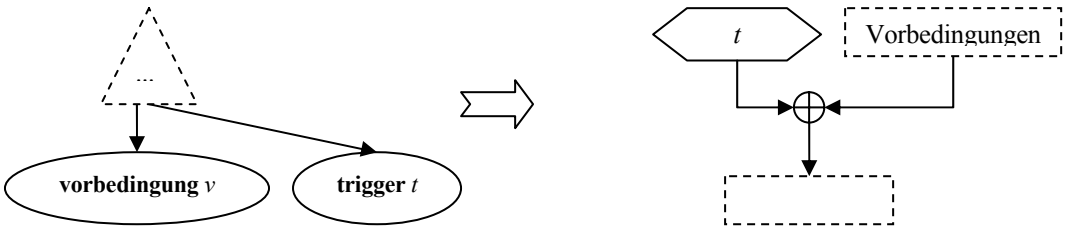


Abbildung 4-10

```
<xsl:template match="trigger[ancestor::useCase/precondition]">
        <xsl:variable name="andID" .../>
        <!-- 1.-->
        <xsl:call-template name="genTriggerEvent" >... </xsl:call-template>
        <!-- 2. -->
        <and>
                <xsl:attribute name="id"><xsl:value-of select="$andID"/></xsl:attribute>
         </and>
        <!-- 3. -->
        <xsl:call-template name="connect">... </xsl:call-template>
        <!-- 4. -->
        <xsl:apply-templates select="ancestor::useCase/precondition">
                <xsl:with-param name="andID" select="$andID"/>
         </xsl:apply-templates>
        <!-- 5. -->
        <xsl:call-template name="connectToNextStep">...</xsl:call-template>
        <!-- 6.-->
```

```
        <xsl:call-template name="genProcessInterfacesToPrecondition">...</xsl:call-template>
</xsl:template>
```

Das Template zur Transformation von Vorbedingungen erhält die id des AND Konnektors als Parameter übergeben, damit die transformierten Vorbedingungen mit diesem AND Konnektor verbunden werden können.

Da eine Vorbedingung, genau wie der Trigger einem Startereignis im EPK- Modell entspricht, werden die Vorbedingungen auch nach demselben Prinzip übersetzt:

1. zu jeder Vorbedingung wird ein Ereignis *v* erzeugt, das denselben Namen hat
2. *v* wird mit dem AND Konnektor verbunden, der bei der Transformation des Triggers erzeugt wurde
3. es werden Prozesswegweiser zu *v* erzeugt und damit verbunden

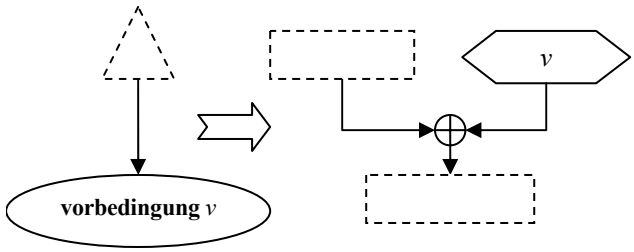


Abbildung 4-11

```
<xsl:template match="precondition">
        <xsl:param name="andID" as="xs:double"/>
        <!-- 1. -->
        <xsl:call-template name="genPreconditionEvent">
                <xsl:with-param name="name" .../>
        </xsl:call-template>
        <!-- 2.-->
        <xsl:call-template name="connect"><xsl:with-param ..."/></xsl:call-template>

        <!-- 3.-->
        <xsl:call-template name="genProcessInterfacesToPrecondition">
                <xsl:with-param ..."/>
        </xsl:call-template>
</xsl:template>
```

Nach dem die Übersetzung der Vorbedingungen und der Trigger besprochen wurde, werden im Weiteren die Übersetzung der Schritte, der Erweiterungen und der Nachbedingungen eines Use Case diskutiert.

Wie in Kapitel 2 festgestellt entspricht ein Use Case Schritt einer Funktion im EPK Modell, wie genau ein Schritt übersetzt wird hängt von folgenden Faktoren ab:

1. der Schritt enthält weder Erweiterungen noch eine goto- Anweisungen
2. der Schritt enthält mindestens eine Erweiterung aber keine goto- Anweisung
3. der Schritt enthält eine goto Anweisung

Da die Übersetzung variiert, je nach dem welche dieser 3 Bedingungen zutrifft, ist jede dieser Bedingungen durch ein Template abgedeckt. Da jedes dieser Templates auf einen Schritt Knoten passt entsteht ein Konflikt, dieser Konflikt wird dadurch aufgelöst, dass die Templates jeweils einen priority Attribut erhalten, bei einem Konflikt wird das Template mit dem höchsten priority Wert ausgewählt.

Sei zunächst angenommen, dass der Schritt weder Erweiterungen noch eine goto- Anweisung enthält, seine XML Baum Darstellung sieht dann so aus, wie im unteren Bild dargestellt. Falls so ein Schritt s erkannt wird, werden folgende Operationen ausgeführt:

1. es wird eine Funktion *f* erzeugt mit dem gleichen Namen und Description Element wie beim Use Case Schritt
2. es wird ein Ereignis *e* erzeugt mit dem Namen N(s). N(s) := „ok“ + s/@name falls keine Nachbedingung spezifiziert ist, die durch diesen Schritt erfüllt wird, anderenfalls ist N(s) := name der Nachbedingung die durch s erfüllt wird
3. *f* wird mit *e* verbunden
4. *e* wird mit seinem Nachfolger verbunden, falls dieser Nachfolger existiert

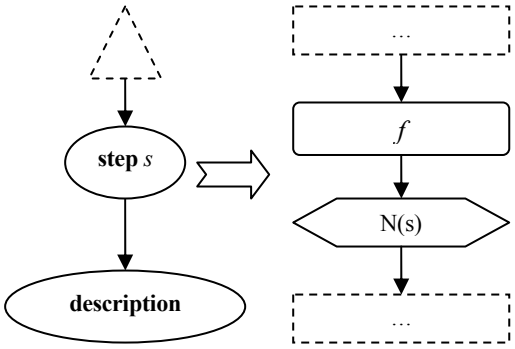


Abbildung 4-12

```
<xsl:template match="step" priority="1" mode="create">
        <!-- Aktionen 1. – 3. --> ...
        <!-- Aktion 4. Falls diesem Schritt noch mindestens ein weiterer Schritt nachfolgt, so wird eine
        Verbindung von der Funktion, die die Übersetzung dieses Schritts darstellt, zu der Funktion, die die
        Übersetzung des nachfolgenden Schritts ist aufgebaut. Die beiden Funktionen haben dieselben id's wie
        die Schritte, die Erstellung der Verbindung ist deshalb möglich bevor die Funktionen erstellt sind  -->
        <xsl:if test="following-sibling::step">
                <xsl:call-template name="connectToNextStep">
                        <xsl:with-param name="sourceID" select="..."/>
                        <xsl:with-param name="targetID" select="following-sibling::step[1]/@id"/>
                        <xsl:with-param name="arcID" select="..."/>
                </xsl:call-template>
        </xsl:if>
</xsl:template>
```

Das Template *connectToNextStep* ist dafür zuständig eine Verbindung von der Übersetzung des Use Case Schritts mit der id *sourceID* zur Übersetzung des Schritts mit der id *targetID* herzustellen. In Kapitel 2 wurde festgestellt, dass eine goto Anweisung einem XOR- Join entspricht, falls es also goto Anweisungen zum Schritt mit der id gleich *targetID* existieren, so werden folgende Aktionen durchgeführt:

a) es wird ein XOR Konnektor *x* erzeugt
b) es wird eine Verbindung vom Element mit sourceID zum *x* erzeugt
c) die Übersetzungen der Schritte, die eine goto Anweisung zum Schritt mit der id gleich *targetID* enthalten werden mit *x* verbunden.
d) *x* wird mit der Übersetzung des folgenden Schritts verbunden

Sind keine goto Anweisungen zum Schritt mit id gleich *targetID* vorhanden, so werden folgende Aktionen durchgeführt:

1. es wird eine Verbindung vom Element mit *sourceID* zum Elementen mit *targetID* hergestellt

In Abbildung 4-13 sind diese zwei Fälle dargestellt:

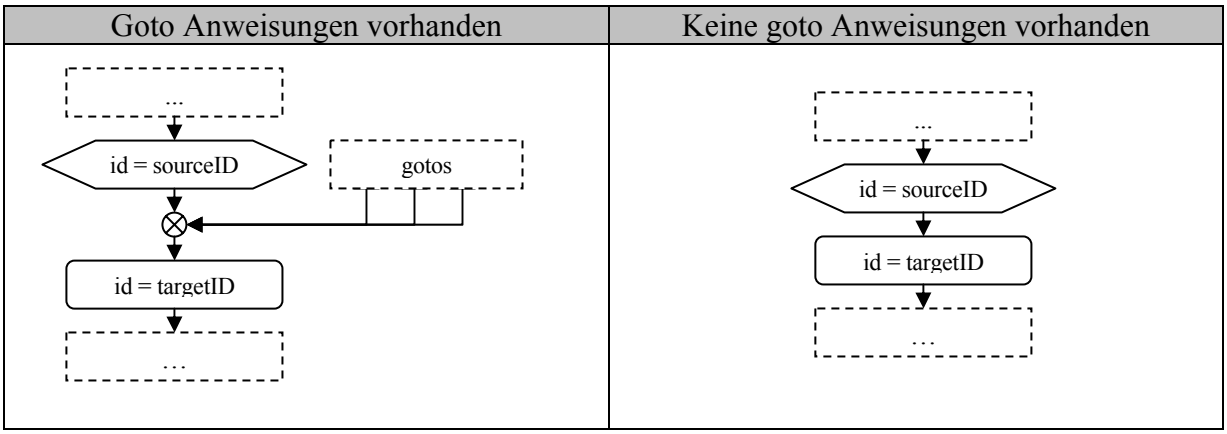


Abbildung 4-13

```
<xsl:template name="connectToNextStep">
        <xsl:param name="sourceID" as="xs:double"/>
        <xsl:param name="targetID" as="xs:double"/>
        <xsl:param name="arcID" as="xs:double"/>

        <!-- Zum target Schritt existieren Sprunganweisungen -->
        <xsl:choose><xsl:when test="ancestor::useCase//step[@goto=$targetID]">
                <!-- Aktionen a. – b.--> ...
                <!-- Aktion c. -->
                <xsl:apply-templates select="ancestor::useCase//step[@goto=$targetID]" mode="connect">
                        <xsl:with-param name="toXorID" select="$xorID"/>
                </xsl:apply-templates>
                <!-- Aktion d. -->
        </xsl:when><xsl:otherwise>
                <!-- Aktion 1. -->
        </xsl:otherwise>
</xsl:template>
```

Enthält ein Schritt eine goto- Anweisung, so muss:

1. eine Funktion *f* zu diesem Schritt erzeugt werden
2. ein Ereignis *e* muss erzeugt werden, welches das erfolgreiche Abarbeiten dieser Funktion repräsentiert
3. *f* muss mit *e* verbunden werden
4. *e* muss mit dem XOR Konnektor verbunden werden, der dem Sprungziel vorangeht (das ist der XOR Konnektor, der durch das Template *connectToNextStep* erzeugt wurde)

Die Ausführung dieser Aktionen sind auf zwei Templates, mit unterschiedlichen mode Attributwerten aufgeteilt, in mode = create wird Aktion 1. ausgeführt, in mode = connect Aktionen 2. – 4.

```
<xsl:template match="step[@goto]" priority="3" mode="create">
        <!-- Aktion 1. -->
</xsl:template>

<xsl:template match="step[@goto]" priority="3" mode="connect">
        <xsl:param name="toXorID"/>
        <!-- Aktion 2. – 4. -->
</xsl:template>
```

Die Aufteilung musste deshalb vorgenommen werden, weil für die Erstellung der Verbindung von der Übersetzung eines Schritts zum Ziel seiner goto Anweisung, die id des XOR Konnektors bekannt sein muss, diese id ist aber bei der Übersetzung des goto Schritts nicht immer bekannt, weil zu dem Zeitpunkt der XOR Konnektor (die Rede ist von dem XOR Konnektor, der im Template *connectToNextStep* erzeugt wird) noch nicht erzeugt wurde. Der XOR Konnektor kann auch nicht bei der Übersetzung eines goto Schritts erzeugt werden, da es mehrere Schritte mit demselben Sprungziel geben kann und der XOR Konnektor nur einmal erzeugt werden muss, die Information über die Erzeugung dieses Konnektors in einer globalen Variablen zu speichern und den Wert dieser Variablen abzufragen ist in XSLT aber leider nicht möglich (da die Werte der Variablen nicht geändert werden können).

Falls ein Schritt mindestens eine Erweiterung aber keine goto- Anweisung enthält, also seine Baumdarstellung so aussieht, wie in der unteren Abbildung rechts dargestellt ist, werden folgende Aktionen durchgeführt um diesen Schritt zu übersetzen:

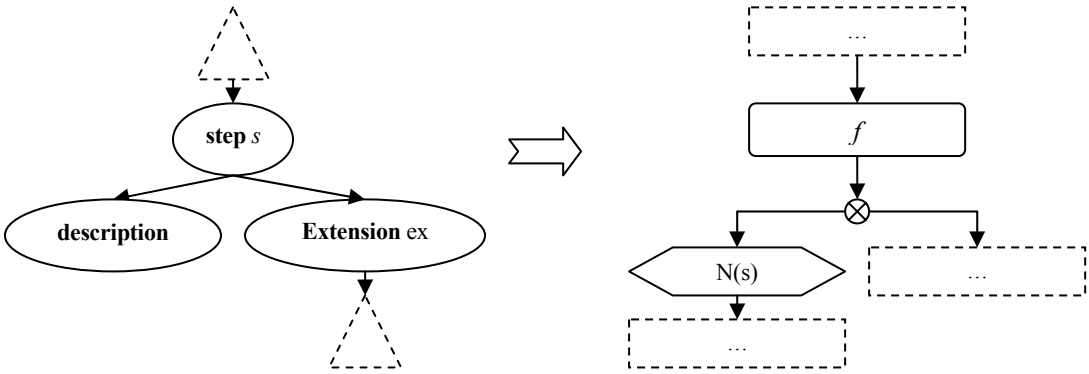


Abbildung 4-14

1. es wird eine Funktion *f* erzeugt mit dem gleichen Namen wie der Name- Attribut des step Elements
2. es wird ein XOR Konnektor *x* erzeugt
3. es wird eine Verbindung von *f* nach *x* hergestellt
4. es wird ein Ereignis *e* erzeugt, das das erfolgreiche Abarbeiten von *f* repräsentiert
5. es wird eine Verbindung von *x* zu diesem Ereignis hergestellt
6. jedes Extension Element wird übersetzt und mit x verbunden

```
<xsl:template match="step[extension]" priority="2" mode="create">
        <!-- Aktionen 1. – 5. --> ...
        <!-- Aktion 6.  -->
        <xsl:apply-templates select="./extension">
                <xsl:with-param name="parentXorID" select="$xorID"/>
        </xsl:apply-templates>
</xsl:template>
```

Das Template zur Übersetzung der extension- Elemente kriegt die id des XOR Konnektors als Parameter übergeben, damit die übersetzten extension- Elemente eine Verbindung dazu aufbauen können. Die Übersetzung der extension- Elemente geschieht wie folgt:

1. es wird ein Ereignis *e* erzeugt mit dem gleichen Namen wie der name- Element des extension Elements
2. *e* wird mit dem XOR Konnektor verbunden, dessen id als Parameter übergeben wurde

3. *e* wird mit dem ersten Schritt des extension- Elements verbunden
4. alle Schritte des extension- Elements werden transformiert

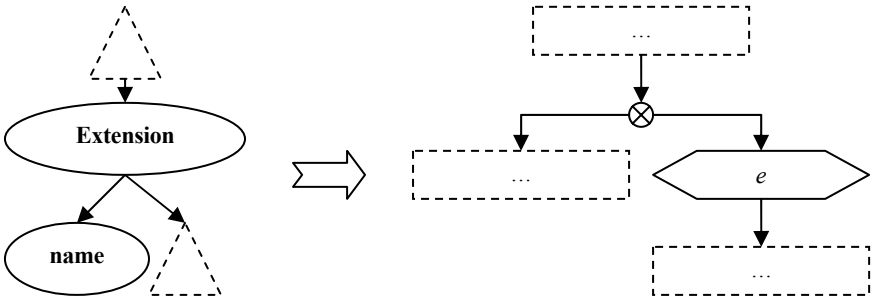


Abbildung 4-15

```
<xsl:template match="extension">
        <xsl:param name="parentXorID"/>
        <!-- Aktionen 1. – 2. --> ...
        <!-- Aktion 3. -->
        <xsl:call-template name="connectToNextStep">...</xsl:call-template>
        <!-- Aktion 4. -->
        <xsl:apply-templates select="./step" mode="create"/>
</xsl:template>
```

Es fehlt noch der Fall wo ein Schritt sowohl eine goto Anweisung als auch mindestens eine Erweiterung enthält, die Übersetzung ist der Übersetzung eines Schritts mit Erweiterungen aber ohne goto, wie sie in Abbildung 4-14 dargestellt wurde sehr ähnlich, der Unterschied besteht nur darin, dass zusätzlich eine Verbindung zum Sprungziel hergestellt werden muss. Falls also ein Schritt mit Erweiterungen und einer goto Anweisung vom XSLT Prozessor gelesen wird, werden folgende Aktionen durchgeführt:

1. es wird ein Ereignis *e* erzeugt, das dengleichen Namen wie der name Attribut des step Elements hat
2. es wird ein XOR Konnektor *x* erzeugt
3. es wird eine Verbindung von *e* nach *x* erstellt
4. es wird eine Verbindung von der Funktion, die die Übersetzung des step Elements darstellt zu *x* erstellt
5. es wird eine Verbindung von *e* zum Sprungziel der goto Anweisung erstellt
6. es werden alle Erweiterungen übersetzt

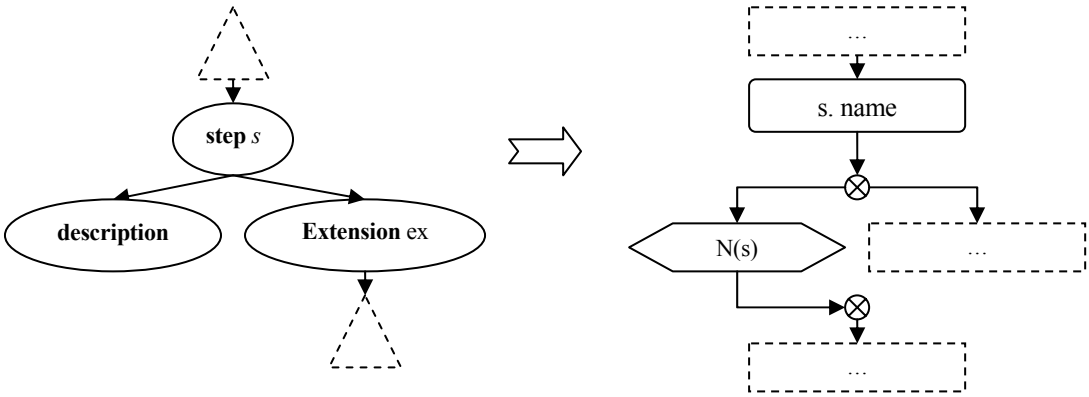


Abbildung 4-16

```
<xsl:template match="step[@goto and extension]" priority="4" mode="connect">
    <xsl:param name="toXorID"/>
    <!-- Aktionen 1.-6. -->
</xsl:template>
```

Bis jetzt wurden die Transformationen der Trigger, der Vorbedingungen, der Schritte und der Erweiterungen vorgestellt, mit der Diskussion der Übersetzung der Nachbedingungen ist dieser Abschnitt komplett.

Eine Nachbedingung entspricht einem Ereignis im EPK- Modell. Die Übersetzung einer Nachbedingung ist jedoch davon abhängig ob diese Nachbedingung von einem oder von mehreren Schritten eines Use Case erfüllt wird. Sei zunächst einmal angenommen, dass die zu übersetzende Nachbedingung von einem Use Case Schritt erfüllt wird, das guarantee Element hat also genau ein stepID Element als Nachfolger, in diesem Fall werden folgende Aktionen durchgeführt:

1. ein Ereignis *e* wird erzeugt, welches dengleichen Namen hat wie das name Element des guarantee Elements
2. Falls nötig werden Prozesswegweiser erzeugt und mit *e* verbunden
3. der stepID Nachfolger wird übersetzt

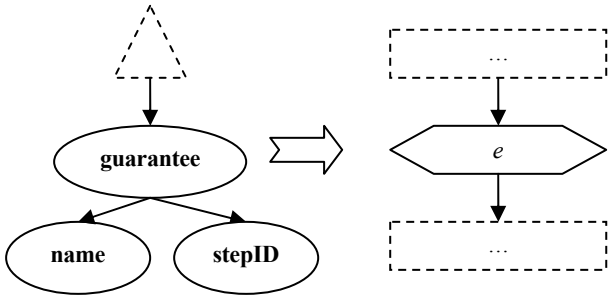


Abbildung 4-17

Sollte ein guarantee Element von mehreren Use Case Schritten erfüllt werden, d.h. es hat mehrere stepID Nachfolger, dann werden folgende Aktionen ausgeführt:

1. ein Ereignis *e* wird erzeugt, welches dengleichen Namen hat wie das name Element des guarantee Elements
2. Falls nötig werden Prozesswegweiser erzeugt und mit *e* verbunden
3. ein XOR Konnektor wird erzeugt und mit *e* verbunden
4. alle stepID Nachfolger werden übersetzt

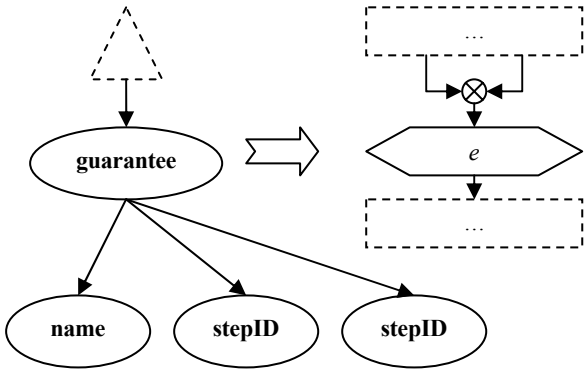


Abbildung 4-18

Diese zwei Fälle werden durch das folgende Template abgedeckt:

```
<xsl:template match="guarantee">
        <!-- 1. -->
        <xsl:variable name="eventID" ...
        <!-- 2. -->
        <xsl:call-template name="genProcessInterfacesToPostCondition">...</xsl:call-template>
        <!-- diese Garantie wird durch mehrere Use Case  Schritte erfüllt -->
        <xsl:choose><xsl:when test="fn:count(./stepID) gt 1">
                <!-- 3. -->
                <xsl:variable name="xorID"...
                <!-- 4. -->
```

```
            <xsl:apply-templates select="./stepID">
                <xsl:with-param name="guaranteeObjectID" select="$xorID"/>
            </xsl:apply-templates>

        <!-- diese Garantie wird durch einen Use Case  Schritt erfüllt -->
        </xsl:when><xsl:otherwise>
            <!-- 4. -->
            <xsl:apply-templates select="./stepID">
                <xsl:with-param name="guaranteeObjectID" select="$eventID"/>
            </xsl:apply-templates>
        </xsl:otherwise>
</xsl:template>
```

Das Template *genProcessInterfacesToPostCondition* erzeugt für eine Nachbedingung *n* die erforderlichen Prozesswegweiser:

1. Falls *n* in keinem anderen Use Case als Vorbedingung vorkommt werden keine Prozesswegweiser erzeugt
2. Falls *n* in genau einem anderen Use Case *u* als Vorbedingung vorkommt wird ein Prozesswegweiser *p* erzeugt, der den Titel von *u* als Bezeichnung hat, und es wird eine Verbindung von *n* zu *p* hergestellt
3. Falls *n* in mehreren Use Cases $u_1 - u_k$ als Vorbedingung vorkommt, wird ein AND Konnektor *a* erzeugt, es wird eine Verbindung von *n* nach *a* generiert, anschließend wird für jedes u_i ein Prozesswegweiser p_i erzeugt der den Titel von u_i als Bezeichnung trägt und es wird eine Verbindung von *a* nach p_i erstellt

Im oberen Template ist die Übersetzung der stepID Elemente durch einen Aufruf eines Templates realisiert, in diesem Template wird eine Verbindung zwischen dem erzeugten Nachbedingungsereignis b.z.w. dem XOR Konnektor und der Funktion der Use Case Schritts, der diese Nachbedingung erfüllt hergestellt.

4.3 Transformation der EPKs

Dieser Abschnitt bietet eine Übersicht über die Implementierung der Abbildung der EPKs auf Use Cases, die verwendeten Algorithmen werden wie im letzten Abschnitt sowohl als Pseudocode als auch graphisch dargestellt. Es wird hier davon ausgegangen, dass die EPKs nach dem Schema aus Abbildung 2-19 aufgebaut sind und keine Benutzereingabe für die Transformation mehr notwendig ist.

Die Syntaxgraphen aus den Abbildung 4-19 - Abbildung 4-23 geben einen Überblick über die vorhandenen Templates zur Transformation der EPKs sowie die Reihenfolge in der diese Templates sich gegenseitig aufrufen.

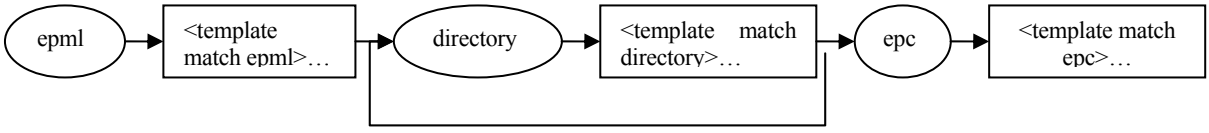


Abbildung 4-19

Der Übersetzungsprozess startet mit dem Root Element jeder EPK, dem *epml* Element, dieses Element kann beliebig viele *directory* Elemente enthalten, jedes *directory* Element enthält andere *directory* Elemente und *epc* Elemente (Abbildung 4-19).

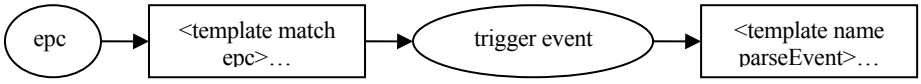


Abbildung 4-20

Die Übersetzung jeder EPK (entspricht dem *epc* Elementen im EPML Format) erfolgt dadurch, dass ihre Knoten (angefangen mit dem Knoten, der als Trigger ausgezeichnet ist) in topologischer Reihenfolge besucht werden, abhängig von dem besuchten Knoten werden dabei die entsprechenden Use Case Elemente erzeugt. Anders als die Use Cases sind die EPKs Graphen (und keine Bäume), die Schleifen enthalten können, damit jeder EPK Knoten nicht mehr als einmal besucht wird, erhält jedes der im Folgenden vorgestellten Templates eine Liste der IDs der EPK Knoten als Parameter, die schon besucht wurden.

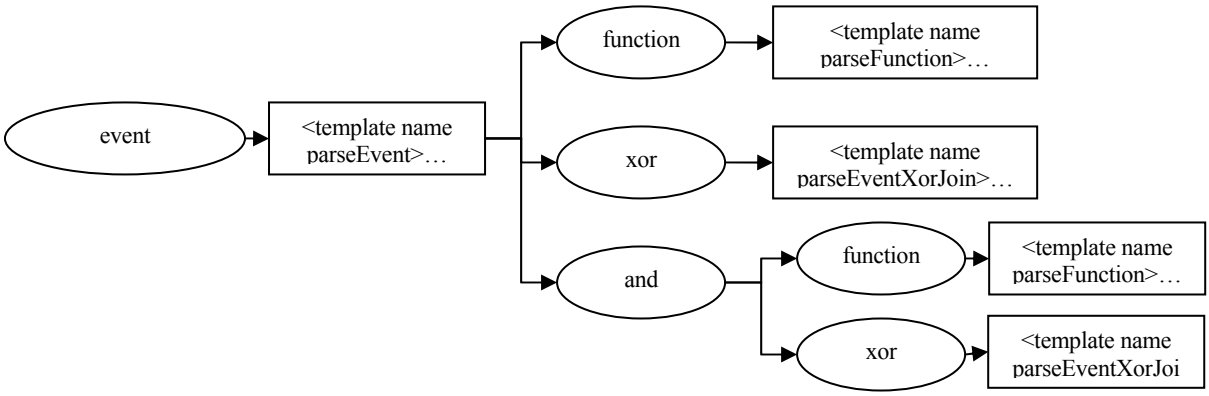


Abbildung 4-21

In einem EPK Graphen, der dem Schema aus Abbildung 2-19 entspricht, folgt einem Ereignis entweder eine Funktion oder ein XOR Konnektor oder ein AND Konnektor oder ein Prozesswegweiser. Da jeder Prozesswegweiser auf eine andere EPK verweist und diese EPK später in einen eigenständigen Use Case transformiert wird, müssen Prozesswegweiser nicht übersetzt werden. Einem AND Konnektor der einem Ereignis nachfolgt kann entweder eine Funktion oder ein XOR Join nachfolgen, was durch die entsprechenden Templates in der oberen Abbildung abgedeckt ist.

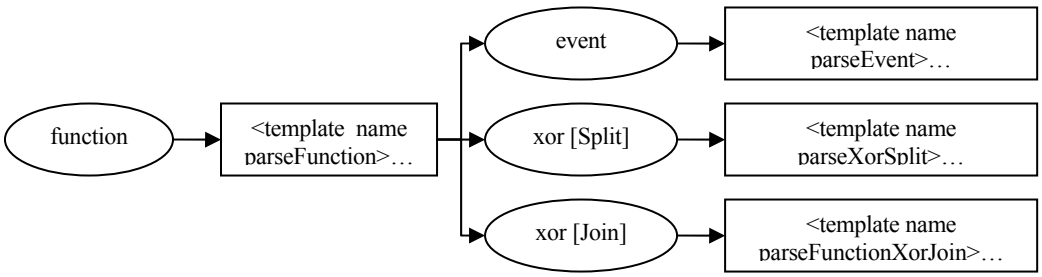


Abbildung 4-22

Nach einer Funktion kann in einem EPK Graphen entweder ein Ereignis oder ein xor Konnektor mit einer eingehenden und mehreren ausgehenden Kanten, also ein xor Split oder ein xor Konnektor mit mehreren eingehenden und einer ausgehenden Kante also ein xor Join auftreten.

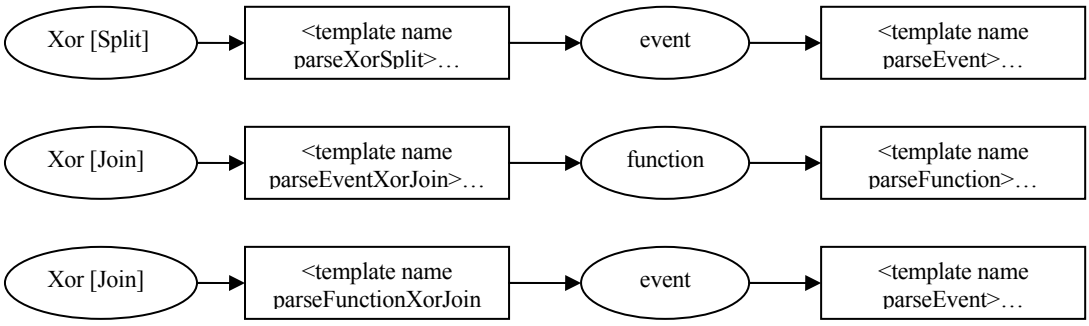


Abbildung 4-23

Nach einem XOR Split können nur Ereignisse im EPK Graphen vorkommen, da ein Xor Split nur einem Funktionsknoten folgen kann. Nach einem Ereignis- XOR Join können nur Funktionen und nach einem Funktionen- XOR Join können nur Ereignisse vorkommen. Aus diesem Zusammenhang ergeben sich die in Abbildung 4-23 dargestellten Syntaxgraphen.

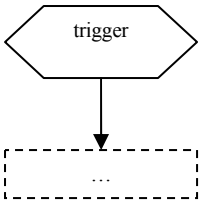
In Kapitel 2 wurde festgestellt, dass für die Transformation der EPKs einige zusätzliche Informationen, benötigt werden, die in den EPKs selbst nicht enthalten sind:

- es müssen diejenigen Ereignisse ausgezeichnet werden, die als Trigger, Vorbedingungen und Erfolgsgarantien übersetzt werden sollen
- bei jedem XOR Split muss ein Ereignis bestimmt werden, welches die Fortsetzung des Standardszenarios darstellt (die anderen Ereignisse markieren jeweils den Anfang einer Use Case Erweiterung).
- bei jedem XOR Join müssen diejenigen Ereignisse angegeben werden, die Sprunganweisungen von einem Erweiterungspfad darstellen.

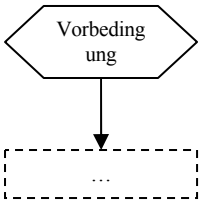
Es gibt zwei Möglichkeiten alle diese Informationen im EPML Format unterzubringen, das Format könnte redefiniert werden oder es könnten einfach die dafür vorgesehenen Elemente von EPML verwendet werden. Der Typ jeden Knotenelements einer EPK ist von dem folgenden Typ abgeleitet.

```
<xs:complexType name="tExtensibleWithAttributes">
                <xs:sequence>
                        <xs:element name="documentation" type="xs:anyType" minOccurs="0"/>
                        <xs:element name="toolInfo" type="xs:anyType" minOccurs="0"/>
                </xs:sequence>
                <xs:anyAttribute namespace="##other" processContents="lax"/>
</xs:complexType>
```

Das Element *toolInfo* dieses Typs ist dazu vorgesehen, zusätzliche nicht unbedingt EPK spezifische Informationen in die EPML Elementen abzulegen. Mit diesem Element werden auch die oben angegebenen Informationen entsprechend folgender Abbildungen abgespeichert.

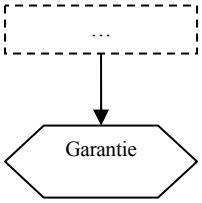



```
<event id="1">
        <name> trigger </name>
        <toolInfo> <trigger/> </toolInfo>
</event>
```



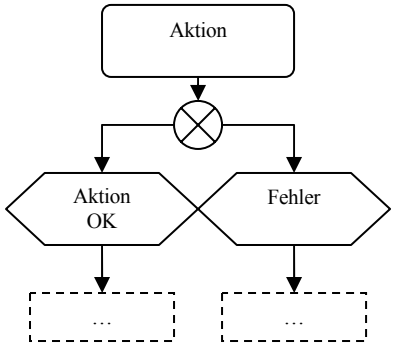

```
<event id="1">
        <toolInfo> <precondition/> </toolInfo>
        <name> Vorbedingung </name>
</event>
```

Falls das Ereignis als Trigger oder Vorbedingung übersetzt werden soll, so enthält das *toolInfo* Element entsprechend ein *trigger*- oder *precondition*- Element.



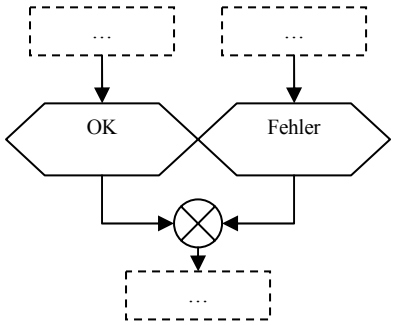

```
<event id="1">
        <toolInfo>
                <guarantee>
                        <stepID value="111"/>
                </guarantee>
        </toolInfo>
        <name> Garantie </name>
</event>
```

Das *value* Attribut des *stepID* Elements verweist dabei auf die Id der Funktion, von der die Garantie erfüllt wird.




```
<event id="1">
        <toolInfo>
                <default/>
        </toolInfo>
        <name> Aktion OK </name>
</event>
```

Dasjenige Ereignis, das keinen Ausnahmefall darstellt, wird mit dem *default*- Element gekennzeichnet.



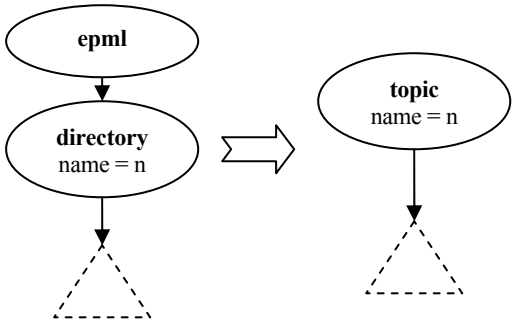

```
<event id="1">
        <toolInfo>
                <goto/>
        </toolInfo>
        <name> Fehler </name>
</event>
```

Ein Ereignis, das das Ende einer Erweiterung darstellt, wird mit dem *goto*- Element gekennzeichnet.

Die oben dargestellten Informationen müssen in jeder EPK, die in ein Use Cases konvertiert wird, vorhanden sein. Sie können entweder manuell eingegeben werden, oder auch mit der dieser Arbeit beigefügten Benutzeroberfläche (siehe Kapitel 6) erzeugt werden. Der Rest dieses Abschnitts befasst sich mit der Diskussion der Funktionsweise der oben angegebenen Templates.

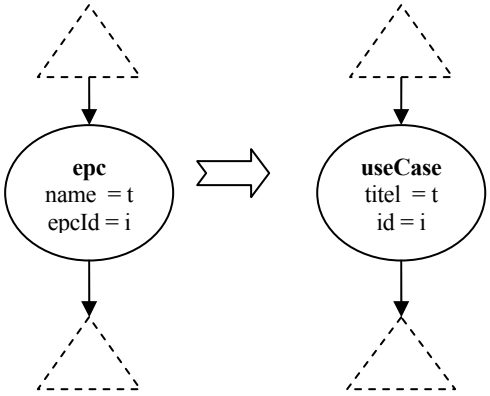
Wie Schon früher festgestellt entspricht das *directory* Element dem *topic* Element der Use Cases, falls also ein *directory* Element vom XSLT Prozessor erkannt wird, werden folgende Aktionen durchgeführt:

1. ein *topic* Element *e* wird erzeugt
2. alle direkten *epc* Nachfolger werden in *useCase* Elemente umgewandelt und unter *e* angehängt
3. Alle direkten *directory* Nachfolger werden in *topic* Elemente umgewandelt und unter *e* anhängt



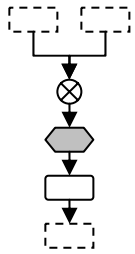
Die Übersetzung des *epc* Elementes erfolgt folgendermaßen:

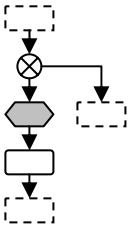
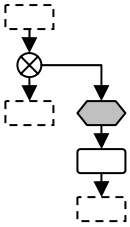
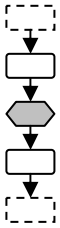
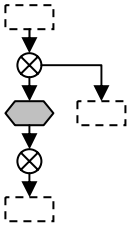
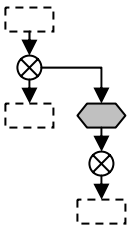
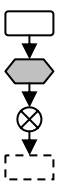
1. Es wird ein *useCase* Knoten *e* erzeugt und die entsprechenden Attributwerte gesetzt
2. Alle als Vorbedingungen markierten Ereignisse werden in die entsprechenden Vorbedingungen umgewandelt und an *e* angehängt
3. Alle als Nachbedingungen markierten Ereignisse werden in die Nachbedingungen umgewandelt und an *e* angehängt
4. Alle Ereignisse und Funktionen mit ihren Konnektoren werden in *useCase* Elemente entsprechend dem in Kapitel 2 vorgestellten Konzept übersetzt und an *e* angehängt. Die Übersetzung jedes *epc* Elementes wird mit dem als Trigger ausgezeichneten Ereignis gestartet.

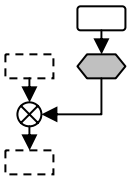
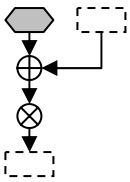
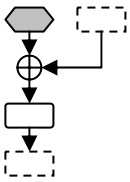


Die Übersetzung eines EPK Ereignisses ist davon abhängig in welchem Kontext dieses Ereignis auftritt, in der folgenden Tabelle sind alle möglichen Konstellationen zusammengestellt. Die Übersetzung der Ereignisse wird mit dem Template *parseEvent* durchgeführt, auf den XSLT-Code wird im folgenden verzichtet, da es sich dabei nur um simple Erzeugung von XML-Knoten handelt.

<xsl:template name="parseEvent">

Konstellation	Transformationsvorschrift
 Der direkte Nachfolger des Ereignisses ist ein Funktionsknoten und der direkte Vorgänger ist ein XOR Join	Es wurde festgestellt, dass nur Ereignisse, die als Trigger, Vor- oder Nachbedingungen und Erweiterungsereignisse ausgezeichnet sind, eine Entsprechung im Use Case Modell besitzen. Das Ereignis im links dargestellten Kontext ist nichts davon, deshalb wird mit der Übersetzung des nachfolgenden Knotens fortgefahren (durch den Aufruf von *parseFunction* Templates auf dem nachfolgenden Funktionsknoten).

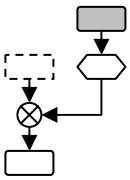
 Der direkte Nachfolger des Ereignisses ist ein Funktionsknoten und der direkte Vorgänger ist ein XOR Split und das Ereignis gehört zum Standardfluss (d.h. es ist kein Ereignis, das eine Use Case Erweiterung definiert)	Da dieses Ereignis ebenfalls keine Entsprechung im Use Case Modell hat (vergleiche auch Abbildung 2-16), wird mit der Übersetzung des nachfolgenden Knotens fortgefahren (Aufruf von *parseFunction* Templates auf dem nachfolgenden Funktionskonten).
 Der direkte Nachfolger des Ereignisses ist ein Funktionsknoten und der direkte Vorgänger ist ein XOR Split und das Ereignis gehört nicht zum Standardfluss (d.h. dieses Ereignis stellte eine Use Case Erweiterung dar)	Dieses Ereignis definiert einen Erweiterungsfall des Use Case. Ein Extension Element *e* wird erzeugt, die Übersetzung (Aufruf von *parseFunction*) des direkten Nachfolgers wird an *e* angehängt. (vergleiche Abbildung 2-14)
 Der direkte Nachfolger des Ereignisses ist ein Funktionsknoten und der direkte Vorgänger ebenfalls	Da Ereignisse zwischen zwei Funktionsknoten keine Entsprechung im Use Case Modell besitzen (Abbildung 2-13), wird hier nur der direkte Nachfolger transformiert und keine weiteren Aktionen durchgeführt.
 Der direkte Nachfolger des Ereignisses ist ein XOR Knoten und der direkte Vorgänger ist ebenfalls ein XOR Knoten und das Ereignis gehört zum Standardfluss	Dieses Ereignis besitzt keine Entsprechung im Use Case Modell (Abbildung 2-16), es wird mit der Übersetzung des Nachfolgers fortgefahren (durch den Aufruf des *parseEventXorJoin* Template).
 Der direkte Nachfolger des Ereignisses ist ein XOR Knoten und der direkte Vorgänger ist ebenfalls ein XOR Knoten und das Ereignis gehört nicht zum Standardfluss	Dieses Ereignis definiert eine Erweiterung. Es wird ein Extension Element *e* erzeugt, der nachfolgende Knoten wird mit dem Aufruf von *parseEventXorJoin* (es kann sich nur um einen Join handeln, da ein Split nach einem Ereignis nicht Erlaubt ist) übersetzt und die Übersetzung wird an *e* angehängt (Abbildung 2-14)
 Der direkte Nachfolger des Ereignisses ist ein XOR Knoten und der direkte Vorgänger ist ein Funktionsknoten und das Ereignis gehört zum Standardfluss	Es wird mit der Übersetzung des Nachfolgers durch den Aufruf von *parseEventXorJoin* (Abbildung 2-16) fortgefahren, da für dieses Ereignis keine Entsprechung im EPK Modell existiert

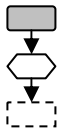
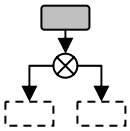
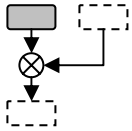
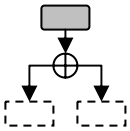
 Der direkte Nachfolger des Ereignisses ist ein XOR Knoten und der direkte Vorgänger ist ein Funktionsknoten und das Ereignis gehört nicht zum Standardfluss (d.h. dieses Ereignis ist Teil einer Erweiterung)	Keine Aktion, der Nachfolgende Xor- Join wurde bereits übersetzt.
 Der direkte Nachfolger des Ereignisses ist ein AND Knoten und der direkte Nachfolger von dem AND Knoten ist ein XOR Knoten	Dieses Ereignis ist ein Trigger oder eine Vorbedingung, der Trigger und die Vorbedingungen wurden schon übersetzt, es wird lediglich der Nachfolger transformiert (durch den Aufruf von *parseEventXorJoin*).
 Der direkte Nachfolger des Ereignisses ist ein AND Knoten und der direkte Nachfolger von dem AND Knoten ist ein Funktionsknoten	Dieses Ereignis ist ein Trigger, er wurde schon übersetzt, es wird lediglich der Nachfolger transformiert (durch den Aufruf von *parseFunction*).

Die Implementierung der oben dargestellten Konstellationen erfolgte durch simple if… then… kaskadierte XPath- Anwesungen innerhalb des *parseEvent* Template. Sämtliche Transformationsskripte befinden sich auf der beigefügten CD und können bei vorhandenem Interesse im Verzeichnis src\ma\model\transform\ressources\xml gefunden werden.

Die Transformation der EPK Funktionen erfolgt durch das Template *parseFunction,* die Muster und die entsprechenden Aktionen sind in der folgenden Tabelle zusammengestellt.

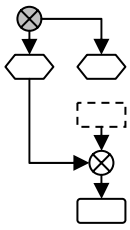
<xsl:template name="parseFunction">

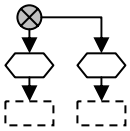
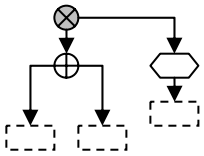
Konstellation	Transformationsvorschrift
 Der direkte Nachfolger der Funktion ist ein Event Knoten, der mit einem goto markiert ist.	Ein Use Case Schritt wird erzeugt, das goto Attribut dieses Schritts wird auf die ID der Funktion gesetzt, die dem XOR Join nach dem Ereignis folgt. Anschließend wird durch den Aufruf des *parseEvent* Templates das Nachfolgerereignis transformiert. Vergleiche auch Abbildung 2-16.

 Der direkte Nachfolger der Funktion ist ein Event Knoten, der nicht mit einem goto markiert ist.	Ein Use Case Schritt wird erzeugt, anschließend wird das nachfolgende Ereignis transformiert (mit dem Aufruf des *parseEvent* Templates). Siehe auch Abbildung 2-13
 Der direkte Nachfolger der Funktion ist ein XOR Split	Das Template *parseXorSplit* (siehe Unten) wird aufgerufen
 Der direkte Nachfolger der Funktion ist ein XOR Join	Das Template *parseFunctionXorJoin* (siehe Unten) wird aufgerufen
 Der direkte Nachfolger der Funktion ist ein AND Konnektor	Diese Funktion erfüllt mehrere Nachbedingungen, die bereits übersetzt wurden. Es wird nur ein Use Case Schritt entsprechend der Abbildung 2-13 erzeugt.

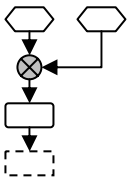
Folgende Tabelle beschreibt die Konstellationen und die dazugehörenden Übersetzungsregeln eins XOR Splits.

<xsl:template name =“parseXorSplit“>

Konstellation	Transformationsregeln
 Das default Ereignis (das Ereignis, das keine Erweiterung ist) dieses Splits ist mit einem goto markiert	Ein Use Case Schritt *s*, mit dem entsprechenden Wert des goto Attributes wird erzeugt. Alle nicht default Ereignisse werden mit dem Aufruf des *parseEvent* Templates transformiert und an *s* angehängt. Der default Ereignis wird mit dem Aufruf von *parseEvent* transformiert. Vergleiche auch Abbildung 2-16

 Das default Ereignis dieses Splits ist nicht mit einem goto markiert	Ein Use Case Schritt *s* wird erzeugt. Alle nicht default Ereignisse werden mit dem Aufruf des *parseEvent* Templates transformiert und an *s* angehängt. Der default Ereignis wird mit dem Aufruf von *parseEvent* transformiert. Vergleiche auch Abbildung 2-16
 Es ist kein default Ereignis angegeben	Die Funktion vor dem Split (es kann nur eine Funktion sein) erfüllt mehrere Nachbedingungen, die mit einem AND verbunden sind und bereits transformiert wurden. Es bleiben nur diejenigen Ereignisse übrig, die mit dem XOR verbunden sind. Es wird also ein Use Case Schritt *s* erzeugt, diese Ereignisse werden transformiert und an *s* angehängt.

Die beiden folgenden Tabellen veranschaulichen die Funktionsweise des *parseEventXorJoin* sowie die des *parseFunctionXorJoin* Templates.

<xsl:template name="parseEventXorJoin">	
Konstellation	Transformationsregeln
	Durch den Aufruf des *parseFunction* Templates wird mit der Transformation des Nachfolgers fortgefahren.

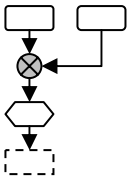
<xsl:template name="parseFunctionXorJoin">	
Konstellation	Transformationsregeln
	Auch in diesem Template wird nichts anderes gemacht als durch den Aufruf des *parseEvent* Templates den Nachfolger zu transformieren

Abbildung 4-24

4.4 User Interface

Dieser Abschnitt bietet eine Übersicht über Implementierung des User Interface zur Erstellung und Transformation der EPKs und Use Cases.

Das Interface wurde in Form einer Java Applikation auf Swing Basis implementiert. Zur Visualisierung der EPK Diagramme wurde eine open source Version der JGraph [JGraph] Bibliothek verwendet, zur Ausführung der XSLT- Transformationen wurde eine open source Version des Saxon [Saxon] Prozessors benutzt. Die Entscheidung fiel auf diese beiden Bibliotheken, weil für beide jeweils eine aktuelle und relativ vollständige Dokumentation vorhanden ist. Im Gegensatz zu dem XSLT- Prozessor, der im JDK eingebaut ist, implementiert der Saxon Prozessor die XSLT 2.0 Spezifiktionen, so dass bei der Programmierung der Transformationen auch auf eine umfangreichere Sammlung von Funktionen zurückgegriffen werden konnte.

Der einfachste Weg zum Starten des Programms besteht in der Ausführung des ant- Skriptes, der in dem src- Verzeichnis auf der mitgelieferten CD enthalten ist. Dazu wird dieses Verzeichnis auf die Festplatte kopiert und auf der Konsole wird in dem kopierten Verzeichnis „ant“ eingegeben. Apache Ant [ApacheAnt] und Java 1.5 [Java] müssen auf dem Rechner installiert sein. Bei der Kompilierung mit dem Java Compiler oder einer anderen Entwicklungsumgebung ist darauf zu achten, dass die mitgelieferten XML- Schemata und XSLT- Skripte in die richtigen Verzeichnisse (siehe ant- Script) kopiert werden, da ohne diese Dateien die Oberfläche nicht funktionieren kann.

Die graphische Oberfläche wurde zum Testen, Visualisieren und Präsentieren der XSLT- und der in Kapitel 2.2 definierten Transformationen (Ersetzung der AND und OR Konnektoren durch die Verschmelzung der entsprechenden EPK Äste) entwickelt. Folgende Funktionalitäten wurden dabei realisiert:

- Die Use Cases, die nach dem XML- Schema aus Kapitel 3.2 aufgebaut sind, können geladen und in tabellarischer Ansicht angezeigt werden.

- Es können neue Use Cases in der Oberfläche angelegt, editiert und abgespeichert werden (in einem Format, der durch das oben erwähnte XML- Schema definiert ist).

- Aus den geladenen Use Cases können EPK Diagramme generiert und dargestellt werden.

- EPKs, die nach dem EPML Schema aufgebaut sind, können geladen, und als Graphen dargestellt werden. Es können auch EPKs in dem Programm angelegt und in dem EPML Format abgespeichert werden.

- Aus geladenen EPKs können Use Case Darstellungen erzeugt werden. Die dazu notwendigen Informationen (Kapitel 2.2) können im Programm eingegeben werden. Sollte eine geladene EPK nicht in ein Use Case transformiert werden können (Kapitel 2.2), so kann diese EPK mit dem Programm so weit verändert werden, dass die Transformation möglich wird. (Vergleiche das Beispiel aus Abbildung 2-27).

Es sollte erwähnt werden, dass die Oberfläche zwar die Möglichkeit bietet EPK Diagramme zu erstellen, als ein vollwertiger EPK Editor kann sie jedoch nicht bezeichnet werden (Dessen Entwicklung war auch nicht Ziel dieser Arbeit). Es fehlt die Möglichkeit hierarchische Funktionen zu erstellen, und die Validierung der erstellten EPK Diagramme auf ihre syntaktische Richtigkeit, wurde im Programm nur rudimentär implementiert. Es bleibt also in der Verantwortung des Benutzers richtige und sinnvolle EPK Diagramme zu erstellen.

Wegen der guten Strukturierungsmöglichkeiten wurde für die Implementierung der Oberfläche das MVC[1] Pattern verwendet. Der Quellcode ist auf drei Haupt- Packages aufgeteilt, das Package model enthält Klassen, die die Struktur der Use Cases und der EPKs nachbilden. Die UML Diagramme aus den Abbildungen Abbildung 1-4 und Abbildung 3-2 werden in diesem Package implementiert. Das Package view enthält Fenster und Dialog Klassen, sowie die auf EPK Diagrame angepassten JGraph- Klassen. In dem Package controller sind die Listener- und Action Klassen der Swing Bibliothek zusammengestellt, die dafür zuständig sind, die Benutzereingabe zu empfangen und entsprechend darauf zu reagieren.

4.5 Zusammenfassung

XSLT ist das Standard Verfahren zur Transformation von XML Dokumenten, da die EPKs in einem XML Format vorliegen und für die Use Cases ebenfalls ein XML Format entwickelt wurde, wurde auch XSLT für die Beschreibung der Transformationen benutzt. Die Transformationen wurden nach der Methode des rekursiven Abstiegs implementiert, d.h. es wurden Templates definiert, die eine bestimmte Ausgabe erzeugen und dann jeweils mit den passenden Parametern und eventuell rekursiv weitere Templates aufrufen.

Die in XSLT beschriebenen Transformationen haben den großen Pluspunkt komplett Plattformunabhängig zu sein. Doch die Verwendung von XSLT anstatt einer Programmiersprache hat auch gewisse Nachteile. XSLT ist eine Scriptsprache und besitzt daher keine Strukturierungsmöglichkeiten wie die Definition von Paketen oder Klassen, die Konstrukte der Vererbung und Polymorphie stehen ebenfalls nicht zur Verfügung, sogar die Definition von Variablen ist in XSLT nicht möglich. Die Implementierung einfacher Transformationen stellt in XSLT kein Problem dar, doch die Implementierung komplexer Sachverhalte kann sich schwierig gestalten und als Resultat sehr große Skripte hervorbringen. Dadurch, dass XSLT- Anweisungen in XML formuliert werden, und deshalb eine enorme Anzahl von auf- und zugehenden Klammern enthalten, wird die Übersichtlichkeit dieser Skripte auch nicht gerade gesteigert.

Die in Kapitel 2 definierten Abbildungen wurden mit XSLT realisiert und diese Realisierung wurde in diesem Kapitel dokumentiert. Im früheren Verlauf dieser Arbeit wurde festgestellt, dass für die Transformation der EPKs Benutzereingaben unumgänglich sind, bei der Implementierung der Transformationen wurde davon ausgegangen, dass die EPKs in einer Form vorliegen, in der sämtliche zur Transformation benötigte Information bereits vorhanden ist. Im Allgemeinen ist eine solche Voraussetzung nicht erfüllt, deshalb wurde eine Benutzeroberfläche entwickelt, mit deren Hilfe alle diese Informationen sich bequem eingeben lassen und die Ergebnisse der Transformation angeschaut werden können. Bei dieser Oberfläche handelt es sich um ein Präsentations- und Testwerkzeug, die Programmierung eines Use Case- und EPK Editors war nicht Ziel dieser Arbeit. Folgendes Kapitel enthält ein Anwendungsszenario in dem auch die Benutzung der Benutzeroberfläche vorgestellt wird.

[1] Model View Controller: eine umfassende Beschreibung ist z.B. bei [Wikipedia] zu finden.

5 Anwendungsszenario

Im Verlauf dieser Arbeit wurde ein Konzept zur Abbildung zwischen EPKs und Use Cases sowie dessen Realisierung vorgestellt, sowohl die Use Cases als auch die EPKs liegen dabei im XML- Format vor. Obwohl XML sich heutzutage als Standard Datenaustauschformat durchgesetzt hat, sind die XML- Dokumente für den Menschen nur mit Mühe zu lesen und zu verstehen. Um die Transformationsergebnisse der Use Cases bzw. der EPKs besser nachvollziehen zu können wurde eine graphische Benutzeroberfläche entwickelt, mit der sich beliebige EPKs und Use Cases erstellen und transformieren lassen ohne auch nur eine Zeile XML schreiben oder lesen zu müssen. In diesem Kapitel werden einige Einsatzmöglichkeiten und Vorzüge der oben genannten Transformationen vorgestellt, dieses Kapitel stellt auch eine Benutzeranleitung für die Benutzeroberfläche dar.

5.1 Anlegen und Transformieren der Use Cases

Im Folgenden wird dargestellt, wie mit dem Programm Use Cases erstellt und transformiert werden können und welche Vorteile sich davon ergeben. Seien zunächst die vier in den Abbildung 5-1 und Abbildung 5-2 dargestellten Use Cases gegeben. Durch diese Use Cases werden die Aktionen, die für die Durchführung einer Abschlussarbeit an einer Universität notwendig sind beschrieben. Die Beschreibung erfolgt immer nur aus der Sicht desjenigen, der an diesen Aktionen beteiligt ist. So beschreiben die Use Cases 1- 2 das, was ein Student zu tun hat, die Aufgaben eines Prüfungsamtmietarbeiters und des Betreuers werden mit den Use Cases 3- 4 beschrieben. Der Vorteil einer Anforderungsbeschreibung aus der Sicht des Hauptakteurs besteht darin, dass für ihre Erstellung nur dieser eine Akteur interviewt werden muss, die anderen werden die Beschreibung nicht durch unnötige Details negativ beeinflussen können.

1

Titel	Student meldet sich für seine Abschlussarbeit an
Akteur	Student
Stakeholder	Student: Möchte sich unkompliziert und schnell anmelden. Prüfungsamtmietarbeiter: Möchte korrekt ausgefüllte Formulare.
Erfolgsgarantie	Die Anmeldung ist abgegeben
Trigger	Student möchte eine Abschlussarbeit anfertigen
Szenario	1. **Ausfüllen der Formulare:** Student besorgt sich die Anmeldeformulare seines Prüfungsamtes und füllt sie aus. 2. **Formulare Abgeben:** Der Student gibt die ausgefüllten Formulare in seinem Prüfungsamt ab

2

Titel	Student sucht ein Thema
Akteur	Student
Stakeholder	Student: Möchte ein interessantes Thema
Erfolgsgarantie	Student hat ein interessantes Thema gefunden
Trigger	Student möchte eine Abschlussarbeit anfertigen
Szenario	1. **Angebote durchsuchen:** Der Student durchsucht die Angebotenen Themen, dazu verwendet er das Internet und die Aushangstafeln der Institute 2. **Thema auswählen:** Der Student wählt ein Thema aus, das er interessant und ansprechend findet 3. **Betreuer fragen:** Der Student gibt dem Betreuer über sein Interesse bescheid.

Abbildung 5-1

	3
Titel	Prüfungsamt bestätigt die Anmeldung
Akteur	Prüfungsamtmietarbeiter
Stakeholder	Prüfungsamtmietarbeiter: Möchte korrekt ausgefüllte Formulare.
Erfolgsgarantie	Student kann die Arbeit beginnen
Invariante	Die Daten des Studenten werden entsprechend den Datenschutz-vorschriften behandelt
Trigger	Die Anmeldung ist abgegeben
Szenario	1. **Überprüfung der Anforderungen:** Prüfungsamtmietarbeiter überprüft ob der Student 80% der erreichbaren CPs erreicht hat. 2. **Bestätigung der Anmeldung:** Prüfungsamtmietarbeiter bestätigt die Anmeldung
Erweiterungen	1.1 **Nicht genug CPs:** Student hat weniger als 80% der erreichbaren CPs 1.1.1 **Anmeldung verweigern:** Prüfungsamtmietarbeiter verweigert die Anmeldung

	4
Titel	Betreuer gibt die Aufgabenstellung aus
Akteur	Betreuer
Stakeholder	Betreuer: Möchte kein Papierkrieg
Erfolgsgarantie	Student hat sein Thema erhalten
Vorbedingungen	1. Student hat sich für ein Thema entschieden
Trigger	Kein
Szenario	1. **Thema Ausgabe:** Betreuer händigt dem Studenten die genau Aufgabenstellung in Schriftlicher Form aus. 2. …

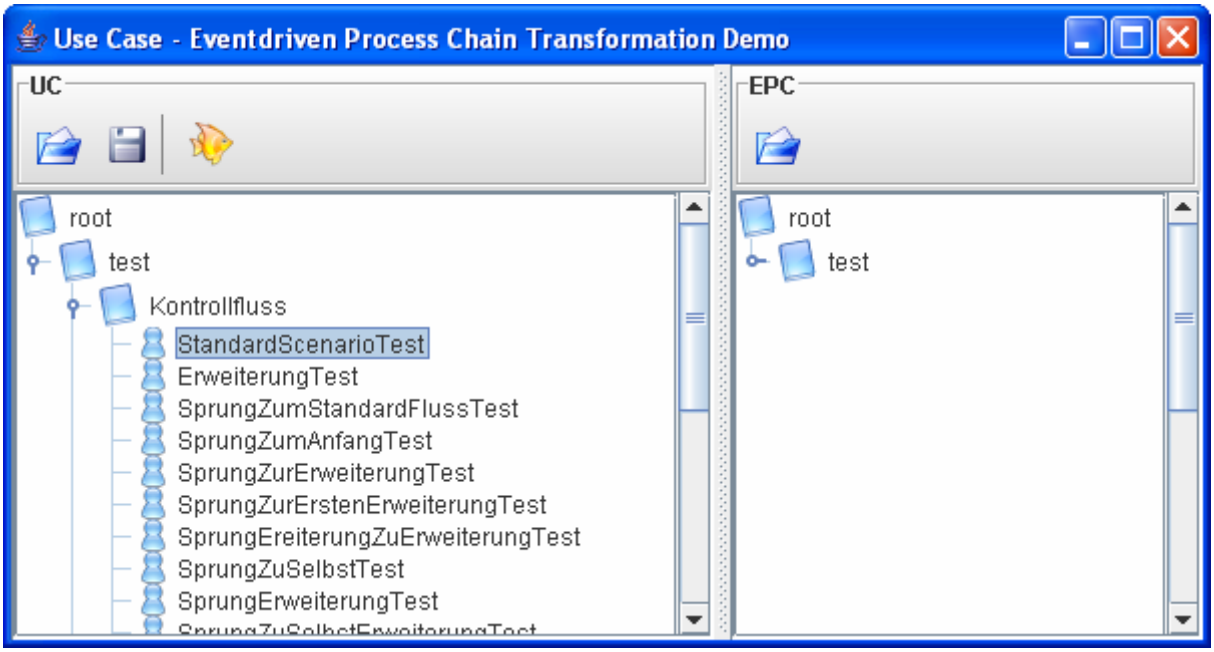
Abbildung 5-2

Jeder Use Case beschreibt Anforderungen aus einer anderen Sicht. Es kann passieren, dass die Anforderungen, die aus unterschiedlichen Sichten aufgenommen wurden und in unterschiedlichen Use Cases stehen zusammenhängen. Dieser Zusammenhang wird jedoch wegen der lokale Sicht, die Use Cases bieten nicht erkannt. So ist z.B. durch das Betrachten der vier oben angegeben Use Cases nicht sofort klar welche Use Case Aktionen unabhängig von welchen anderen Aktionen und welche Aktionen nur in einer bestimmten Reihenfolge ausgeführt werden können. Falls es eine bestimmte Reihenfolge gibt, so ist auch nicht sofort zusehen welche Bedingungen in welchen Use Cases diese Reihenfolge erzwingen. Daher ist es auch schwer nachzuvollziehen ob alle diese Bedingungen richtig und vollständig festgehalten wurden.

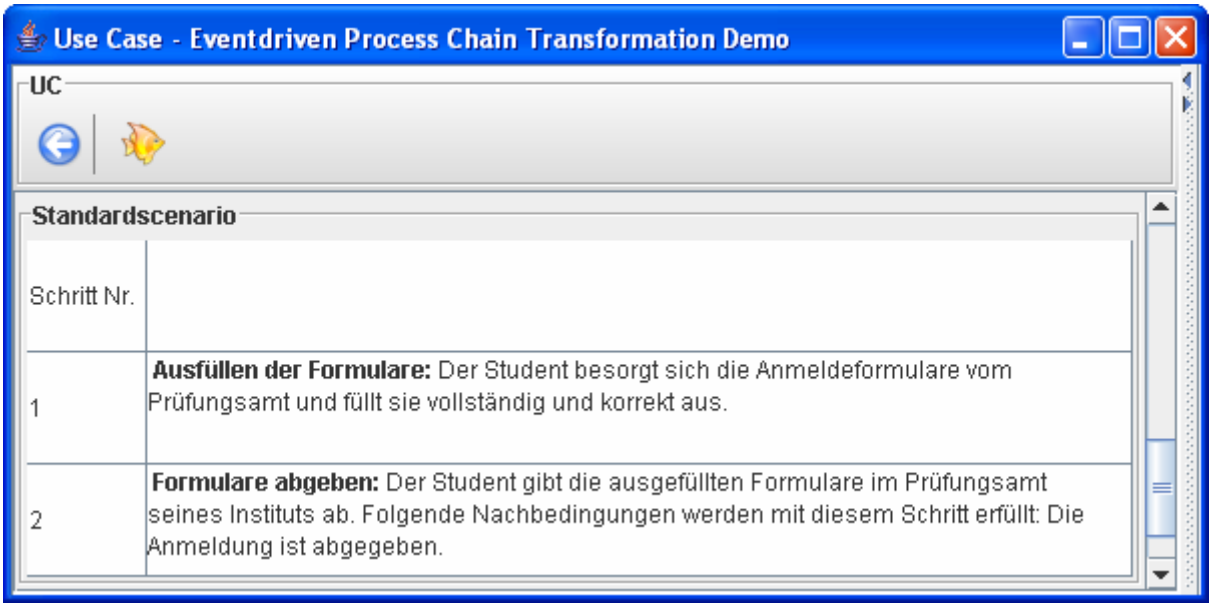
Jeder Use Cases bietet eine fokussierte Sicht auf einen kleinen Ausschnitt der Anforderungen, bereits bei 4 sehr kleinen Use Cases, die dazu auch noch nebeneinander aufgestellt sind und auf eine Seite passen ist eine globale Übersicht nur sehr schwer zu gewinnen. Als Folge davon können Fehler und Mehrdeutigkeiten in den Anforderungsbeschreibungen auftreten. Zusätzlich zu einer fokussierten lokalen kann also auch eine globale Sicht wertvoll sein, wie sie z.B. von den EPK- Diagrammen geboten wird.

Für die Erzeugung der EPK- Diagramme muss zunächst die Benutzeroberfläche gestartet und die Use Cases geladen werden. Das Kompilieren und Starten des Programms ist in Kapitel 4.4 beschrieben.

Nach dem das Programm gestartet ist, erscheint ein Fenster, das wie folgt aussieht. Im linken Teil werden Use Cases dargestellt und bearbeitet im rechten die EPKs. Die Testfälle werden beim Start automatisch geladen, so dass der Benutzer weder neue Use Cases noch EPKs anlegen muss um sich ein Bild von der Funktionsweise des Prototyps zu machen.

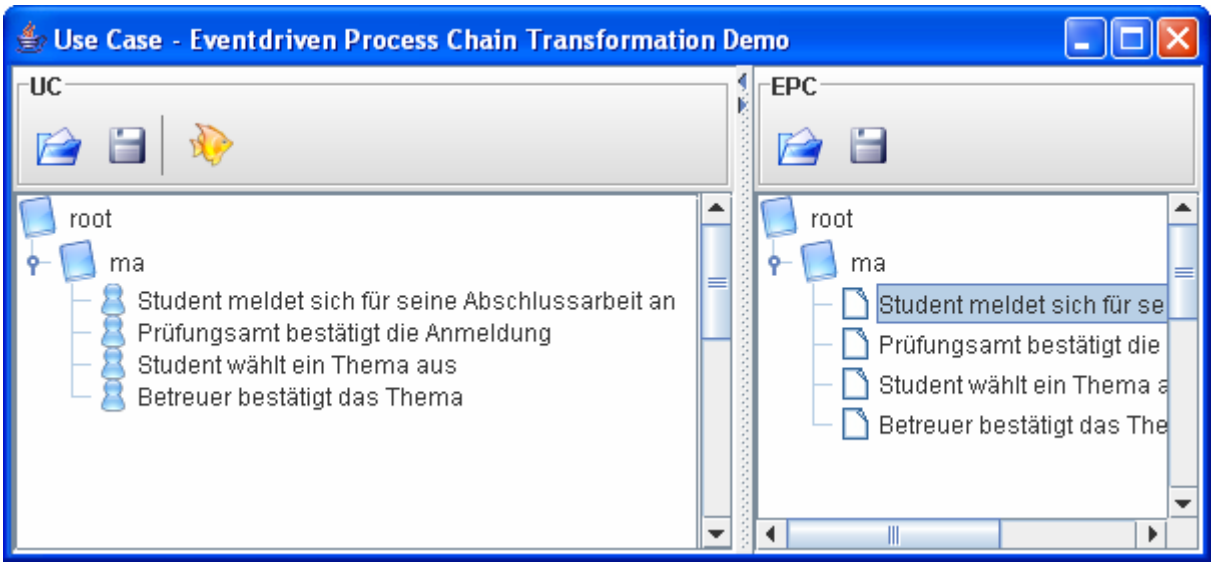


Zum Anlegen eines neuen Use Case wird im rechten Fensterteil ein Verzeichnis ausgewählt und mit der rechten Maustaste angeklickt, daraufhin erscheint ein Popup Menu, in dem die Erzeugung eines neuen Use Case als Option zur Verfügung steht. Durch einen Doppelklick auf einen angelegten Use Case wird die Tabellenansicht des Use Case dargestellt, wie sie in folgender Abbildung skizziert ist.

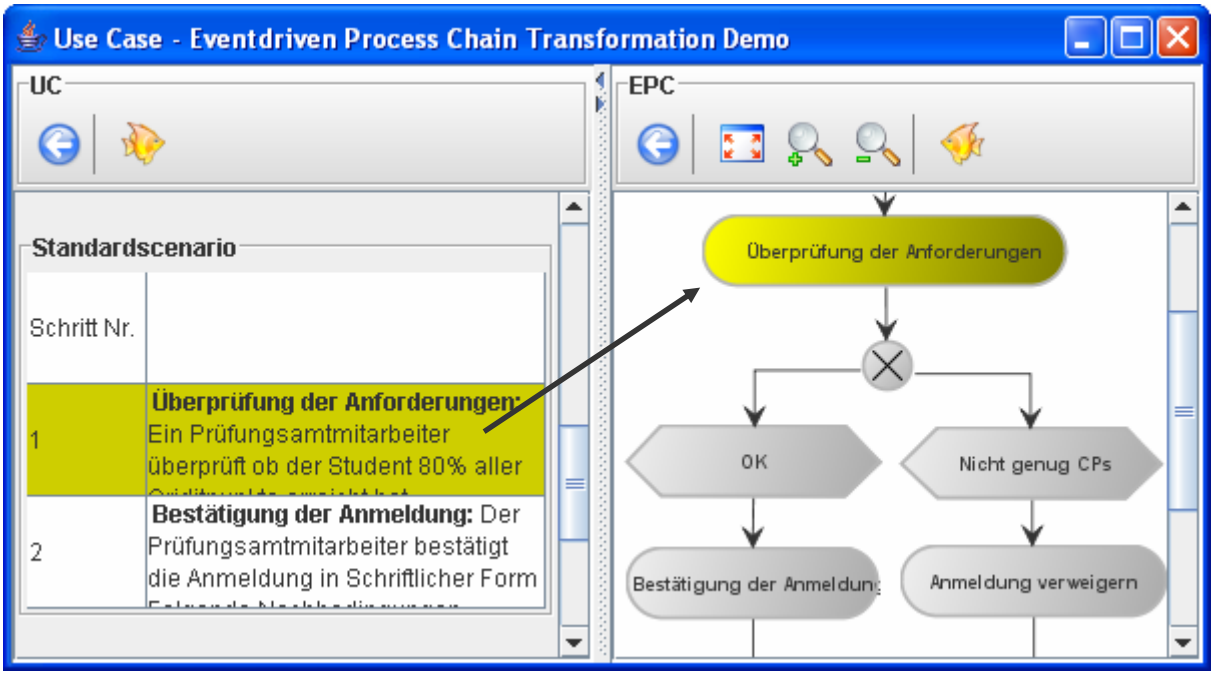


Durch einen Doppelklick auf eine Zeile, können die entsprechenden Attribute wie z.B. der Aktions- Name oder Beschreibung editiert werden. Durch einen Rechtsklick auf eine Zeile erscheint ein Popup Menu mit den Optionen neue Zeilenelemente anzulegen oder zu löschen. Nach dem die Use Cases aus den Abbildung 5-1 und Abbildung 5-2 angelegt sind, können durch das Betätigen des Übersetzungsbuttons auf der oberen Taskleiste die EPK Diagramme

generiert werden. Der Übersetzungsbutton (Fischform) muss aus der Baumansicht betätigt werden, falls die EPK Diagramme über Prozesswegweiser entsprechend den Use Cases verbunden werden sollen. Nach dem Übersetzungsprozess werden im rechten Fensterteil die EPKs in derselben Verzeichnisstruktur abgebildet.



Durch einen Doppelklick auf eine EPK kann deren graphische Darstellung betrachtet werden. Beim Anklicken der Use Case Elemente in der tabellarischen Sicht, werden die äquivalenten Elemente der EPK graphisch hervorgehoben, so lässt sich gut nachvollziehen, welche Aktionen, Vor- oder Nachbedingungen in welche EPK Ereignisse oder Funktionen umgesetzt wurden.



In der folgenden Abbildung sind die Übersetzungen der vier Use Cases so wie sie vom Prototyp erzeugt werden zusammengestellt. Die Prozesswegweiser über die die EPKs verknüpft werden, wurden dabei durch die EPKs auf die sie verweisen substituiert (Doppelklick auf den Prozesswegweiser), so kann der globale Kontrollfluss, der über mehrere Use Cases geht übersichtlich in einem Diagramm dargestellt werden.

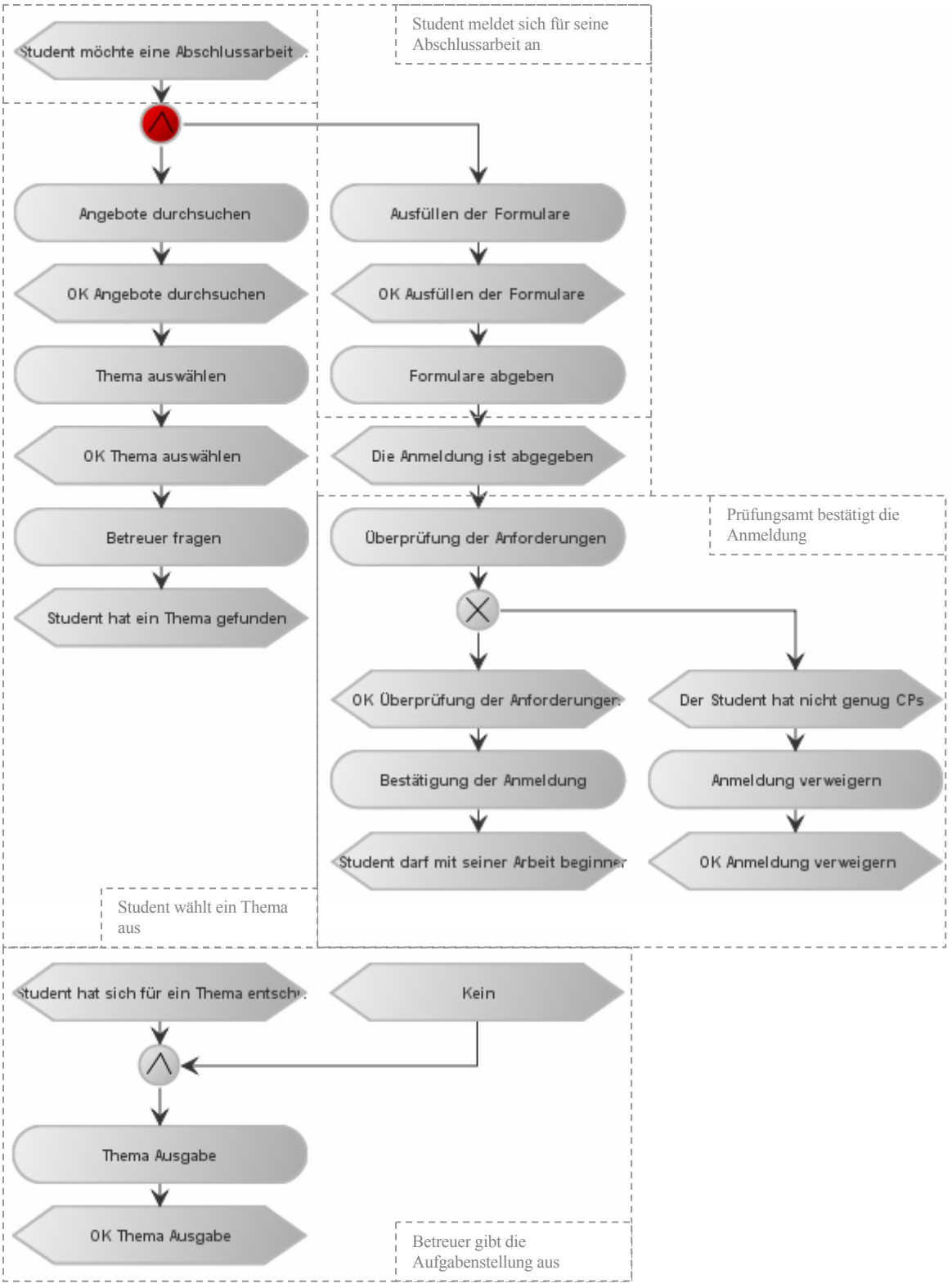


Abbildung 5-3

Bei der Betrachtung der EPK Diagramme aus Abbildung 5-3 können sofort folgende Abhängigkeiten festgestellt werden:

- Die Bedingung „Student möchte eine Abschlussarbeit beginnen“ löst sowohl den Use Case „Student meldet sich für seine Abschlussarbeit an“ als auch den Use Case „Student wählt

ein Thema aus“. Eine Änderung dieser Bedingung in einem der beiden Use Cases hätte eine Auswirkung auf den Kontrollfluss.

- Die Aktionen des Use Case „Prüfungsamt bestätigt die Anmeldung“ werden erst ausgeführt nach dem der Use Case „Student meldet sich für seine Abschlussarbeit an“ abgearbeitet wurde. Die Bedingung “Die Anmeldung ist abgegeben“ ist eine Erfolgsgarantie in einem und Trigger im anderen Use Case.

Bei der weiteren Betrachtung des EPK Diagramms, kann leicht festgestellt werden, dass die Aktionen des Use Case „Betreuer gibt die Aufgabenstellung aus“ ganz unabhängig von den anderen drei Use Cases abgearbeitet werden. Da die Aufgabenstellung erst ausgegeben werden kann nach dem der Student sich für ein Thema entschieden hat und nachdem das Prüfungsamt bestätigt hat, dass der Student beginnen kann, handelt es sich hier eindeutig um einen Fehler. Obwohl die Beiden Bedingungen „Student hat ein Thema gefunden“ im Use Case „Student wählt ein Thema aus“ und „Student hat sich für ein Thema entschieden“ im Use Case „Betreuer gibt die Aufgabenstellung aus“ unterschiedlich benannt sind, muss es sich hier um ein und dieselbe Bedingung handeln.

Auffallend ist auch das Ereignis, das die Bezeichnung „kein“ trägt, offensichtlich wurde im Use Case „Betreuer gibt die Aufgabenstellung aus“ die Eingabe des Triggers vergessen. Die Tatsache, dass die Aufgabenstellung erst dann ausgegeben werden kann, wenn das Prüfungsamt bestätigt, dass der Student seine Abschlussarbeit schreiben darf, spricht dafür die Erfolgsgarantie des Use Case „Prüfungsamt bestätigt die Anmeldung“ hier als Vorbedingung oder Trigger zu definieren.

Nach dem alle entdeckten Mängel an den vier Use Cases beseitigt sind, kann die Transformation erneut ausgeführt werden, das Ergebnis ist in der Abbildung 5-4 dargestellt.

Es ist ganz offensichtlich, dass die Abhängigkeiten und Fehler in den Anforderungen sehr viel leichter entdeckt werden können, wenn zusätzlich zu den Use Cases auch noch Kontrollflussgraphen, z.B. in Form von EPKs vorliegen oder automatisch generiert werden können. Neben der Validierung der Anforderungen kann die automatische Erzeugung der EPK-Diagramme aus Use Cases auch in folgenden Fällen nützlich sein:

- So genannte Service Orientierte Architekturen (SOA) sehen eine Menge voneinander unabhängiger Dienste vor. Die Logik der Anwendungssysteme wird durch die Aneinanderreihung dieser Dienste komponiert. Das Design der SOA Systeme beinhaltet also auch Prozessbeschreibungen, die automatische Generierung dieser Beschreibungen aus den Anforderungen spart Zeit und ist nicht so Fehleranfällig wie die manuelle.

- Für Use Cases existieren zahlreiche Schablonen, sowie Tipps und Tricks zu ihrer Erstellung. Ein Verfasser von Geschäftsprozessen könnte in den Interviews von allen diesen Methoden profitieren, in dem er zunächst Use Cases verfasst und daraus dann automatisch die Geschäftsprozesse erzeugt

- Die Visualisierung der Anforderungen kann schließlich gut für Präsentations- und Kommunikationszwecke genutzt werden.

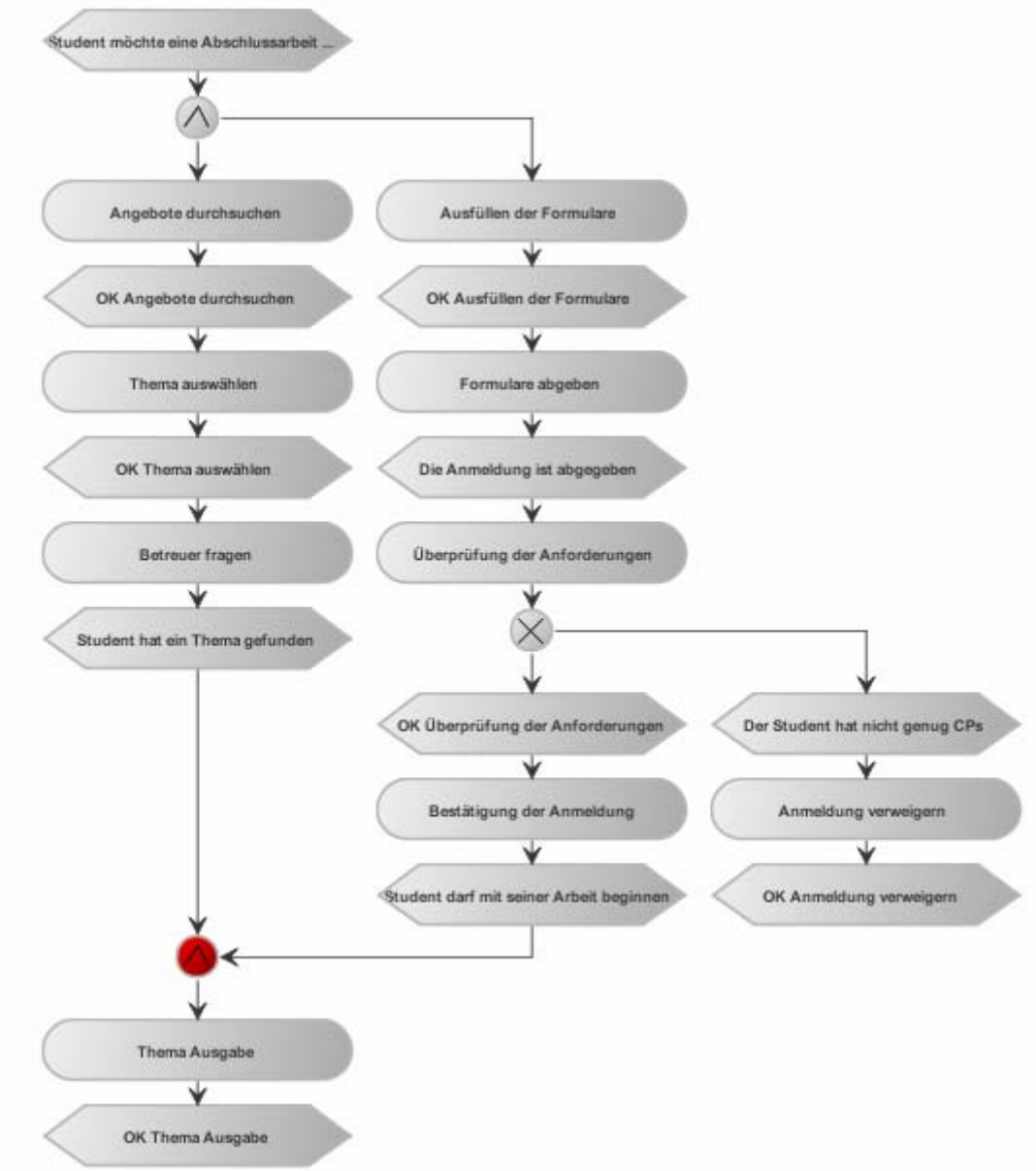


Abbildung 5-4

5.2 Anlegen und Transformieren der EPKs

In diesem Abschnitt wird präsentiert wie EPKs angelegt und in Use Cases transformiert werden können, es wird auch auf die Vorzüge solcher Transformation eingegangen. Im letzten Abschnitt wurde die Transformation der Use Cases, die die Anmeldeprozedur für eine Abschlussarbeit an einer Universität beschreiben, vorgestellt. Sei nun umgekehrt diese Anmeldeprozedur durch die vier folgenden EPKs modelliert.

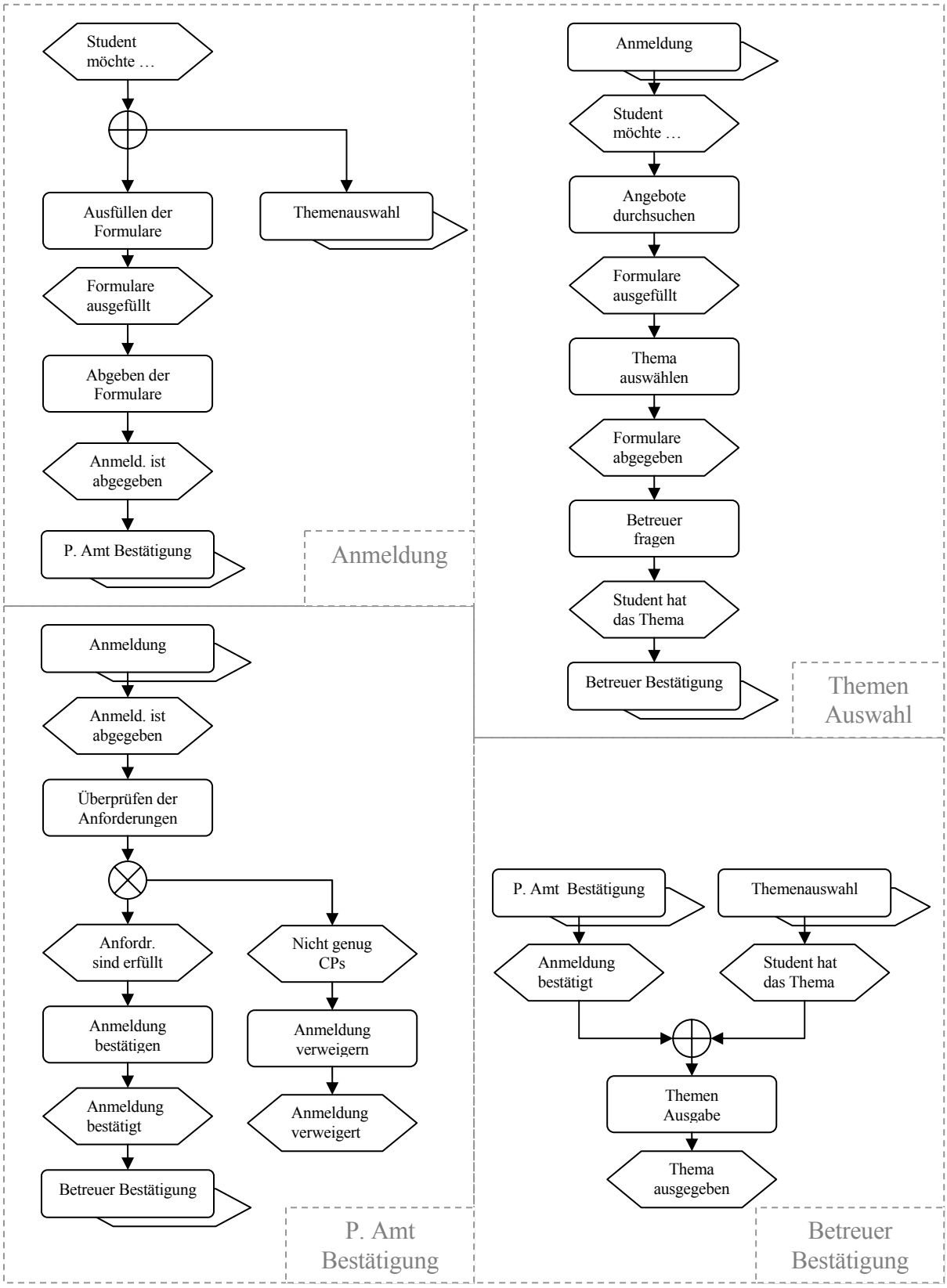
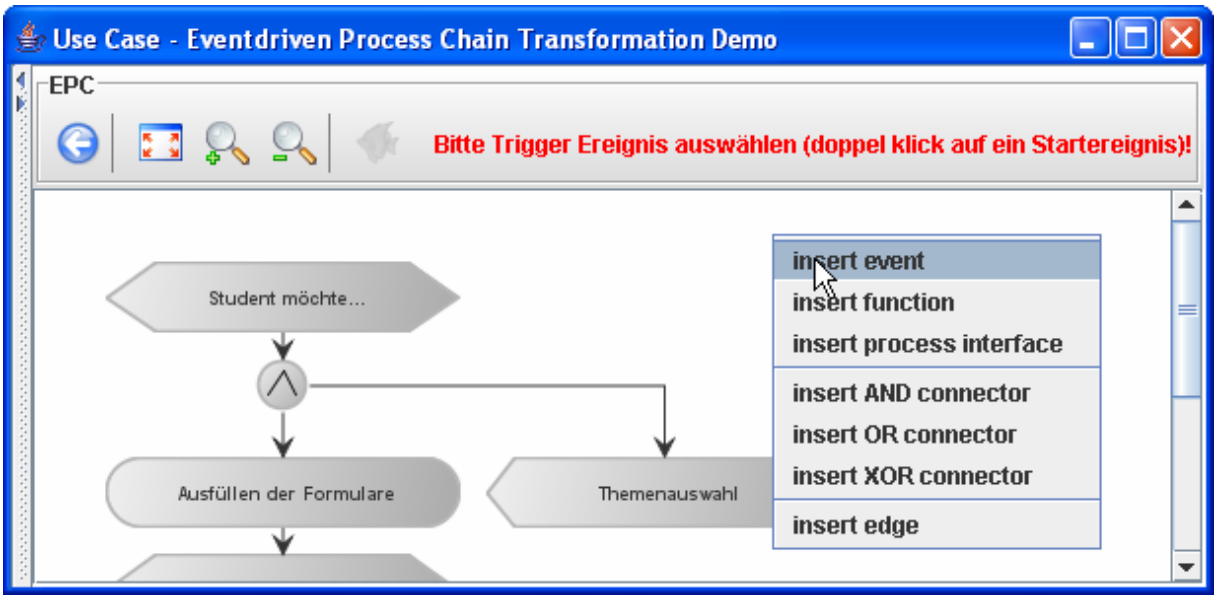


Abbildung 5-5

Für die Erzeugung der entsprechenden Use Cases ist es zunächst erforderlich diese EPKs ins EPML Format zu übertragen, alternativ dazu kann die Benutzeroberfläche gestartet und die Diagramme dort eingegeben werden.

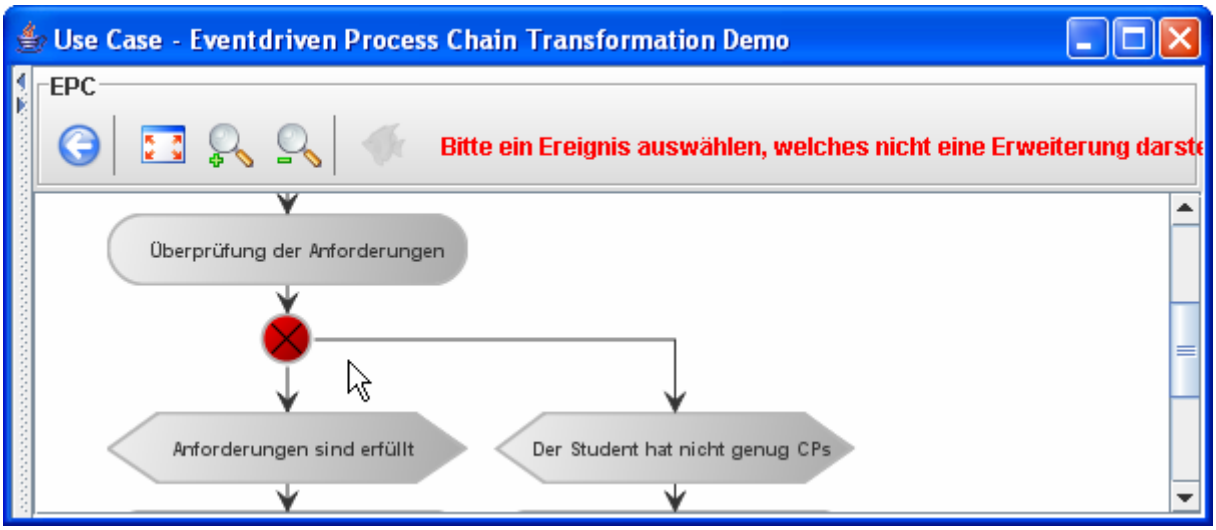
Das Anlegen einer neuen EPK funktioniert exakt auf die gleiche Weise, wie auch das Anlegen eines Use Case, was im letzten Abschnitt vorgestellt wurde. Nach dem also eine EPK in der Verzeichnissahnsicht angelegt wurde, wird mit einem Doppelklick auf sie in die graphische Ansicht gewechselt. Hier können die EPK Elemente, wie Ereignisse, Funktionen und Prozesswegweiser angelegt und durch Kontrollflusskanten miteinander verbunden werden. Durch das Drücken der rechten Maustaste erschein ein Popupmenu, das die dazu notwendigen Optionen zur Verfügung stellt.



Falls die erstellte EPK den semantischen Regeln nicht genügt, oder bestimmte Eingaben, die zur Transformation ins Use Case Modell notwendig sind, nicht gemacht wurden, wird in der oberen Taskleiste eine Fehlermeldung angezeigt, die darauf hin weist. So muss z.B. für die Übersetzung in ein Use Case mindestens ein Triggerereignis ausgewählt werden. Dazu wird ein Startereignis zweimal angeklickt, daraufhin erschein ein Dialog wo die Attribute des Ereignisses editiert werden können, hier kann auch eingegeben werden, ob das Ereignis als Trigger oder Vorbedingung oder nichts von beiden übersetzt werden soll.

Zu beachten ist auch, dass nicht nur Trigger und Vorbedingungen unter den EPK Ereignissen markiert werden müssen, sondern auch die Erfolgsbedingungen. So wird bei der EPK „Anmeldung“ das Ereignis „Anmeldung ist abgegeben“ als Erfolgsbedingung markiert, bei den EPKs „P. Amt Bestätigung“, „Themenauswahl“ und „Betreuer Bestätigung“ sind es die Ereignisse „Anmeldung bestätigt“, „Student hat das Thema“ und „Thema ausgegeben“.

Wie aus Kapitel 2 bekannt, müssen bei jedem XOR Split diejenigen Ereignisse ausgewählt werden, die als Use Case Erweiterungen übersetzt werden sollen, oder was dazu äquivalent ist, dasjenige Ereignis, das nicht als eine Erweiterung übersetzt werden soll. Die EPK „P. Amt Bestätigung“ enthält so ein XOR Split. Nach der Eingabe dieser EPK wird der XOR Konnektor rot markiert und der Benutzer dazu aufgefordert ein nicht- Erweiterungsereignis auszuwählen. Durch einen Doppelklick auf den markierten Konnektor wird ein Dialog geöffnet, in dem die Erforderliche Auswahl vorgenommen werden kann, das Ereignis „nicht genug CPs“ gehört zu einer Erweiterung, das Ereignis „Anforderungen sind Erfüllt“ muss also ausgewählt werden.



Nach dem alle vier EPKs fehlerfrei eingegeben wurden, kann der Transformationsprozess durch das Betätigen des gelben Übersetzungsbuttons in der oberen Taskleiste gestartet werden. Das Ergebnis dieses Prozesses sind die in der folgenden Abbildung dargestellten Use Cases. (Auf die Screenshots der Benutzeroberfläche wurde diesmal verzichtet, weil diese weniger kompakt sind und deshalb nicht so übersichtlich dargestellt werden können).

1	
Titel	Anmeldung
Akteur	
Stakeholder	
Erfolgsgarantie	Die Anmeldung ist abgegeben
Trigger	Student möchte eine Abschlussarbeit anfertigen
Szenario	1. **Ausfüllen der Formulare:** 2. **Formulare Abgeben:**

2	
Titel	Themen Auswahl
Akteur	
Stakeholder	
Erfolgsgarantie	Student hat das Thema
Trigger	Student möchte eine Abschlussarbeit anfertigen
Szenario	1. **Angebote durchsuchen:** 2. **Thema auswählen:** 3. **Betreuer fragen:**

3	
Titel	P. Amt Bestätigung
Akteur	
Stakeholder	
Erfolgsgarantie	Anmeldung bestätigt
Trigger	Die Anmeldung ist abgegeben
Szenario	1. **Überprüfung der Anforderungen:** 2. **Bestätigung der Anmeldung:**
Erweiterungen	**1.1 Nicht genug CPs:** 1.1.1 **Anmeldung verweigern:**

4	
Titel	Betreuer Bestätigung
Akteur	
Stakeholder	
Erfolgsgarantie	Student hat sein Thema erhalten
Vorbedingungen	1. Student hat das Thema 2. Anmeldung bestätigt
Trigger	Anmeldung bestätigt
Szenario	1. **Thema Ausgabe:**

Abbildung 5-6

Es fällt sofort auf, dass diese Use Cases erstens unvollständig sind zweitens grobe Verletzungen der Regeln enthalten, die beim Verfassen der Use Cases befolgt werden sollten:

- Der Titel eines Use Case sollte eine Zielformulierung enthalten, welche das Ziel des Primärakteurs nennt, bei allen Vier Use Cases ist das nicht der Fall.

- Die Primärakteure, die Stakeholder mit ihren Interessen sowie die Angaben zum Umfang und Ebene fehlen. Dadurch bleibt es gänzlich unklar welche Akteure welche Ziele verfolgen und wie sie diese Ziele erreichen können.

- Die Use Case Schritte sind nicht als Ziele formuliert, und nicht mal als richtige Sätze, die Absichten der Akteure werden nicht deutlich.

- usw.

Die vielen Regelverstöße sind offensichtlich, sogar für diejenigen, die nur wenig Erfahrung mit dem Schreiben von Use Cases haben.

Dieses Beispiel bestätigt, nur das, was von vorneherein mehr oder weniger klar war: es können keine hinreichend genauen Anforderungsbeschreibungen aus etwas erzeugt werden, das die nötigen Informationen überhaupt nicht enthält. Wie man leicht feststellen kann, enthalten die EPKs aus Abbildung 5-5 weder Angaben zu den Primärakteuren noch zu den Stakeholdern und ihren Interessen, und die Formulierungen in den generierten Use Cases sind auch exakt dieselben, wie in den EPKs. Es bleibt also nur der Schluss: Durch die Transformation der EPKs in Use Cases mit einem Computerprogramm können keine Anforderungsbeschreibungen gewonnen werden, die ohne manuelle Nachbearbeitung von Nutzen wären.

Es wurde bereits festgestellt, dass die vielen Regelverstöße und das Nichtvorhandensein von notwendigen Informationen in den computergenerierten Use Cases offensichtlich sind. Dieselben Regelverstöße und die fehlenden Informationen sind natürlich auch in den EPK-Diagrammen enthalten[1], es stellt sich nun die Frage ob sie auch ohne die computergenerierten Use Cases genau so offensichtlich sind.

Mit einer computergestützten Transformation der EPKs in Use Cases können zwar keine finalen Anforderungsbeschreibungen gewonnen werden, dafür aber eine ausgezeichnete, interdisziplinäre Dokumentationsmethodik, die auf vielfach benutztem, gut erforschtem und sich bewährtem Konzept der Use Cases basiert. Mithilfe des Computers werden auf die EPK zugeschnittenen, teilweise ausgefüllten Use Case Schablonen erzeugt und mit ihr verknüpft. Der manuelle Aufwand beschränkt sich auf die Ausfüllung oder Vervollständigung einiger Felder dieser Schablonen, wodurch erstens der Zeitaufwand gegenüber einer manuellen Dokumentation reduziert und zweitens die Vollständigkeit der notwendigen Angaben sichergestellt wird. Das unabsichtliche Nichtausfüllen der generierten und offen gelassenen Felder ist fast ausgeschlossen. Die Verknüpfung[2] der EPK- Elemente mit den sie dokumentierenden Use Case- Elementen erleichtert zudem die Navigation und lässt die Herkunft der Anforderungen transparent.

[1]Damit sind die Regelverstöße und das Nichtvorhandensein von Informationen aus der Sicht eines Softwareentwicklers in Bezug auf die Anforderungen von Computerprogrammen gemeint, nicht aus der Sicht des Geschäftsprozessmodellierers.

[2] Die Verknüpfung erfolgt durch die gleichen IDs dieser Elemente. Die IDs sind Bestandteil des jeweiligen XML Formats, und die Verknüpfung ist deshalb von dem in dieser Arbeit entwickelten Prototypen unabhängig.

6 Schluss

6.1 Verwandte Arbeiten

Zur Erhöhung der Produktivität und Senkung der Kosten ist es heutzutage üblich die Geschäftsprozesse in betriebliche Informationssysteme umzusetzen, dabei entstehen häufig Probleme, weil die Methoden, Technologien und Werkzeuge, die von den Softwareentwickler verwendet werden sich unabhängig von denen der Geschäftsprozessmodellierung entwickelt haben und deshalb deutliche Unterschiede aufweisen. Um dieses Problem zu entschärfen werden in zahlreichen Veröffentlichungen Strategien zur parallelen Benutzung oder der Übersetzung der Softwareentwicklungs- und Geschäftsprozessmodellierungsmethoden propagiert:

- So wird in [GeUML2003] ein kompletter Leitfaden zur Modellierung der Geschäftsprozesse mit den Methodiken der UML vorgestellt. Das Werk beschreibt Praktiken über die Erstellung, Vervollständigung und Verfeinerung der Geschäftsprozess- Use Cases bis hin zur Modellierung der Unternehmensstruktur und Abläufe mit den UML Klassen-, Aktivitäts- und Zustandsdiagrammen.

- In [AC2003] wird ein Kapitel der Modellierung von Geschäftsprozessen mit den Use Cases gewidmet.

- In [LoosAllweyer1998], [NFZ1998] und [Dandl999] werden EPK Diagramme mit Klassen, Sequenz-, Aktivitäts- und Use Case Diagrammen (aber nicht Use Cases selbst) verglichen und einige Ideen skizziert, wie EPK Diagramme und die oben genannten UML Diagramme in einander transformiert werden können.

- In [SNZ1997] wird eine objektorientierte Erweiterung für EPKs vorgestellt, die es dem Modellierer erlaubt Klassen und Methoden, wie sie in einem UML Klassendiagramm verwendet werden, darzustellen.

Alle diese Arbeiten bieten Methoden und Praktiken an, die nicht formal definiert sind, sondern eher an die menschliche Intuition und das umfassende Verständnis der entsprechenden Sachverhalte appellieren. Einen anderen Ansatz verfolgen folgende Arbeiten, zu denen auch diese gehört. Hier werden ebenfalls Methoden vorgeschlagen, die etwas mit der Transformation von Geschäftsprozessen in Form von EPKs zu tun haben, allerdings sind diese Methoden so konzipiert, dass sie nicht manuell durch der Menschen sondern automatisch oder zumindest halbautomatisch durch den Computer ausgeführt werden sollen.

- In [LLSG2006] werden EPK Erweiterungen vorgeschlagen, die es ermöglichen aus Ereignisgesteuerten Prozessketten Webservices und Userinterfaces zu generieren.

- Zum Zwecke der Verifikation und Simulation wird in [VDA] ein Verfahren zur Transformation der EPKs in Petri Netze präsentiert.

- Im [MZ] wird ein Verfahren diskutiert, wie aus BPEL Beschreibungen EPK Diagramme erzeugt werden können.

- Eine Transformationsmethode von EPK Modellen in das POP* Format ist in der Arbeit von [KK] zu finden.

6.2 Zusammenfassung

Um Use Cases und EPK Diagramme in einander konvertieren zu können ist zunächst eine genaue Definition der Syntax dieser beiden Sprachen erforderlich. Im Falle der EPKs wurde die Syntax in der Arbeit von [MendlingNüttgens] in Form eines XML Schema definiert, weil für die Use Cases keine Syntax Definition vorlag, wurde in dieser Arbeit ein XML Schema entwickelt das die Syntax der Use Cases nach dem Buch von [AC2003] beschreibt.

Wegen der unkomplizierten Handhabung und des hohen Verbreitungsgrades wurde die XSLT-Technologie gewählt um die notwendigen Transformationen zur Konvertierung zwischen Use Cases und EPKs zu beschreiben. Es hat sich dabei herausgestellt, dass die einfache Form der Use Cases, die auch in dem oben erwähnten Buch von [AC2003] propagiert wird, problemlos und ohne jegliche Benutzereingaben sich auf ein EPK Diagramm abbilden lässt. Bei der umgekehrten Abbildung, also der Abbildung eines EPK Diagramms auf ein Use Case hat sich gezeigt, dass bestimmte Benutzereingaben unerlässlich sind, das heißt also, dass die Use Cases gewisse Informationen (vergleiche Kapitel 2.2 und 5.2) enthalten, die durch die EPK Diagramme nicht erfasst werden, woraus die Nichtäquivalenz zwischen Use Cases und EPKs folgt.

Weiterhin wurde festgestellt, dass die EPK Diaramme, die durch Transformation der Use Cases entstehen, alle ein bestimmtes Muster (Kapitel 2.2) aufweisen, die Abbildung der Use Case Menge auf die Menge der EPKs ist deshalb nicht surjektiv (es gibt solche EPKs zu denen kein passender Use Case als Uhrbild existiert).

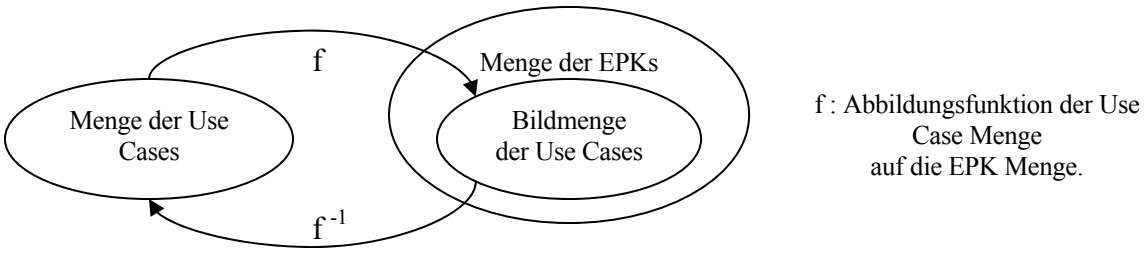
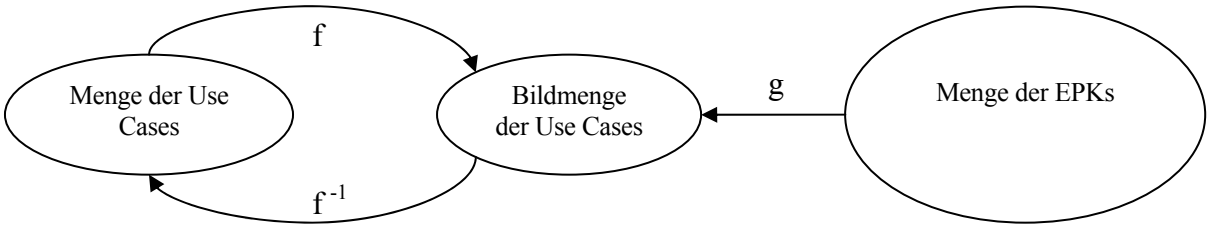


Abbildung 6-1

Um die Abbildungsvorschriften für die EPKs zu definieren, reichte es also nicht aus einfach die Umkehrabbildung der Use Cases zu nehmen, da mit dieser Umkehrabbildung nicht alle EPKs erwischt worden wären, wie in der Abbildung 6-1 zu sehen ist. Um dieses Problem zu beheben wurde eine weitere Abbildung angegeben, mit der die EPKs in die Bildmenge der Use Cases abgebildet werden können. Damit wurde schließlich erreicht, dass jede EPK auf ein Use Case abgebildet werden konnte (in dem sie zunächst in die Bildmenge und dann mit der Umkehrabbildung in die Menge der Use Cases abgebildet wurde, siehe auch Abbildung 6-2).



f : Abbildungsfunktion der Use Case Menge auf die EPK Menge.

g : Abbildungsfunktion, die beliebige EPKs auf in Use Case Modell abbildbare EPKs abbildet.

Abbildung 6-2

Da sowohl die Use Cases als auch die EPKs im XML Format vorlagen, der für den Menschen recht unübersichtlich ist, wurde im Rahmen dieser Arbeit eine Benutzeroberfläche entwickelt, mit der es möglich ist Use Cases und EPKs graphisch (nicht auf XML Ebene) zu erstellen und in einander zu überführen.

6.3 Ausblick

Das Konzept der Konvertierung und der Verknüpfung zwischen Use Cases und EPKs kann in vielen Situationen hilfreich sein. Die Konvertierung der EPKs in Use Cases kann z.B. für die Dokumentation der EPKs verwendet werden. Solche Dokumentationsform wird sowohl von den Softwareentwicklern als auch von Wirtschaftswissenschaftlern verstanden und ist vor allem dann hilfreich wenn die betriebswirtschaftlichen Prozesse durch Software automatisiert werden und diese beiden Gruppen aufeinander treffen. Aber auch dann wenn die Use Cases in Projekten verwendet werden, die mit Geschäftsprozessen nichts zu tun haben, hilft die Konvertierung in die EPK Diagramme den Überblick über die zusammenhängenden Use Cases zu verschaffen und ist somit bei der Validierung der Use Cases hilfreich.

In der vorliegenden Arbeit wurde deshalb ein Konzept vorgeschlagen und implementiert, mit dem vollautomatisch Use Cases in EPKs konvertiert werden können und eine halbautomatische Konvertierung der EPKs in Use Cases möglich ist. Diese Konvertierung ist allerdings nur auf solche EPKs beschränkt, die aus den EPK- Basiselementen (Ereignis, Funktion, Prozesswegweiser und AND-, OR-, XOR- Konnektoren) bestehen.

Das Konzept der EPKs wurde inzwischen um zahlreiche Erweiterungen (die in dieser Arbeit nicht berücksichtigt wurden) ergänzt, zu diesen Erweiterungen gehören z.B. die Aris-Sprachkonstrukte [ScheerThomas2005], die Finanzmittel, menschliche Arbeitsleistung, maschinelle Ressourcen, Computerhardware, Anwendungssoftware etc. repräsentieren. Mit der objektorientierten Erweiterung [SNZ1997] lassen sich auch Klassen und ihre Methode in einer EPK darstellen. Für die Zukunft wäre es also auch interessant zu untersuchen inwieweit diese Erweiterungen sich auf UML Konstrukte (nicht nur auf Use Cases) abbilden lassen. Insbesondere die objektorientierte Erweiterung mit ihren Klassen und Methoden, die auch in UML- Klassendiagrammen modelliert werden, verspricht gute Integrationsmöglichkeiten.

7 Literaturverzeichnis

[AC2003] Alistair Cockburn. Use Cases effektiv erstellen. Mitp 2003.

[Clemmons2006] Roy K. Clemmons. Project Estimation With Use Case Points. CROSSTALK The Journal of Defense Software Engineering

[KNS1992] Keller, Nüttgens, Scheer. Semantische Prozessmodellierung auf der Grundlage „Ereignisgesteuerter Prozessketten (EPK)" Veröffentlichungen des Instituts für Wirtschaftsinformatik, Heft 89

[EPK2002] Nüttgens, Rump. EPK 2002 Proceedings des Arbeitskreises Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten

[MendlingNüttgens] Jan Mendling, Markus Nüttgens. Konzeption eines XML-basierten Austauschformates für Ereignisgesteuerte Prozessketten (EPK).

[GrunLaue] Volker Gruhn, Ralf Laue. Einfache EPK- Semantik durch praxistaugliche Stilregeln.

[Standishgroup] Chaos Report 1994. www.standishgroup.com

[AWScheer2001] August-Wilhelm Scheer. ARIS Vom Geschäftsprozess zum Anwendungssystem. 4 Auflage. Springer 1998.

[GeUML2003] Bernd Oestereich, Christian Weiss , Claudia Schröder, Tim Weilkiens, Alexander Lenhard. Objektorientierte Geschäftsprozessmodellierung mit der UML. Dpunkt Verlag 2003.

[ScheerThomas2005] Prof. Dr. Dr. August-Wilhelm Scheer, Dipl.-Kfm. Oliver Thomas, Geschäftsprozessmodellierung mit der Ereignisgesteuerten Prozesskette 2005.

[Dandl1999] Jörg Dandl. Objektorientierte Prozessmodellierung mit der UML und EPK Arbeitspapiere WI, Nr. 12/99

[NFZ1998] Dr. Markus Nüttgens, Thomas Feldt, Volker Zimmermann. Business Process Modelling with EPC and UML Transformation or Integration? Institut für Wirtschaftsinformatik Universität des Saarlandes.

[SNZ1997] A.W. Scheer, M. Nüttgens, V.Zimmermann. Objektorientierte, Ereignisgesteuerte Prozesskette (oEPK). Veröffentlichung des Instituts für Wirtschaftsinformatik Universität des Saarlandes Heft 141.

[LoosAllweyer1998] Peter Loss, Thomas Allweyer. Process Orientation and Object Orientation, An Approach for Integratin UML and Event-Driven Process Chains. University of Saarland, Saarbrücken. March 1998

[LLSG2006] Daniel Lübke, Tim Lüecke, Kurt Schneider, Jorge Marx Gomez. Using EPCs for Model-Driven Development of Business Applications. Multikonferenz Wirtschaftsinformatik 2006, volume 2. GITO Verlag, 2006.

[VDA] W.M.P van der Aalst. Formalization and Verification of Event-driven Process Chains. Department of Mathematics and Computer Science, Eindhoven University of Technology.

[MZ] Jan Mending, Jörg Ziemann. Transformation of BPEL Processes to EPCs. Vienna University of Economics and Business Administration, Institute for Information Systems University of Saarland.

[KK] Timo Kahl, Florian Kupsch. Transformation und Mapping von Prozessmodellen in verteilten Umgebungen mit der Ereignisgesteuerten Prozesskette. Institut für Wirtschaftsinformatik Saarbrücken.

[ApacheAnt] http://ant.apache.org

[Java] http://java.sun.com

[Wikipedia] http://de.wikipedia.org

[W3C] http://www.edition-w3c.de

[JGraph] www.JGraph.com

[Saxon] http://saxon.sourceforge.net

....