universität wien

MASTERARBEIT

Titel der Masterarbeit

„Differenzen-basierte Versionskontrolle von Geschäftsprozessen“

verfasst von

Lukas Schreier, BSc.

angestrebter akademischer Grad

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2014

Studienkennzahl lt. Studienblatt:	A 066 926
Studienrichtung lt. Studienblatt:	Wirtschaftsinformatik
Betreuerin / Betreuer:	Univ.-Prof. Dipl.-Math. Dr. Stefanie Rinderle-Ma

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Beispieleverzeichnis

Formelverzeichnis

Listingsverzeichnis

1 Einleitung

In Zeiten von agilen Geschäftsfeldern und schnell wachsender Konkurrenz, müssen Unternehmen und Institutionen ihre Geschäftsprozesse permanent anpassen und umändern [REDA00][AADM00], um so ihre Ziele zu erreichen und in weiterer Folge die Geschäftsstrategien zu wahren [JJNJ14]. Gründe für Geschäftsprozessänderungen können dabei vielfältig und unterschiedlich motiviert sein. Eine offensichtliche Geschäftsprozessänderung entsteht dann, wenn durch kontinuierliche Verbesserungen von Technologien oder durch das Inkrafttreten neuer regulatorischer Richtlinien eine Prozessadaptierung notwendig wird. Solche Änderung zielen auf eine längerfristige Umstellung der Prozesse ab [RIRD04]. Prozessänderungen können jedoch auch im „täglichen Betrieb“ von Geschäftsabläufen getätigt werden. Solche Adhoc-Änderungen werden dann durchgeführt, wenn zum Beispiel Fehlersituationen innerhalb des Ablaufs auftreten [RNAH06]. Ein weiterer Grund für die Abänderung eines Prozessmodells kann das Customizing sein. Darunter versteht man die Anpassung von Produkten oder Dienstleistungen eines Serienproduktes an die Bedürfnisse des Kunden. Unterschiedliche Kundenanforderungen können es notwendig machen vom Standardprozess abzuweichen und gezielte Anpassungen für diesen Kunden im Prozessablauf einzubauen.

Mit wachsender Prozesskomplexität bei einer gleichzeitigen Steigerung der Flexibilität im Customizing, wird das Verwalten von Prozessen zunehmend schwieriger, aufwendiger und intransparent. Dies führt zwangsläufig zu langen Reaktionszeiten und verursacht Kosten [WESK07]. Lösungen zur Versionsverwaltung von Prozessmodellen sind dabei eine Abhilfe, um diesen Entwicklungen entgegenzuwirken. Mit geeigneten Versions-verwaltungssystemen ist auch langfristig die Wartbarkeit und Nachvollziehbarkeit der Abänderungen von Geschäftsprozessen gegeben. In weitere Folge bedeutet dies ein schnelleres Deployment neuer oder adaptierter Prozesse. Dadurch entstehen nicht nur Wettbewerbsvorteile für Unternehmen sondern auch Kosteneinsparungspotenziale.

1.1 Motivation und Problemstellung

Die Motivation für diese Arbeit liegt in zwei Problemstellungen begründet. Eine Problemstellung entwickelte sich während eines Seminars an der Universität, die andere ist eine generelle Lücke in der Verwaltung von Versionen im Kontext des Process Engineerings.

Der erste Fall entstand im Seminar „Evolution“ im Studienspezialisierungsfach „Semantische Informationssysteme“. Hier mussten die Studierenden eine Softwarelösung zur Differenzerkennung und Merging von Prozessmodellen erstellen. Dabei wurde von jeder Gruppe eine eigene Prozessmodellnotation definiert und fiktive Prozesse erstellt. Anschließend wurde über das Semester hinweg die Software implementiert. Es stellte sich heraus, dass alle Gruppen eine implizite Annahme getroffen haben: Nämlich, dass alle Prozess-Objekte eindeutig erkannt werden können, zum Beispiel über einen uniquen Identifizierer. Eine Annahme, welche in der Praxis nicht immer getroffen werden kann, wenn eine Differenzenanalyse zwischen zwei Prozessmodellen in einer beliebigen Prozessnotation durchgeführt wird. Ebenso waren die gefundenen Differenzen (Deltas) nicht einheitlich strukturiert und stark an die jeweilige Prozessnotation angepasst. In der Endphase der Projekte sollte eine gewisse Interoperabilität zwischen den Implementierungen hergestellt werden. Ziel war es, die Prozessmodelle der anderen Gruppen mit dem gewählten Ansatz und der Implementierung zu analysieren. Um dies zu erreichen, mussten die Gruppen den Sourcecode teilweise anpassen aber auch Mappings erstellen, um die „fremde“ Notation zu prozessieren. Auch bei den Mappings wurden einige Probleme erkannt: Der Aufbau der einzelnen Notationen der Gruppen unterschied sich zum Teil fundamental. Bei einer Gruppe war die Anordnung der Prozess-Objekte wichtig. Andere Gruppen wiederum konnten die Prozess-Objekte unabhängig ihrer Abfolge im Prozess abspeichern und die tatsächliche Anordnung wurde mittels Kanten beschrieben. Resultat dieser Übung war es, dass für jede Prozessnotation ein eigener Algorithmus zum Erkennen von Differenzen und zum Mergen der Modelle entstand. Ein Umstand, der bei einem Einsatz von mehreren Prozessnotationen nicht praktikabel erschien.

Der zweite Fall entsteht durch die Diskrepanz der verwendeten Workflow-Tools. In

Workflow-Tools sind die Versionskontrollen zwar im Tool integriert, aber auf eine bestimmte Prozess-Objekt-Struktur des Tools ausgelegt [WBRR08]. Vorgaben und Restriktionen in einem Workflow-Tool können somit mit den Gruppenprojekten in dem oben erwähnten Seminar verglichen werden. Werden daher Prozesse in zwei unterschiedlichen Workflow-Tools entwickelt, sind zumeist die jeweiligen Tools für die Versionsverwaltung zuständig. Ein Import der Prozesse in ein anderes Tool ist oft nicht oder nur mit hohem Aufwand möglich.

Lösungen, wie sie zur Zeit in der Softwareentwicklung bestehen, gibt es im Process Engineering noch nicht. In der Softwareentwicklung ist es für das Versionsmanagement unbedeutend in welcher Programmiersprache, oder mit welcher Entwicklungsumgebung der Code implementiert wird. Systeme wie SVN, Git, Mercurial oder dergleichen können in Sourcedateien unabhängig der Gegebenheiten die Differenzen erkennen und auch versionieren.
Da Prozessmodelle zumeist in einem Textformat abgelegt werden, können zwar die bereits bestehenden Systeme der Softwareentwicklung herangezogen werden, allerdings besitzen diese Tools keine adäquate Darstellung der Änderungen für Prozessmodelle. Die Änderungen werden hier für jede Zeile und Buchstaben angezeigt. Für die Prozessmodelle ist das eine zu detaillierte Darstellung und das Gesamtbild der Änderungen könnte somit nicht mehr erkannt werden.

Aus den erwähnten Problemstellungen entwickelte sich folgende zentrale Forschungsfrage:

Wie kann eine generische Versionskontrolle von blockstrukturierten Prozessmodellen, unabhängig von der verwendeten Prozessnotation, umgesetzt werden?

Darauf bezogen konnten weitere Forschungsfragen abgeleitet werden, die zur Beantwortung der Kernfrage, ebenfalls in dieser Master-Thesis behandelt werden:

- Welche Mindestanforderungen muss ein abstraktes Prozessmodell (Metamodell) erfüllen, um ein beliebiges blockstrukturiertes Prozessmodell ohne Informations-verlust in dieses Metamodell zu konvertieren?
- Wie sieht eine atomare Form (Change-Primitives) von Änderungsoperationen in

einer generischen Differenzerkennung von Prozessmodellen aus?

- Welcher Algorithmus ist geeignet um Change-Primitives zwischen zwei blockstrukturierten Prozessmodellen einer beliebigen Prozessnotation zu erkennen?
- Wie können Change-Primitives zu hochwertigen Änderungsoperationen (Highlevel-Changes) überführt werden, sodass diese im Hinblick auf Prozessmodelle eine geeignete semantische Aussagekraft besitzen?
- Wie sieht eine geeignete Struktur zur Verwaltung von Prozessmodellversionen aus, wenn lediglich die Highlevel-Changes zur Vorgängerversion (Delta) ablegt werden sollen?
- Wie lässt sich über chronologisch angeordnete Deltas eine performante Menge an Änderungsoperationen erstellen, um das ursprüngliche Prozessmodell zu rekonstruieren?

1.2 Methodik

Zur Beantwortung der Forschungsfragen dieser Master-Thesis wird ein *konstruktiv-wissenschaftliches Paradigma (Design Science)* [TWTH06] verwendet. Demzufolge werden ein Metamodell, Algorithmen und Formeln zunächst theoretisch ausgearbeitet, um anschließend eine lauffähige Software zu konstruieren. Dabei wird in dieser Arbeit das folgende Methodenspektrum angewendet:

- Jedes Kapitel enthält einen *theoretischen Methodenteil,* welcher mit einer Literaturrecherche durchgeführt wurde. Hier wird bereits bestehendes Wissen aus unterschiedlichen Themengebieten der Informatik (u.a. Information Retrieval, Workflow-Technology, Plagiatserkennung und Versionskontroll-Management) ausgearbeitet. Anschließend erfolgt mit einer *argumentativ-deduktiven Analyse* eine Reflexion des gewonnenen Wissens sowie eine Erweiterung hinsichtlich des Kerns dieser Arbeit.

- Resultierend aus dem theoretischen Teil werden mittels *Referenzmodellierung,* sukzessive Konzepte zur generischen Versionskontrolle inklusive Differenzenerkennung von blockstrukturierten Geschäftsprozessen entwickelt. Im Referenzmodell werden die Algorithmen, Formeln, Modelle und Strukturen entwickelt um die zentralen Forschungsfragen zu beantworten.

- Anschließend wird mittels einer *qualitativ konstruktiven Methode* ein Prototyp für das Referenzmodell der Differenzenerkennung erstellt. Die Implementierung erfolgt dabei in der Programmiersprache Java.

- Eine *praktische Evaluierung* erfolgt mit einem Workflow-Tool namens CPEE-Cockpit, welches an der Universität Wien von der Forschungsgruppe „Workflow-Systems and Technology“ entwickelt wurde und einer weiteren, weitverbreiteten Prozessmodellierungsnotation namens BPMN.
  Als Prozessmodelle dienen Geschäftsprozesse, einer österreichischen Bank zur Berechnung und Reporting einer neuen Kennzahl aus dem Basel-III-Framework für das Monitoring der 30-Tage-Liquidität [BCBS14].
  Ausgehend von einem Basisprozess, werden an diesem Modifikationen durchgeführt, um anschließend die Implementierung und Konzepte der Differenzerkennung und Ähnlichkeitsanalyse zu testen. Da die Änderungen der Prozessmodelle im Vorfeld bekannt sind, kann durch eine Analyse, der von der Implementierung produzierten Log-Files, eine Aussage darüber getroffen werden, ob die Menge an Änderungsoperationen vollständig ist.
  Um die entworfenen Konzepte und Modelle für die Versionskontrolle auf Korrektheit zu evaluieren, werden die zuvor generierten Prozessmodelle und Log-Files herangezogen um die ursprünglichen Prozessmodelle, mit den erzeugten Daten, wieder zu generieren.
  Die Konzeptprüfungen werden dabei mit Beispielen unterlegt und sind in jedem Kapitel für das jeweilige Thema integriert.

1.3 Kapitelübersicht

Diese Master-Thesis gliedert sich in sieben Kapitel, wobei der Kern der Arbeit in zwei Bereiche gegliedert werden kann: Der erste Teil setzt sich mit der Differenzerkennung zwischen Prozessmodellen auseinander. Der zweite Teil schlägt ein Konzept zur Verwaltung von Prozess-Versionen vor. Bevor jedoch im Detail auf die zwei Kernpunkte eingegangen wird, erfolgt zuerst ein einführendes theoretisches *erstes Kapitel*, das sich mit Definitionen über Prozesse und deren Aufbau (Blockstrukturierung) befasst. Ebenso erfolgt eine Einführung in die Materie des Workflowmanagements und grundsätzliches zum Thema Versionskontrolle. Das Kapitel wird ergänzt durch Informationen über die Speicherung von Änderungsoperationen von Geschäftsprozessen (Change-Logs) und einer Erläuterung über typische Änderungsoperationen (Change-Pattern) in einem Geschäftsprozess.

Das *zweite Kapitel* leitet ein Konzept für eine generische Prozessnotation ab. Ziel dieses Kapitel ist es, ein Metamodell zu spezifizieren, in welches beliebig andere Prozess-notationen abgebildet werden, ohne dabei Informationsverluste zu verursachen. Nach der Metamodell-Spezifikation wird im dritten *Kapitel* eine Definition über die kleinsten Einheiten von Änderungsoperation zwischen zwei Prozessmodellen gegeben. Des weiteren wird gezeigt, wie solche Änderungsoperationen erkannt werden können. Ebenso werden die entsprechenden Algorithmen angeführt. Aufbauend auf den Ergebnissen von Kapitel drei soll anschließend in *Kapitel vier* gezeigt werden, wie die zuvor gefundenen Änderungen in höherwertige Änderungsoperationen überführt werden können. Hierzu wird eine Ähnlichkeitsanalyse (Similarity-Measure) zwischen den Differenzen der zwei Prozessmodellen durchgeführt, sowie eine Methodik zum Erkennen von verschobenen Prozessobjekten (Move-Detection) vorgestellt.

Ab dem *fünften Kapitel* wird dann auf die Versionskontrolle von Prozessmodellen eingegangen. In diesem Kapitel wird eine Spezifikation und grundsätzliche Gestaltung einer Repository-Struktur zur Verwaltung von Prozess-Versionen vorgeschlagen. Gleichzeitig wird auf die Besonderheiten zwischen der Versionsverwaltung von Prozessen und der Versionskontrolle in der Softwareentwicklung hingewiesen. Der letzte Abschnitt dieses Kapitels beschäftigt sich mit der Abspeicherung der Informationen über die

Änderungen zwischen zwei Prozessversionen. Aufbauend auf diesem Entwurf wird in *Kapitel sechs* gezeigt, wie mittels einer minimalen Menge an Änderungsoperationen eine beliebige Version eines Prozesses wiederhergestellt werden kann.

Das *letzte Kapitel* fasst die gewonnenen Ergebnisse der Master-Thesis zusammen. Es folgt eine Diskussion über die gewählten Methoden und Algorithmen. Die Diskussion enthält auch einen Abschnitt über Ideen zur Weiterentwicklung und Verfeinerungen.

1.4 Grobkonzept

Nachfolgend wird ein kurzer Überblick über den Ablauf des Grundkonzepts dieser Arbeit gegeben. Der Fokus liegt nicht auf einer detaillierten Beschreibung der einzelnen Bestandteile, sondern auf der Interaktion der Bestandteile untereinander. Gezeigt wird, wie ein Prozessmodell in das Versionskontrollsystem eingecheckt wird, welche Methodiken innerhalb der Differenzenerkennung angewandt werden, welche Daten beziehungsweise Dateien erstellt werden und wie ein unter der Versionskontrolle stehendes Prozessmodell wieder hergestellt wird.

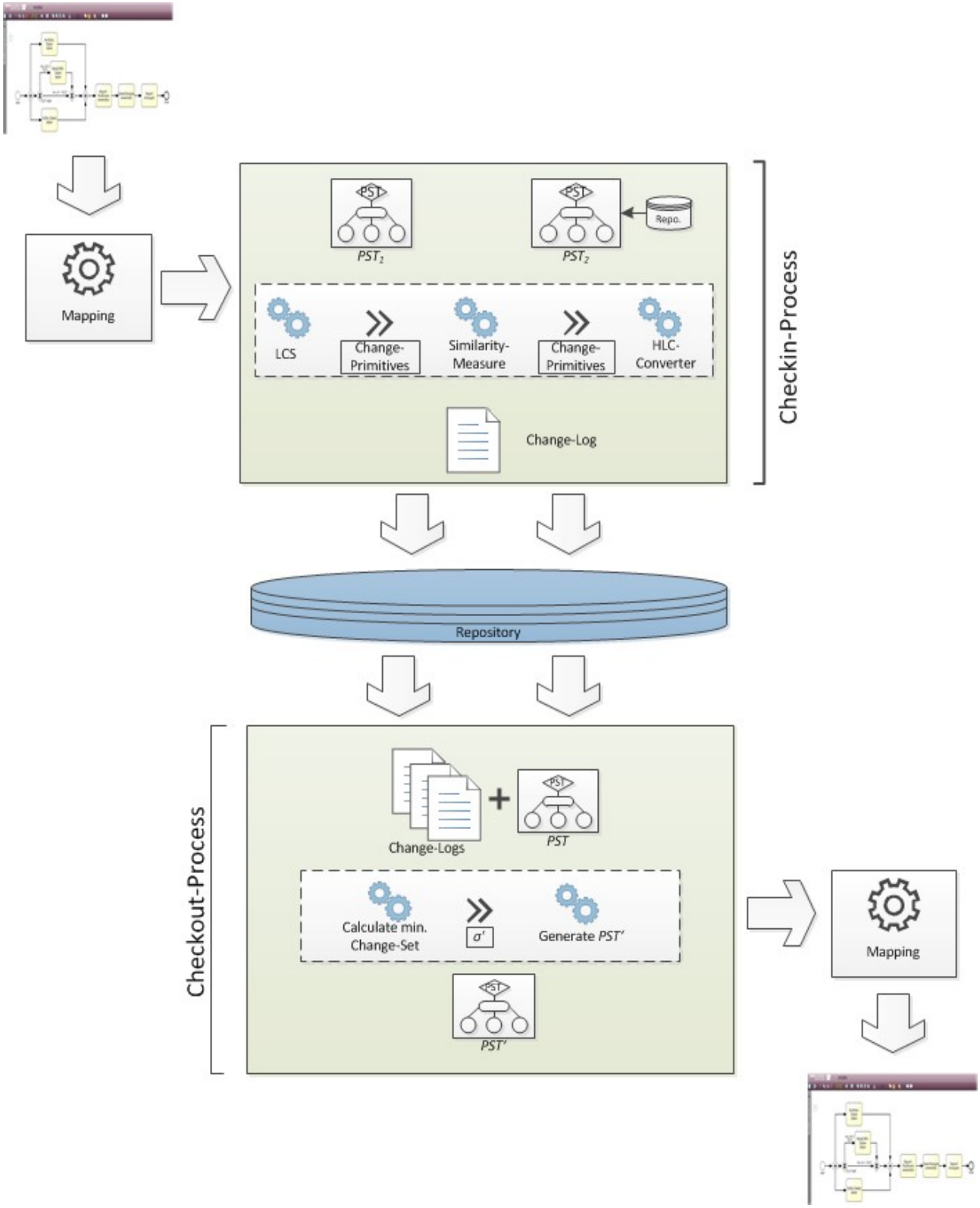


Abbildung 1: Grobkonzept

Quelle: Eigene Erstellung

Wie das Grobkonzept in Abbildung 1 zeigt, teilt sich der Ablauf in zwei Prozesse auf. Der Checkin-Prozess und der Checkout-Prozess. Im Checkin-Prozess wird das Prozessmodell

aus einer speziellen Modellierungsnotation konvertiert, analysiert, die benötigten Informationen für das Versionskontrollsystem (Change-Log) aufbereitet und so abgelegt, dass eine spätere Wiederherstellung des Modells möglich wird. Das Prozessmodell durchläuft zuerst ein Mapping (Mapping-Level), welches das spezielle Prozessmodell in ein eigens erstelltes Meta-Modell transformiert. Aufgrund der Vielzahl von Prozessmodell-notationen, muss für jede Prozessmodellnotation ein Mapping geschaffen werden. Anschließend erfolgt eine automatische Transformation in eine Datenstruktur, welche es ermöglicht die genauen Positionsinformationen, sowie die Vorgänger-Nachfolger-Beziehungen der Prozessobjekte, im Prozessmodell zu bestimmen. Diese Datenstruktur ist ein sogenannter Process-Structure-Tree. Da im Repository zwischen zwei Versionen nur die Differenzen abgelegt werden, wird der Process-Structure-Tree PST_1 mit dem Vorgängermodell PST_2 verglichen. Das Vorgängermodell wird dabei direkt aus dem Repository errechnet und existiert daher schon in der übersetzten Form, wodurch eine sofortige Erstellung des Process-Structure-Trees möglich ist.

Existieren nun beide Process-Structure-Trees, beginnt die Analyse. Hier wird zuerst mittels Longest-Common-Subsequence Algorithmus (LCS) eine Menge an Add- und Delete-Operationen erstellt, sogenannte Change-Primitives. Diese Change-Primitives geben an, auf welchen Positionen in PST_2 Objekte gelöscht oder hinzugefügt werden müssen um PST_1 zu generieren. Es ist jedoch von Interesse Change-Primitives in Highlevel-Changes zu transformieren. Dies sind höherwertige Änderungsoperationen, welche eine stärkere semantische Aussagekraft besitzen als die Anreihung von Add- und Delete-Operationen. Bevor diese Umwandlung möglich ist, muss noch ein Zwischenschritt durchgeführt werden, da in der Menge der Change-Primitives Add- und Delete-Operationen über ein und dasselbe Prozessobjekt gelistet sein können. Solch ein Szenario kann zum Beispiel durch Umbenennungen oder Verschiebungen von Prozessobjekten entstehen. Demzufolge werden mit einer Ähnlichkeitsanalyse die Change-Primitives untersucht. Semantisch gleichwertige Prozessobjekte können dadurch erkennt und zusammengeführt werden. Mit dieser Kombinationsbildung entstehen wichtige Informationen um die Highlevel-Changes abzubilden. Wurde die Ähnlichkeitsanalyse durchgeführt und Kombinationen von Add- und Delete-Operationen gefunden, startet die Konvertierung in die Highlevel-Changes (HLC-Converter). Während der Konvertierung werden unterschiedliche Regeln und Berechnungen hinsichtlich Prozessobjektpostionen durchgeführt. Das Ergebnis der Highlevel-Change-Umwandlung ist ein Change-Log mit

Informationen über die Änderungen zwischen PST_1 und PST_2 . Dieses Change-Log wird anschließend im Repository als eine Version abgelegt. Die Change-Logs werden im Repository so abgelegt, dass die unterschiedlichen Versionen einen sogenannten Versionsbaum darstellen. Der Versionsbaum selbst ist dabei ein direkter azyklischer Graph, der aus einer Hauptentwicklungslinie und mehreren Abzweigungen besteht.

Die zweite Phase bildet der Checkout. Hier wird ein Prozessmodell in einer beliebigen Version aus dem Repository entnommen und über die zuvor generierten Change-Logs der entsprechende Process-Structure-Tree von diesem Modell rekonstruiert. Diese „Rückrechnung“ teilt sich in drei Prozessschritte auf. Als Ergebnis erhält man PST' . Der erste Schritt ist die Sammlung der Change-Logs der letzten Versionen in einem Versionsast des Versionsbaumes, bis zu der Version, welche aus dem Repository extrahiert werden soll. Zusätzlich wird auch der Process-Structure-Tree PST der letzten Prozessversion des Versionsastes entnommen. Dieser Baum dient als Referenz-Prozessmodell, auf welchen dann die Änderungen durchgeführt werden um PST' zu erhalten.
Anschließend werden die Change-Logs chronologisch sortiert und eine Change-Menge errechnet, welche die minimalste Anzahl an Änderungs-Operationen beinhaltet, die notwendig sind, um die gewünschte Version in Abhängigkeit von PST zu erzeugen. Dieses Change-Log enthält dann nicht nur die minimalste Anzahl an Operationen, sondern auch die inversen Change-Operationen. Dies ist notwendig, da im Versionsbaum nur die Vorgänger-Nachfolger-Änderungen enthalten sind. Wird das minimale inverse Change-Log σ' als Eingangsdaten für `generate PST'` verwendet, erhält man den „alten“ Process-Structure-Tree PST' .
Schlussendlich wird wieder mittels eines Mappings die interne Repräsentation auf die spezielle Prozessmodellierungsnotation transformiert.

1.5 Prozessmodellnotationen

Um Geschäftsprozesse abzubilden und um eine einfache Schnittstelle zwischen Benutzer und Prozessen zu ermöglichen, werden Modellierungssprachen/Prozessmodellnotationen eingesetzt. Diese arbeiten mit graphischen Elementen, welche die unterschiedlichen Bestandteile des Workflows zeigen. So können eigene Symbole für Start- und End-Knoten existieren, verschiedene Abbildungen für Aktivitätstypen sowie spezielle Symbole für Workflow-Bestandteile wie Parallelität, Kondition und Iteration. Meist werden in der graphischen Notation Kanten verwendet um Verbindungen zwischen den Objekten klar zu definieren.

Über die Jahre hat sich eine Vielzahl von Modellierungsnotationen und Sprachen herausgebildet. Diese Arbeit beschränkt sich jedoch lediglich auf zwei Notationen. Hierbei fiel die Wahl auf die Business Process Modeling Notation (BPMN) und Cloud Process Execution Engine (CPEE).

BPMN wurde aufgrund der Bekanntheit, der weiten Verbreitung sowie der intuitiven Verständlichkeit gewählt. Im Laufe der Arbeit werden die unterschiedlichen Konzepte und Ableitungen mit Beispielen erläutert, die alle mit BPMN modelliert worden sind. Da BPMN als ein Standard definiert ist, können Hersteller von Modellierungsnotationen dennoch diesen Standard mit Abweichungen implementieren, insbesondere bei der Serialisierung der Prozessobjekte. Alle BPMN-Modelle wurden mit der Modellierungsplattform Signavio erstellt.

CPEE wurde gewählt, da CPEE im Gegensatz zu BPNM eine Workflow-Engine beinhaltet und somit auch der Datenfluss berücksichtigt wird. Des weiteren ist CPEE auf Grund der technischen Konzeption für diese Arbeit von großem Interesse, da CPEE immer blockstrukturiert ist. Es wird in der internen Repräsentation des Modells auf Kanten verzichtet und die einzelnen Prozessobjekte werden nicht eindeutig identifiziert.

Die Auswahl dieser Kombination liegt in den großen Unterschieden der beiden Modellierungsnotationen. Diese Unterschiede erzwingen die Gestaltung der Konzepte und Methoden in dieser Arbeit so zu wählen, dass eine Hersteller-Unabhängigkeit garantiert

wird. Obwohl der Schwerpunkt auf BPMN und CPEE liegt, wurden dennoch weitere Modellierungssprachen wie EPK und andere Workflow-Engines ,wie zum Beispiel AristaFlow[1], Adept$_{flex}$ [RMDP98] oder UC4[2] untersucht, um auch hier Erkenntnisse zu sammeln und diese in die Arbeit mit einfließen zu lassen.

1.5.1 BPNM

BPMN ist die Kurzform für Business Process Modeling Notation und ist auf die Darstellung von Geschäftsprozessen spezialisiert. Sie wurde im Jahr 2001 von der OMG (Object Management Group) zum ersten Mal veröffentlich und existiert nun in der Version 2.0.

Die Grundelemente stammen noch aus der Version 1.0 und sind neben den Aktivitäten, sowie Start-/End-Knoten, auch die Konnektoren und die sogenannten Gateways. Aktivitäten sind atomare Aufgaben innerhalb des Geschäftsprozesses. Der Start- und der End-Knoten markieren den Anfang und das Ende des Prozesses. Konnektoren verbinden dabei die Aktivitäten untereinander und bilden so den Kontrollfluss ab. Ein weiteres Grundelemente sind Gateways. Gateways dienen zur weiteren Verfeinerung des Kontrollflusses und setzen dabei Parallelitäten, Konditionen und Iterationen um. Mittels unterschiedlicher Symbole lassen sich dabei Parallelitäten sowie Konditionen und Iterationen erkennen.

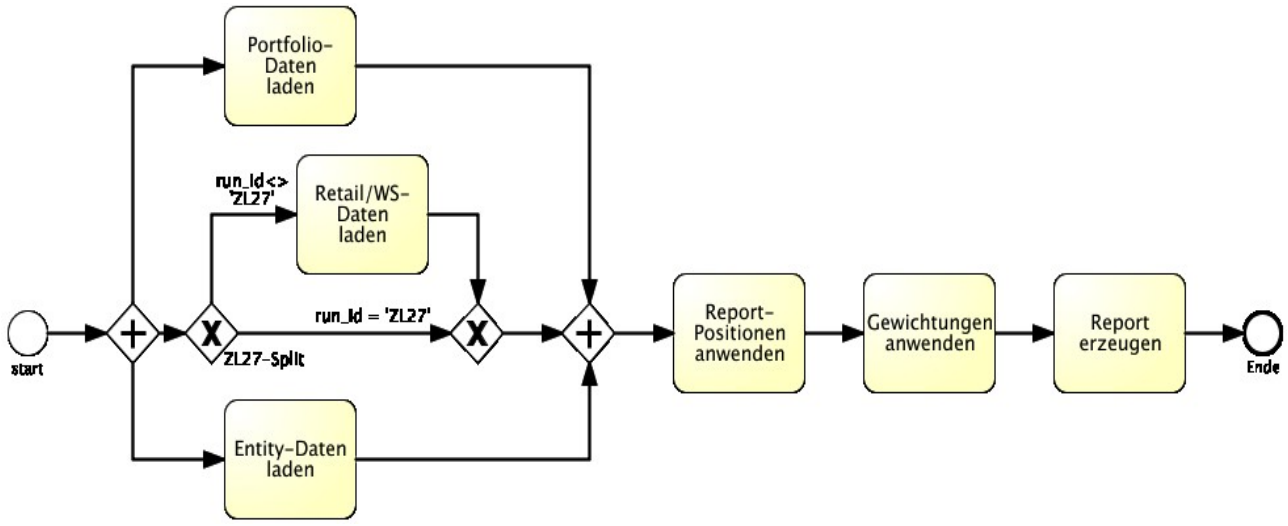


Abbildung 2: Beispiel eines BPMN-Modells

Quelle: Eigene Erstellung, mit Signavio

1 *www.**aristaflow**.com/*

2 *www.**uc4**.com/*

1.5.2 CPEE

Die Cloud Process Execution Engine kurz CPEE ist ein RESTfull Interface, welches auf der WEE-Library (Workflow Execution Engine Library) aufbaut und wurde von der Forschungsgruppe „Workflow Systems and Technology“ der Universität Wien entwickelt. Neben unterschiedlichen Aktivitätstypen (Task/Call) unterstützt die CPEE zur Zeit die meisten Workflow-Patterns, verglichen mit anderen Workflow-Engines [MJSS10]. Die nächste Abbildung zeigt den in Abbildung 2 gezeigten Workflow in CPEE modelliert.

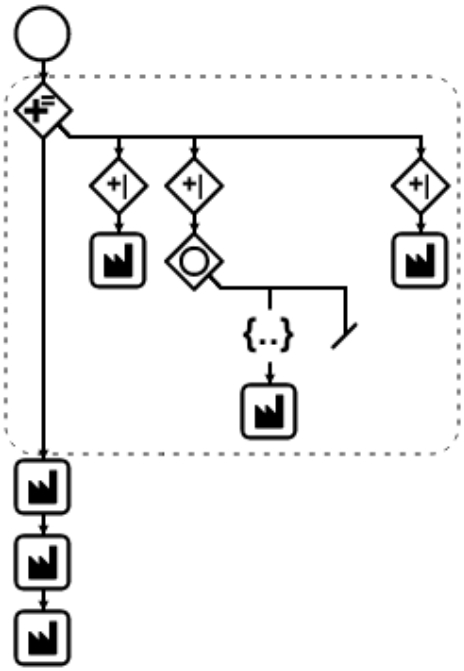


Abbildung 3: Ein CPEE-Modell

Quelle: Eigene Erstellung, mit CPEE-Cockpit

2 Grundlagen

2.1 Grundlangen über Prozesse

In diesem Kapitel werden fundamentale Definitionen festgelegt, welche für das Verständnis notwendig sind und als Basis für die weiteren Betrachtungen dienen.

Als ein *Prozessmodell* wird ein spezieller gerichteter Graph verstanden, mit einer bedingten Anordnung von auszuführenden Aufgaben [WESK07]. Vereinfacht lässt sich ein Prozessmodell wie folgt formal definieren [REMA00]: Sei $P=(N,E)$ ein Prozessmodell mit N, eine Menge von Knoten, welche zugleich die auszuführenden Aufgaben im Prozess darstellen. Sowie E eine Menge von Kanten, wobei gilt $E \subset (N \times N)$, also eine Teilmenge aus allen möglichen Kombinationen aus der Menge der Knoten. Wichtig ist, dass das Prozessmodell, oder auch Prozess-Schema, lediglich eine Vorlage für die tatsächliche Prozessausführung ist. Wird der Prozess nun ausgeführt, entsteht aus dem Prozessmodell eine Prozess-Instanz [REMA00]. Eine *Prozess-Instanz* ist definiert durch: Sei I eine Prozessinstanz, bestehend aus $I=(P,NS,ES,\sigma)$, wobei P das Prozessmodell darstellt, NS den Status der Knoten der ausgeführten Instanz (zum Beispiel 'Active', 'Canceled', 'Completed'), ES den Ausführungsstatus der Kanten in der Instanz und σ beinhaltet die Ausführungshistorie der Instanz. Typischerweise besteht zwischen dem Prozessmodell und der Prozessinstanz eine 1:n-Beziehung.

Abbildung 4 zeigt den Unterschied zwischen einem Prozessmodell und den dazugehörigen Instanzen. Aus dem `Modell` wird eine lauffähige Instanz erzeugt, wobei aus einem Modell mehrere Instanzen entstehen können. Die Instanzen eins bis vier befinden sich in unterschiedlichen Zuständen: Instanz eins wird initialisiert, Instanz zwei wird gerade die Aktivität `Report-Positionen anwenden` durchgeführt. Die Instanz drei wurde erfolgreich beendet, da alle Aktivitäten durchlaufen wurden und Instanz vier ist in der Aktivität `Gewichtungen anwenden` abgebrochen.

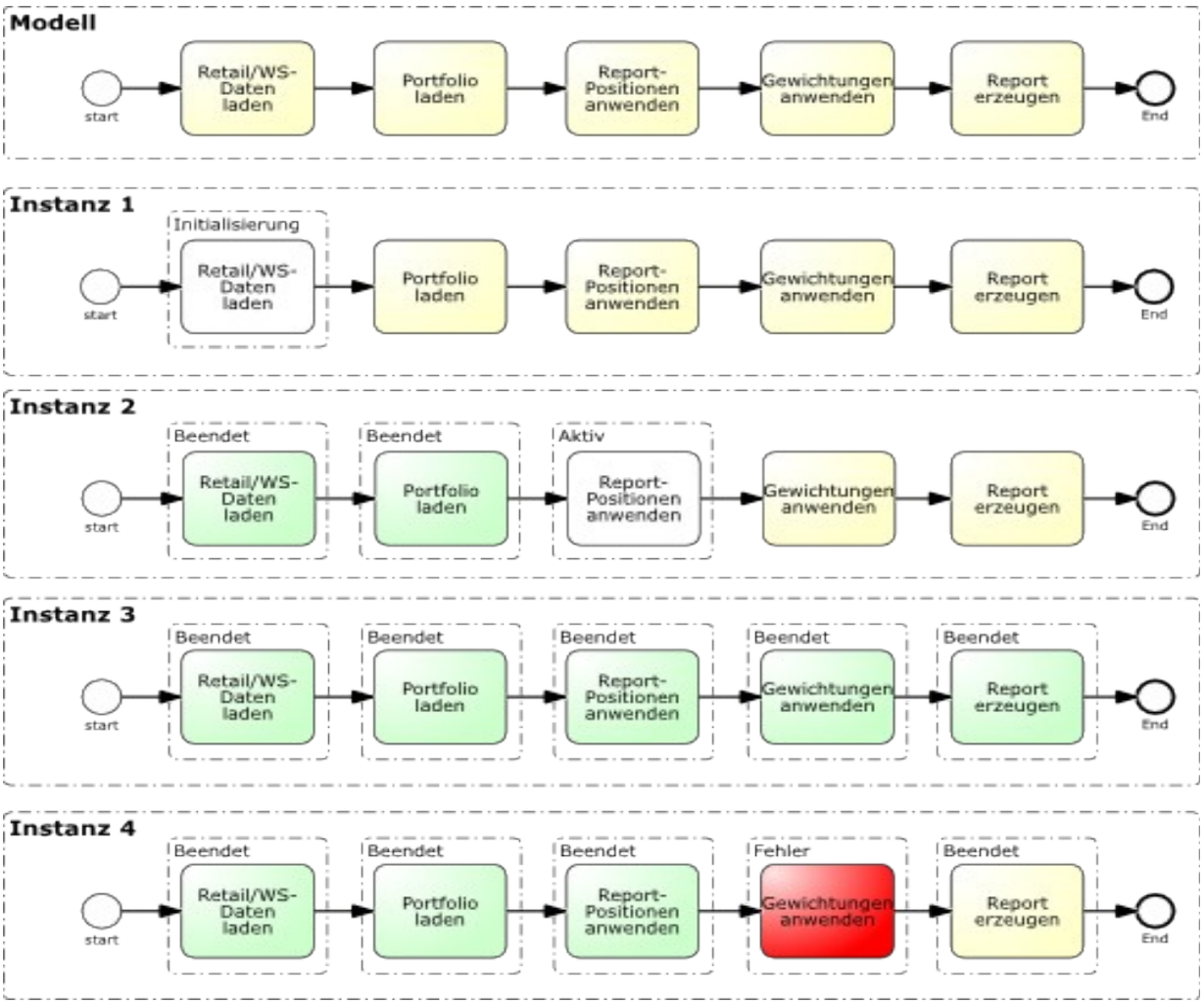


Abbildung 4: Prozessmodell und Prozessinstanzen

Quelle: Eigene Erstellung

Die oben gegebene Definition über ein Prozessmodell ist jedoch sehr weit gefasst und lässt auch relativ unstrukturierte Modelle zu. Daher wird zur Darstellung und Ausführung von Geschäftsprozessen häufig ein sogenanntes Workflow-Modell verwendet. Ein Workflow-Modell hat zwei wichtige Eigenschaften [AALS98]:

- *Start-Knoten:* Der Start-Knoten ist ein Knoten, welcher keine eingehende Kante, dafür aber eine ausgehende Kante besitzt.
- *Ende-Knoten:* Der Ende-Knoten ist ein Knoten im Workflow-Diagramm, welcher nur eine eingehende Kante besitzt, aber keine ausgehende Kante.

Abbildung 5 zeigt ein Workflow-Modell mit jeweils einem Start-Knoten (`Start`) und einem Ende-Knoten (`End`).

Abbildung 5: Workflow-Modell mit Start- und End-Knoten

Quelle: Eigene Erstellung

Des weiteren können in Workflow-Modellen für die Abbildung von Geschäftsprozessen fundamentale Prozess-Fragmente [AHKB03] verwendet werden. Grundsätzlich können diese in vier Gruppen unterteilt werden:

- *Sequenz*: Aktivitäten (Knoten) werden in der angeführten Reihenfolge (aufeinander folgend) durchgeführt.
- *Parallelität (AND)*: Zwei oder mehrere Aktivitäten werden gleichzeitig durchgeführt.
- *Kondition (OR)*: Eine oder mehrere Aktivitäten werden erst dann ausgeführt, wenn eine zuvor definierte Bedingung erfüllt ist.
- *Iteration (LOOP)*: In einer Iteration werden Workflow-Fragmente solange wiederholt, bis eine zuvor definierte Bedingung die Schleife unterbricht.

Die nachfolgenden Abbildungen 6 bis 9 zeigen die Fragmente mittels Workflow-Modellen:

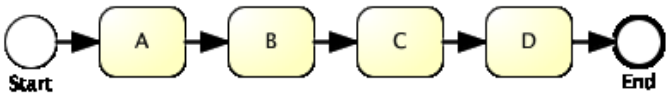


Abbildung 6: Fragment Sequenz

Quelle: Eigene Erstellung

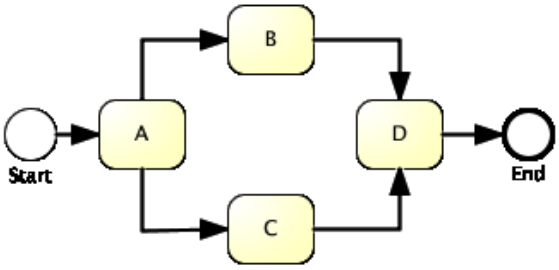


Abbildung 7: Fragment Parallelität (AND)

Quelle: Eigene Erstellung

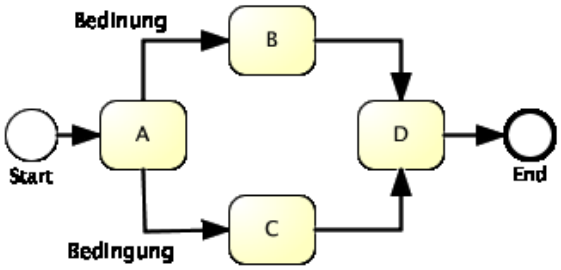


Abbildung 8: Fragment Kondition (OR)

Quelle: Eigene Erstellung

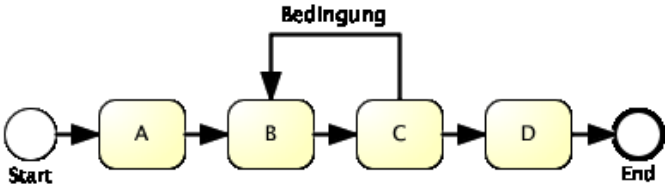


Abbildung 9: Fragment Iteration (LOOP)

Quelle: Eigene Erstellung

In dieser Arbeit wird – um die Komplexität zu reduzieren – angenommen, dass alle Workflow-Modelle blockstrukturiert sind. Unter Blockstrukturierung versteht man, dass Verzweigungen wie Parallelitäten, Konditionen und Iterationen einen Knoten besitzen, welcher die Verzweigung „öffnet“, der sogenannte Split-Knoten, und nachdem die Verzweigung wieder zusammenführt, einen Synchronisation-Knoten (der Join-Knoten) enthält [RMDP98][KBHB00]. Blockstrukturierte Workflow-Modelle haben den Vorteil, dass sie einfacher zu lesen sind [MJRC07], da die expliziten Split/Join-Knoten die Verzweigungen durch genau abgegrenzte Blöcke sichtbar machen. Durch Blockstrukturierung sind Prozessmodelle ebenfalls weniger fehleranfällig [MJNA07]. Split/Join-Fragmente können verschachtelt werden, jedoch muss zuerst das Kind-Fragment geschlossen werden, bevor das Eltern-Segment zusammengeführt wird.

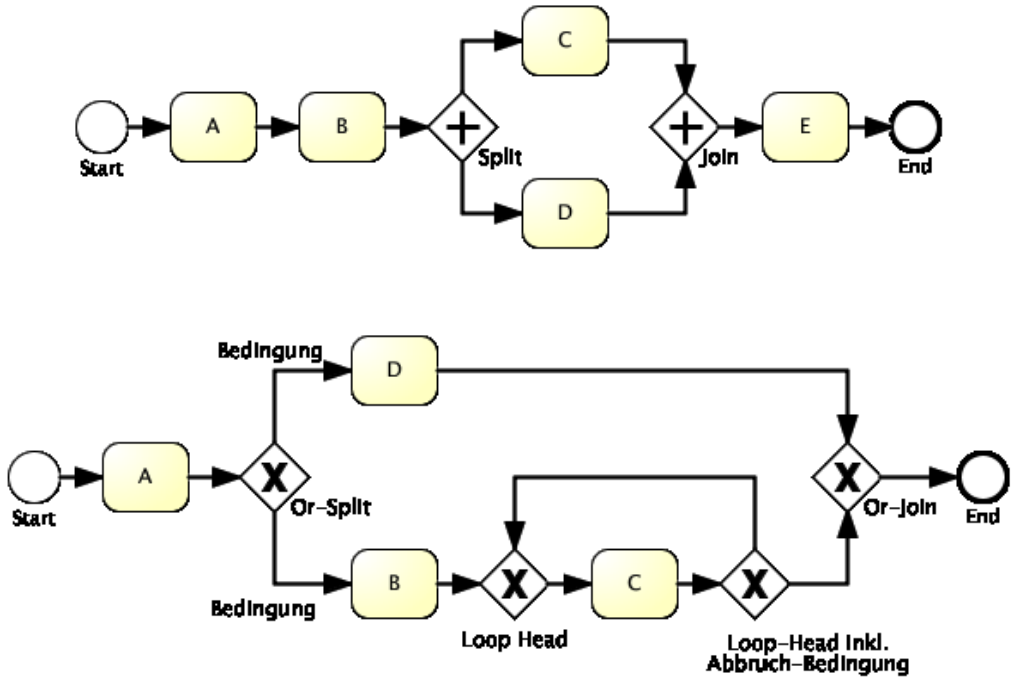


Abbildung 10: Blockstrukturierte Workflow-Modelle

Quelle: Eigene Erstellung

In Abbildung 10 werden zwei blockstrukturierte Modelle gezeigt. Das erste Modell beinhaltet eine Parallelität (AND), welche durch den `AND-Split`-Knoten aufgemacht und mit dem `AND-Join`-Knoten wieder geschlossen wird. Das zweite Modell zeigt eine Verschachtelung mehrerer Split/Join-Fragmente. Zuerst wird eine Kondition `OR-Split` aufgemacht. Wird der Alternativ-Zweig mit Aktivität `B` gewählt, wird eine Iteration (`Loop-Head`) durchlaufen, welche ein Kind-Fragment aufmacht. Die Blockstrukturierung verlangt, dass dieses Kind-Fragment wieder geschlossen wird, gefolgt vom Eltern-Fragment, welches die Kondition ist. Dieses Fragment wird durch den Knoten `OR-Join` geschlossen.

Nachfolgend werden Beispiele für nicht-blockstrukturierte Workflow-Modelle gezeigt:

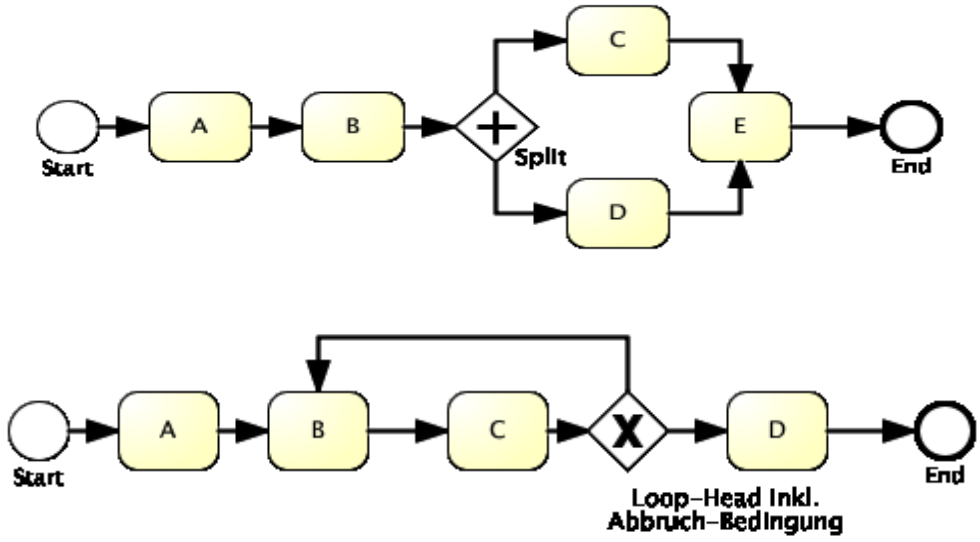


Abbildung 11*:* Nicht-blockstrukturierte Workflow-Modelle

Quelle: Eigene Erstellung

Im weiteren Verlauf der Arbeit wird der Term „Prozessmodell“ und „blockstrukturiertes Workflow-Modell“ gleichgesetzt.

2.1.1 Schwache Blockstrukturierung

Eine strenge Anwendung der Blockstrukturierung kann das Prozessmodell jedoch in der Ausführbarkeit und Semantik beschränken. In der Blockstrukturierung wird die Annahme getroffen, dass der korrespondierende Join-Knoten immer vom selben Typ wie der Split Knoten ist. Zum Beispiel muss ein AND-Split mit meinem AND-Join-Knoten geschlossen werden. Moderne Workflow-Tools unterstützen Split-Join-Kombinationen, die über dieses Verhalten hinausgehen [AHKB03], jedoch Blockstrukturierung aufweisen. Korrespondieren die beiden Split- und Join-Knoten nicht, wird aber die Blockstrukturierung eingehalten, spricht man von einer schwachen Blockstrukturierung. Des weiteren kann mit der schwachen Blockstrukturierung eine spezielle Sync-Kante eingeführt werden, welche den Prozessablauf insbesondere innerhalb von Parallelitäten beeinflussen kann. Solch eine Kante dient zur Synchronisation von Alternativ-Zweigen [RMDP98]. Durch diese Sync-Kanten können Aktivitäten dazu gezwungen werden erst dann zu feuern, wenn eine andere Aktivität einen definierten Zustand in der Workflow-Instanz erreicht hat.

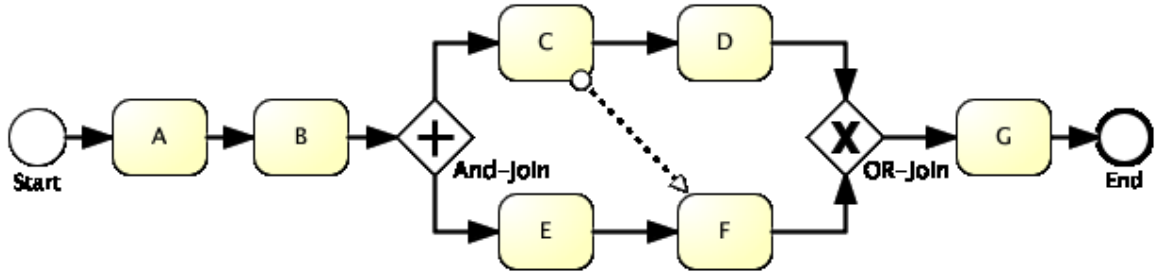


Abbildung 12*:* Schwach blockstrukturierte Workflow-Modelle

Quelle: Eigene Erstellung

In Abbildung 12 ist ein schwach blockstrukturiertes Prozessmodell zu sehen. Es existiert eine Sync-Kante zwischen den Aktivitäten C und F. Mittels dieser Sync-Kante feuert die Aktivität F erst dann, wenn Aktivität C einen bestimmten Zustand erreicht hat. Ebenso befindet sich ein Fragment im Prozessmodell, welches einen AND-Split-Knoten besitzt und mit einem OR-Join-Knoten endet. Obwohl die beiden Fragment-Knoten unterschiedlich sind, ist das Prozessmodell blockstrukturiert.

2.1.2 Sichten auf Prozessmodelle

Bislang wurden Prozessmodelle nur als eine Anreihung von Aufgaben (Knoten) gesehen. Ein Prozessmodell im Sinne eines Geschäftsprozess beinhaltet jedoch weitere Informationen [AHKB03]. So kann es für die Ausführung notwendig sein, dass eine Aufgabe nur von einer bestimmten Maschine oder Rolle durchgeführt werden darf, oder Daten im Prozess verarbeitet und erzeugt werden. Dementsprechend kann ein Prozessmodell in die folgenden drei Sichten unterteilt werden:

- *Control-Flow-Sicht* [RNHM06]: Die Control-Flow-Sicht beschreibt in einem Prozessmodell die Ausführungsreihenfolge der einzelnen Aktivitäten, durch aufeinanderfolgende Anreihungen oder durch spezielle Fragmente wie die Parallelität, Kondition oder Iteration.
- *Data-Flow-Sicht* [RAEH05]: In der Data-Flow-Sicht werden prozessspezifische Daten und Variablen beschrieben. Charakteristisch für diese Sicht ist die Definition der Daten hinsichtlich ihrer Sichtbarkeit für andere Prozess-Komponenten, der Interaktion der Daten mit den Aktivitäten im Prozess, des Datentransfers und der Beeinflussbarkeit des Control-Flows (zum Beispiel bei einer Iteration).

- *Ressourcen-Sicht* [AEHR05]: Die Resource-Sicht beinhaltet sämtliche Ressourcen, die an einem Prozess teilnehmen. Eine Ressource kann dabei eine menschliche Arbeitskraft sein, eine Maschine oder ein spezielles Equipment. Bestandteil der Resource-Sicht ist es, zuvor definierte Ressourcen-Gruppen zu den Prozess-aktivitäten zuzuordnen, um dann bei der Prozessausführung einer Aktivität, eine konkrete Ressource zu delegieren.

Gelegentlich wird auch noch eine vierte Dimension, das *Exception-Handlings* [RNAH06], hinzugefügt. Beim Exception-Handling wird auf die Fehlerbehandlung bei einer unvorhergesehenen Abweichung vom vordefinierten Prozess eingegangen. Das inkludiert Fehler in allen drei Dimensionen, sowie die Auflösung der Ausnahmesituation. Exception-Handling spielt eine wichtige Rolle bei Prozessinstanzen.

2.1.3 Korrektheit von Prozessmodellen

Um zu gewährleisten, dass das Prozessmodell einen eindeutigen Control-Flow besitzt und bei der Ausführung keine Laufzeitfehler aus der Semantik des Prozessmodells hervorgehen, sollten Prozessmodelle einem Korrektheitskriterium unterliegen. Dabei unterteilt man das Korrektheitskriterium in die strukturelle Korrektheit und in die Verhaltenskorrektheit.
Prozessmodelle werden als strukturell-korrekt bezeichnet, wenn die folgenden Kriterien auf das Modell zutreffen [WESK07]:

- Es gibt einen Start-Knoten, welcher keine eingehende Kante besitzt.
- Es gibt einen Ende-Knoten, welcher keine ausgehende Kante besitzt.
- Jeder Knoten im Prozessmodell ist über einen Weg beginnend vom Start-Knoten weg erreichbar und es existiert mindestens ein Weg über die nachfolgenden Knoten welche den Ende-Knoten erreichen.

Dabei ist anzumerken, dass die ersten zwei Punkte bereits durch die vorangegangenen Definitionen über Workflow-Modelle gegeben sind.

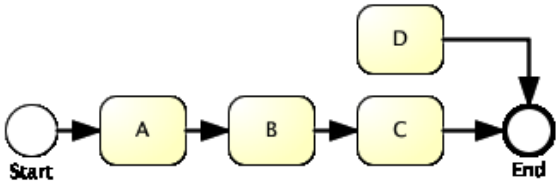


Abbildung 13: Ein strukturell-inkorrektes Workflow-Modell

Quelle: Eigene Erstellung

Die Abbildung 13 zeigt ein Modell, welches eine Sequenz „A“, „B“, „C“ enthält. All diese Aktivitäten sind vom Start-Knoten aus erreichbar. Die Aktivität „D“ ist jedoch nie über den Start-Knoten erreichbar. Das Modell ist erst dann wieder strukturell korrekt, wenn zum Beispiel die Aktivität „D“ entfernt wird oder eine entsprechende Kondition eingebaut wird.

Ist nun das Prozessmodell strukturell-korrekt, ist jedoch noch keine Garantie für eine korrekte Terminierung des Prozesses gegeben. [WESK07] unterscheidet dabei zwischen zwei Fällen: Den sogenannten Deadlocks und den Livelocks. Als ein Deadlock bezeichnet man dabei eine Situation, in der ein Prozess „hängen“ bleibt (der Status des Prozesses ändert sich nicht) und keine Terminierung mehr möglich ist.

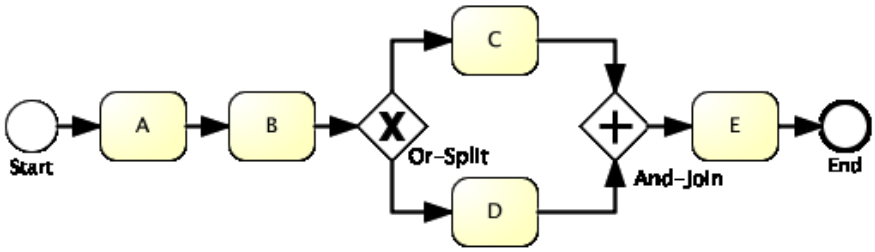


Abbildung 14: Deadlock durch falschen Join

Quelle: Eigene Erstellung

Dagegen ist ein Livelock eine Situation, in der die Aktivitäten in einem Prozess ausgeführt werden, sich der Status des Prozesses entsprechend ändert, es aber zu keiner Terminierung des Prozesses kommt. Es entsteht eine Endlos-Schleife.

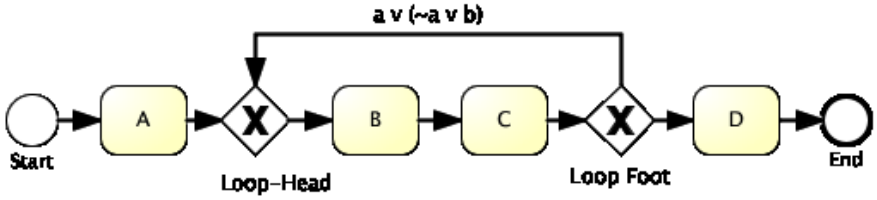


Abbildung 15: Livelock durch Tautologie

Quelle: Eigene Erstellung

Wie die Abbildungen 14 und 15 zeigen, können solche Verhaltensproblematiken auch dann entstehen, wenn die obigen drei Bedingungen für die strukturelle Korrektheit erfüllt sind.

Daher wird das Korrektheitskriterium um Verhaltensbedingungen erweitert [WESK07]:

- Wird der Start-Knoten ausgeführt, darf keine andere Aktivität im Prozessmodell aktiv sein. Dies wird auch als Initial-Zustand bezeichnet.
- Wird der Ende-Knoten terminiert, darf keine andere Aktivität mehr aktiv sein – der sogenannte End-Zustand.

Weiters

- In jedem beliebigen Zustand der Prozessinstanz muss eine Sequenz existieren, welche zum End-Zustand führt.
- Der End-Zustand muss über den Start-Zustand erreicht werden.
- Es dürfen keine toten Aktivitäten im Prozessmodell vorhanden sein. Sprich, jede Aktivität muss zumindest in einer Prozessinstanz enthalten sein.

2.2 Differenzen zwischen Prozessmodellen

Werden zwei Prozessmodelle PM_1 und PM_2 miteinander verglichen, sind sie verschieden (es existieren Änderungen), wenn es einen Transformationsprozess Δ mit Änderungen gibt, der PM_1 durch Anwendung von Δ zu PM_2 überführt. Änderungen beziehen sich dabei auf alle drei Sichten eines Prozessmodells [WBRR08]. [LCRW06] definiert den Transformationsprozess wie folgt: Seien PM_1 und PM_2 Prozessmodelle, Δ eine Menge von Änderungen und $\sigma = \{\Delta_1, \Delta_{2,\dots}, \Delta_n\}$ eine Sequenz von Transformationsprozessen, dann gilt:

- $PM_1 [\Delta \rangle PM_2$ nur dann wenn Δ auf PM_1 anwendbar ist, entsteht durch Anwendung von Δ auf PM_1 das Prozessmodell PM_2
- $PM [\sigma \rangle PM'$ nur dann wenn $\exists PM_1, PM_{2,\dots} PM_{n+1}$ mit $PM = PM_1$, $PM' = PM_{n+1}$ und $PM_i [\Delta_i \rangle PM_{i+1}$ wobei $i = \{1,2,\dots n\}$.

Die erste Definition spiegelt eine einfache Transformation zwischen zwei Prozessmodellen wider. Während die zweite Definition eine geordnete Menge von Transformationen (Änderungs-Sequenz) beschreibt. Hier wird die evolutionäre Entwicklung eines Prozessmodells beschrieben. Ausgehend von einem initialen Prozessmodell PM_1 und der Anwendung von Δ_1 , entsteht das Prozessmodell PM_2 . Wird auf PM_2 nun Δ_2 angewendet, ergibt sich PM_3 , bis sich schlussendlich PM_{n+1} ergibt.
Des weiteren wird angenommen, wenn $\Delta = \emptyset$ dann gilt, die beiden Prozessmodelle sind äquivalent und somit existieren keine Änderungen zwischen den beiden Modellen.

Informationen über die Abänderungen zwischen zwei Prozessmodellen, kann auf zwei verschiedene Arten erfolgen. [RRJK06] unterscheidet dabei zwischen Change-Primitives und den High-Level-Changes.

Change-Primitives sind dabei einfache Manipulationen auf den Prozessgraphen. Diese Manipulationen können direkt auf den Graphen ausgeführt werden.

Das Set an Change-Primitives setzt sich dabei aus denn folgende vier Operationen zusammen:

- `AddNode(n)`: Hinzufügen eines Knoten `n` zum Prozessmodell.
- `DeleteNode(n)`: Entfernen eines Knoten `n` vom Prozessmodell.
- `AddEdge(n1,n2)`: Hinzufügen einer Kante zum Prozessmodell, wobei die Kante zwischen den beiden Knoten `n1` und `n2` platziert wird. Dabei ist `n1` das Objekt mit der ausgehenden Kante und `n2` jenes mit der einmündenden Kante.
- `RemoveEdge(n1,n2)`: Entfernen einer Kante zwischen den Prozessobjekten `n1` und `n2`.

Zwei Beispiele sollen die Anwendung von Change-Primitives verdeutlichen:

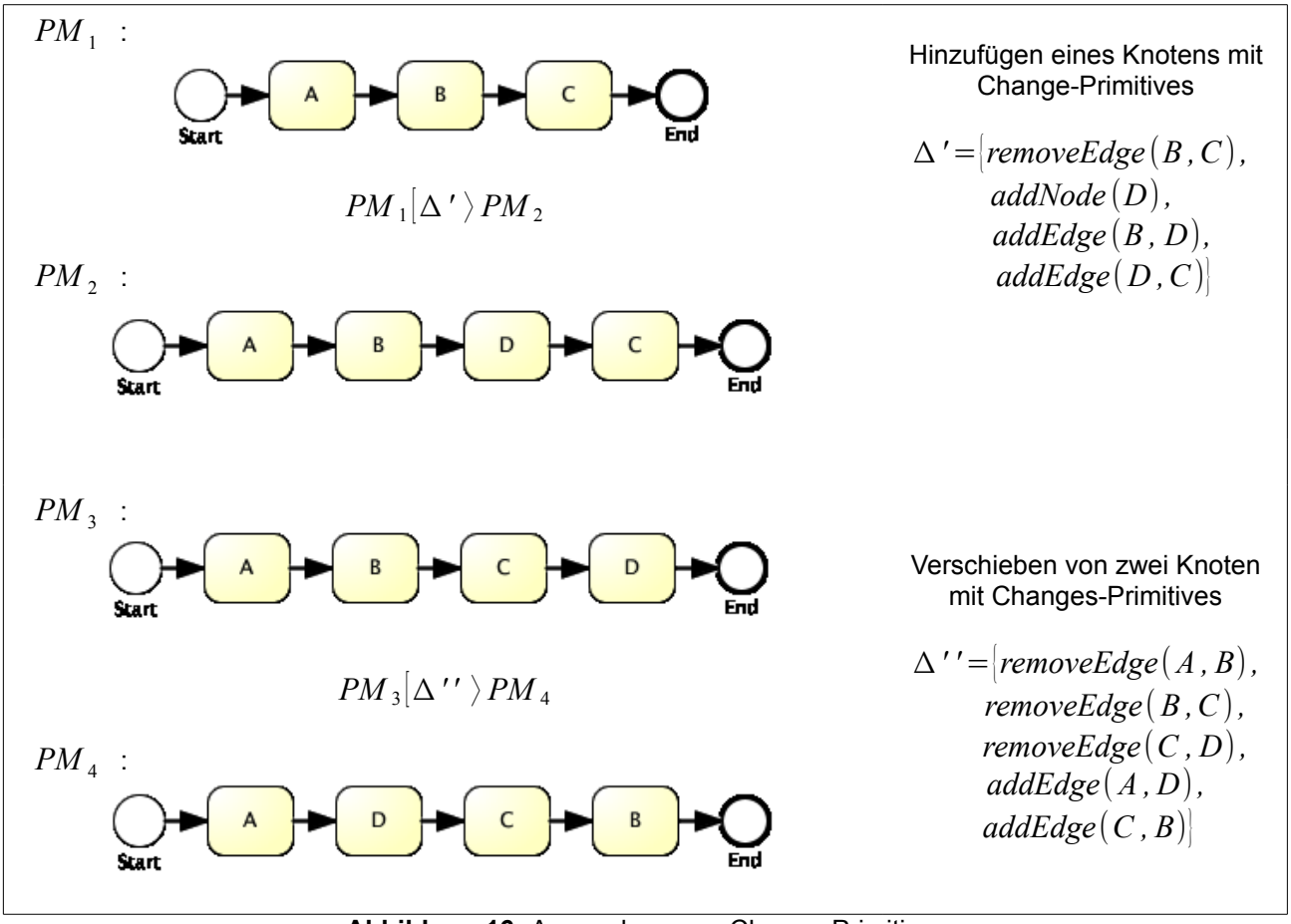


Abbildung 16: Anwendung von Change-Primitives

Die Beispiele in Abbildung 16 zeigen die Anwendung von Change-Primitives. Im ersten Beispiel wird ein Knoten zum Prozessmodell PM_1 hinzugefügt. Da nur Change-Primitives angewendet werden, besteht die Menge Δ' aus vier Graphenoperationen. Im

zweiten Beispiel wird das Verschieben von zwei bereits vorhanden Knoten gezeigt. Dies kann über Change-Primitives mittels fünf Lösch- und Hinzufüge-Operationen vollzogen werden.

Werden die Änderungen als Change-Primitives beschrieben, kann die Menge an durchzuführenden Änderungsoperationen relativ stark anwachsen. Eine semantische Auswertung durch den Benutzer ist somit nur schwer möglich und bei unvollständiger Anwendung von Change-Primitives können inkorrekte Prozessmodelle entstehen [LCRW08].
Abhilfe schaffen hier High-Level-Changes. Sie sind semantisch höherwertige Informationen über die Abänderungen eines Prozessmodells. Anstatt einer Menge von Graphenmanipulationen – wie im obigen Beispiel gezeigt – wird nur mehr eine Änderungsoperation angeführt. Angewendet auf das „Add“-Beispiel bedeutet dies, dass statt der Änderungssequenz (`removeEdge, addNode, addEdge, addEdge`) nur mehr die High-Level-Change-Operation `add(Obj,zw1,zw2)` ausgewiesen wird. `add(Obj,zw1,zw2)` besagt somit, dass das Prozessobjekt `Obj` zwischen den Prozessobjekten `zw1` und `zw2` angelegt wird. Die entsprechenden Change-Primitives scheinen nicht mehr in den Änderungen auf, werden aber implizit durch den High-Level-Change ausgeführt.

Durch die Anwendung von High-Level-Change-Operationen entstehen, ähnlich wie in der Softwareentwicklung, Adaptierungspattern [WBRR08] welche weniger fehleranfällig sind. Dadurch wird die Korrektheit des Prozessmodells garantiert [RMDP98]. Gleichzeitig sind diese intuitiver als Change-Primitives [LCRW08]. Die sechs wichtigsten High-Level-Changes zur Manipulation von Geschäftsprozessen sind:

- *Add*: Hinzufügen eines neuen Knotens.
- *Delete*: Löschen eines Knotens.
- *Move*: Verschieben eines Knotens an eine andere Position.
- *Replace*: Ein Knoten wird durch einen anderen Knoten ersetzt.
- *Swap*: Knoten werden innerhalb einer Sequenz vertauscht.
- *Update*: Die Eigenschaften eines Knotens werden geändert.

Analysen von [WBRR08] haben ergeben, dass weitere acht Adaptierungspatterns für die strukturelle Anpassung von Prozessmodellen existieren:

- *Condition/OR-Embedment:* Einfügen einer Kondition innerhalb einer Sequenz, wenn Prozessobjekte nur unter bestimmten Bedingungen ausgeführt werden sollen.
- *Parallelism/And-Embedment:* Einfügen einer Parallelität, wenn Prozessobjekte simultan ausgeführt werden können.
- *Iteration/LOOP-Embedment*: Einfügen einer Wiederholung von Prozessobjekten.
- *Alternative-Add*: Einfügen einer zusätzlichen Kante. Anders als bei dem Change-Primitive `addEdge`, handelt es sich bei Alternative-Add um das Hinzufügen eines zusätzlichen Alternativ-Zweiges bei einer Parallelität, Iteration oder einer Kondition, welcher Informationen über den Kontrollfluss beinhaltet.
- *Alternative-Remove*: Entfernen einer Alternativ-Kante in einer Parallelität, Iteration oder Kondition.
- *Extract:* Um Prozessmodelle zu modularisieren, beziehungsweise semantisch zusammenhängende Fragmente zu gruppieren, gibt es die Möglichkeit Subprozesse einzupflegen [AHKB03][RWMH11]. Das Entfernen von Bestandteilen eines Prozesses um diese als Subprozess zu implementieren, wird mit dem Highlevel-Change Extract beschrieben.
- *Inline:* Inline ist die entgegengesetzte Operation zu Extract. Sie fügt die Prozessobjekte aus dem Submodell in das eigentliche Prozessmodell ein. Der Aufruf des Subprozesses ist somit nicht mehr vorhanden.
- *Copy*: Erstellen von Duplikaten von Prozessobjekten oder Fragmenten mit anschließendem Einfügen an einer anderen Position im Prozessmodell. Die originalen Objekte bleiben dabei weiterhin im Prozessmodell bestehen.

Um sicherzustellen, dass die Nachvollziehbarkeit von evolutionären Veränderungen von Prozessmodellen gewährleistet ist, sollten die gefundenen Differenzen gespeichert werden. Eine Speicherung der Deltas erlaubt dann eine vereinfachte Weiterverarbeitung hinsichtlich des Change-Minings [GRRA06]. Änderungen können in einem Change-Log abgelegt werden. Ein Change-Log ist dabei nach [RRJK06] wie folgt aufgebaut:

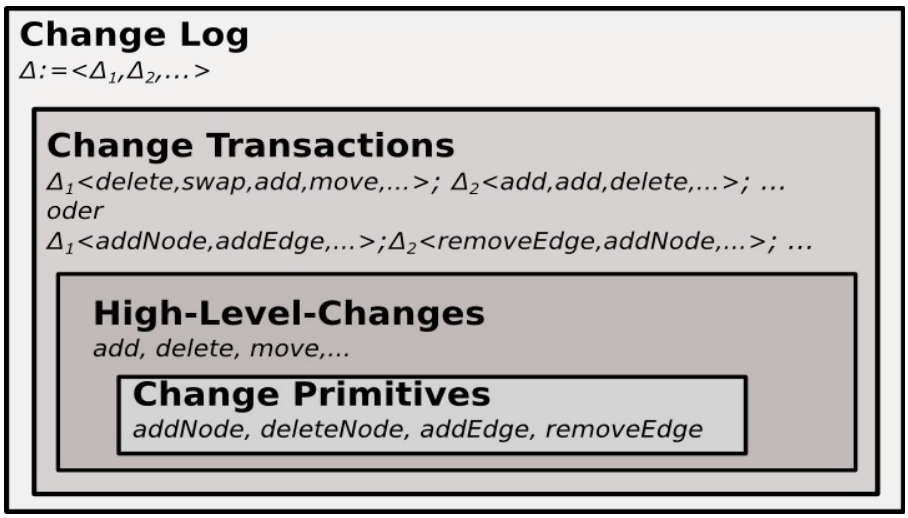


Abbildung 17: Aufbau eines Change-Logs

Quelle: in Anlehnung an [RRJK06]

Abbildung 17 zeigt die grundsätzliche Zusammenstellung eines Change-Logs. Das Change-Log beinhaltet die Change-Primitives oder die High-Level-Changes. Um die Korrektheit der Änderungen zu gewähren, kann es notwendig werden sowohl Change-Primitives als auch die High-Level-Changes zu speichern. Weiters beinhaltet eine Change-Log die sogenannten Change-Transactions. Unter Change-Transactions ist die Abfolge der Änderungen zu verstehen. Änderungsabfolgen sind dann notwendig, wenn bei Anwendung eines beliebigen Change-Primitives oder High-Level-Changes ein inkonsistentes Prozessmodell entstehen würde. Ein Beispiel soll dies verdeutlichen:

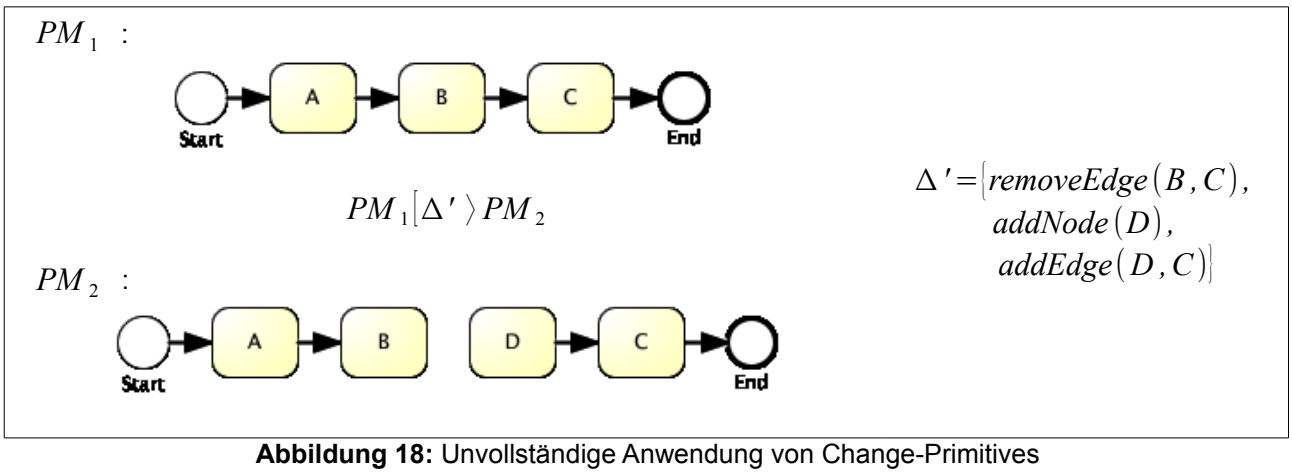


Abbildung 18: Unvollständige Anwendung von Change-Primitives

Das Abbildung 18 zeigt ein Hinzufügen der Aktivität „D“. Wie aus dem Beispiel „Hinzufügen eines Knotens mit Change-Primitives“ von Abbildung 16 auf Seite 33 bereits bekannt ist, werden für das Hinzufügen einer neuer Aktivität vier Change-Primitives

benötigt. Im obigen Beispiel wurden aber fälschlicherweise nur drei Change-Primitives (`removeEdge`, `addNode` und `addEdge`) angewendet. Es fehlt ein Change-Primitive, `addEdge`, welches die Aktivitäten „B“ und „D“ miteinander verbindet.

Um dies zu verhindern, wird die Abfolge der Änderungen in Change-Transactions zusammengefasst. Wird eine Änderungsoperation durchgeführt, müssen auch alle anderen Änderungsoperationen in der Change-Transaction, ebenfalls durchgeführt werden.
Die Menge aller Change-Transactions ergibt dann das Change-Log und bildet alle Änderungen eines Prozessmodells ab.
Formal lässt sich ein Change-Log und eine Change-Transaction wie folgt beschreiben [RRJK06]: Sei PM ein Prozessmodell und Δ ein Change-Log mit einer geordneten Menge von Change-Transactions $\Delta := \langle \Delta_{1,} \Delta_{2,\ldots} \Delta_{n} \rangle$, welche auf PM angewendet werden. Eine Change-Transaction $\Delta_i := \langle cOp_1, \ldots, cOp_k \rangle (j = 1, \ldots, n)$ beinhaltet eine Sequenz von k -Change-Operationen cOp . cOp können dabei entweder Change-Primitives oder Highlevel-Changes sein, wobei alle $cOp_1, \ldots, cOp_k$ in Δ_j angewendet werden oder kein einziges cOp .

Eine weiterer Vorteil bei Verwendung von Change-Logs, betreffend der Versionskontrolle von Prozessen, ist die Unterlegung der Change-Logs mit versionskontroll-relevanten Metainformationen ,wie zum Beispiel, der Zeitpunkt der Änderung, der Kommentar zur Änderung oder die Person, welche die Änderung durchgeführt hat.

3 Vereinheitlichung von Prozessmodellen

Geschäftsprozessnotationen existieren bereits seit den 1970er-Jahren [ZMUM04] und werden für spezielle Domänen erstellt [DELP03]. Die frühe Entstehung von Prozessnotationen und der bis heute fehlende endgültige Standard in der Industrie für Prozessmodelle [HLBB13], führt zu heterogenen Meta Modellen zwischen den verschiedenen Prozessmodellnotationen. Hinsichtlich einer generischen Versionskontrolle sind die folgenden Unterschiede zwischen den Prozessmodellnotationen in den Mittelpunkt zu stellen und müssen daher entsprechend überwunden werden:

- *Terminologie der Prozessobjekte [HLBB13]:* Prozessobjekte, welche grundsätzlich die gleiche Semantik besitzen, werden unterschiedlich bezeichnet. So zum Beispiel, wird ein Task in BPNM „Activity“, in EPC als „Funktion“ und in CPEE-Cockpit als „Task“ bezeichnet.

- *Abbildungen des Kontrollflusses [MJNN05]:* Beziehungen zwischen den einzelnen Prozessobjekten können auf unterschiedliche Arten abgelegt werden. So werden in manchen Tools, welche BPNM unterstützen, die Objekte explizit mit Kanten-Objekten verbunden. Eine genaue Anordnung der Prozessobjekte - wie zum Beispiel in Woflan und UC4 – ist nicht notwendig, da Vorgänger- und Nachfolger-Beziehungen gesondert abgebildet sind. Jedoch kann auch eine Blockorientierung mit Verschachtelungen, welche aufgrund der Objektanreihung den Kontrollfluss bereits bestimmen, existieren.

- *Komplexität und Umsetzung von Patterns:* Analysen von [VDAW04], [RAEH05] und [AEHR05] haben ergeben, dass nicht alle – untersuchten – Hersteller den gesamten Bestand an identifizierten Patterns unterstützen. Insbesondere in der Unterstützung der Daten- und Ressourcen-Perspektive sind teils große Unterschiede zu erkennen. Auffallend ist auch, dass bei der Pattern-Identifizierung herstellerspezifische Pattern existieren, welche von keinen der anderen Hersteller implementiert wurde.

In den letzten zehn Jahren wurden Versuche unternommen dem entgegenzuwirken, indem Standards eingeführt wurden. So bildeten sich unterschiedliche Empfehlungen heraus wie von OMG, OASIS, BPMI, UN/CEFACT oder WfMC/XPDL. Dennoch gibt es auch hier starke Abweichungen. [DELP03][KRLL09].
Sollen in einer Analyse mehrere Prozessmodellnotationen berücksichtigt werden, oder auch bei einer Vereinheitlichung von unterschiedlichen Notationen, schlagen [MJNN05], [HLBB13] und [LBKB06] eine Transformation der spezialisierten Prozessmodellnotationen in eine weniger spezialisierte und abstrakte Prozessnotation vor. Die Vereinheitlichung erfolgt dabei über eine Projektion auf ein eigens geschaffenes Meta-Modell. Da eine generische Versionskontrolle nicht auf eine bestimmte Prozessnotation restriktiert werden darf, wird im Folgenden ein Meta-Modell, welches sich sowohl für die Differenzerkennung, als auch für die Versionskontrolle eignet, erstellt.

Das Meta-Modell, als ein Zwischenformat in der Versionskontrolle, nimmt dabei mehrere Funktionen wahr: Wie schon erwähnt, können durch das eigens erstelle Meta-Modell Unterschiede in den speziellen Modellierungsnotationen eliminiert werden, wodurch eine Prozessmodell-unabhängige Implementierung der Algorithmen erfolgen konnte. So musste die Differenzenerkennung nur noch für eine „Sprache“ umgesetzt werden. Das Meta-Modell kann in einem sowohl für Menschen also auch Maschinen lesbaren Format abgelegt werden. Dadurch kann ein Mapping zwischen der speziellen Modellierungs-notation und den notwendigen Informationen für die Differenzenerkennung und Versionskontrolle durchgeführt werden. Jede Modellierungsnotation benötigt hierbei je zwei, eigens erstellte, Mappings. Das erste Mapping wird notwendig wenn die spezielle Notation zum Meta-Modell transformiert wird, das zweite Mapping wenn ein Prozess-modell aus der Versionskontrolle ausgebucht wird, also vom Meta-Modell zurück in die ursprüngliche Notation gewandelt wird. Des weiteren ermöglicht das vereinheitlichte Meta-Modell Prozessmodelle homogen in einem Repository wie es im Kapitel „Versionskontrolle“ beschrieben wird abzulegen. Dies hat den Vorteil, dass effizientere Methoden für die Versionskontrolle, hier insbesondere für das Kapitel „Rückrechnung von Prozessmodellen“, erstellt werden können, da Mapping-Prozesse innerhalb der Versionskontrolle wegfallen.
Die beschriebenen Algorithmen und Konzepte bauen auf einem sogenannten „Process Structure Tree“ auf. Das Meta-Modell dient auch gleichzeitig zur die Automatisierung der Serialisierung dieses Baums.

3.1 Das Meta-Modell

Das Meta-Modell ist ein abstraktes Modell, welches eigens für einen bestimmten Zweck erstellt wird und ist auch nur dafür einsetzbar. Das nachstehende Meta-Modell wurde mit dem Ziel erstellt, ein möglichst herstellerunabhängiges „Superset“ für die Differenzenerkennung und Versionskontrolle von blockstrukturierten Prozessmodellen zu schaffen. Die verschiedenen Aspekte eines Prozessmodells nach Curtis [CBKO92] spielen in diesem Meta-Modell eine untergeordnete Rolle. Wichtiger ist es, ein minimales Modell zu erstellen, welches den Fokus auf einem deskriptiven und strukturellen Aspekt der einzelnen Prozessobjekte des Prozessmodells hat. Als deskriptiven Aspekt sind die Attribute und deren Attributwerte von Prozessobjekten-Eigenschaften zu verstehen. Unter dem strukturellen Aspekt werden die Position sowie Vorgänger- als auch Nachfolger-Beziehungen von Objekten zusammengefasst.

Anforderung an das Meta-Modell ist di Unterstützung der folgenden grundlegenden Workflow-Konzepte:

- *Aktivitäten-Anreihungen*: Abbildung von Sequenzen – Objekte mit einem Vorgänger und Nachfolger.
- *Split*: Abbildung von Verzweigungen – Objekte, welche einen Vorgänger, aber mehrere Nachfolger besitzen.
- *Join:* Zusammenführung eines Splits – Objekte mit mehreren Vorgängern, aber nur einem Nachfolger.
- *Iterationen*: Darstellung von Schleifen in einem Prozessmodell

sowie die, in der Einführung unterstellte, Blockstrukturierung und Definitionen über Workflows.

Wie schon erwähnt sind die traditionellen Aspekte [CBKO92] eines Prozessobjektes für die Differenzenerkennung und Versionskontrolle nicht notwendig. In der obigen Aufzählung fehlen daher bewusst die verhaltensorientierten Pattern nach [AHKB03] wie Parallelisierung (AND) und Kondition (OR). Das spezielle Verhalten eines Prozessobjektes in einer Prozessinstanz, wird von der Workflow-Engine umgesetzt. Hier ist es unerlässlich wie sich ein Split-Knoten und ein Join-Knoten verhält. Die Workflow-Engine determiniert,

ob nun alle Alternativen aufgrund eines Parallelisierungs-Knotens ausgeführt werden dürfen, oder ob nur ausgewählte Zweige, welche bestimmten Konditionen entsprechen, feuern. Die genaue Ausführbarkeit der einzelnen Prozessobjekte ist nicht Bestandteil der Versionskontrolle und muss auch nicht gesondert bei der Differenzenerkennung behandelt werden.

Für den zugrunde liegenden Zweck, kann die Spezialisierung des Prozessobjektes über den deskriptiven Charakter angegeben werden. Bei einer späteren Transformierung zwischen Meta-Modell und der spezialisierten Notation, kann diese Information wieder verwendet werden, um die Prozessobjekte auf die ursprüngliche Terminologie zu mappen.
Split- und Join-Knoten werden in das Meta-Modell aufgenommen, um so die Struktur des Prozessmodells analysieren zu können. Der Split-Knoten macht einen neuen Block auf und ein Join-Knoten wieder zu. So lassen sich wichtige Informationen über die Hierarchien und Positionen von Prozessobjekten herleiten.
Zwar kann eine Iteration als eine Spezialform einer Kondition gesehen werden, allerdings wird diese, wie auch beim WfMC-Standard [WfMC98], als ein eigenständiges Prozessobjekt (beziehungsweise als Block) gesehen. Die Iterationen unterliegen einer besonderen Behandlung, da die diese eine Rückwertskante zum Split-Knoten besitzen. Weiters kann bei einer Iteration die Abbruch-Bedingung im Schleifen-Kopf, als auch im Schleifen-Fuß enthalten sein. Aufgrund dieser zwei Gegebenheiten benötigt die Iteration eine gesonderte Behandlung, obwohl sie strukturell gesehen, ebenfalls als ein Block darstellbar ist.

Komplexe Workflow-Pattern, welche von [AHKB03] identifiziert wurden, können grundsätzlich als spezielle Ausprägungen der obigen vier Punkte betrachtet werden. So zum Beispiel ist ein „Discriminator“ oder ein „Multi-Merge“, eine Spezialform eines Join-Knotens. Ein „Sub-Flow“, welcher einen anderen Workflow aufruft, ist in dieser Domäne eine Aktivität.

Des weiteren existieren in manchen Modellierungsnotationen verhaltensorientierte Aktivitäten, welche gezielt auf den Typ der Aktivität ausgerichtet sind. So zum Beispiel hat UC4 je nach Task-Typ (UNIX-Job, Data-Stage-Job, File-Patch-Job, u.a.), unterschiedliche Objekte mit entsprechenden Konfigurationen. Im CPEE-Cockpit wiederum existieren drei unterschiedliche Ausprägungen für eine Aktivität. Jede dieser Ausprägungen führt dann die

entsprechenden Aufgaben durch.

Im Meta-Modell werden daher alle Prozessobjekte, die genau einen Vorgänger sowie genau einen Nachfolger besitzen, als Aktivitäten betrachtet.

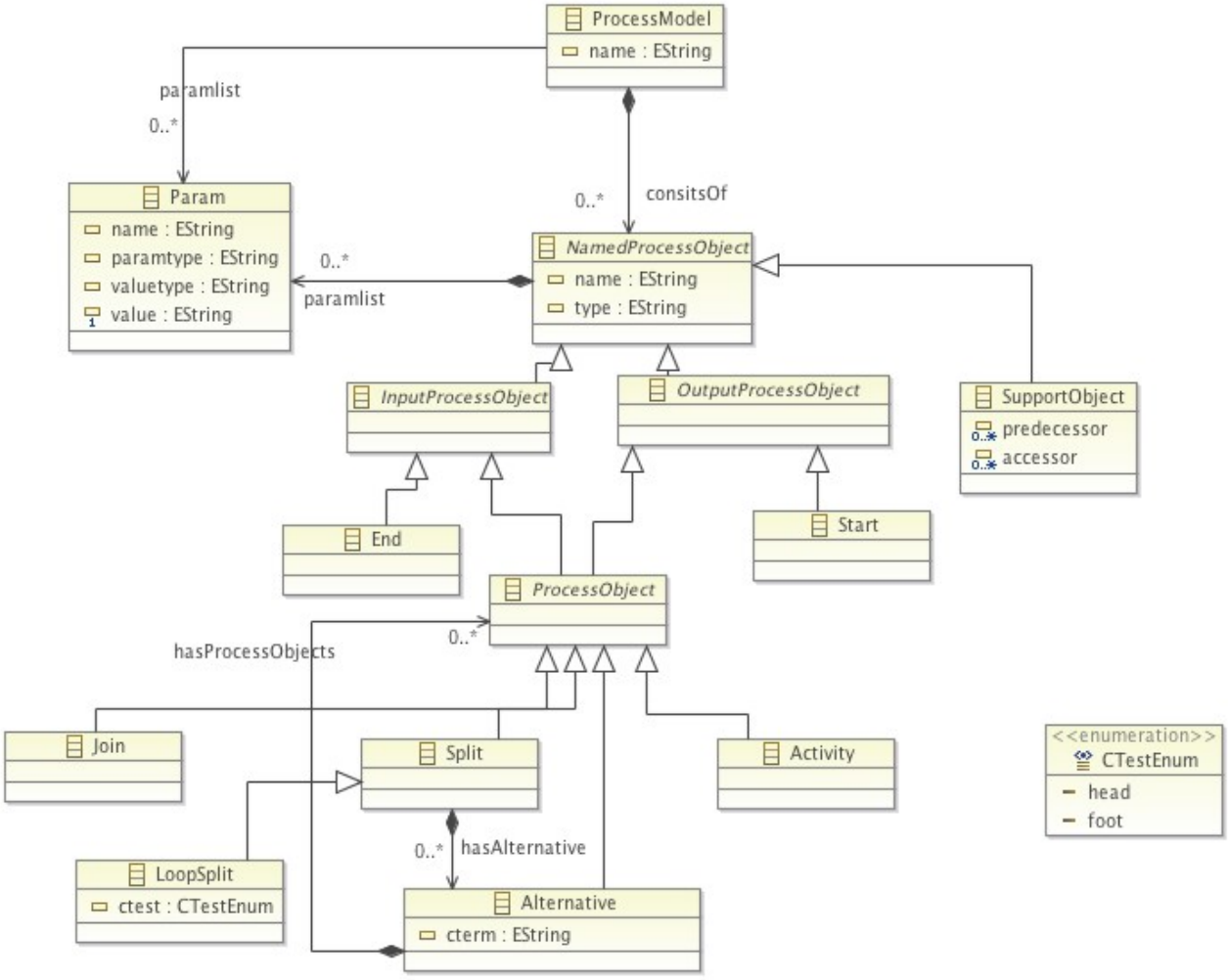


Abbildung 19: Meta-Modell für „Differenzenerkennung und Versionskontrolle“

Quelle: Eigene Erstellung

Abbildung 19 zeigt ein minimales Meta-Modell für die Domäne der Differenzenerkennung und Versionskontrolle für Prozessmodelle, in welchem alle Informationen eines Prozessobjektes abgebildet werden können. In dieser Abbildung ist ersichtlich, dass ein abstraktes Objekt `NamedProcessObject` existiert. Aus diesem Objekt lassen sich alle anderen Objekte, welche im Prozessmodell enthalten sind, ableiten. Dabei hat das `NamedProcessObject` ein Attribut `name`, welches die Kurzbeschreibung des Prozessobjektes sein. kann In vielen Prozessmodellierungsnotationen ist dies die Label-Beschriftung eines Objektes. Der Name sollte dabei nicht mit der ID verwechselt werden

wie sie von [XPDL12][WfMC98] vorgeschlagen wird. Der Name beziehungsweise die Kurzbeschreibung kann sich – anders als ein eindeutiger Identifizierer – im Laufe der Zeit durchaus ändern. Abänderungen des Attributs `name` Aufschluss über den Umfang der Änderungen des Objektes eher widerspiegeln als eine ID (siehe nachfolgende Kapitel).

Das Attribut `type` gibt den Prozessobjekttyp in der speziellen Notation an. So gibt es zum Beispiel, in UC4 unterschiedliche Job-Typenobjekte wie „Unix-Job“ oder „Data-Stage-Job“. In CPEE-Cockpit existieren Prozessobjekttypen, wie „Script-Task“ oder „Service Call“. Durch das Setzen des `type`-Attribut kann wieder eine Rücktransformation in die spezielle Notation erfolgen.

Die Summe aller `NamedProcessObjects` ergibt dann das komplette Prozessmodell der speziellen Modellierungsnotation.

3.1.1 Abbildung der Control-Flow-Perspektive

Aus dem `NamedProcessObjekt` lassen sich zwei weitere abstrakte Objekte ableiten. Das `InputProcessObject`, welches einen Prozess-Vorgänger besitzt und das `OutputProcessObject`. Dieses besitzt einen Prozess-Nachfolger. Über diese zwei abstrakten Objekte lassen sich die konkreten Start- und End-Knoten abbilden. Wie in der Einleitung definiert, müssen Workflows genau einen Start- und einen End-Knoten besitzen, wobei der Start-Knoten kein Vorgänger-Objekt und der End-Knoten kein Nachfolger-Objekt aufweisen darf.
Objekte, welche sowohl Vorgänger, als auch Nachfolger ausweisen, werden im abstrakten Objekt `ProcessObject` zusammengefasst. Aus diesem werden dann die konkreten Objekte `Activity`, `Split`, `Join` und `Alternative` abgeleitet.
Da in blockstrukturierten Prozessmodellen alle Aktivitäten nur einen Vorgänger und Nachfolger besitzen [JRPP94], erbt dieses Objekt die jeweiligen Vorgänger- und Nachfolger-Eigenschaften vom `InputProcessObject`-Objekt und `OutputProcessObject`-Objekt.
Die `Split`- und `Join`-Objekte werden ebenfalls als `ProcessObject` abgebildet, treten aber in der tatsächlichen Modellierung als ein Paar auf. Diese Kombination aus Split- und Join-Knoten ergibt sich bereits aus der Blockstrukturierung, da es hier zu jedem Split-

Knoten auch einen Join-Knoten geben muss. Das Split-Objekt hat einen Vorgänger und mehrere Nachfolger, beziehungsweise Abzweigungen. Diese Abzweigungen werden als `Alternative` bezeichnet. Eine Alternative enthält dabei wiederum mehrere Objekte, welche vom abstrakten Typ `ProcessObject` sein müssen. Der Split-Knoten in der speziellen Notation kann dabei alle Alternativ-Kanten oder nur bestimmte Alternativen, wenn diese die angegebene Bedingung erfüllen, feuern. Wann eine Alternative feuert spielt in der zu modellierenden Domäne keine Rolle. Die einzelnen Bedingungen zum Feuern werden direkt in der Alternative im Attribute `cterm` modelliert.

Das `LoopSplit`-Objekt ist eine Spezialisierung des Split-Objektes, da die Schleifenabbruchbedingung einer OR-Kondition gleichkommt [GTKW08]. Der Unterschied besteht darin, dass die Abbruch-Bedingung nicht zwingend im Split-Knoten auftreten muss, sondern diese auch im Schleifenfuß angegeben sein kann. Ob es sich um eine kopf- oder fußgesteuerte Schleife handelt, wird im Attribut `ctest` (= conditiontest) angegeben. Des weiteren wird die Schleife als eigenständiges Objekt beschrieben, da bei einer Rücktransformation vom Meta-Modell in die spezielle Sprache, eine gesonderte Behandlung der Rückwertskante gegeben ist.

3.1.2 Prozessobjekt-Attribute

In manchen Workflow-Tools können die Prozessobjekte noch zusätzlich vom Benutzer genauer konfiguriert werden. Jedes Prozessobjekt hat daher spezielle Attribute, welche mit Werten beladen werden können. Solche Informationen sollten unter eine Versionskontrolle gestellt werden. Des weiteren müssen die Attribute abgebildet werden, um das Prozessobjekt wieder vollständig aus dem generalisierten Meta-Modell wieder in die spezielle Notation zu überführen. Fehlen diese Attribute, kann nach einer Rücktransformation die korrekte Funktionsweise des Objektes, nicht mehr garantiert werden. Dies kann zu fehlerhaften Prozessinstanzen führen. Demzufolge werden die Attribute der Prozessobjekte in einem Objekt `Param` abgelegt. Das `Param`-Objekt ist dabei ein Dreier-Tupel, bestehend aus dem Namen des Attributs, dem Wertetyp und dem gesetzten Attributwert oder Attributwerten.

3.1.3 Daten und Ressourcen-Perspektive im Meta-Modell

Prozessobjekte, welche nicht direkt der Control-Flow-Perspektive zuzuordnen sind, werden mit Hilfe des `SupportObject`-Objektes abgebildet. Typischerweise werden hier Objekte aus der Daten-Perspektive und der Ressourcen-Perspektive modelliert, sofern diese Objekte nicht schon direkt als Attributwert in einem `NamedProcessObject` angegeben sind. Beispiele für solche Objekte sind Variablen, Dokumente oder Hardware-Ressourcen, aber auch spezielle Objekte wie „Swimlanes“ oder „Gruppen“ in BPNM.
Hardware-Systeme, Dokumente oder Steuervariablen können in einem Prozessmodell mehrfach verwendet werden, bei nur einer einzigen Angabe des Objekts im Modell selbst [RAEH05][AEHR05]. Daher wird in den `SupportObejcts`-Objekt, anders als beim `ProcessObject`, die Möglichkeit gegeben, mehrere `predecessor` (Vorgänger) und `accessor` (Nachfolger) anzugeben.

3.1.4 Kanten im Meta-Modell

Im Meta-Modell sind explizit keine Kanten ausgewiesen, da Prozessmodellnotationen existieren (unter anderem CPEE und BPEL), welche aufgrund der Blockstrukturierung der Quelldatei und durch Verschachtelungen von Prozessobjekten, bereits die Vorgänger- und Nachfolger-Beziehungen ergeben. Eine spezielle Angabe von Kanten ist in solchen Modellierungsnotationen nicht notwendig. Die Kanten werden lediglich nur auf der graphischen Oberfläche für den Benutzer angezeigt.
Sollten dennoch Kanten eine besondere Rolle, zum Beispiel zur Darstellung unterschiedlicher Kanten-Ausprägungen (u.a. in BPNM), spielen oder um eine schwache Blockstrukturierung, wie zum Beispiel die Sync-Kanten in ADEPTflex [RMDP98] zu erreichen, können diese über das `SupportObject`-Objekt modelliert werden.

4 Process-Structure-Tree (PST)

Bevor Change-Primitives abgeleitet werden können, muss die Anreihung der Prozessobjekte im Prozessmodell erkannt werden. Die genaue Anreihung ist aus zwei Gründen wichtig: Erstens, kann über die festgelegte Reihenfolge der Objekte erkannt werden, ob ein Objekt an einer Position hinzugefügt oder gelöscht worden ist. Zweitens, können Positionsnummern abgeleitet werden, welche für die Überführung in höherwertige Änderungs-Operationen verwendet werden.
Da Prozessmodelle gerichtete azyklische Graphen (DAG) sind [WSMO00][VAHV02] und demzufolge keine Kreise im Graph entstehen, kann, beginnend beim Start-Knoten, über Vorgänger-Nachfolger-Beziehungen, die Reihenfolgen abgeleitet werden.

Die Eigenschaften eines gerichteten azyklischen Graphen und die Blockstrukturierung von Prozessmodellen ermöglichen es, einen Process-Structure-Tree (PST) zu erstellen. Mit einem Process-Structure-Tree erfolgt eine hierarchische Aufgliederung der Prozessobjekte in einzelne Teilfragmente [VJVK09], wodurch eine Baumstruktur mit allen Teilfragmenten des Prozessmodells entsteht. Diese Teilfragmente enthalten wiederum geordnete Sequenzen von Prozessobjekten. Eine Sequenz besteht dabei aus mehreren sogenannten Single-Entry-Single-Exit-Komponenten (SESE-Komponenten). [JRPP94] Eine SESE-Komponente ist ein Objekt, welches genau eine Eingangskante (Entry-Kante) und eine Ausgangskante (Exit-Kante) besitzt. Ist nun die Exit-Kante gleichzeitig die Entry-Kante eines Nachfolgers, so entsteht eine Sequenz. Sequenzen können wiederum Teilfragmente (Kinder-Fragmente) tragen, welche weitere Sequenzen enthalten. Ein Kind-Fragment kann nur dann erstellt werden, wenn der aktuelle, zu parsende Knoten vom Supertyp „Fragment“ aus dem vorgestellten Meta-Model ist. Die Aktivitäten des Prozessmodells werden als Blätter des Baumes eingetragen [VVLO07].

Der Process-Structure-Tree ist in der Analyse von Prozessmodellen bereits eine etablierte Datenstruktur. Dieser findet, unter anderem, Anwendungen in der Erkennung von Control-Flow Fehlern [VVLO07], im Compilerbau – hier heißen sie Programm-Stucture-Tree [JRPP94], in der Überleitung von unstrukturierten Prozessmodellen nach blockstrukturierten Prozessmodellen [HFKV06][VJVK09], in der Transformation von Prozessmodellen in einer Quell-Modellierungsnotation zu einer Ziel-Modellierungsnotation

[GBLU08] oder auch in der Pattern-Anwendung innerhalb von Prozessmodellen [GTKW08].

Die Erstellung eines Process-Structure-Trees bietet auch Vorteile hinsichtlich der Analyse der Differenzen in einem Prozessmodel: Durch geeignete Traversierungs-Algorithmen wie zum Beispiel Depth-First-Search können die Positionsnummern der Prozessobjekte berechnet werden. Werden alle Knoten mit der gleichen Methodik wie die Positionsnummern-Errechnung besucht, kann aus der hierarchischen Darstellung eine genau definierte lineare Liste (im Weiteren auch als „flache Struktur“ bezeichnet) erstellt werden. Diese „flache“ Struktur dient zur Erkennung der Change-Primitives. Weiters entstehen bei der Dekomposition des Prozessmodells in einen Process-Structure-Tree zwischen den Sequenzen und Teilfragmenten keine Überlappungen im Baum [JRPP94]. Damit kann sich die spätere Analyse für die Highlevel-Changes auf die Teilfragmente beschränken [VVLO07]. Des Weiteren kann für jedes Prozessobjekt im Baum ein eindeutiger Pfad bis zum Wurzel-Knoten des Baumes erstellt werden. Diese Pfade werden unter Anderem für die Verschiebung von Prozessobjekten innerhalb eines Prozessmodells verwendet.
Ebenfalls wird der Process-Structure-Tree für die Rückrechnung von Prozessmodellen auf Vorgänger-Versionen verwendet, indem auf diesem die errechneten Differenzen angewendet werden.

4.1 Dekomposition eines DAGs in einen PST

Beginnend mit dem Start-Knoten wird mittels Depth-First-Search [JRPP94][KGFE08] der DAG geparst. Wobei jede Aktivität eine SESE-Komponente in einer Sequenz darstellt und demzufolge ein Blatt-Knoten wird. Die Blockstrukturierung stellt dabei sicher, dass Aktivitäten nur eine Eingangs- und Ausgangskante haben. Das nachfolgende Beispiel soll ein einfaches Prozessmodell ohne Kind-Fragmente zeigen.

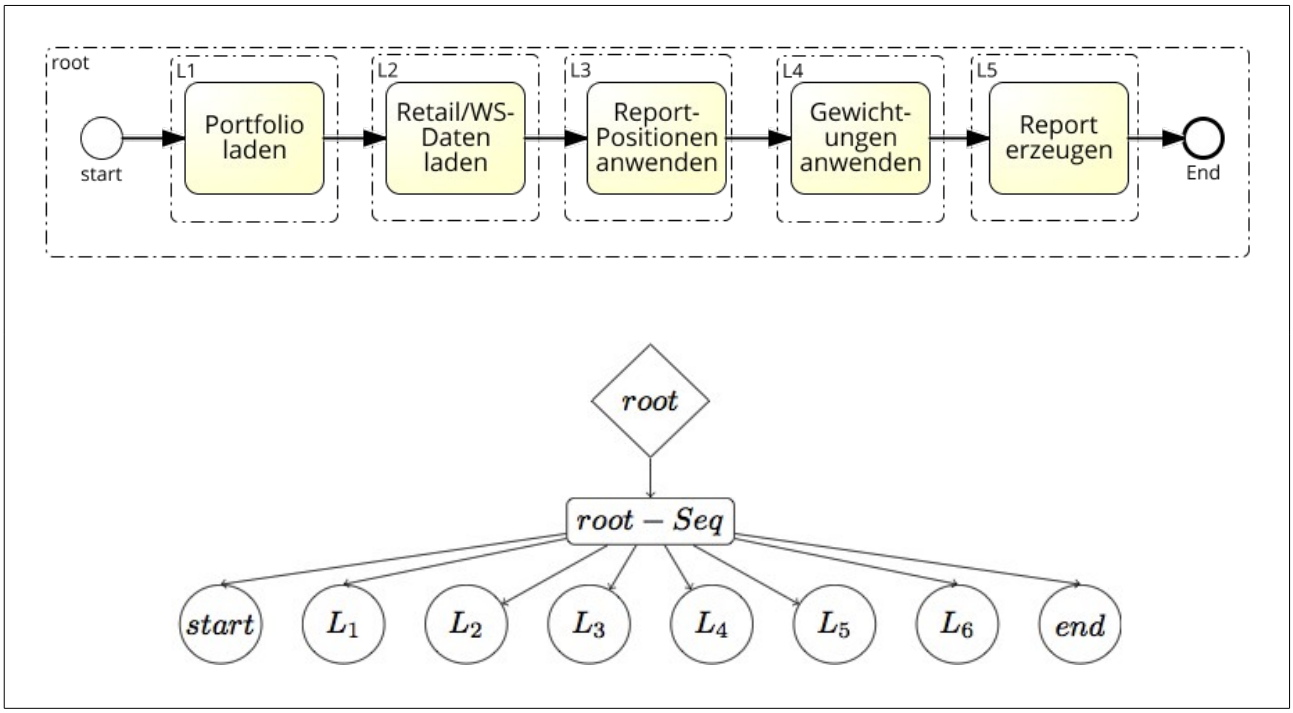


Abbildung 20: Beispiel eines einfach Process Structure Tree

Quelle: Eigene Erstellung

Im Prozessmodell aus Abbildung 20 wurden die SESE-Komponenten durch die gestrichelten Rechtecke gekennzeichnet. Um den PST kompakt zu halten, werden anstatt der Prozessobjektbeschreibungen, die Kurznamen der gestrichelten Rechtecke aus dem Prozessmodell verwendet. In Abbildung 20 gibt es im PST nur eine einzige Sequenz - die `root`-Sequenz - welche alle SESE-Komponenten beinhaltet. Ersichtlich ist auch, dass durch das Verfolgen der einzelnen Kanten eine geordnete Sequenz der Prozessobjekte im Prozessbaum entsteht. Ohne die Prozessnotation zu kennen, beziehungsweise ohne einer graphischen Darstellung des Modells, kann die Anordnung und die Position eines jeden Prozess-objektes über den Process-Structrue-Tree genau abgeleitet werden. So zum Beispiel kann durch die Rückverfolgung des Blatt-Knotens `L1` (dies entspricht „Report

Position anwenden“) bis zum `root`-Knoten bestimmt werden, dass dies eine Aktivität ist, welche in der `root`-Sequenz enthalten ist und an Position vier liegt.

Eine weitere Abbildung soll die Dekomposition von mehreren Teilfragmenten in einem PST zeigen. Im nachfolgenden Prozessmodell existiert eine Parallelisierung, wobei eine Alternativ-Kante der Parallelisierung zusätzlich ein OR-Framgment beinhaltet.

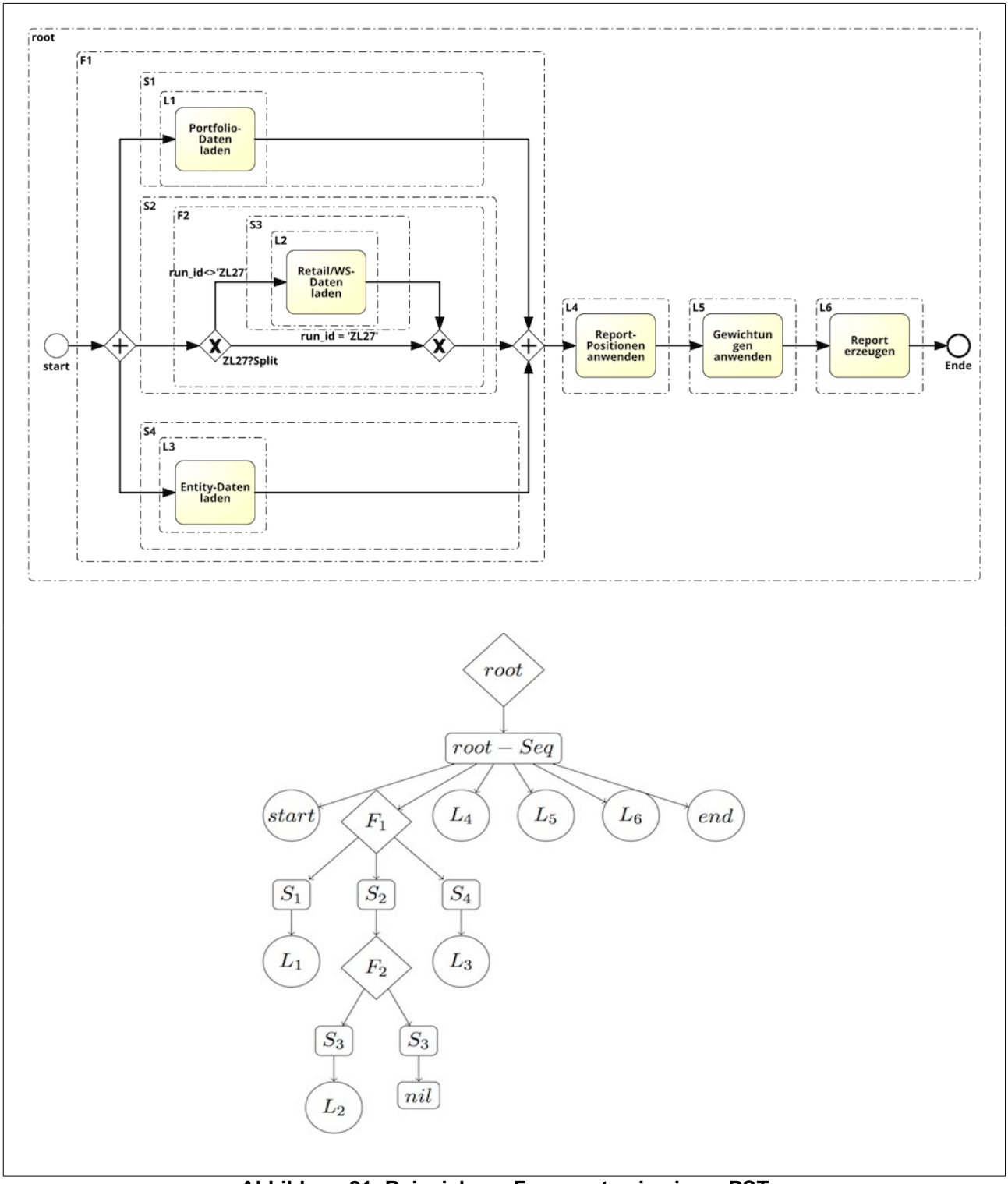


Abbildung 21: Beispiel von Fragmenten in einem PST

Quelle: Eigene Erstellung

Der PST aus Abbildung 21 zeigt, die `root`-Sequenz beginnend mit den `Start`-Knoten, gefolgt von einem AND-Fragment `F1`, dem Blatt-Knoten `L4` (Report-Positionen anwenden), `L5` (Gewichtungen anwenden) sowie `L6` und dem `End`-Knoten.
Da der nächste Knoten nach dem Start-Knoten eine Parallelisierung ist, wird anstatt eines Blatt-Knoten ein Fragment-Knoten (`F1`) im Baum erstellt. Dieser Knoten beinhaltet als Kind-Knoten alle weiteren Knoten in diesem AND-Fragment. In der `root`-Sequenz, wird das komplette AND-Fragment bis inklusive dem Join-Knoten als eine SESE-Komponente behandelt [VVLO07]. Dies hat den Vorteil, dass alle Prozessobjekte in einer hierarchischen Stufe, isoliert von allen anderen Stufen, analysiert werden können. Da nun ein neues Fragment durch den AND-Split-Knoten aufgemacht worden ist, werden ausgehend vom Split-Knoten die einzelnen Alternativ-Kanten mittels Depth-First-Search bis zum Join-Knoten traversiert, wobei jede Alternativ-Kante eine neue Sequenz einleitet. Aus dem Prozessmodell ist ersichtlich, dass die Parallelisierung drei Alternativ-Kanten beinhaltet. Es entstehen somit drei Kinder-Sequenzen (`S1`, `S2` und `S3`). Jede dieser Sequenzen beinhaltet die Prozessobjekte der entsprechenden Alternative. In der zweiten Sequenz (`S2`) befindet sich noch zusätzlich ein OR-Split, welcher wieder ein Fragment (`F2`) aufmacht. Die Alternativen des Fragments `F2` erstellen wiederum neue Sequenzen. Der OR-Split wird ebenfalls mit einem Depth-First-Search traversiert. Erst wenn alle Alternativen im OR-Split geparst wurden und es keine weiteren Fragment-Knoten gibt, wird über den OR-Join-Knoten hinausgegangen und das Fragment geschlossen. Ebenso wird auch beim AND-Fragment vorgegangen. Erst wenn es keine Alternativ-Kanten sowie keine weiteren Fragmente mehr gibt, wird über den AND-Join-Knoten hinausgegangen. Wird über den And-Join-Knoten hinausgegangen, befindet man sich wieder in der `root`-Sequenz und das Prozessmodell wird weiter bis zum End-Knoten geparst. Da sich keine weiteren Fragmente in dieser Sequenz befinden, werden alle nachfolgenden Objekte als Blätter in diese Sequenz aufgenommen. Anzumerken ist noch, dass der Join-Knoten nicht explizit aufgenommen wird. Die Split/Join-Kombination sowie die Alternativen sind indirekt über den Fragment-Knoten und den Sequenz Knoten enthalten. Des weiteren kann gleichzeitig mit der Erstellung des PSTs jeder einzelne Knoten nummeriert werden. Dies führt zur einer Positionierung jedes einzelnen Knotens. Diese Positionsnummern werden dann beim Erstellen der Highlevel-Changes benötigt. Die beiden Process-Structure-Trees aus diesem Abschnitt befinden sich mit den Positionsnummern der einzelnen Prozess-objekte im Anhang zu diesem Kapitel.

5 Differenzenerkennung zwischen zwei Prozessmodellen

Wie schon in der Einleitung beschrieben wurde, sollten zuerst Change-Primitives (Adds und Deletes) erkannt werden, um diese anschließend in höherwertige Änderungsoperationen zu transformieren – den sogenannten Highlevel-Changes [LCRW08] [RMDP98]. Um Change-Primitives zu erkennen, werden zwei Prozessmodelle PM_1 und PM_2 mit dem Ziel analysiert, zwei Mengen-Sets $addCL_{prim}$ und $delCL_{prim}$ zu erstellen, wobei [BCRS13]:

- $addCL_{prim} := PM_2 / PM_1$
- $delCL_{prim} := PM_1 / PM_2$

Zur Bestimmung der beiden Mengen kann ein mengenorientierter Ansatz zu Grunde liegen, indem mittels Differenz-Funktion aus der Mengenlehre, die beiden Mengen bestimmt werden. Hierbei wird überprüft, ob ein Element in der jeweilig anderen Menge enthalten ist oder nicht.
Diese Vorgehensweise mag zwar als intuitiv und einfach erscheinen, birgt aber Nachteile. So muss die Prüfung einmal für $addCl_{prim}$ und einmal für $delCL_{prim}$ durchgeführt werden, da die Differenzen für jede Menge nur einseitig bestimmt werden. Des Weiteren werden keine Informationen über die Position eines Prozessobjektes berücksichtigt. Verschiebt sich ein Prozessobjekt, geht die Information darüber verloren, dass ein Objekt an einer Position entfernt und an einer anderen Position wieder eingefügt wurde. Dies bedeutet, dass mit diesem Ansatz keine Änderungsreihenfolgen bestimmt werden und in weiterer Folge das Ableiten von Highlevel-Changes, wie zum Beispiel Moves oder Swaps, erschwert wird. Ziel sollte es jedoch sein, dass ein verschobenes Prozessobjekt sowohl im Set $addCL_{prim}$, als auch in $delCL_{prim}$ enthalten ist.

Eine weitere Möglichkeit, Change-Primitives zu erkennen und dabei die Reihenfolgen und Positionen der Prozessobjekte zu berücksichtigen, ist es sich das Konzept der Longest-Common-Subsequence (kurz: LCS) zu Nutze zu machen. Die LCS wird unter anderem beim Vergleichen von DNA-Strängen in der Bioinformatik [NSWC70][KAOH12], zur

Differenzenerkennung von Dateien (Unix-diff-Tool) oder zur Erkennung von Rechtschreibfehlern in Dokumentenverarbeitungssoftware [WAFI74] eingesetzt. Die zu Grunde liegende Idee ist es, aus zwei Zeichenketten die längste Teilzeichenkette, welche die beiden gemeinsam haben, zu errechnen [HIDA77]. Aufbauend auf der errechneten LCS kann dann ein sogenanntes Editier-Skript (ES) generiert werden [WAFI74][MWME85] [MYER86]. Das Editier-Skript ermöglicht, bei Anwendung auf einer der beiden Zeichenketten, die Generierung der zweiten Zeichenkette. Dabei zeigt das Editier-Skript genau an, an welcher Stelle ein Zeichen eingefügt wird, wann ein Zeichen gelöscht werden kann, sowie welche Zeichen an der gleichen Position unverändert übernommen werden können. Umgelegt auf das Themengebiet des Prozess-Engineerings würden nun anstatt einzelner Zeichen, die Prozessobjekte herangezogen werden, um ein Editier-Skript zu erzeugen. Im nachfolgenden Abschnitt soll näher auf die Generierung des Editier-Skript beziehungsweise auf die Extraktion der Change-Primitives eingegangen werden.

5.1 Extraktion der Change-Primitives

Um nun ein Editier-Skript zu erhalten, wird ein Editier-Graph zwischen zwei Zeichenketten errechnet [MYER86]. Teilstrecken des Editier-Graphens geben dabei die notwendigen Änderungsoperationen für jede Kombination von Subzeichenketten an. Je nachdem in welche Richtung der Editier-Graph verläuft, werden Änderungs-Operationen in das Editier-Skript aufgenommen. Da jedes einzelne Zeichen einer Zeichenkette mit allen Zeichen aus der zweiten Kette überprüft wird und um den Verlauf des Graphens effizient zu errechnen, bietet sich zur Abbildung aller potentiellen Wege eine Matrix-Darstellung an [NSWC70] [SANK77][HIDA77]. Jedes Element der Matrix M hält dabei die längste gemeinsame Teilfolge einer Subzeichenkette. Dieser Wert errechnet sich, wie in Formel 1 auf Seite 55 dargestellt, aus bereits zuvor errechneten Teilfolgen. Aufbauend auf dieser „LCS-Längen“-Matrix kann sich der Editier-Graph dann horizontal, senkrecht beziehungsweise diagonal durch M bewegen. Dabei soll der Editier-Graph den kürzesten Weg von $M_{0,0}$ nach $M_{n,m}$ finden. Ausschlaggebend sind dabei die diagonalen Verbindungen zwischen den einzelnen Koordinaten in M. Eine diagonale Verbindung zwischen zwei Koordinaten

$M_{i-1,j-1}$ und $M_{i,j}$ entsteht dann, wenn die beiden Zeichen auf den Koordinaten i und j gleich sind. Horizontale und vertikale Wege sind Verbindungen zwischen den Nachbarzeichen der jeweiligen Zeichen. Ein Beispiel mit zwei Zeichenketten $A=abbcbabc$ und $B=cabcbaba$ ist in Abbildung 22 angeführt.

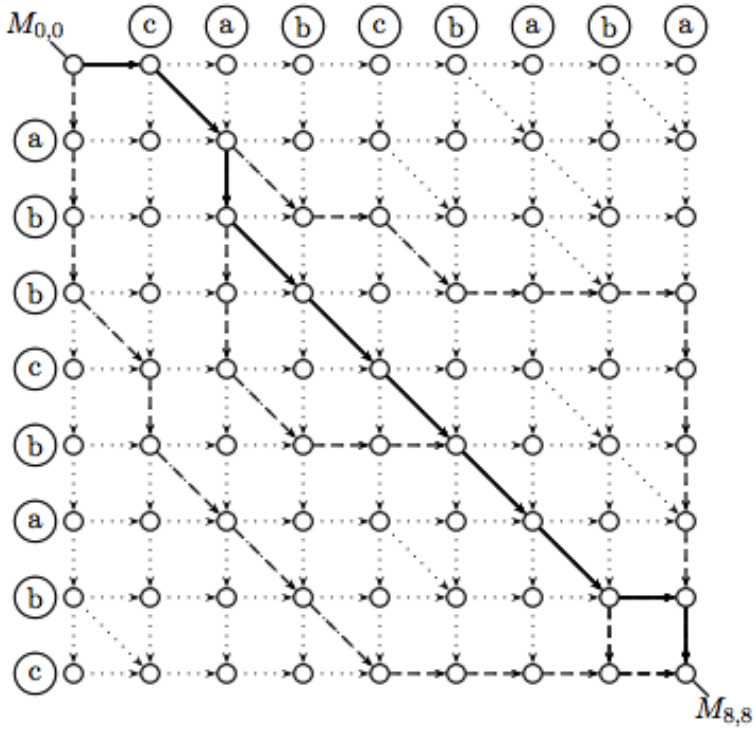


Abbildung 22: Editier-Graph(en) zweier Zeichenketten.

Quelle: Eigene Erstellung

Die Abbildung 22 zeigt unterschiedliche Pfade um von $M_{0,0}$ nach $M_{n,m}$ zu gelangen. Aus der Grafik ist zu erkennen, dass nicht nur ein Editier-Graph existiert, sondern gleich mehrere Varianten. In der Grafik sind mögliche Wege von $M_{0,0}$ nach $M_{n,m}$ durch die hervorgehobenen Pfeile markiert. Betrachtet man den Pfad mit der durchgehenden dickeren Linie, kann durch das Folgen des Pfades ein Editier-Skript erstellt werden. Eine horizontale Verbindung repräsentiert eine Hinzufügen-Aktion, eine senkrechte Verbindung eine Löschen-Aktion und bei einer diagonalen Verbindung bleibt das Zeichen unverändert. Das Editier-Skript beinhaltet jedoch nur die horizontalen beziehungsweise vertikalen Verbindungen des Editier-Graphens. Zusätzlich wird die Position der Änderungs-Aktion in der jeweiligen Zeichenkette angegeben. Für eine Hinzufügen-Aktion (add) bedeutet dies, dass das Zeichen an dieser Postion aus einer Zeichenkette übernommen werden muss. Die Positionsnummer einer Löschen-Aktion ($delete$) besagt, dass das Zeichen an

dieser Position aus der Zeichenkette nicht übernommen werden darf. Da es für eine Zeichenkette A nur $delete$ -Operationen geben kann und für die Zeichenkette B nur add -Operationen, ist es auch eindeutig ableitbar für welche Zeichenketten die Aktionen durchzuführen sind.

Daraus ergibt sich für den gewählten dicken Pfad das folgende Editier-Skript:
$ES_{A,B} := \langle add(1,c), delete(2,b), add(8,a), delete(8,c) \rangle$.

Ausgehend von der Zeichenkette A und unter Zuhilfenahme des Editier-Skripts aus dem obigen Beispiel, kann nun die Zeichenkette B reproduziert werden: Füge an erster Stelle ein `c` ein. Das kein Eintrag im Editier-Skript zum ersten Buchstaben aus Zeichenkette A existiert, wird das Zeichen `a` übernommen. Anschließend lösche (beziehungsweise übernehme nicht) aus der Zeichenkette A das zweite Zeichen, welches das `b` wäre. Da wieder keine Informationen über die Zeichen an Position drei bis sieben im Editier-Skript enthalten sind, werden die nachfolgenden Buchstaben aus A übernommen. Die letzten zwei Anweisungen fügen an der Stelle acht ein `a` ein, während in der alten Zeichenkette A , die achte Position entfernt werden muss – sie wird also nicht in die neue Zeichenkette B transferiert. Durch die Angabe von lediglich vier Änderungs-operationen konnte mit Hilfe des Editier-Skripts die Zeichenkette A in die Zeichenkette B transferiert werden.

Anzumerken ist, dass durch eine Invertierung des Editier-Skripts und ausgehend von der zweiten Zeichenkette B , die ursprüngliche Zeichenkette A rekonstruiert werden kann [MYER86]. Dies ist eine wichtige Eigenschaft und wird – in einem späteren Abschnitt – zur Anwendung kommen, um aus den Change-Primitives, Highlevel-Changes zu konstruieren, sowie eine Versionskontrolle von Prozessmodellen zu erreichen.

Wie in Abbildung 22 bereits ersichtlich, ergeben sich mehrere Editier-Skripte. Wobei nicht alle Skripte für die Weiterverarbeitung von Relevanz sind. Beispielsweise wäre der Pfad von $M_{0,0}$ nach $M_{8,0}$ und anschließend nach $M_{8,8}$, ein solches Editier-Skript. Anders ausgedrückt, würde durch diesen Pfad die Zeichenkette B zuerst hinzugefügt und anschließend die komplette Zeichenkette A gelöscht werden. Durch diesen Pfad entstehen zu viele Lösch- und Hinzufüge-Aktionen sowie ein Verlust von Informationen

über gleichbleibende Zeichen. Demzufolge sind nur all jene Pfade von Interesse, welche die kürzesten Editier-Skripte erzeugen, beziehungsweise all jene Pfade welche die längste gemeinsame Teilzeichenkette (LCS) beinhaltet. Das Finden jenes Editier-Skripts mit der minimalsten Anzahl an Änderungsoperationen ist äquivalent mit dem Finden der LCS [BLHR00]. Beide Probleme können, bei einer Prüfung aller möglichen Pfade, zu einem NP-vollständigen Problem anwachsen [MADA78]. Um die Komplexität zu reduzieren wird ein Verfahren zur Berechnung der LCS von [SANK72] eingesetzt. Dabei wird für jedes Element $x_{i,j} \in M$ die entsprechende LCS aus Zeichenketten A und B nach folgender Logik errechnet:

$$M(i,j) = \begin{cases} 0 & \text{wenn } i = 0 \text{ oder } j = 0 \\ M(i-1, j-1) + 1 & \text{wenn } A_i = B_j \\ max(M(i-1,j), M(i,j-1)) & \text{sonst} \end{cases}$$

Formel 1: LCS-Längen nach [SANK72]

Wendet man die obige Formel auf die beiden Zeichenketten A und B an, erhält man die in Abbildung 23 gezeigte Matrix M mit den korrespondierenden LCS-Längen für jede Zeilen-Spalten-Kombination:

		(c)	(a)	(b)	(c)	(b)	(a)	(b)	(a)
	0	0	0	0	0	0	0	0	0
(a)	0	0	1	1	1	1	1	1	1
(b)	0	0	1	2	2	2	2	2	2
(b)	0	0	1	2	2	3	3	3	3
(c)	0	1	1	2	3	3	3	3	3
(b)	0	1	1	2	3	4	4	4	4
(a)	0	1	2	2	3	4	5	5	5
(b)	0	1	2	3	3	4	5	6	6
(c)	0	1	2	3	4	4	5	6	6

Abbildung 23: LCS-Längen zwischen Zeichenkette A und B

Quelle: Eigene Erstellung

Aus dieser Matrix können die einzelnen LCS-Längen abgelesen werden. Aus jeder dieser Positionen kann bereits eine LCS erstellt werden [HIDA77] und theoretisch auch schon ein Editier-Skript. So zum Beispiel, ist für Position $M_{5,3}$ die LCS-Länge drei. Daraus ergibt sich eine LCS von abb [3]. Bei einer Subzeichenkette $A' = abb$ und $B' = cabcbaba$ sowie der errechneten LCS können die folgende Editier-Skripte erstellt werden:

$$ES_1 := \{add(1,c), add(4,c), add(6,a), add(7,b), add(8,a)\}$$
$$ES_2 := \{add(1,c), add(3,b), add(4,c), add(6,a), add(8,a)\}$$
$$ES_3 := \{add(1,c), add(4,c), add(5,b), add(6,a), add(8,a)\}$$

Für die Position $M_{8,8}$ ergibt sich eine LCS-Länge von sechs und eine LCS $abcbab$. Somit lässt sich unter Anwendung der LCS, die Zeichenkette A mit den folgenden Editier-Skripten zur Zeichenkette B überführen:

$$ES_1 := \{add(1,c), delete(3,b), add(8,a), delete(8,c)\}$$
$$ES_2 := \{add(1,c), delete(2,b), delete(8,c), add(8,a)\}$$
$$ES_3 := \{add(1,c), delete(3,b), delete(8,c), add(8,a)\}$$
$$ES_4 := \{add(1,c), delete(2,b), add(8,a), delete(8,c)\}$$

Durch die Berechnung der LCS konnten somit die möglichen Editier-Skripte auf lediglich drei, beziehungsweise vier Stück beschränkt werden, was die Komplexität zur Berechnung des Editier-Skriptes wesentlich reduziert. Jedoch zeigen die obigen zwei Beispiele zu den Editier-Skripten, dass es zu einer errechneten LCS, mehrere mögliche Editier-Skripte geben kann. Für eine Transformation ist jedoch nur ein einziges Skript notwendig. Um genau ein Editier-Skript zu errechnen, können weiterhin die einzelnen LCS-Längen in der Matrix hergezogen werden. Die einzelnen Übergänge in den LCS-Längen geben Aufschluss über die durchzuführende Änderungsoperation. Beginnend von Position $M_{n,m}$ kann, unter Anwendung von drei Regeln [MWME85], ein Editier-Graph nach $M_{0,0}$ erstellt werden [MYER86]:

- *Regel 1*: Eine diagonale Verbindung im Editier-Graph kann erstellt werden, wenn gilt $(M_{i,j} = M_{i-1,j-1} + 1) \wedge (A_i = B_j)$. Bedeutet, dass die Zeichen äquivalent sind.
- *Regel 2*: Eine Verbindung zum linken Nachbarn wird erstellt, wenn $M_{i,j} = M_{i,j-1}$
- *Regel 3*: Der Editier-Graph wird nach oben hin erweitert, wenn gilt: $M_{i,j} > M_{i,j-1}$

3 Eine genaue Darstellung und die Ableitung der LCS befindet sich im Anhang.

Die nachstehende Abbildung 24 zeigt die Anwendung der drei Regeln, auf die zuvor errechnete Matrix aus der Abbildung 23 auf Seite 55.

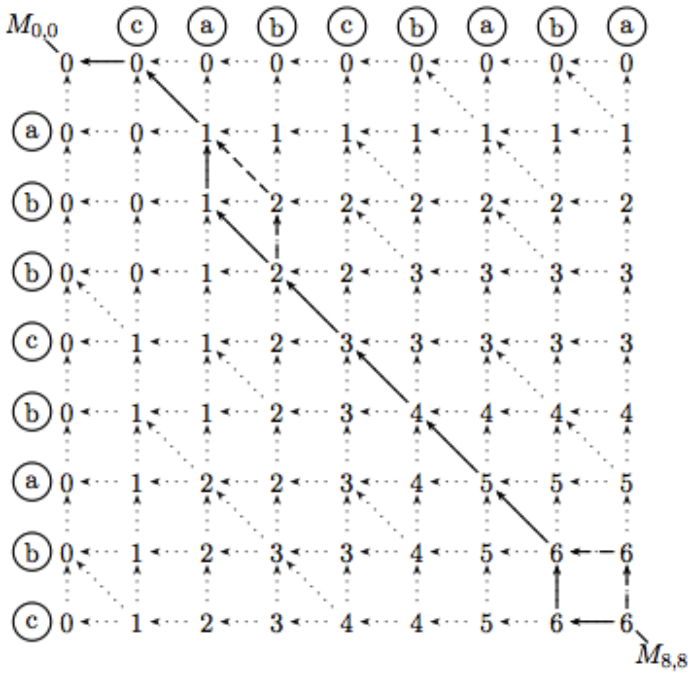


Abbildung 24: Editier-Skript mittels Backtracking

Quelle: Eigene Erstellung

Durch die Anwendung der drei Regeln kann ein einziger Editier-Graph errechnet werden:

$$ES := \{add(1,c), delete(2,b), delete(8,c), add(8,a)\}$$

Nachdem das Editier-Skript – mittels des Editier-Graphens – abgeleitet wurde, können die beiden Sets für die Change-Primitives daher wie folgt belegt werden: $addCL_{prim}$ mit $\{(1,c),(8,a)\}$ und $delCL_{prim} := \{(2,b),(8,a)\}$ abgeleitet werden.

5.2 LCS, Editier-Skript und Prozessmodelle

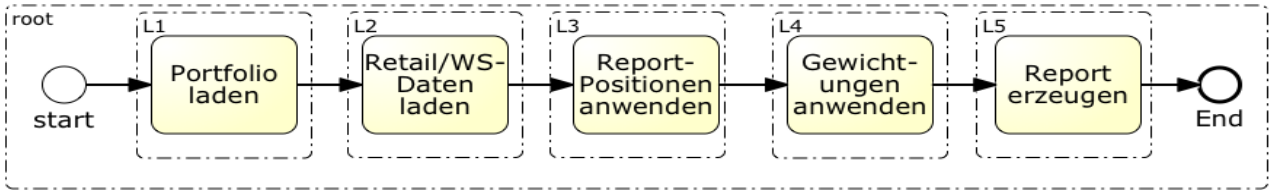
Das im ersten Abschnitt beschriebene Verfahren kann auf Prozessmodelle und die enthaltenen Prozessobjekte umgelegt werden, indem anstatt der Buchstaben einer Zeichenkette, Objekt-Beschreibungen verwendet werden. Dabei können als Objekt-Beschreibungen die Namen der Prozessobjekte dienen. Zwar können sich die Namen der Objekte laufend ändern, dies hat aber zur Konsequenz, dass die beiden Objekte als nicht mehr gleich betrachtet werden und somit zwei Einträge in den Sets für Change-Primitives entstehen. Dies ist jedoch aus den folgenden Gründen erwünscht. Erstens, um feststellen zu können, welche Prozessobjekte garantiert gleich geblieben sind. Diese müssen dann nur noch auf die Highlevel-Change-Operation „Update“ geprüft werden. Zweitens, kann durch das Hinzufügen zu den Change-Primitives der Prozessobjekte, bei Namensänderung mittels eines Ähnlichkeitstests (siehe nachfolgendes Kapitel) geprüft werden, wie stark die Überschneidungen der beiden Objekte sind. Sollte nur eine schwache bis keine Überschneidung vorhanden sein, können Highlevel-Change-Operationen, wie „Replace“ angewendet werden, welche die durchgeführte Operation im Prozessmodell genauer beschreiben kann, als ein „Update“. Drittens, dienen garantiert unveränderte Prozessobjekte als Fixpunkte in der Move-Erkennung von weiteren Objekten. Ein weiteres Argument für die Aufnahme von umbenannten Prozessobjekten in die Change-Primitive-Sets $addCL_{prim}$ und $delCL_{prim}$, mit einer anschließenden Ähnlichkeitsprüfung ist, dass so Änderungsketten erstellt werden können wie zum Beispiel ein „Update“ durch Namensänderung mit gleichzeitiger Verschiebung des Objektes.

Nachfolgend werden Prozessmodelle inklusive deren Change-Primitives-Sets analysiert und beschrieben. Da die Anordnung der Prozessobjekte eine wichtige Rolle für die Erstellung der Change-Primitives und der Positionsnummern im Editier-Skript einnimmt, werden die Prozessobjekte in der LCS-Längen-Matrix M, beginnend vom Start-Knoten über deren Vorgänger-Nachfolger-Beziehung, bis hin zum End-Knoten geordnet gelistet. Basis für diese Anordnung ist der jeweilige Process-Structure-Tree, der bereits die Prozessobjekte in der entsprechenden Reihenfolge hält. Jedoch sind die Process-Structure-Trees eine hierarchische Repräsentation der Prozessmodelle, M. Da diese Form der Darstellung aber nicht unterstützt wird, werden die beiden Bäume in ihre flache

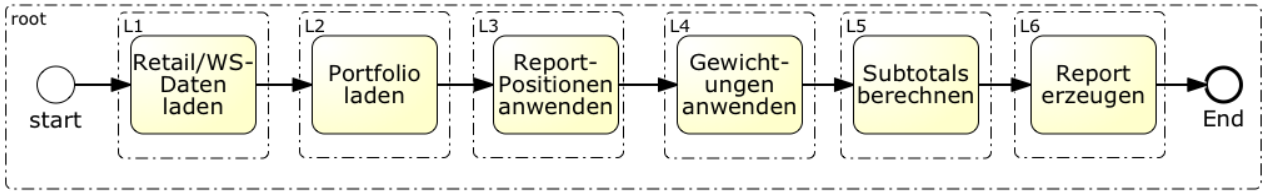
Struktur transformiert. Die Traversierung erfolgt wieder mittels „Depth-First-Search“-Ansatzes [CLRS09][KGFE08], wobei erst über ein Fragment hinausgegangen wird – sprich die Traversierung des Fragment-Join – wenn alle Alternativ-Kanten durchlaufen wurden. Die so erstellten flachen Strukturen aus den beiden Process-Structure-Trees dienen dann als die beiden „Zeichenketten“ für M .

Beispiel 1: Angenommen, es existieren die folgenden zwei Prozessmodelle PM_1 und PM_2 , so wie die zwei Process-Structure-Trees PST_1 und PST_2 für diese zwei Modelle.

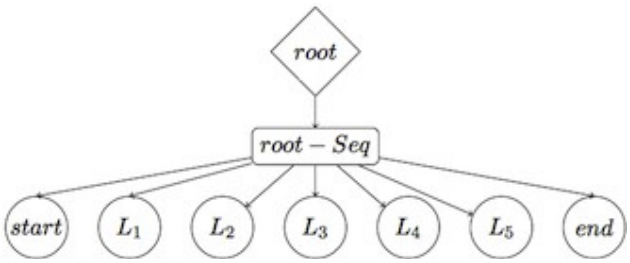
PM_1



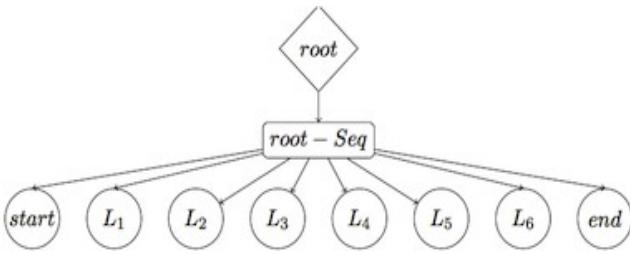
PM_2



PST_1



PST_2



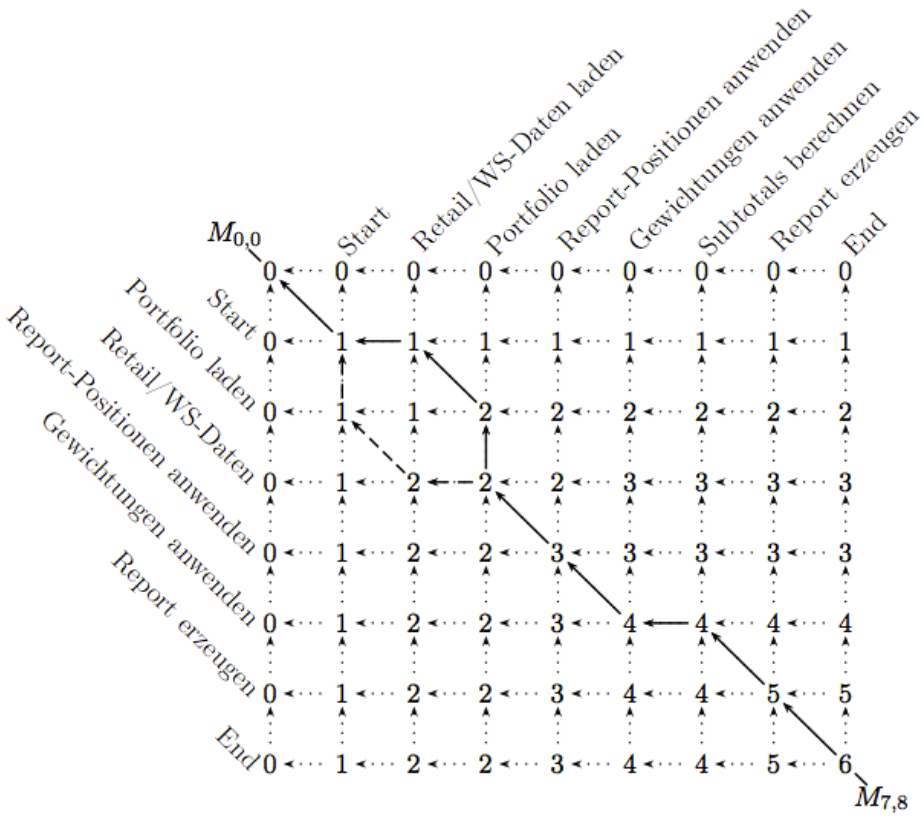
$fPST_1$: `start, Portfolio laden, Retail/WS-Daten laden, Report-Postionen anwenden, Gewichtung anwenden, Report erzeugen, End`

$fPST_2$: `start, Retail/WS-Daten laden, Portfolio laden, Report-Postionen anwenden, Gewichtung anwenden, Subtotals berechnen, Report erzeugen, End`

Beispiel 1: Changes-Primitives – Einfaches Modell

Zuerst wird die flache Struktur aus den beiden Process-Structure-Trees PST_1 und

PST_2 abgeleitet. Da es in diesem Beispiel keine weiteren Fragmente außer das `root`-Fragment gibt, ergibt sich die flache Struktur $fPST_1$ und $fPST_2$ direkt aus der Anreihung der Prozessobjekte in PST_1 und PST_2. Aus Platzgründen wurden die Knoten in Process-Structure-Trees abgekürzt und entsprechen dabei den Beschriftungen aus den Umrandungen. $fPST_1$ und $fPST_2$ können dann für die Erstellung der LCS-Längen-Matrix M herangezogen werden.



Editier-Skript $ES_{PM1,PM2}$

```
add('Retail/WS-Daten laden', 2); delete('Retail/WS-Daten laden',2); add('Subtotals berechnen', 6)
```

$$addCL_{prim} := \{(2, 'Retail/WS-Daten\ laden'), (6, 'Subtotals\ berechen')\}$$

$$delCL_{prim} := \{(3, 'Retail/WS-Daten\ laden')\}$$

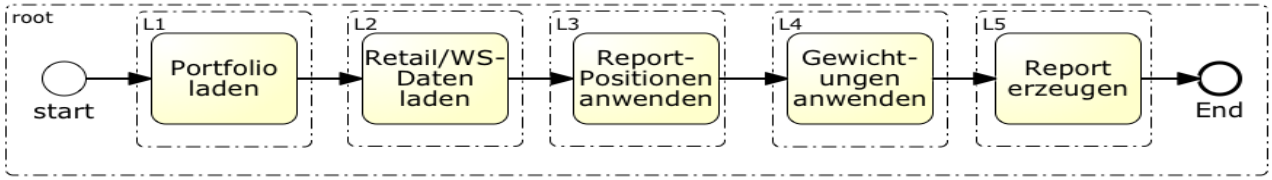
Abbildung 25: LCS-Längen-Matrix, Editier-Skript und Change-Primitives aus Beispiel 1

In Abbildung 25 ist zu erkennen, dass der Prozess-Task „Retail/WS-Daten laden“ in PM_1 von der dritten Position gelöscht wird. Jedoch wird dieser Task in PM_2 an einer anderen Stelle wieder eingefügt. Diese $add/delete$ -Kombination kann dann bei der Überführung zu höherwertigen Änderungs-Operationen verwendet werden, um zum Beispiel Moves, Swaps oder Replaces zu erkennen. Des Weiteren kann bei Aufnahme

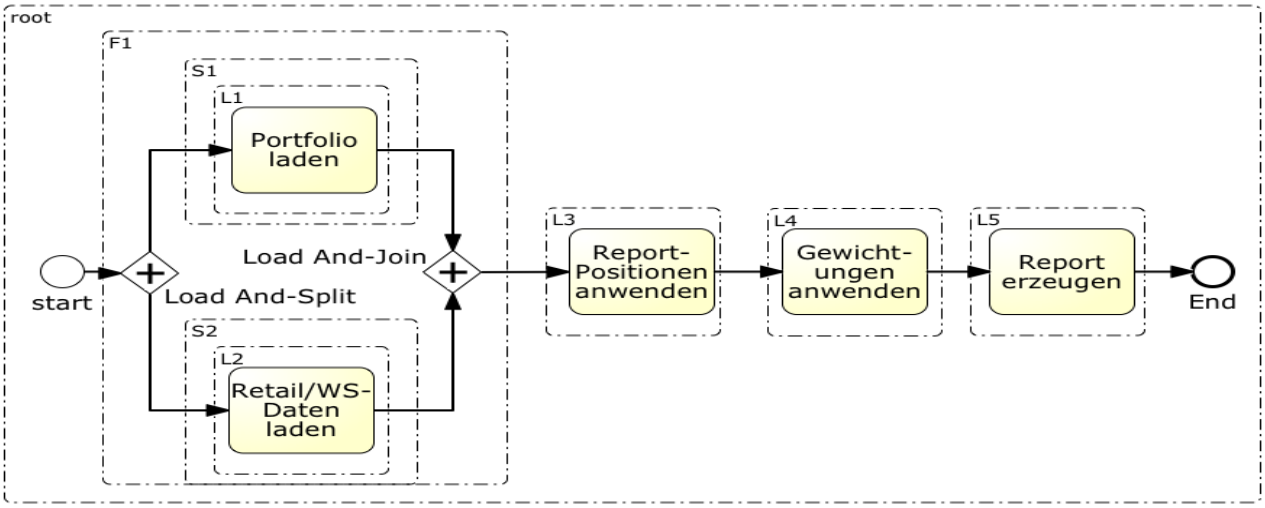
solcher Kombinationen und durch Invertierung der Change-Primitives-Sets über dem Prozessmodell PM_2 wieder das ursprüngliche Prozessmodell PM_1 erstellt werden. Zusätzlich ist aus dem Change-Primitive-Set $addCL_{prim}$ noch zu erkennen, dass im Prozessmodell PM_2 ein neuer Task, „Subtotals berechnen“, an der Position sechs hinzugefügt wurde.

Beispiel 2: Angenommen, es existiert ein weiteres Prozessmodell PM_3, welches mit dem Prozessmodell PM_1 aus Beispiel 1, auf Differenzen analysiert werden soll. Im Prozessmodell PM_3 wurden Prozessoptimierungen durchgeführt, wodurch die beiden Prozess-Tasks `Portfolio laden` und `Retail/WS-Daten laden` parallelisiert wurden.

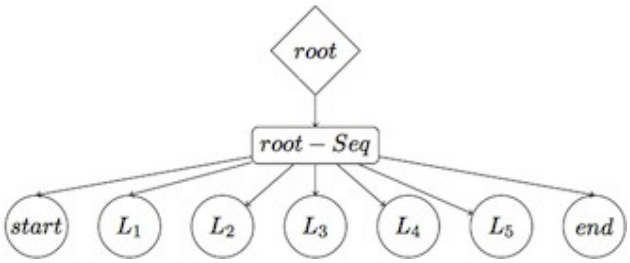
PM_1



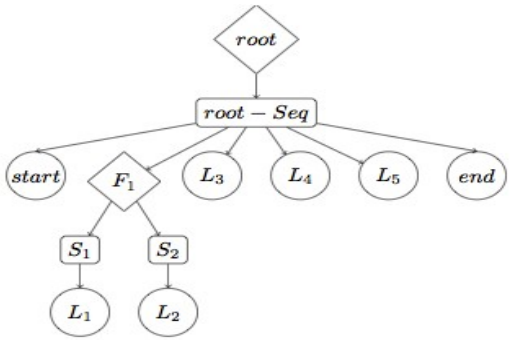
PM_3



PST_1



PST_3

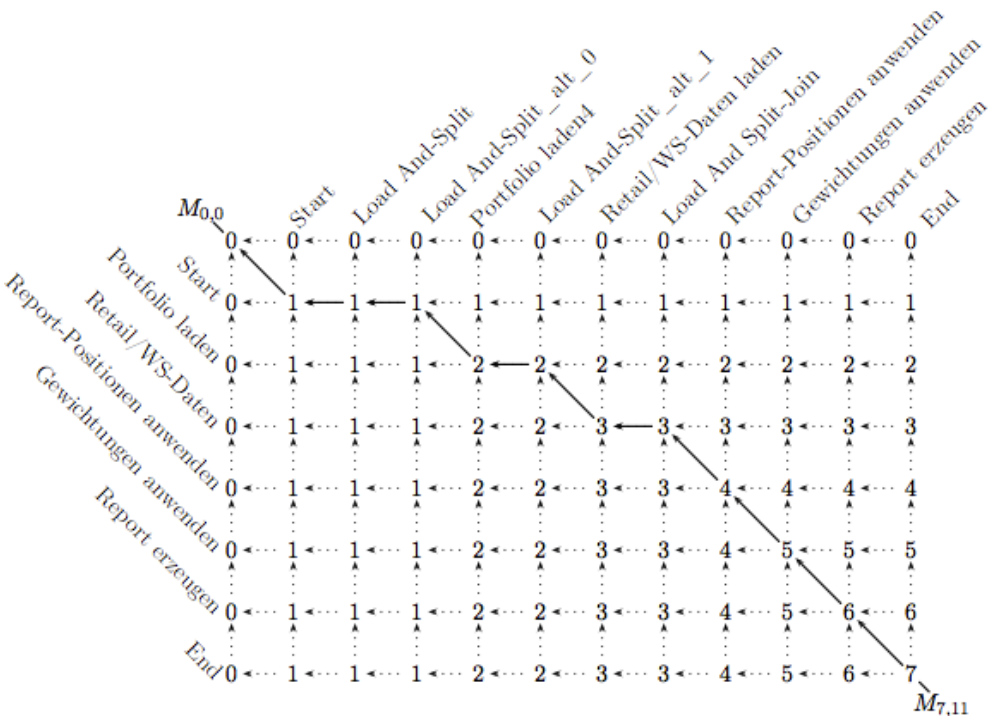


$fPST_1$: `start, Portfolio laden, Retail/WS-Daten laden, Report-Postionen anwenden, Gewichtung anwenden, Report erzeugen, End`

$fPST_3$: `Start, Load And-Split, Load And-Split_alt_0, Portfolio laden, Load And-Split_alt_1, Retail/WS-Daten laden, Load And-Join, Report-Positionen anwenden, Gewichtungen anwenden, Report erzeugen, End`

Beispiel 2: Changes-Primitives – Modell mit Fragment

In Beispiel 2 hat das Prozessmodell PM_3 ein Parallelisierungs-Fragment. Dieses Fragment wird in PST_3 im Knoten F_1 gezeigt. Um die flache Struktur zu erstellten, wird mittels Depth-First-Search zuerst der Zweig mit der Alternative `Load And-Split_alt_0` (S1) traversiert. Wurden alle Prozessobjekte aus dieser Sequenz ermittelt, wird mit der Alternative `Load And-Split_alt_1` (S2) weitergemacht. Existiert keine weitere Alternative in diesem Fragment, wird der Join-Knoten zur flachen Struktur $fPST_3$ hinzugefügt. Daraus ergibt sich für das Parallelisierungs-Fragment die folgende Anreihung in $fPST_3$: `Load And-Split, Load And-Split_alt_0, Portfolio laden, Load And-Split_alt_1, Retail/WS-Daten laden, Load And-Join.` Die beiden flachen Strukturen werden wieder für die Generierung der LCS-Längen-Matrix herangezogen:



Editier-Skript $ES_{PM1,PM3}$

```
Add('Load And-Split', 2); Add('Load And-Split_alt_0',3); Add('Load And-
Split_alt_0',5); Add('Load And-Join',7)
```

$$addCL_{prim} := \{(2,'Load\ \text{And-Split}'),(3,'Load\ \text{And-Split\_alt\_0}'),(5,'Load\ \text{And-Split\_alt\_1}'),(7,'Load\ \text{And-Join}')\}$$

Abbildung 26: LCS-Längen-Matrix, Editier-Skript und Change-Primitives aus Beispiel 2

Aus $addCl_{prim}$ kann man erkennen, dass auch die Alternative-Kanten der Parallelisierung

aufgenommen wurden. Dies ist notwendig, um die Anzahl von Alternative-Kanten in einem Fragment feststellen zu können. Des Weiteren dienen diese ebenfalls der Erstellung von Highlevel-Changes (hierfür siehe das Kapitel über die Generierung von Highlevel-Changes).

Obwohl sich die beiden Tasks `Portfolio laden` und `Retail/WS-Daten laden` scheinbar verschoben haben, sind diese nicht in den Change-Primitives aufgenommen worden. Da durch den „Depth-First-Search“-Ansatz die oberste Alternative-Kante zuerst gewählt wurde, erscheint in der Ableitung der „flachen“ Struktur der Task `Portfolio laden` zuerst auf. Betrachtet man die „flache“ Struktur in PM_1, dann ist der Task `Portfolio laden` ebenfalls vor dem Task `Retail/WS-Daten laden`. Daher erkennt der Algorithmus zum Erstellen der Change-Primitives, beziehungsweise des Editier-Skripts, die Tasks `Portfolio laden` und `Retail/WS-Daten laden` als beizubehaltende Objekte.

6 Ähnlichkeitsanalyse

Basierend auf den errechneten Change-Primitives aus dem letzten Kapitel, wird noch eine weitere Analyse dieser Änderungen durchgeführt, bevor diese in die geforderten Highlevel-Change-Operationen umgewandelt und in einem Change-Log abgelegt werden [RRJK06]. Diese Analyse ist notwendig, um die Anzahl der zu analysierenden Prozessobjekte, welche in Highlevel-Change-Operationen transformiert werden können, zu reduzieren. Dadurch werden erst bestimmte Highlevel-Changes wie, zum Beispiel ein Update ermöglicht.

Da der verwendete Algorithmus zur Erkennung von Änderungen zwischen zwei Prozessmodellen, nur Hinzufüge- und Entfern-Operationen von Prozessobjekten erkennt, entsteht folgende Situation: Der Algorithmus produziert mehr Add- und Delete-Operationen als eigentlich notwendig wären. Dies ist in einem ersten Schritt erwünscht um einerseits Positionsinformationen von Änderungen zu erhalten, andererseits sind überschüssige Add- und Delete-Operationen ein erster Hinweis auf eine Transformationsmöglichkeit zu Highlevel-Change-Operationen. Überschüssige Änderungsoperationen entstehen durch:

- Verschiebungen: Wenn ein Prozessobjekt an eine andere Position verschoben wird, entsteht eine Add- und eine Delete-Operation für das gleiche Objekte.
- Umbenennungen: Wird ein Prozessobjekt unbenannt, behandelt der LCS-Algorithmus das unbenannte Objekt als ein neues Objekt. Somit entsteht wieder eine Delete- und eine Add-Operation.

Ziel der Ähnlichkeitsanalyse ist es, all jene Prozessobjekte aus den beiden Mengen $addCL_{prim}$ und $delCL_{prim}$ zu identifizieren, welche, semantisch betrachtet, gleich sind und diese für die Transformation in die Highlevel-Change-Operationen als add/delete-Kombination zu markieren. Später muss dann nur noch der passende Highlevel-Change für diese add/delete-Kombinationen errechnet werden.

Beispiel 3: Das folgende Beispiel soll die „überschüssigen“ Add- und Delete-Operationen für die Verschiebungen und Umbenennungen zeigen. Hierfür wird angenommen, es existieren zwei Prozessmodelle PM_1 und PM_2 und es wurde bereits eine Change-

Primitives-Erkennung durchgeführt.

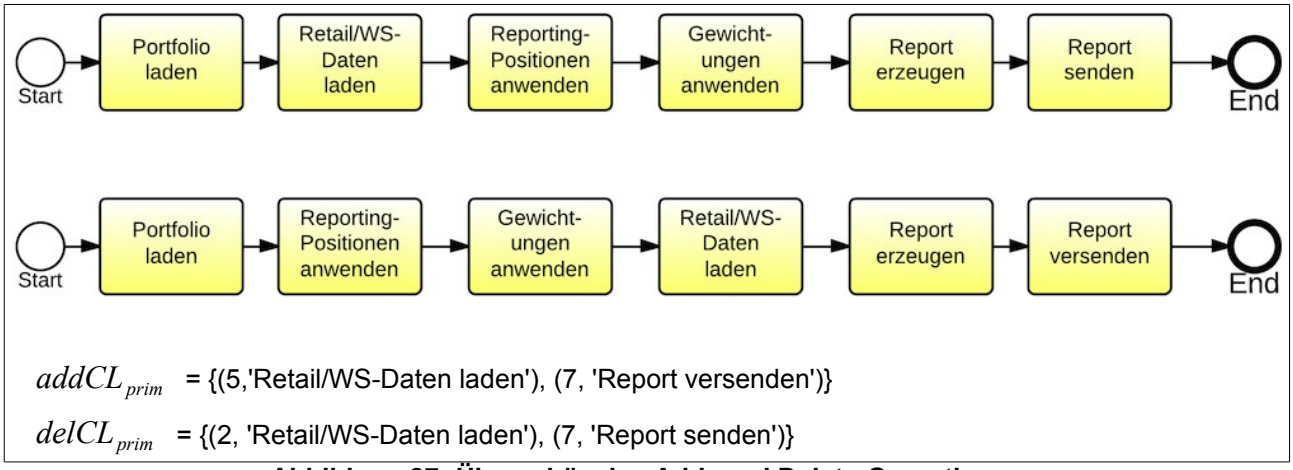


Abbildung 27: Überschüssige Add- und Delete-Operationen

Quelle: Eigene Erstellung

Die Abbildung 27 zeigt, dass lediglich durch die Umbenennung des Prozessobjektes `Report senden` in `Report versenden`, eine Delete-Add-Kombination entsteht, diese aber im Sinne der Highlevel-Change-Operationen ein Update darstellen könnte. Selbige Kombination entsteht auch dann, wenn Prozessobjekte verschoben werden, wie es im Beispiel mit dem Objekt `Retail/WS-Daten laden` gemacht wurde. Das Objekt wird an der Postion zwei entfernt, um es dann wieder an Postion fünf einzufügen.

Um möglichst viele Change-Primitives in Highlevel-Changes umwandeln zu können, werden alle Prozessobjekte in den Mengen $addCL_{prim}$ und $delCL_{prim}$ auf deren Ähnlichkeit untereinander geprüft. Das Problem dabei ist, dass eine Menge $CL_{prim} \subseteq delCL_{prim} \times addCL_{prim}$ entsteht. [KGFE08] bezeichnet dies als eine many-to-many Korrespondenz, welche zu einer one-to-one Korrespondenz aufgelöst werden sollte. Erst wenn one-to-one Korrespondenzen entstanden sind, was eindeutigen add/delete-Kombinationen entspricht, kann mit der Transformation in Highlevel-Changes begonnen werden.

Die Ableitung der add/delete-Kombinationen beziehungsweise der one-to-one-Korrespondenzen wird mit zwei Schritten erreicht:

- Zuerst erfolgt die Ähnlichkeitsmessung für jedes Delete-Objekt über alle Add-Objekte in $addCL_{prim}$. Jede bereits gerechnete add/delete-Kombination erhält dabei einen Ähnlichkeitswert. Da bei der Messung ein Kreuzprodukt zwischen der Menge $delCL_{prim}$ und $addCL_{prim}$ entsteht und um im zweiten Schritt der Ähnlichkeitsmessung weniger Kombinationen vergleichen zu müssen, werden nur all jene Kombinationen in Betracht gezogen, welche eine starke Ähnlichkeit aufweisen. Dies hat zur Folge, dass lediglich die Kombinationen mit der besten Fitness „überleben“ und in die zweite Auswahlrunde gelangen. Das Ergebnis der ersten Runde bringt ein Mengenset $PObj_{pot}$ hervor. Erst wenn alle potenziellen Delete-Add-Kombinationen aus CL_{prim} gefunden wurden, beginnt die Evaluierung der zweiten Runde.
- Die zweite Evaluierungsrunde erzeugt ein Set $PObj_{best}$ durch Selektion aus $PObj_{pot}$, indem Delete-Add-Kombinationen so gewählt werden, dass es durch die Auswahl einer Kombination, keine andere Kombination mit dem selben Delete- oder Add-Objekt gibt, welche einen höheren Ähnlichkeitswert aufweist. $PObj_{best}$ beinhaltet nur mehr one-to-one Korrespondenzen mit den höchsten Ähnlichkeits-werten, welche in Highlevel-Changes wie ein 'Update' oder 'Move' umgewandelt werden können.

Im nachfolgenden Abschnitt wird nun genauer auf die Errechnung des Ähnlichkeitswertes eingegangen.

6.1 Prozessobjekt-Eigenschaften zur Ähnlichkeitsmessung

In der Literatur werden zur Messung der Gleichheit von Prozessmodellen auch die einzelnen Prozessobjekte herangezogen. [DDDK11,RDRJ09, MEKO07]. Die Messung kann dabei auf folgende drei Arten erfolgen:

- Label-Gleichheit der Prozessobjekte
- Attribute-Gleichheit
- Positions- bzw. Verhaltensgleichheit

6.1.1 Label-Gleichheit

Mittels der Label-Gleichheit wird festgestellt, ob zwei Prozessobjekte die gleiche Beschreibung aufweisen. Diese Prüfung wird schon teilweise bei der Erkennung der Change-Primitives angewendet. Eine paarweise Überprüfung der Change-Primitives kann so gleiche Prozessobjekte wieder „zusammenführen“ und diese gegebenenfalls zu Highlevel-Changes verarbeiten. Eine gesonderte Überprüfung auf Label-Gleichheit ist dann von Nutzen, wenn sich Prozessobjekte lediglich verschoben haben.

Beispiel 4: Gegeben sind die Change-Primitives Mengen $addCL_{prim}$ und $delCL_{prim}$ aus Abbildung 27. Die enthaltenen Prozessobjekte sollen auf Label-Gleichheit geprüft werden.

$addCL_{prim}$ ={(5,'Retail/WS-Daten laden'), (7, 'Report versenden')}

$delCL_{prim}$ ={(2, 'Retail/WS-Daten laden'), (7, 'Report senden')}

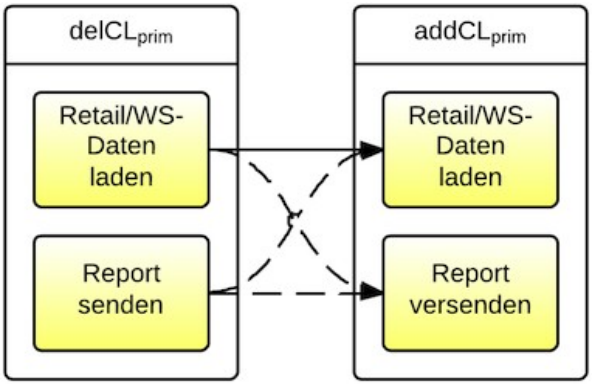


Abbildung 28: Prüfung auf Label-Gleichheit

Quelle: Eigene Erstellung

Die Abbildung 28 zeigt eine Prüfung der Label-Beschriftungen der Prozessobjekte. Jedes Prozessobjekt aus $delCL_{prim}$ wird dabei mit den Objekten aus $addCL_{prim}$ verglichen. Die Prüfung aus dem Beispiel ergibt, dass das Prozessobjekt `Retail/WS-Daten laden` ein korrespondierendes Prozessobjekt in $addCL_{prim}$ enthält. Somit sind die beiden Objekte äquivalent. Bei einer Label-Überprüfung kann für das Prozessobjekt `Report senden keine` Übereinstimmung gefunden werden.

6.1.2 Attribut-Gleichheit

Eine Überprüfung auf exakte Label-Übereinstimmung, wie das eben gezeigte Beispiel, ist nicht immer ausreichend. Durch eine Umbenennung von Prozessaktivitäten greift die Label-Gleichheit nicht mehr. Die Ausstattung von Prozessobjekten mit Attributen kann allerdings, als Hilfe der Bestimmung der Gleichheit von Prozessobjekten dienen [RDRJ09, MEKO07]. Ein Attribut besteht dabei aus einem Bezeichner, den Attribute-Namen und einem Wert. Existieren zwei Prozessobjekte in der Menge der Change-Primitives, welche genau die gleichen Attribut-Namen mit den selben Attribut-Werten aufweisen, kann auf Gleichheit geschlossen werden.

Beispiel 5: Seien wieder die zwei Change Menge $addCL_{prim}$ und $delCL_{prim}$ aus Abbildung 27 vorhanden. Dieses mal soll auf eine Attribute-Gleichheit geprüft werden.

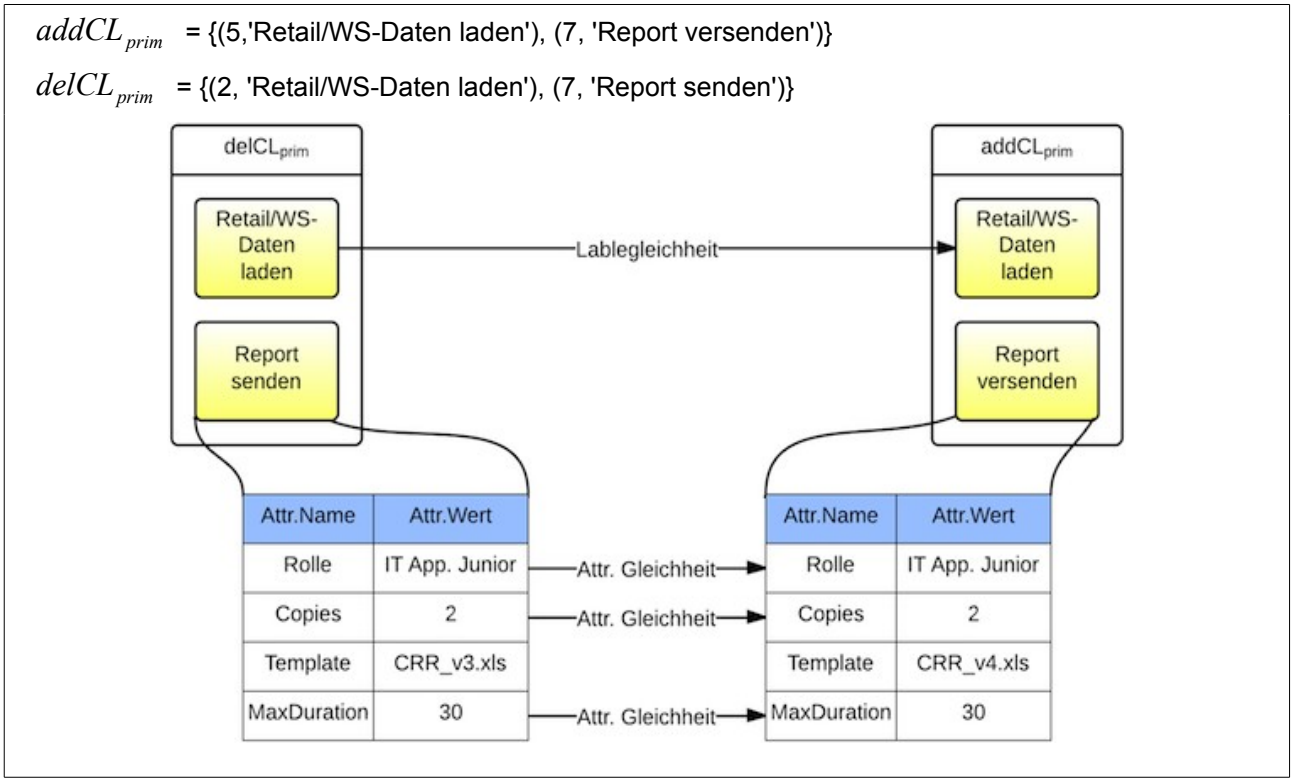


Abbildung 29: Prüfung auf Attribut-Gleichheit

Die beiden Prozessobjekte `Report senden` und `Report versenden` weisen keine Label-Gleichheit auf, folglich sollte auf Attribut-Gleichheit geprüft werden. Dabei wird jedes gleichnamige Attribut auf deren Wertegleichheit getestet. Eine Attributgleichheit tritt in diesem Beispiel nur bei drei von vier Attributen auf, somit sind die beiden Objekte – streng betrachtet – nicht gleich. Erst wenn das Attribut `Template` fehlen würde oder angepasst wird, kann man von einer Objektgleichheit aufgrund der Attribute sprechen.

6.1.3 Positions- und Verhaltensgleichheit

Eine weitere Möglichkeit um die Gleichheit zweier Prozessobjekte zu bestimmen besteht darin, die relative Position der Objekte im Prozessgraphen zu errechnen [MEKO07, KGFE08]. Befindet sich ein untersuchtes Prozessobjekt an der selben relativen Position, können diese als äquivalent betrachtet werden. [RDRJ09] und [DDDK11] versuchen die Position eines Prozessobjektes mittels deren Vorgänger- und Nachfolger-Objekten zu erkennen. Für [RDRJ09] ist ein Objekt an der selben Position, wenn dieses, unter Berücksichtigung der hinzugefügten Objekte, die gleichen Nachbarn aufweist. Während [DDDK11] bei gleichen Nachbarn von Verhaltensgleichheit spricht.

Eine Positions- bzw. Verhaltensgleichheit, ist bei der Erkennung der Ähnlichkeit zwischen einer add/delete-Kombination noch nicht von Interesse. Zuerst wird in der Ähnlichkeitsanalyse festgestellt, ob sich überhaupt zwei Change-Primitives, hinsichtlich der Labels und den gesetzten Attributen, gleichen. Gibt es einen Match, wird auf ein Update hin geprüft und die daraus gewonnenen Informationen fließen zusätzlich in die Move-Detection mit ein. Ein Test auf die Positionsgleichheit wird daher erst nach der Ähnlichkeitsanalyse durchgeführt.

6.2 Maßzahl zur Bestimmung der Ähnlichkeit

Generell kann für die Bestimmung der Ähnlichkeit, der Wert 0 für keine Übereinstimmung und der Wert 1 für komplette Übereinstimmung gesetzt werden [SABU88]. Es zeigt sich, dass diese Annahme jedoch zu grob für eine Ähnlichkeitsmessung ist. Würden nur die Werte 0 bzw. 1 herangezogen werden, könnte die „Ähnlichkeitsmessung“ nur auf Gleichheit hin geprüft werden. Im obigen Beispiel wären dann die zwei Labels `Report versenden` und `Report senden`, unter Verwendung eines naiven String-Matching-Algorithmus [CLRS09]: $sim_{naiv}('Report\ senden', 'Report\ versenden') = 0$.

Offensichtlich besteht aber zwischen den beiden Termen eine bestimmte Ähnlichkeit. [SABU88] schlägt daher ein Intervall zwischen den Grenzen 0 und ein 1 vor. Man erhält dadurch eine Ratio zur Übereinstimmung der beiden Terme. Auch hier gilt wieder: Sind sich die Terme überhaupt nicht ähnlich, werden diese mit 0 bewertet, sind beide Terme gleich, so erhalten sie einen Wert von 1. Werte zwischen den beiden Grenzen geben dann die genaue Ähnlichkeit der Terme an. Geht dabei die Ratio gegen 1, sind sich die Werte stark ähnlich, geht sie jedoch eher gegen 0, kann angenommen werden, dass die Terme nur wenig ähnlich sind. Im Folgenden wird, wie auch bei [DDDK11], angenommen, dass Terme welche eine Ratio von über 0.5 aufweisen ähnlich sind. Dies bedeutet, dass sich zwei vergleichbare Terme zu über 50 % gleichen. Zusätzlich zu dieser Annahme wird der Begriff der starken Ähnlichkeit, ab einer Ratio von 0.75, definiert. Diese Grenze wurde gewählt, da: Erstens, schon bei nur einem gleichen Attribut Ratios von 0.5 entstanden sind. Das würde bedeuten, dass Prozessobjekte schon als ähnlich markiert werden, welche aufgrund des Prozesskontexts prinzipiell ähnliche Attribute aufweisen. Ein solches Szenario entsteht zum Beispiel dann, wenn ein Prozess von einem bestimmten Akteur mit einer gewissen Rolle durchgeführt wird. Ist ein Prozess IT-lastig, kann die Rolle „IT Application Manager“ häufig vorkommen oder sogar in jeder Prozessaktivität enthalten sein. Bei einer Ratio von 0.5 und unter der Annahme, dass das einzige gemeinsame Attribut über alle Prozessobjekte hinweg, die Rolle ist, entstehen in der Menge CL_{pot} add/delete-Kombinationen, welche alle eine Ratio von 0.5 aufweisen würden. Folglich würde bei der Abarbeitung der Ähnlichkeitsmaxima, eine sequenzielle Zuweisung der one-to-one Korrespondenzen stattfinden, was einer zufälligen Zuweisung entspricht.

Zweitens, ist die Grenze von 0.75 gerade noch eine Marke, die es erlaubt, dass $sim_{naiv}(Label1, Label2) = 0$, die Attribute sich leicht ändern dürfen, dennoch aber eine

signifikante Anzahl an vergleichbaren Attribute, inklusive Attributwerten im Prozessobjekt enthalten sein müssen, um bestimmen zu können, dass sich zwei Objekte stark ähnlich sind.

Beispiel 6: Gegeben sind die zwei Change-Primitive-Mengen $addCL_{prim}$ und $delCL_{prim}$ aus Abbildung 27. Mittels Attribut-Gleichheit soll der Grenzwert von 0.75 für eine starke Ähnlichkeit von zwei Prozessobjekten gezeigt werden, wenn die Label-Gleichheit 0 beträgt.

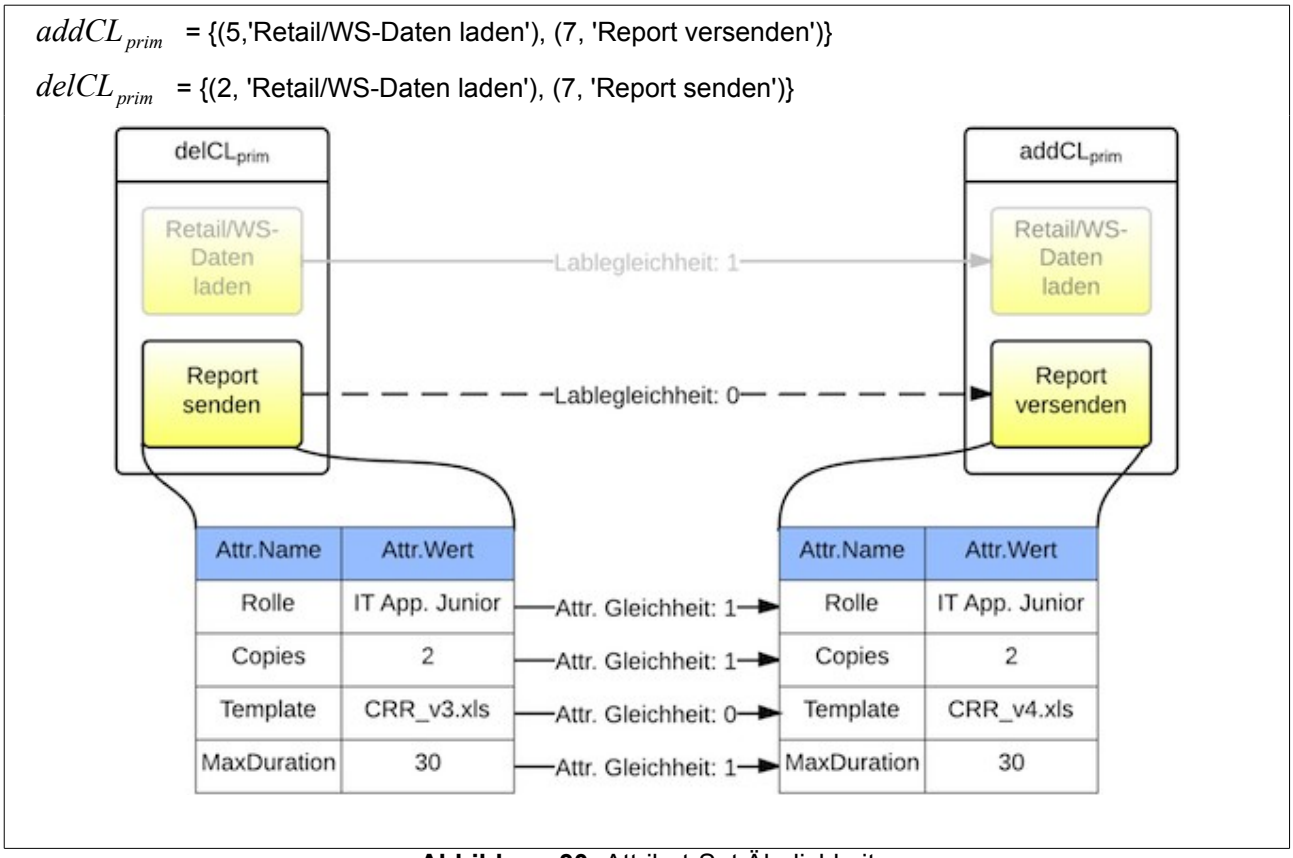


Abbildung 30: Attribut-Set Ähnlichkeit

In Abbildung 30 wird eine Ähnlichkeitsratio über das Attribute-Set eines Objektes errechnet. Da $sim_{naiv}('Report\ senden', 'Report\ versenden') = 0$ ist, werden die vier Attribute der beiden Objekte herangezogen. Die Messung erfolgt über eine Methode $sim_{naiv,AttrSet}$, welche die übereinstimmenden Attribute durch die Gesamtanzahl der existierenden Attribute errechnet. So ergibt sich für $sim_{naiv,AttrSet}(Report\ senden, Report\ versenden) \approx 0{,}75$. Obwohl keine Label-Gleichheit gegeben ist, kann dennoch über die Attributsliste der beiden Objekte eine starke Ähnlichkeit errechnet werden.

7 Erweitertes Konzept zur Ähnlichkeitsmessung

Zur Ähnlichkeitsmessung kann die Label-Beschriftung und die Menge der Attribute herangezogen werden. Allerdings wurde im vorhergehenden Abschnitt über die Möglichkeiten zum Messen der Ähnlichkeit von Prozessobjekten, nur eine Gleichheit der Labels und Attribut-Menge unterstellt. Da diese Annahme, bei sich ständig verändernden Geschäftsprozessen, als unrealistisch betrachtet werden kann, ist es eine Anforderung an die Ähnlichkeitsmessung diese Veränderungen zu berücksichtigen. Im Gesamtkontext des Vergleichens, werden Übereinstimmungen und Abweichungen aufgewogen und zu einer Ratio zusammengefasst. Solch Übereinstimmungen und Abweichungen treten bei den zwei Labels `Report senden` und `Report versenden` auf. Die Funktion soll erkennen, dass die beiden Fragmente `Report` und `senden` in beiden Labels enthalten sind und das Teilfragment '`ver`' nur in einem der beiden Labels vorkommt. Entsprechend soll die Funktion die Ratio für die Ähnlichkeit der zwei Labels `Report senden` und `Report versenden` heruntersetzen.[4]

Zusätzlich werden nach [RIMM92] für Funktionen zur Messung der Ähnlichkeit zweier Objekte, die folgenden Bedingungen vorausgesetzt: Sei sim eine Funktion zur Messung zweier Objekte, so muss gelten:

$$sim(x,x) = 1 \quad (1)$$
$$sim(x,y) = sim(y,x) \quad (2)$$

Formel 2: Eigenschaften einer Simularitätsfunktion

Dieses besagt, dass die Messung der Ähnlichkeit mit einer Funktion sim , welche ein Objekt x mit sich selbst vergleicht, immer zu 100 % übereinstimmt. Außerdem, beim Vergleich zweier unterschiedlicher Objekte, ungeachtet der Vergleichsreihenfolge, liefert die Funktion sim immer den selben Funktionswert zurück. Der Funktionswert liegt – wie schon im Abschnitt „Maßzahl zur Bestimmung der Ähnlichkeit“ angeführt - im Intervall $[0,1]$.

4 Später wird gezeigt, dass die Menge an Unterschiede lediglich auf 'v' reduziert werden kann.

7.1 „Bag of Words“ und Vector-Space-Modell

Um zwei Texte miteinander zu vergleichen, können die Texte in ihre einzelnen Bestandteile aufgespalten werden, sodass für jeden Text eine Menge $D=\{t_{1,...}, t_n\}$ entsteht. D Ist ein Dokument, das den Text widerspiegelt und die Terme $t_{1,...}t_n$ beinhaltet [SAMG86]. Ein Term t kann dabei ein Wort oder Textfragment aus dem Text sein. Diese Repräsentationsform eines Dokuments ist in der Literatur auch als „bag-of-words“-Modell bekannt. Bei diesem Modell werden die enthaltenen Terme mit der Häufigkeit ihres Auftretens, auch *term-frequency tf* genannt [SGFH83], gespeichert. Die Reihenfolge der Terme spielt dabei keine Rolle [WALL06]. Es wird argumentiert [GWCY92,BUMO06], dass ein BOW-Modell für die Analyse von Texten nur schlecht geeignet ist, da durch den Verlust der Term-Reihenfolge viel Semantik verloren gehen kann, welche für einen Vergleich von Bedeutung sein könnte. Ein Beispiel soll die Funktionalität und den Kritikpunkt veranschaulichen. Gegeben sind zwei Texte mit den folgenden Inhalten:

$Text_1 =$"Komponente A wird in die Komponente B integriert"

$Text_2 =$"Komponente B wird in die Komponente A integriert"

Daraus entsteht eine von mehreren möglichen Term-Anordnungen in den Mengen D_1 und D_2 . Hier wurden diese nach deren alphabetischen Reihenfolge sortiert: Für $Text_1$ erhält man so $D_1=\{A, B, die, Komponente, in, integriert, wird\}$ und für $Text_2$ die Menge $D_2=\{A, B, die, Komponente, \text{in}, integriert, wird\}$. Die korrespondierenden tfs sind folglich $D_{1,tf}=\{1,1,1,2,1,1,1\}$ und $D_{1,tf}=\{1,1,1,2,1,1,1\}$.

Das Beispiel verdeutlicht, dass sowohl D_1 , also auch D_2 die gleichen Terme beinhalten und die tf für jeden Term ebenfalls übereinstimmt. Somit können die beiden Texte – auf einer syntaktischen Ebene – als gleich angesehen werden. Die Semantik spielt dabei keine Rolle. Für das BOW-Modell ist es unerheblich ob - wie in diesem Beispiel - die Komponente A in B integriert wird oder umgekehrt.

Trotz der propagierten Schwäche dieses Ansatzes wird in den nachstehenden Abschnitten gezeigt, dass ein BOW-Modell dennoch für die Ähnlichkeitsmessung zweier Prozessobjekte als sinnvoll erscheint und eben dieser Reihenfolgen-Verlust für die Ähnlichkeitsanalyse von Prozessobjekten vorteilhaft ist.

Aufbauend auf der Grundlage des BOW-Modells wird das Vector Space Model [SGWY75]

angewendet. Werden die Häufigkeiten der einzelnen Terme unter Anwendung eines Referenzdokuments Q in einem Dokument D bestimmt, entsteht ein multidimensionaler Häufigkeitsvektor $\vec{d}$ für D. Jede Dimension repräsentiert einen enthaltenen Term, inklusive der tf im Dokument. Existieren Terme in Q mit Q/D, werden diese in $\vec{d}$ mit 0 gewichtet. Dies ist notwendig um die Vektor-Dimensionen gleich zu halten und beide Vektoren vergleichbar zu machen. Die Simularität zweier Vektoren kann dann mittels Winkelfunktion errechnet werden.

Werden zwei Dokumente D_1 und D_2 von Prozessobjekten mit dem Vector Space Model verglichen, so muss eines der beiden Prozessobjekte Q repräsentieren. Die Terme aus D_1 werden somit zu Q und daraus folgt, dass $\vec{d}_1 = D_{1,tf}$. $\vec{d}_2$ anschließend in Abhängigkeit der enthaltenen Terme in Q erstellt wird. Dies bedeutet, dass nur jene Terme in den Häufigkeitsvektor für D_2 aufgenommen werden, welche auch im Vektor von D_1 enthalten sind.

Beispiel 7: Gegeben sind die beiden Dokumente D_1 mit $D_1 = \{Report, senden\}$ und $D_2 = \{Report, versenden\}$. Als Search-Query wird $Q = D_1$ festgelegt. Somit entstehen für die beiden Dokumente die folgenden Vektoren: $\vec{d}_1 = \{1,1\}, \vec{d}_2 = \{1,0\}$. Zeichnet man nun die beiden zwei-dimensionalen Vektoren in ein Koordinatensystem ein, kann der Winkel leicht eruiert werden:

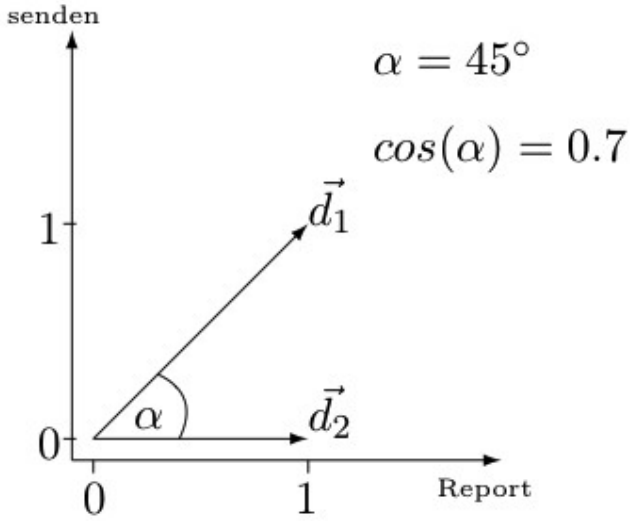


Abbildung 31: Ähnlichkeit zweier Vektoren

In der obigen Abbildung 31 wurde als Winkelfunktion der Kosinus verwendet. Die Kosinus-Funkion hat die Eigenschaft, dass diese im Intervall [-1,1] definiert und für die

Simularitätsmessung geeignet ist. Je spitzer der Winkel, umso ähnlicher sind sich die beiden Vektoren. Sind sich zwei Vektoren gleich, ergibt dies einen Kosinus von 1 (was einer Simularität von 100 % entspricht). Würde man mit einem Winkel-Grad messen, so entsprechen dann 0° der Gleichheit zweier Vektoren.

Der nächste Abschnitt soll zeigen, wie die Winkelfunktion für eine Ähnlichkeitsmessung ausgelegt werden kann.

7.2 Kosinus-Simularität

Werden beide Konzepte – Vector Space Modell und Messung des Winkels mit einer Kosinus-Funktion – kombiniert, dann spricht man von der sogenannten Kosinus-Simularität [HUAN08]. Sind zwei Dokumente D_1 und D_2 sowie deren Vektoren $\vec{d}_1$ und $\vec{d}_2$ bekannt, berechnet sich ihre Kosinus-Simularität aus:

$$sim_{cos} = \frac{\vec{d_1} \cdot \vec{d_2}}{\|\vec{d_1}\|\|\vec{d_2}\|} = \\ = \frac{\sum_{i=1}^{n} d_{1i} \times d_{2i}}{\sqrt{\sum_{i=1}^{n} (d_{1i})^2} \times \sqrt{\sum_{i=1}^{n} (d_{2i})^2}}$$

Formel 3: Kosinus-Simularität

Die Kosinus-Simularität lässt sich auch auf negative Gewichtungen in einem Vektor anwenden - eine gegenläufige Simularität $sim_{cos}(\vec{d}_1, \vec{d}_2) < 0$ wäre die Konsequenz. Dieser Fall tritt im Kontext des Information Retrieval nicht auf: Denn für die Gewichtung bzw. term-frequency eines Terms muss gelten $tf \geq 0$. Entweder ist der Term in D_i enthalten ($tf > 0$) oder der Term ist nicht enthalten, dann gilt $tf = 0$. Aufgrund dieser Bedingung verkleinert sich der Gültigkeitsbereich der Vektor-Elemente auf den Zahlenbereich, $\mathbb{N} \cup \{0\}$ und somit auch den Funktionswertebereich der Kosinus-Simularität auf das

Intervall [0,1]. Nachfolgend werden Beispiele gezeigt, wie sich Kosinus-Simularität bei unterschiedlichen Vektoren verhält:

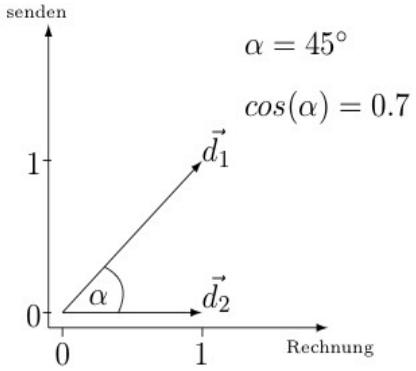
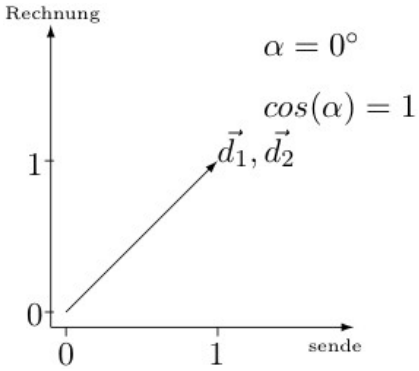
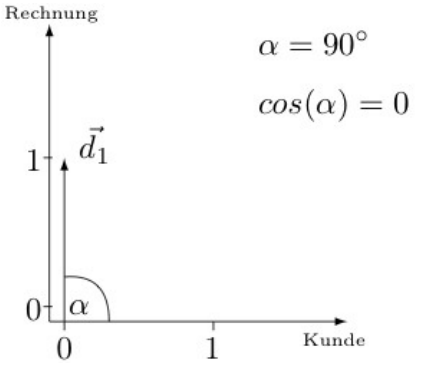
Vektoren-Ähnlichkeit	Vektoren-Gleichheit	Vektoren-Ungleicheit
$D_1=\{Report, senden\}$	$D_1=\{sende, Report\}$	$D_1=\{Report\}$
$D_2=\{Report, versenden\}$	$D_2=\{sende, Report\}$	$D_2=\{Portfolio\}$
$Q=D_1$	$Q=D_1$	$Q=D_1$
$\vec{d}_1=\{1,1\}, \vec{d}_2=\{1,0\}$	$\vec{d}_1=\{1,1\}, \vec{d}_2=\{1,1\}$	$\vec{d}_1=\{1\}, \vec{d}_2=\{0\}$
		

Abbildung 32: Kosinus-Simularität Verhalten

Quelle: Eigene Erstellung

Mit dem bereits gezeigten, können mittlerweile zwei Bedienungen (siehe Formel 2) für die Simularitätsfunktion erfüllt werden: Reflexivität - siehe mittleres Beispiel in Abbildung 32 mit $\alpha=0°$ - und der Funktionswert liegt im Intervall [0,1].

Zu prüfen ist daher nur noch die Symmetrie-Eigenschaft zwischen den beiden zu vergleichenden Objekten. Eine Verletzung der Symmetrie-Eigenschaft ist dennoch gegeben, da die Auswahl des Prozessobjektes für das Referenzdokument Q immer auf das erste Prozessobjekt gefallen ist. Durch diese Vorgehensweise wird der Häufigkeitsvektor des zweiten Prozess-Objektes beeinflusst. Ein Beispiel soll diese Verletzung veranschaulichen:

Beispiel 8: Gegeben seien zwei Dokumente von Prozessobjekten mit $D_1=\{Report, senden\}$ und $D_2=\{Report, an, Oenb, per, Mail, versenden\}$, wobei $Q=D_1$. Daraus entstehen dann $\vec{d}_1=\{1,1\}$, sowie $\vec{d}_2=\{1,0\}$ mit einer entsprechenden $sim_{cos}(\vec{d}_1,\vec{d}_2)=0.7$. Wird der Q nicht D_1 zugewiesen, sondern $Q=D_2$, erhält man für die Dokumente folgende zwei Vektoren $\vec{d}_1=\{1,0,0,0,0,0\}$ und $\vec{d}_2=\{1,1,1,1,1,1\}$, dann

errechnet sich $sim_{cos}(\vec{d}_{2,}\vec{d}_1) \approx 0.41$.

Wie das Beispiel zeigt, ist dies der Verstoß gegen die geforderten Symmetrie-Eigenschaften für die Simularitätsfunktion. Um diese Bedingung zu erfüllen, kann eine Anpassung des Referenzdokumentes vollzogen werden, sodass $Q = D_1 \cup D_2$ gilt. Dann existiert ein unabhängiges Referenzdokument, welches alle Dimensionen der zu vergleichenden Objekte beinhaltet und nicht nur jene aus D_1 . Fehlt eine Dimension im Vektorraum, wird diese ergänzt und mit einer Gewichtung von $tf = 0$ belegt und im jeweiligen Dokument ergänzt. Durch das Hinzufügen von künstlichen Dimensionen, werden fehlende Informationen aus dem Gegenüber hinzugefügt. Dies ermöglicht die Erfüllung der Symmetrie-Eigenschaften:

Beispiel 9: Angelehnt an das obige Beispiel, welches noch gegen die Symmetrie-Eigenschaft verstößt, soll Q standardisiert werden: $Q = D_1 \cup D_2 = \{Report, an, Oenb, per, Mail, versenden, senden\}$. Folglich erhält man die beiden Vektoren $\vec{d}_1 = \{1,0,0,0,0,0,1\}$ und $\vec{d}_2 = \{1,1,1,1,1,1,0\}$ mit $sim_{cos}(\vec{d}_{1,}\vec{d}_2) \approx 0.28$ als auch $sim_{cos}(\vec{d}_{2,}\vec{d}_1) \approx 0.28$.

Das Beispiel zeigt, dass sowohl eine Unabhängigkeit der Simularitätsanalyse vom Referenz-Objekt erreicht wird, als auch eine Steigerung der Präzision der Messung, wenn das Referenzdokument Q normalisiert wird.

Bisweilen wurde für den Term in einem Dokument D unterstellt, dass es sich um ein Wort handelt. Das Beispiel `Report senden` und `Report versenden` zeigt deutlich, dass bereits durch sehr geringe Änderungen die Vektoren stark von einander abweichen können.

Es ist möglich von jedem Wort die Stammform herzuleiten, was im Allgemeinen als „stemming“ bekannt ist. Dabei werden Flexionsformen von, zum Beispiel, Verben getrennt um den Wortstamm zu bilden [JBLO68]. Das Verb `versenden` wird somit nach `senden` abgeleitet. Demzufolge würde sich der Vektorraum um eine Ebene verkleinern, da die beiden Terme 'sende' und 'versenden' den gleichen Wortstamm tragen.

Das Stemming erweist sich jedoch für den Vergleich von Prozessobjekten als nachteilig: Erstens, kann nicht sichergestellt werden welche Sprache im Prozessmodell verwendet wird. Zweitens muss für jede Sprache ein eigener Stemmer-Algorithmus erstellt werden.

[COBE68][SAJA06]. Drittens, müsste die Bildung von Okkasionalismen und Neologismen in einem bestimmten Kontext, mit dem gleichzeitigen Verwenden von Anglizismen [BAAO10] ebenfalls berücksichtigt werden, was die Ableitung der Stammform erschwert.

Betrachtet man nun einen Term nicht mehr als Wort sondern als einen einzelnen Buchstaben, kann die syntaktische Analyse des Wortes präziser erfolgen. Hierzu wird das Dokument D in die einzelnen Buchstaben aufgeteilt und tf wird anschließend für jedes einzelne Zeichen bestimmt. Bei einer Aufsplittung der Terme auf Buchstaben-Ebene entstehen mehrere Dimensionen, die Gewichtung der einzelnen Vektor-Dimensionen werden dadurch akkurater bestimmt und die Zeichenkette besser im Vektorraum repräsentiert. Daher kann die Simularitätsfunktion auch genauere Ergebnisse liefern. Veränderungen des Wortstammes fallen nicht mehr so stark ins Gewicht wie bei einer Analyse auf Wort-Ebene.

Beispiel 10: Angenommen, es soll ein Vergleich der beiden Prozessobjekt-Dokumente 'Report senden' und 'Report versenden' stattfinden, dann werden die beiden Texte in zwei Dokumente $D_{letter1}$ für den Text `Report senden` und $D_{letter2}$ mit `Report versenden` und deren korrespondierenden Häufigkeitsvektoren angelegt:

$D_{letter1}=\{c,d,e,g,h,n,r,s,u,v\}$ mit $\vec{d_{letter1}}=\{1,1,3,1,1,4,1,1,1,0\}$ und

$D_{letter2}=\{c,d,e,g,h,n,r,s,u,v\}$ mit $\vec{d_{letter2}}=\{1,1,4,1,1,4,2,1,1,1\}$.

$Q=\{e,n,o,p,r,s,t,v\}$

$D_{letter1}=\{e,n,o,p,r,s,t\}$ mit $\vec{d_{letter1}}=\{3,2,1,1,1,1,1,0\}$ und

$D_{letter2}=\{e,n,o,p,r,s,t,v\}$ mit $\vec{d_{letter2}}=\{4,2,1,1,2,1,1,1\}$.

Daraus ergibt sich eine Kosinus-Ähnlichkeit von annähernd 0,97.

Die gezeigten Beispiele in diesem Abschnitt verdeutlichen, dass sich die anfangs erwähnte Kritik gegenüber dem BOW-Modell, als nützlich für die Ähnlichkeitsanalyse von Prozessobjekten herausstellt. Würde bei der Ähnlichkeitsmessung die Wortreihenfolge zum Beispiel zwischen `Report senden` und `sende Report` ebenfalls noch berücksichtigt werden, könnte die Simularitätsfunktion niedrigere Ähnlichkeitswerte berechnen, obwohl die beiden Dokumente offensichtlich nahezu identisch sind.

7.3 Attributwerte und Kosinus-Ähnlichkeit

Grundsätzlich kann bei Attributwerten ebenfalls die Kosinus-Ähnlichkeit angewendet werden. Für die Messung der Attribut-Gleichheit unter der Verwendung der Kosinus-Ähnlichkeit, mussten jedoch Erweiterungen getätigt werden, um aussagekräftige Resultate zu erzielen. Im Unterschied zur Label-Gleichheit müssen bei einer Attribut-Gleichheit mehrere sim_{cos} -Funktionen durchgeführt werden. Wobei nur Ähnlichkeitsfunktionen für Attributwerte gerechnet werden, bei welchen die Attributnamen in beiden Prozessobjekten gleich sind. Dies verhindert ein Kreuzprodukt beim Berechnen der Ähnlichkeiten, stellt aber auch gleichzeitig sicher, dass für jedes vergleichbare Attribut-Paar nur ein Ähnlichkeitswert errechnet wird.

Allerdings fließen bei einer Attribut-Gleichheit nicht nur die Attributwerte in die Berechnung mit ein, sondern auch die Attributnamen. Dabei werden die Attributnamen mittels einer Ähnlichkeitsfunktion $sim_{AttrNames}(PObj_1, PObj_2)$ errechnet, die zwar eine Kosinus-Funktion beinhaltet, jedoch die Attributnamen der Prozessobjekte nicht auf einer Buchstaben-Ebene errechnet, sondern auf einer Wort-Ebene. Es wurde die Wort-Ebene gewählt, damit fehlende Attribute in einem der beiden Prozessobjekte, die Ähnlichkeit stärker beeinflussen. Die beiden Häufigkeitsvektoren $\vec{d}_1$ und $\vec{d}_2$ in der $sim_{AttrNames}$ -Funktion erstellen sich mit $Q = \left(PObj_{1,\,AttrNames} \cup PObj_{2,\,AttNames} \right)$.

Beispiel 11: Seien wieder die zwei Prozessobjekte 'Report senden' und 'Report versenden' in einem Change-Log CL_{prim} vorhanden, wobei diese die folgenden Attribute aufweisen:

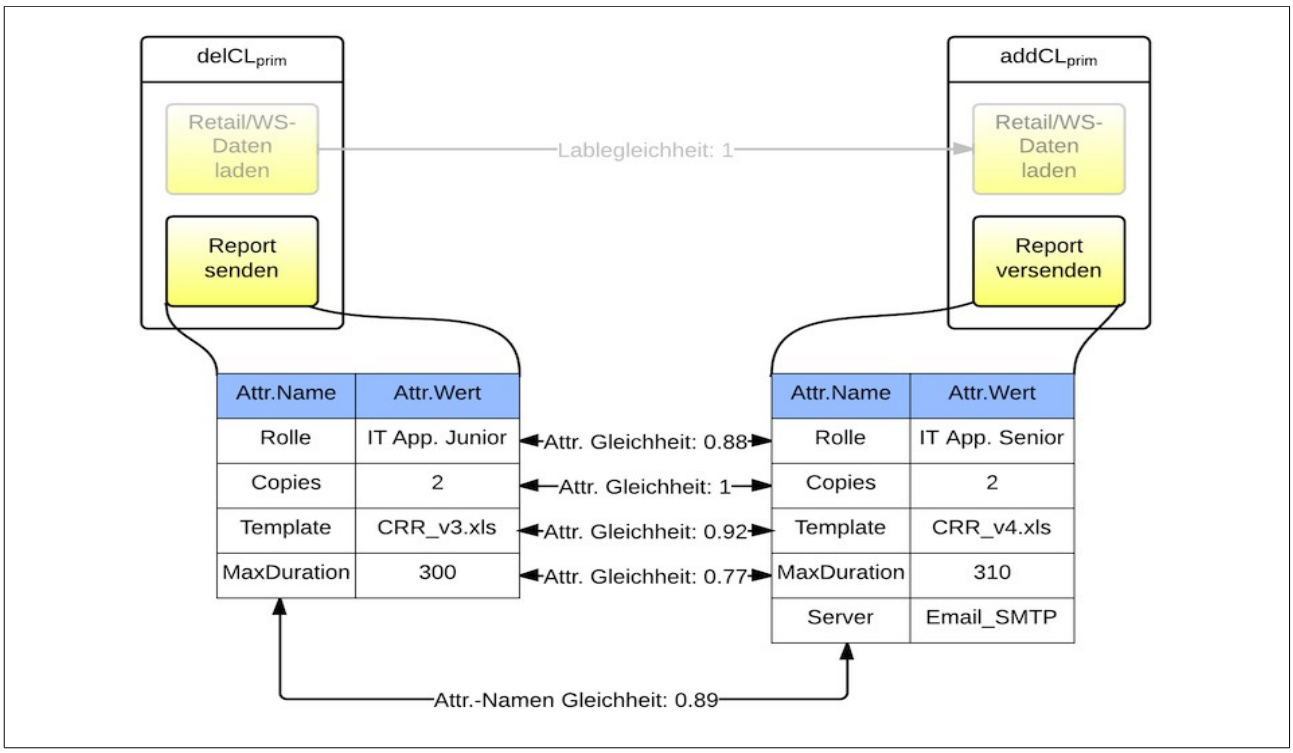


Abbildung 33: Attribut-Gleichheit mit Kosinus-Funktion

Zuerst wird die Menge $A=\{Rolle, Copies, Template, MaxDuration, Server\}$ bestimmt, dann die zwei Vektoren $\vec{o}_1$ und $\vec{o}_2$ mit den korrespondierenden term-frequencies der beiden Prozessobjekte:

$$\vec{o}_1=\begin{pmatrix} Rolle:1 \\ Copies:1 \\ Template:1 \\ MaxDuration:1 \\ Server:0 \end{pmatrix} \quad \vec{o}_2=\begin{pmatrix} Rolle:1 \\ Copies:1 \\ Template:1 \\ MaxDuration:1 \\ Server:1 \end{pmatrix}$$

Entsprechend ist dann $sim_{AttrSet}(POBj_1, POBj_2)\approx 0.89$ und somit sind sich die beiden Prozessobjekte hinsichtlich ihrer Attributmenge stark ähnlich. Der zweite Schritt analysiert anschließend mi t $sim_{AttrValue}(POBj_{1,} POBj_2)$ eine Attributmenge $A_{eq}=\{POBj_{1,AttrNames} \cap POBj_{2,AttrNames}\}$. Wobei der Vergleich zweier übereinstimmender Attributnamen über deren Attributwerte erfolgt und für jedes Paar eine eigene Kosinus-Simularität bestimmt wird. In diesem Fall beinhalten die Vektor-Dimensionen die einzelnen Buchstaben der Attributwerte, um eine genauere Aussage über deren Inhalte treffen zu können. Wird nun auf Buchstaben-Ebene die Simularität der Attributwerte errechnet, so erhält man folgenden Ähnlichkeitswerte für alle Attribute in A_{eq} :

$sim_{AttrValue, Rolle}(POBj_{1,} POBj_2)\approx 0{,}88$, $sim_{AttrValue, Copies}(POBj_{1,} POBj_2)=1$,

$sim_{AttrValue, Template}(POBj_{1,} POBj_2)\approx 0{,}92$ und $sim_{AttrValue, MaxDuration}(POBj_{1,} POBj_2)\approx 0{,}77$.

7.4 Gesamt-Ähnlichkeit

Wie schon im Abschnitt zur 'Maßzahl zur Bestimmung der Ähnlichkeit' vorweggenommen, können die Label-Beschriftung und die Attribute zweier Prozessobjekte die Ähnlichkeit genauer beschreiben, als eine isolierte Betrachtung einzelner Gleichheiten. So sollen nun in diesem Abschnitt die Simularitätswerte aus der Label-Gleichheit und der Attribut-Gleichheit zusammengeführt werden. Dadurch kann eine Gesamt-Ähnlichkeit errechnet werden, welche für den eingangs erwähnten Test, für die kritische Marke von 0.75 herangezogen werden kann. An dieser Stelle ist jedoch anzumerken, dass die nachfolgende Formel zur Gesamt-Ähnlichkeit, nur für all jene Prozessobjekte in $delCL_{prim}$ durchgeführt wird, für die keine Label-Gleichheit von 1 gefunden wurde. Prozessobjekte mit einer Label-Gleichheit von 1 können bereits eindeutig zugeordnet werden.

Um nun endgültig $sim(PObj_1, PObj_2)$ zu errechnen, werden die einzelnen Simularitätswerte wie folgt kombiniert:

$$sim(PObj_1, PObj_2) = \frac{sim_{label}(PObj_1, PObj_2) + sim_{AttrSet}(PObj_1, PObj_2) + \sum_{i \in A_{eq}} sim_{AttrValue}(PObj_{1,i}, PObj_{2,i})}{max(sim_{label}) + max(sim_{AttrName}) + max(sim_{AttrValue})}$$

Formel 4: Gesamt-Ähnlichkeit

Im Nenner werden die maximal möglichen Werte der Simularitätsmessung der entsprechenden sim -Funktion angegeben. Für die beiden Funktionen sim_{label} und $sim_{AttrSet}$ liefert max den Wert 1, da die Label-Gleichheit und das Attributset A nur einmal berechnet werden. Bei $sim_{AttrValue}$ gilt jedoch $max(sim_{AttrValue}) = |A_{eq}|$, weil für jedes $a \in A_{eq}$ eine eigenständige Kosinus-Simularität errechnet wird.

Beispiel 12: Aus dem eben gezeigten Beispiel aus der Attribut-Gleichtheit soll nun die Gesamt-Ähnlichkeit der beiden Objekte `Report senden` und `Report versenden` errechnet werden. Aus den vorhergehenden Beispielen sind die Ähnlichkeitswerte bereits bekannt:

$sim_{label} \approx 0.97$, $sim_{AttrNames} \approx 0.89$, $sim_{AttrValue,Rolle} \approx 0{,}88$, $sim_{AttrValue,Copies} = 1$, $sim_{AttrValue,Template} \approx 0{,}92$ und $sim_{AttrValue,MaxDuration} \approx 0{,}77$. Eingesetzt in die Formel für $sim(PObj_1, PObj_2)$ ergibt sich:

$$sim(PObj_1, PObj_2) = \frac{0.97 + 0.89 + (0.88 + 1 + 0.92 + 0.77)}{1 + 1 + 4} = 0.905$$

Die Gesamt-Ähnlichkeit der beiden Objekte beträgt daher 90,5 %, was einer starken Ähnlichkeit gleichkommt.

7.5 Behandlung von Default-Werten

Workflow Management Systeme wie AristaFlow[5] oder UC4[6], aber auch Modellierungstools wie Signavio[7], bieten für die Prozessobjekte ein vorgefertigtes Set an parametrierbaren Attributen an. Werden Attribute vom Anwender nicht explizit gesetzt, werden diese mit Standardwerten im Modell hinterlegt. Standardwerte können zum Beispiel 'null' oder ein im Vorhinein festgelegter Wert sein. Ist die Attributmenge für Prozessobjekttypen fix festgelegt und werden die Attribute automatisch gesetzt, kann dies zu Verzerrungen bei der Ähnlichkeitsmessung führen. Ebenso denkbar wäre es, dass der Benutzer für ein Attribut immer den selben Wert setzt - auch dann kann von einer Verfälschung des Ergebnisses durch Standardwerte gesprochen werden.
Da bei der Messung die Attributwerte stärker berücksichtigt werden als die Attributnamen, schlägt sich die Verzerrung durch eine Erhöhung der Ähnlichkeitsmessung nieder, obwohl die Prozessobjekte, aufgrund der vom Benutzer gesetzten unterschiedlichen Attributwerte, nicht als ähnlich zu werten sind.

Beispiel 13: Angenommen es existiert ein Prozessmodell mit zwei Prozessobjekten aus einem Workflow Management System, welche ein fixes Set an Attributen haben:

5 *www.**aristaflow**.com/*

6 *www.**uc4**.com/*

7 *www.**signavio**.com/*

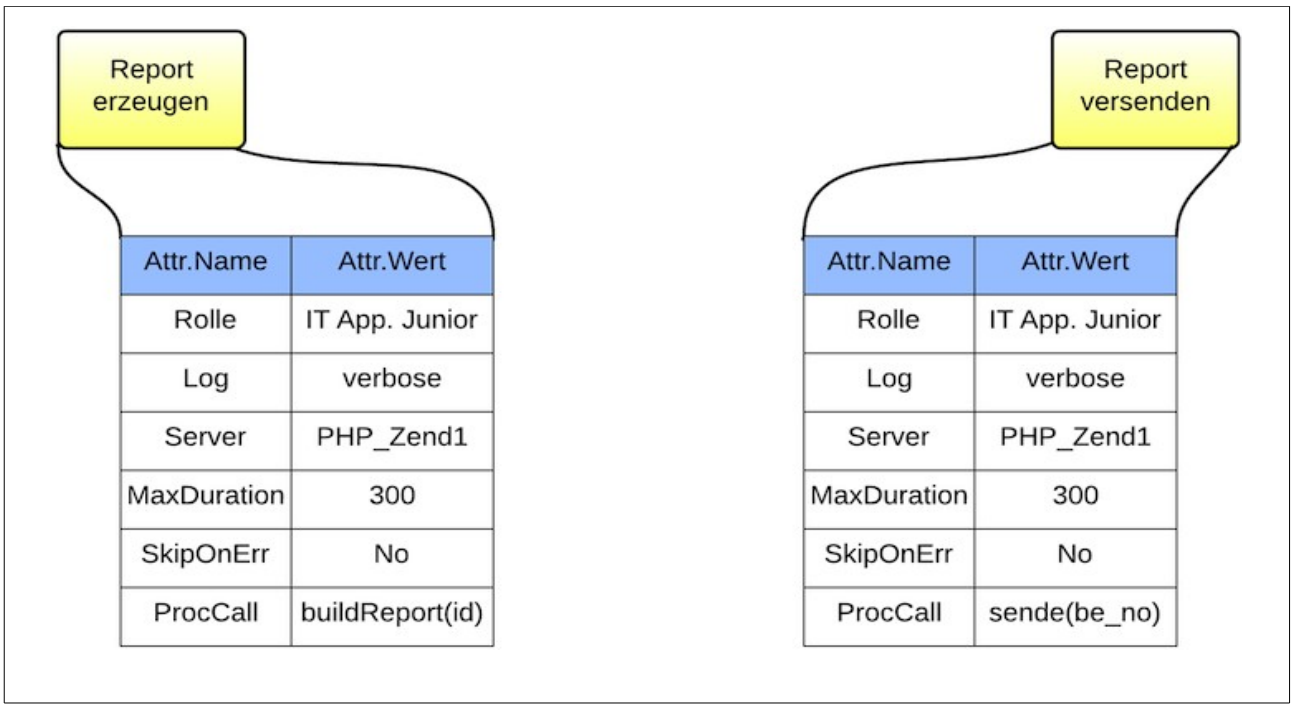


Abbildung 34: Fixe Anzahl von Attributen

Betrachtet man die Attribute, so sind, abgesehen von `ProcCall`, alle Attributwerte gleich. Schon aus den Label-Beschriftungen lässt sich jedoch – semantisch betrachtet – erkennen, dass die beiden Prozessobjekte unterschiedliche Funktionen durchführen. Zu erwarten wäre daher, dass die beiden Objekte nicht ähnlich sind. Eine Analyse zeigt jedoch eine Gesamt-Ähnlichkeit von $sim(POBj_{1,}\ POBj_2) \approx 0.91$: Obwohl $sim_{label} = 0.89$ und $sim_{AttrValue,\ ProcCall} = 0.43$ sogar unter der kritischen Marke von 0.75 liegt, fallen die restlichen Werte mit $sim_{AttrValue,x} = 1$ so stark ins Gewicht, dass die Simularität auf über 0.91 ansteigt. Unter der Annahme, dass Standardwerte erzeugt werden, kann daraus geschlossen werden, dass der Benutzer nur das Attribut `ProcCall` explizit gesetzt hat.

Standardwerte sollten daher erkannt und prinzipiell aus der Ähnlichkeitsanalyse ausgenommen werden. Hierfür werden für alle auftretenden Prozessobjekttypen, die Attributnamen inklusive deren Werte gesammelt und mittels statistischem Test ausgewertet. Ein Attribut in einem Prozessobjekttyp stellt dabei ein Merkmal dar, welches unterschiedliche Ausprägungen enthalten kann. Die Ausprägungen des Attributs werden auf deren Auftrittshäufigkeit hin untersucht, womit eine Häufigkeitsverteilung entsteht. Mittels dieser kann anschließend beurteilt werden, ob eine Ausprägung in einem Attribut signifikant häufiger enthalten ist als andere Werte. Dabei wird unterstellt, dass die einzelnen Ausprägungen der Attribute über alle Prozessobjekte eines bestimmten Typen

gleichverteilt sind. Mit anderen Worten, jede Ausprägung eines Attributs kommt in der Häufigkeitsverteilung genau n/m -mal vor. n ist die Anzahl der Ausprägungen eines Merkmals und m ist die Anzahl der Prozessobjekte eines Prozessobjekttypen in einem Prozessmodell.

Der Test, ob nun die Ausprägungen annähernd gleichverteilt sind, erfolgt mittels Chi-Quadrat-Test und einem Signifikanzniveau (p-Wert) von 0,2. Der p-Wert wird hoch gehalten, um tatsächlich automatisch gesetzte Werte zwischen zufälligen Schwankungen zu unterscheiden. Des Weiteren gilt eine Ausprägung als Standardwert, wenn diese in allen Prozessobjekten eines bestimmten Prozesstyps vorkommt. Also dann, wenn es keinen anderen Wert zu diesem Attribut gibt.

Beispiel 14: Angewendet auf das zuvor gezeigte Beispiel 13, werden die folgenden Attribute mit enthaltenen Standardwerten erkannt, da keine andere Ausprägung für diese Attribute existiert: $A_{std} = \{Rolle, MaxDuration, Server, \log, SkipOnErr\}$.

Entsprechend erfolgt eine Berechnung der Gesamt-Ähnlichkeit von $sim(PObj_{1,} PObj_2) = 0.77$, wobei $A_{eq} = A / A_{std}$. Das Beispiel zeigt, dass durch ein Herausrechnen von Attributen mit Standardwerten die Simularität tatsächlich gesenkt werden kann, da die Menge A_{eq} auf ein Attribut reduziert werden konnte und somit in die Gesamt-Simularität nur der Faktor $sim_{AttrValue, ProcCall} = 0.43$ einfließt.

7.5.1 Straffaktor für Standardwerte

Standardwerte werden prinzipiell[8] nicht in die Berechnung mitaufgenommen, daher wird nur ein entsprechend geringeres Set an Attributen zur Analyse herangezogen. Selbst dann, wenn die betroffenen Attribute aus der Berechnung entfernt werden, entsteht bei einer fixen Menge an Attributen, für die übriggebliebenen Attribute in A_{eq} in der Berechnung von $sim_{AttrSet}$ immer ein Ergebnis von 1: Da den beiden zu vergleichenden Attributmengen die gleiche Anzahl an Attributen entfernt wird. Dies führt im Faktor $sim_{AttrSet}$, in der sim - Funktion zu einer Überbewertung der Attribute aus A_{eq} .

Das zuletzt gezeigte Beispiel zur Entfernung der Attribute mit Standardwerten errechnete $sim(PObj_{1,} PObj_2) = 0.77$. Diese Simularität ist jedoch für die beiden Objekte zu hoch

8 Es gibt allerdings Ausnahmen, wann ein Standardwert in die Analyse wieder aufgenommen werden kann. Hierfür siehe Abschnitt 'Ausnahmen für Standardwerte'

ausgefallen, da $sim_{AttrSet}=1$. Hinsichtlich der Gesamt-Simularität bedeutet dies, dass die zwei Prozessobjekte sich lediglich über das Attribute `ProcCall` beschreiben lassen und das Attributset vollständig ist. Dies ist aber offensichtlich nicht der Fall, da fünf weitere Attribute die Prozessobjekte definieren.

Um bei der Berechnung der Gesamt-Simularität zusätzlich eine Information über das Vorhandensein von Standardwerten hinzuzufügen, sollte das Ergebnis der Funktion $sim_{AttrSet}$ bereinigt werden. Hierfür wird für jedes Attribut, welches einen Standardwert beinhaltet ein Strafpunkt eingeführt. Aus der Summe der Strafpunkte errechnet sich dann ein Straffaktor pf , welcher den Funktionswert normalisiert. Für pf gilt:

$$pf = 1 - \frac{n}{m}$$

Formel 5: Straffaktor bei Standardwerten

n steht für die Anzahl der Attribute mit Standardwerten und m für die Gesamtanzahl an Attributen im Prozessobjekt eines Prozessobjekttypen.

Beispiel 15: Gegeben sind wieder die zwei Prozessobjekte aus dem vorhergehenden Beispiel, wobei dieses mal der Straffaktor pf miteinbezogen wird:
Dieser errechnet sich aus fünf Attributen, welche einen Standardwert beinhalten und der Gesamtmenge von sechs Attributen.

$$pf = 1-(\frac{5}{6})\approx 0.167$$

Die einzelnen Simularitäten ergeben dann: $sim_{label}=0.89$, $sim_{AttrValue,ProcCall}=0.43$ sowie $sim_{AttrSet}=1*pf=0.167$ und eine Gesamt-Simularität von $sim(PObj_{1,}\ PObj_2)\approx 0.49$.

Das Beispiel zeigt, dass durch eine Einführung von pf die Simularität zwischen den beiden Prozessobjekten korrigiert werden kann und so der sim -Wert die beiden Prozessobjekte genauer beschreiben kann. Demzufolge sollte unter Verwendung von Standardwerten die sim -Funktion um den Straffaktor angepasst werden.

$$sim(PObj_1, PObj_2) =$$

$$\frac{sim_{label}(PObj_1,PObj_2)+sim_{AttrSet}(PObj_1,PObj_2)*pf+\sum_{i\in A_{eq}} sim_{AttrValue}(PObj_{1,i},PObj_{2,i})}{max(sim_{label})+max(sim_{AttrName})+max(sim_{AttrValue})}$$

Formel 6: Gesamt-Ähnlichkeit mit Default-Werten

7.5.2 Ausnahmen für Standardwerte

Jedoch ist es zu strikt gesetzt, wenn Standardwerte generell aus dem Vergleich zwischen zwei Prozessobjekten ausgenommen werden. Diese sollten in der Ähnlichkeitsanalyse erst dann wieder berücksichtigt werden, wenn sich zwei Attributnamen gleichen, die Attributwerte – auch wenn einer der beiden Werte als Standard ausgemacht worden ist – unterschiedlich sind. Angenommen es existiert ein Vergleich zwischen $PObj_1$ und $PObj_2$ mit $PObj_{1,\,Rolle} = 'IT\ App.\ Junior'$ und $PObj_{2,Rolle} = 'IT\ App.\ Senior'$, weiters sei der Prozess generell für die Rolle 'IT App. Junior' ausgerichtet, weshalb diese auch als Standardwert gilt. Wird der Standardwert mit dem Attributwert nicht aufgenommen, reduziert sich somit die Attributmenge A_{eq}, womit kein Vergleich zwischen den beiden Attributen erfolgt. Dies hat zur Folge, dass der Ähnlichkeitswert verzerrt wird. Durch eine Aufnahme des Attributs 'Rolle' in A_{eq} können wieder mehr Attributwerte miteinander verglichen werden, wodurch die $sim_{AttrValue}$ berichtigt wird, was wiederum positive Auswirkungen auf die Funktion sim nach sich zieht.

An dieser Stelle sei noch angemerkt, dass eine Analyse auf Standardwerte nicht immer zum gewünschten Ziel führen muss. Eine Prüfung erscheint für ein Attribut nur dann sinnvoll, wenn folgende Bedingung α erfüllt ist:

$$\alpha = \frac{|a|}{|PObjType_x|} = 1$$

Formel 7: Anwendung Analyse auf Standardwert

Das bedeutet, dass die Anzahl eines bestimmten Attributs a gleich häufig auftreten muss, wie es Prozessobjekte von Type x im Prozessmodell gibt. Folglich wird die Entscheidung über eine Prüfung getroffen wenn

$$\alpha = \begin{cases} 1 & \text{Analyse auf Standardwert} \\ < 1 & \text{keine Analyse auf Standardwert} \end{cases}$$

Formel 8: Prüfungsbedingung Standardwerte

8 Von Change-Primitives zu Highlevel-Changes

Nachdem der Process-Structure-Tree erstellt, die Change-Primitives erkannt und Add/Delete-Kombinationen mit der Ähnlichkeitsanalyse erstellt worden sind, kann mit der Transformierung in Highlevel-Changes begonnen werden. In diesem Abschnitt werden die sieben implementierten Highlevel-Changes beschrieben:

- `ADD(Obj, predecessor, successor)`: Hinzufügen von `Obj` zwischen `predecessor` und `successor`.
- `DELETE(Obj, predecessor, successor)`: Löschen von `Obj` zwischen `predecessor` und `successor`.
- `UPDATE(Obj) TYPE`: Update von `Obj`, wobei `TYPE` die genaue Update-Operation angibt.
- `MOVE(Obj, predecessor, successor)`: Verschieben von `Obj` zwischen die beiden Objekte `predecessor` und `successor`.
- `SWAP(Obj1, Obj2)`: Tauschen des Objektes `Obj1` mit Objekt `Obj2`.
- `REPLACE (Obj, ReplacedObj, predecessor, successor)`: Ersetzen vom Objekt `ReplaceObj` mit dem Objekt `Obj`, zwischen den Objekten `predecessor` und `successor`.
- `COPY(Obj, predecessor, successor)`: Das Prozessobjekt `Obj` wurde kopiert und zwischen `predecessor` und `successor` eingefügt.

Im Folgenden wird beschrieben wie die einzelnen Highlevel-Changes aus den zuvor errechneten Change-Primitives abgeleitet werden können. Des Weiteren werden Beispiele angeführt um die Ableitung zu demonstrieren.

8.1.1 UPDATE

Ein Update entsteht dann, wenn ein bereits vorhandenes Prozess-Objekt aktualisiert wurde. Dies kann zum Beispiel durch eine Attribut-Änderung oder Namen-Änderung sein.

Um ein Update zu erkennen, werden zwei Mengen aus der zuvor durchgeführten Differenzenerkennung analysiert: Zuerst wird das Set $POBj_{eq}$ überprüft, welches alle Prozessobjekt-Paare enthält die durch eine Diagonale im Editier-Graphen gekennzeichnet worden sind, dann, das Mengen-Set der $POBj_{add/del}$, welches mittels der Ähnlichkeitsanalyse errechnet worden ist.
Bei einer Analyse auf Updates werden dabei nur die Attribute `name` und `type` sowie das `param`-Objekt aus dem Meta-Modell überprüft. Je nachdem was durch das Update aktualisiert wird, kann noch zusätzlich zwischen den folgenden drei Subtypen eines Updates unterschieden werden:

- `CHANGE`: Ein Attribut oder der Wert eines param-Objektes ist geändert worden.
- `ADD`: Ein neues param-Objekt ist zum Prozess-Objekt hinzugefügt worden.
- `DELETE`: Ein bestehendes param-Objekt wurde vom Prozess-Objekt entfernt.

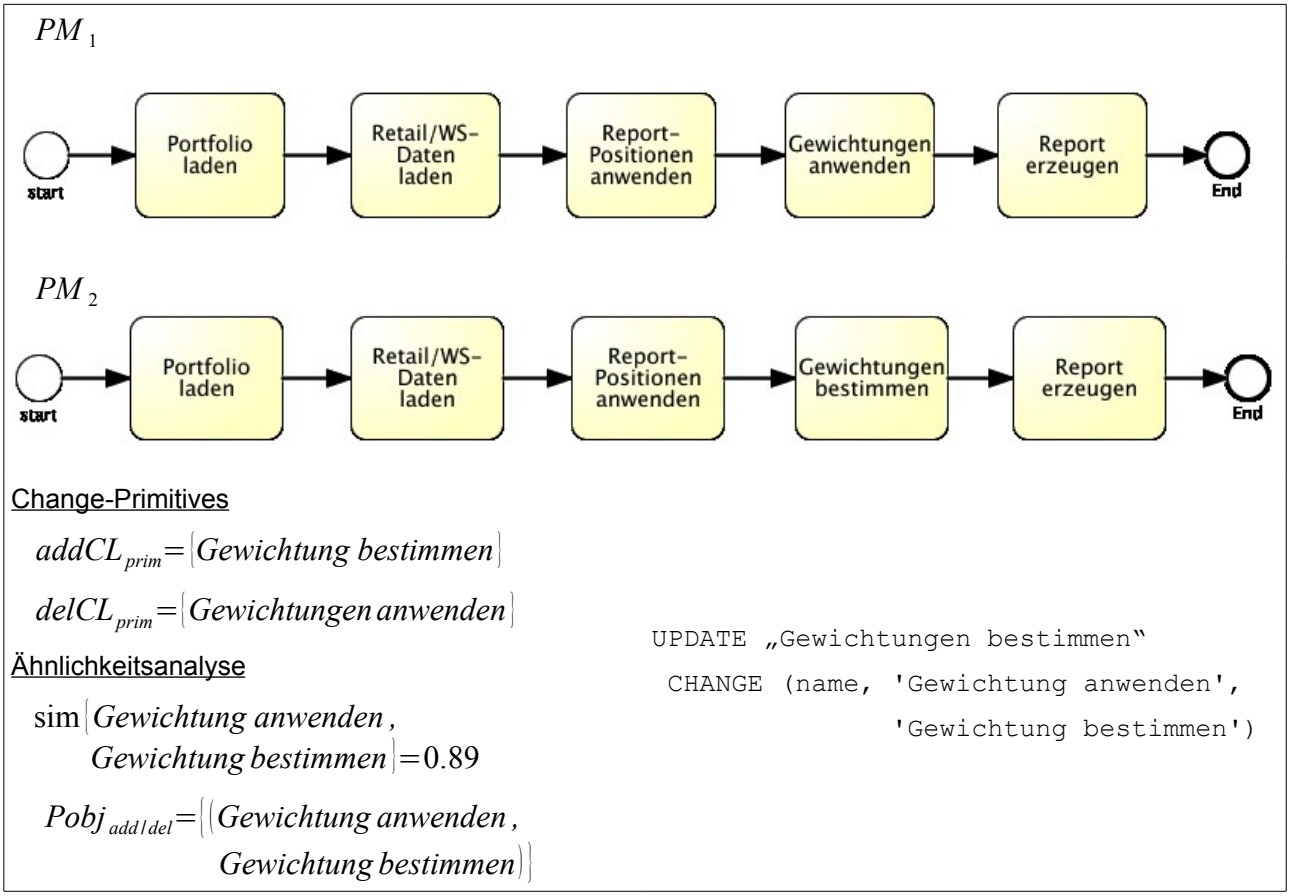


Abbildung 35: Highlevel-Change Update

Das obige Beispiel zeigt eine `name`-Änderung im Objekt `Gewichtungen anwenden`. In

den Change-Primitives scheint `Gewichtungen anwenden` in $delCL_{prim}$ auf und `Gewichtungen bestimmen` in $addCL_{prim}$. Durch die Ähnlichkeitsanalyse wird eine Ähnlichkeit von 89 % errechnet, dies ist über der gesetzten 75 % Marke, womit die beiden Objekte `Gewichtungen anwenden` und `Gewichtungen bestimmen` im Set $PObj_{add/del}$ aufgenommen werden. Die Überprüfung der beiden Kurzbeschreibungen (Attribut `name`) ergibt dann ein Highlevel-Change UDPDATE vom Subtyp `CHANGE`. Jedes Update, egal ob CHANGE, ADD oder DELETE, wird dann in das Highlevel-Change Set HLC_{upd} hinzugefügt.

8.1.2 ADD und DELETE

Grundsätzlich sind all jene Change-Primitives auch ADD- oder DELETE-Highlevel-Changes, wenn diese nicht im Set $PObj_{add/del}$ sind, da sie keine starke Ähnlichkeit mit anderen Objekten aufweisen. Es entstehen für die Highlevel-Changes ADD und DELETE zwei Sets: $HLC_{add} := addCL_{add} / PObj_{add/del}$ und $HLC_{del} := delCL_{prim} / PObj_{add/del}$. Die entsprechenden Differenzen sind dann all jene Objekte, welche tatsächlich im Prozessmodell hinzugefügt oder gelöscht wurden.

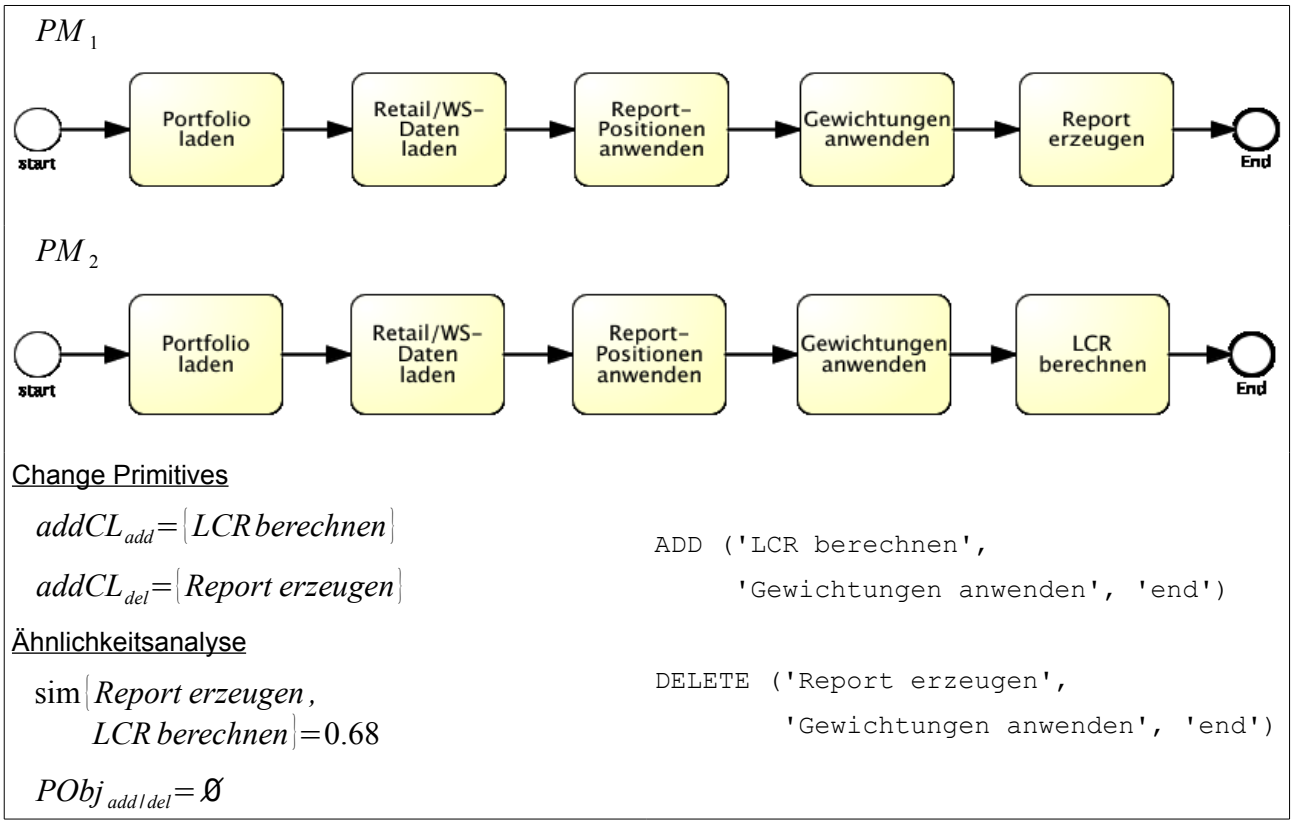


Abbildung 36: Highlevel-Change ADD und DELETE

Abbildung 36 zeigt zwei Prozessmodelle PM_1 und PM_2. In der Erkennung der Change-Primitives beinhaltet $addCL_{add} = \{LCR\,berechnen\}$ und $addCL_{del} = \{Report\ erzeugen\}$. Über die Ähnlichkeitsanalyse konnte kein Paar für die Menge $PObj_{add/del}$ gefunden werden - die Similarität zwischen Objekt $Report\ erzeugen$ und Objekt $LCR\,berechnen$ ist kleiner 0.75. Wird nun die Menge der $addCL_{prim}$ und die Menge der $delCL_{prim}$ von $PObj_{add/del}$ abgezogen, so werden genau diese zwei Objekte direkt in die Highlevel-Changes übertragen, da $PObj_{add/del} = \emptyset$. Als Ergebnis dieser Analyse erhält man ein ADD des Objektes `LCR berechnen` zwischen `Gewichtungen anwenden` und `end` sowie ein DELETE des Objekts `Report erzeugen` zwischen `Gewichtungen anwenden` und `end`, wobei sich die Delete-Operation auf das Objekt in PM_1 bezieht.
Im Abschnitt über REPLACES wird gezeigt, dass aufbauend auf diesen zwei Highlevel-Changes, ein REPLACE erzeugt werden kann.

8.1.3 Add / Delete von Fragmenten

Aktivitäten werden als ADD beziehungsweise als DELETE ausgewiesen. Um zwischen dem Hinzufügen oder Löschen von Fragmenten und Aktivitäten zu unterscheiden, wird hierfür eine eigene Highlevel-Change Operation zur Verfügung gestellt [WBRR08].

Zu Beginn befinden sich die Bestandteile eines Fragments (Split/Join-Knoten sowie die Alternativen) wie auch die Aktivitäten in den Sets HLC_{add} oder HLC_{del}. Jedoch kann über den Knoten-Typ im Process-Structure-Tree erkannt werden, dass es sich dabei um ein Split/Join-Knoten (Fragment) handelt. Da es sich um blockstrukturierte Prozessmodelle handelt und es somit zu jeden Split-Knoten auch einen Join-Knoten gibt, wird ein Add oder ein Delete eines Split- und Join-Knotens in einer Operation zusammengefasst. Bei der Operationsdurchführung wird dann sowohl für den Split- als auch für den Join-Knoten die gleiche Operation angewandt.
Alternativen erhalten ebenfalls einen eigenen Highlevel-Change. So kann erkannt werden, ob Alternativen in bereits existierenden Fragmenten hinzugefügt oder gelöscht worden sind. Ebenso wie Fragmente können aus dem Process-Structure-Tree die Alternativen abgeleitet werden um so den Highlevel-Change zu erzeugen.
Für die Split/Join-Knoten und Alternativen stehen die folgenden vier Erweiterungen von

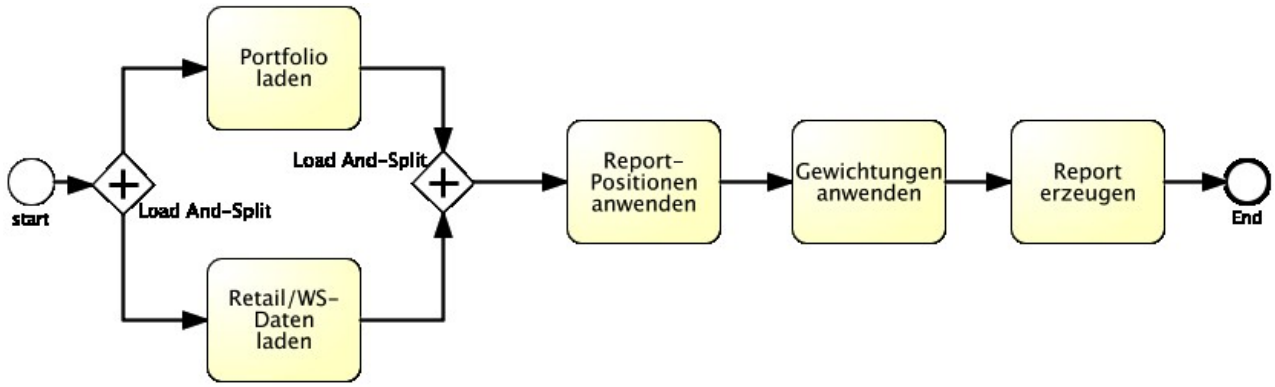
den ADD und DELTE Operationen zur Verfügung:

- `ADDFragment(FragmentName, predecessor, successor)`: Fügt einen Split- und Join-Knoten zwischen zwei Prozess-Objekte ein.
- `DELETEFragment(FragmentName,predecessor, successor)`: Löscht die Split- und Join-Knoten aus einem Modell. Wobei der `predecessor` und `successor` die Prozessobjekte aus dem alten Modell angeben.
- `ADDAlternative(AlternativeName, FragmentName, successor)`: Fügt eine Alternative zwischen einem Split-Knoten (`FragmentName`) und dem `successor` ein.
- `DELETEAlternative(AlternativeName, FragmentName, successor)`: Löscht die Alternative aus dem Fragment. `FragmentName` und `successor` sind dabei die Objekte aus dem alten Prozessmodell.

In Abbildung 37 wird das Hinzufügen eines Fragments dargestellt. Zu Beginn befinden sich die Bestandteile eines Fragments (Split/Join-Knoten sowie die Alternativen) im Set der Change-Primitives $addCL_{prim}$. Über die gleiche Vorgehensweise wie auch schon bei den Aktivitäten werden die drei Prozessobjekte als ADDs behandelt. Da eine positive Prüfung auf die Process-Structure-Knoten-Typen Fragment für `Load And-Split` sowie `Load And-Join` und Sequenzen für die beiden Alternativen erfolgte, werden ein `ADDFragment` und zwei `ADDAlternative` Highlevel-Changes generiert.

PM_1

PM_2



Change-Primitives

$$addCL_{prim} = \{Load\ \text{And-Split},\ Load\ \text{And-Split\_alt\_0},\ Load\ \text{And-Split\_alt\_1},\ Load\ \text{And-Join}\}$$

$$delCL_{prim} = \emptyset$$

//Ähnlichkeitsanalyse

nicht durchführbar, da $delCL_{prim}$ leere Menge

```
ADDFragment(Load And-Split,
            start,
            Report-Positionen anwenden
           )

ADDAlternative(
      Load And-Split_alt_0,
      Fragment Load And-Split,
      Portfolio laden
      )

ADDAlternative(
       Load And-Split_alt_1,
       Load And-Split,
       Retail/WS-Daten laden
      )
```

Abbildung 37: Highlevel-Changes Hinzufügen eines neuen Fragment

8.1.4 MOVE

Moves entstehen dann, wenn sich die Position eines Objekte im Prozessmodell geändert hat. Die genaue Positionsnummer eines Objektes, wird bereits im Proces-Structure-Tree ermittelt. Um ein MOVE zu erkennen, werden die Positionsnummern des Objektes verglichen. Sollte die neue Positionsnummer ungleich der alten Positionsnummer sein, hat sich das Objekt verschoben.
Jedoch können nicht einfach die abgeleiteten Positionsnummern aus dem Process-Structure-Tree herangezogen werden, da durch vorhergehende Adds und Deletes die Positionsnummer im neuen Prozessmodell verzerrt wird. Demzufolge muss versucht werden die alte Position zu errechnen.

Rückrechnung der Objekte

Beginnend vom zu analysierenden Objekt im neuen Prozessmodell, werden die zuvor durchgeführten Change-Primitives in ihrer inversen Form auf das neue Prozessmodell „angewendet“. Ziel ist es, das zu analysierende Objekt an seine ursprüngliche Position im alten Prozessmodell zu verschieben. Daher müssen Adds als Deletes angesehen werden und Deletes als Adds. Die Reihenfolge beziehungsweise welche Change-Primitives vor dem Objekt angewandt wurden, kann direkt aus dem Editier-Skript entnommen werden. Dort ist die Reihenfolge der Änderungsoperationen aufgrund des Editier-Pfades abgelegt.
Jedes Add im Editier-Skript verringert dabei die Positionsnummer des zu analysierenden Objektes im neuen Modell und ein Delete im Editier-Skript erhöht die Positionsnummer. Diese Rückrechnung der zuvor durchgeführten Deletes und Adds bewirkt eine Konsolidierung der flachen Struktur des neuen Prozessmodells. So entsteht für das zu analysierende Objekt eine Positionsnummer, als ob keine Änderungen durchgeführt worden sind. Stimmt die konsolidierte Positionsnummer nicht mit der erwarteten Positionsnummer aus dem alten Prozessmodell überein, wurde das Objekt an eine andere Postion verschoben.

PM_1

PM_2

`Editier-Skript: delete(Retail/WS-Daten laden,2); add('Retail/WS-Daten laden,5);`

Change-Primitives

$$addCL_{prim} = \{Retail/WS-Daten\ laden\}$$

$$delCL_{prim} = \{Retail/WS-Daten\ laden\}$$

Ähnlichkeitsanalyse

$$\text{sim}(Retail/WS-Daten\ laden,\ Retail/WS-Daten\ laden) = 1$$

$$PObj_{add/del} = (Retail/WS-Daten\ laden,\ Retail/WS-Daten\ laden)$$

```
MOVE (Retail/WS-Daten laden,
      Gewichtungen anwenden,
      Report erzeugen
     )
```

Konsolidierung und Positionswerte

$$pos_{neu} = 5 \quad ; \quad pos_{alt} = 3 \quad ; \quad pos_{cons} = 6$$

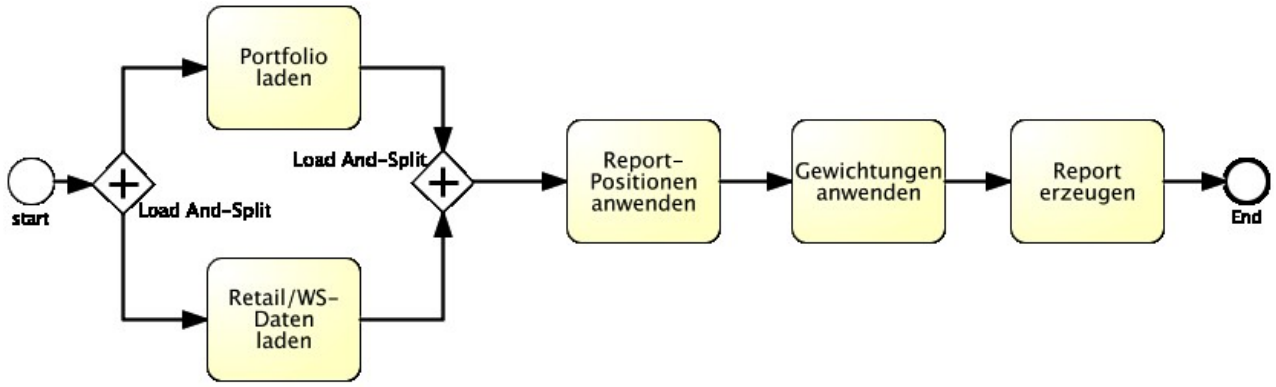
Abbildung 38: Highlevel-Change MOVE

Abbildung 38 zeigt die Rückrechnung eines Prozessobjektes `Retail/WS-Daten laden`. Zuerst wurde das Objekt in Set $Pobj_{add/de}$ aufgrund einer positiven Prüfung in der Ähnlichkeitsanalyse angelegt. Das Objekt wird somit auf eine mögliche Verschiebung geprüft. Im alten Prozessmodell lag das Objekt an $pos_{alt} = 3$ und im neuen Modell nun an $pos_{neu} = 7$. Zuerst werden alle Change-Primitives aus dem Editier-Skript, welche vor dem Objekt im neuen Prozessmodell durchgeführt worden sind, unter folgender Bedingung aufsummiert: Ein Add im Editiert-Skript fließt mit '-1' und ein Delete mit '+1' in den Summand ein. Im Beispiel befindet sich vor dem Objekt `Retail/WS-Daten laden`, welches auf der Position fünf ist, nur eine Operation im Editier-Skipt – nämlich das Delete. Daher ist der Konsolidierungswert $kw = +1$. Daraus ergibt sich die konsolidierte Positionsnummer $pos_{cons} = pos_{neu} + kw = 5 + 1 = 6$. Nun gilt $pos_{cons} \neq pos_{alt}$, was somit auf eine Verschiebung des Prozess-Objektes hindeutet.

Es hat sich herausgestellt, dass auch dann Moves entstehen können, wenn ein Prozessobjekt nicht in einem Change-Primitive-Set enthalten ist. Dies ist dann der Fall, wenn um bereits existierende Prozessobjekte ein neues Fragment (= Split/Join-Konten) angelegt wird. Um dennoch auch diese Verschiebungen zu erkennen, werden alle Objekte in der Menge $POBj_{eq}$ untersucht und eine Konsolidierung errechnet.

PM_1

PM_2



Change-Primitives

$$addCL_{prim} = \{ Load\ \text{And-Split},\ Load\ \text{And-Split\_alt\_0},\ Load\ \text{And-Split\_alt\_1},\ Load\ \text{And-Join} \}$$

$$delCL_{prim} = \emptyset$$

Ähnlichkeitsanalyse

nicht durchführbar, da $delCL_{prim}$ leere Menge

```
ADDFragment(Load And-Split,
            start,
            Report-Positionen anwenden
           )

ADDAlternative(
      Load And-Split_alt_0,
      Load And-Split,
      Portfolio laden
      )
```

Konsolidierung und Positionswerte

Portfolio laden

$$pos_{neu} = 4 \ ; \ pos_{alt} = 2 \ ; \ pos_{cons} = 4$$

Fortsetzung nächste Seite

Retail/WS-Daten laden

$pos_{neu}=6 \;;\; pos_{alt}=3 \;;\; pos_{cons}=6$

```
ADDAlternative(
        Load And-Split_alt_1,
        Load And-Split,
        Retail/WS-Daten laden
       )

MOVE(Portfolio laden,
      Load And-Split_alt_0,
      Load And-Join
     )

MOVE(Retail/WS-Daten laden,
      Load And-Split_alt_1,
      Load And-Join
     )
```

Abbildung 39: Highlevel-Change MOVE ohne Change-Primitives

In Abbildung 39 ist zu sehen, dass um die beiden Prozessobjekte `Portfolio laden` und `Retail/WS-Daten laden` ein neues Fragment erstellt worden ist. Im Editier-Pfad werden die beiden Objekte mit einer diagonalen Kante markiert und somit sind diese nicht in den Change-Primitives vorhanden. Lediglich die neuen Objekte `Load AND-Split`, `Load AND-Split_alt_0`, `Load AND-Split_alt_1` und `Load AND-Join` werden in $addCL_{prim}$ aufgenommen.
Die beiden Objekte `Portfolio laden` und `Retail/WS-Daten laden` sind jedoch im Vergleich zum alten Prozessmodell (so auch im Process-Structure-Tree) in einem anderen Fragment ausgewiesen und Objekte, welche in einem anderen Fragment als zuvor sind, wurden verschoben. [KGFE08].

Dennoch würde aber eine Rückrechnung noch immer nicht zum gewünschten Ergebnis führen, da durch die Rückrechnung wird ein Konsolidierungswert $kw=-2$ für `Portfolio laden` und $kw=-4$ für `Retail/WS-Daten laden` errechnet wurde. Zieht man die beiden kw -Werte von den jeweiligen pos_{neu} -Werten der Objekte ab, erhält man die ursprüngliche Position im alten Prozessmodell und folglich kann auch kein MOVE abgeleitet werden.

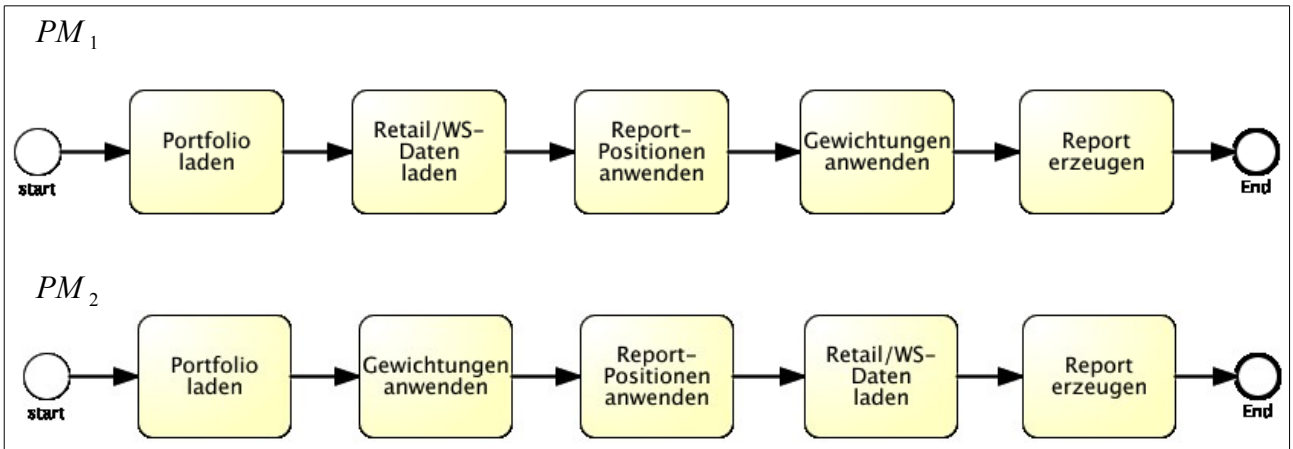
Um dennoch ein MOVE ableiten zu können, werden Splits und Alternativen, sofern diese

hinzugefügt wurden und gleichzeitig der direkte Parent- und Grandparent-Knoten sind, in der Konsolidierung exkludiert. Dadurch wird erreicht, dass die Hierarchie berücksichtigt wird und der Move in ein anderes Fragment ausgewiesen wird: Der kw -Wert für `Portfolio laden` und `Retail/WS-Daten laden` ist daher für beide Objekte null. Im Zuge dessen sind dann auch die beiden pos_{cons} ungleich der pos_{alt} -Werte. Das Highlevel-Change MOVE kann damit für Objekte im Set $PObj_{eq}$ ebenfalls bestimmt werden.

Die verschobenen Elemente werden in das HLC_{move} aufgenommen. Dieses Set kann anschließend für die Erkennung von Swaps herangezogen werden.

8.1.5 SWAP

SWAPs werden aufbauend auf den bereits generierten HLC_{move} erzeugt. Dabei werden zwei Objekte $a, b \in HLC_{move}$ dahingehend geprüft, ob die Objekte a und b im alten Prozessmodell auf der jeweils anderen Position, unter der Berücksichtigung der Rückrechnung im neuen Prozessmodell, abgelegt wurden. Hat also das Prozessobjekt a im neuen Prozessmodell konsolidiert die gleiche Postion wie b im alten und gilt dies auch umgekehrt für b , dann ist ein SWAP ableitbar.




```
Editier-Skript: add(Gewichtung anwenden, 3); delete(Retail/WS-Daten laden);
add('Retail/WS-Daten laden',5); delete(Gewichtung anwenden,5);
```

Fortsetzung nächste Seite

Change-Primitives

$addCL_{prim} = (Gewichtung\ anwenden,\ Retail/WS-Daten\ laden)$

$delCL_{prim} = (Gewichtung\ anwenden,\ Retail/WS-Daten\ laden)$

Ähnlichkeitsanalyse

$\mathrm{sim}(Gewichtung\ anwenden,\ Gewichtung\ anwenden) = 1$

$\mathrm{sim}(Retail/WS-Daten\ laden,\ Retail/WS-Daten\ laden) = 1$

$POb j_{add/del} = (Retail/WS-Daten\ laden,\ Retail/WS-Daten\ laden),\ (Gewichtungen\ anwenden,\ Gewichtungen\ anwenden)$

Konsolidierung und Positionswerte

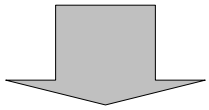
Gewichtung anwenden

$pos_{neu} = 3 \quad ; \quad pos_{alt} = 5 \quad ; \quad pos_{cons} = 3$

Retail/WS-Daten laden

$pos_{neu} = 5 \quad ; \quad pos_{alt} = 3 \quad ; \quad pos_{cons} = 5$

```
MOVE(Gewichtungen anwenden,
     Portfolio laden,
     Report-Positionen anwenden
    )

MOVE(Retail/WS-Daten laden,
     Report-Postionen anwenden,
     Report erzeugen
    )
```



```
SWAP(Gewichtung anwenden,
     Retail/WS-Daten laden)
```

Abbildung 40: Highlevel-Change SWAP

Das Beispiel in Abbildung 40 zeigt, ausgehend von HLC_{move}, wie ein SWAP abgeleitet werden kann. Die pos_{cons} für die beiden als MOVE erkannten Objekte wurde bereits bei der MOVE-Erkennung für `Gewichtung anwenden` mit $pos_{cons} = 3$ und für `Retail/WS-Daten laden` mit $pos_{cons} = 5$ errechnet. Im alten Prozessmodell hatte das Objekt `Gewichtung anwenden` $pos_{alt} = 5$ und das Objekt `Retail/WS-Daten laden` hatte $pos_{alt} = 3$. Um den SWAP zu erkennen, muss die folgende Bedingung positiv validiert werden:

$$Gewichtung\ anwenden.pos_{old} = Retail/WS-Daten\ laden.pos_{cons} \wedge Retail/WS-Daten\ laden.old_{pos} = Gewichtung\ anwenden.pos_{cons}$$

Eine Auswertung des Ausdrucks fällt in diesem Beispiel positiv aus, wodurch der SWAP für diese zwei Prozessobjekte erstellt werden kann, obwohl diese zuerst als MOVE deklariert wurden. Werden SWAPs erkannt, werden diese aus der Menge HLC_{move} entfernt und in HLC_{swap} aufgenommen.

8.1.6 REPLACE

Um REPLACES zu erkennen werden die Objekte aus HLC_{del} und HLC_{add} analysiert. Aus der Menge HLC_{del} wird für die Objekte die Positionsnummer pos_{alt} verwendet. Aus den HLC_{add} werden die konsolidierten Positionsnummern pos_{cons} verwendet, welche der gleichen Rückrechnungslogik wie zur Errechnung von HLC_{Move} unterstehen. Hat ein Objekt aus HLC_{add} die gleiche konsolidierte Position wie ein Objekt in HLC_{del} , dann wurde das Objekt in HLC_{del} durch jenes in HLC_{add} ersetzt.

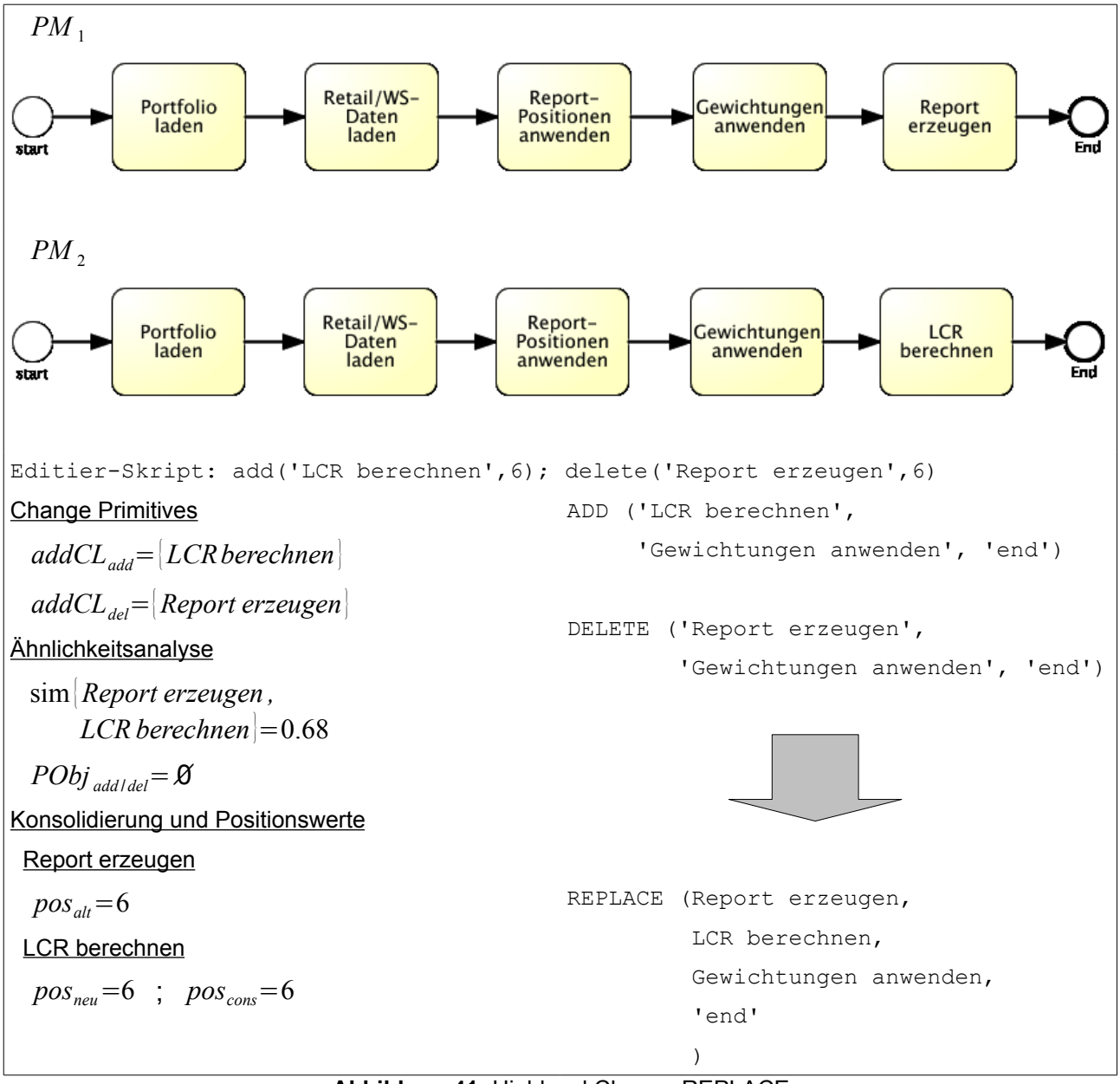


Abbildung 41: Highlevel Change REPLACE

Die Abbildung 41 zeigt die Transformation eines DELETE und ADD in ein REPLACE. Da

das Objekt `LCR berechnen` einen kw -Wert von $kw=0$ hat, bleibt das Objekt an der ursprünglich eingefügten Postion im neuen Prozessmodell, gleichzeitig wurde aber auf der Position sechs im alten Prozessmodell das Objekt `Report erzeugen` entfernt. Wird also der Ausdruck

$$LCR\,berechen.pos_{cons} = Report\,erzeugen.pos_{alt}$$

positiv validiert, kann zwischen einem ADD und DELETE ein REPLACE abgeleitet werden.

8.1.7 COPY

Der Highlevel-Change COPY kann dann erstellt werden, wenn Prozessobjekte in der Menge HLC_{add} existieren und diese auch im alten Prozessmodell vorhanden waren. Hierzu werden die HLC_{add} Objekte mit all jenen Objekten verglichen, welche im Editier-Pfad mit einer Diagonale markiert worden sind. Anders ausgedrückt: $HLC_{copy} := HLC_{add} \cap PObj_{eq}$. Die Prüfung erfolgt dabei über den Namen der Prozess-objekte.

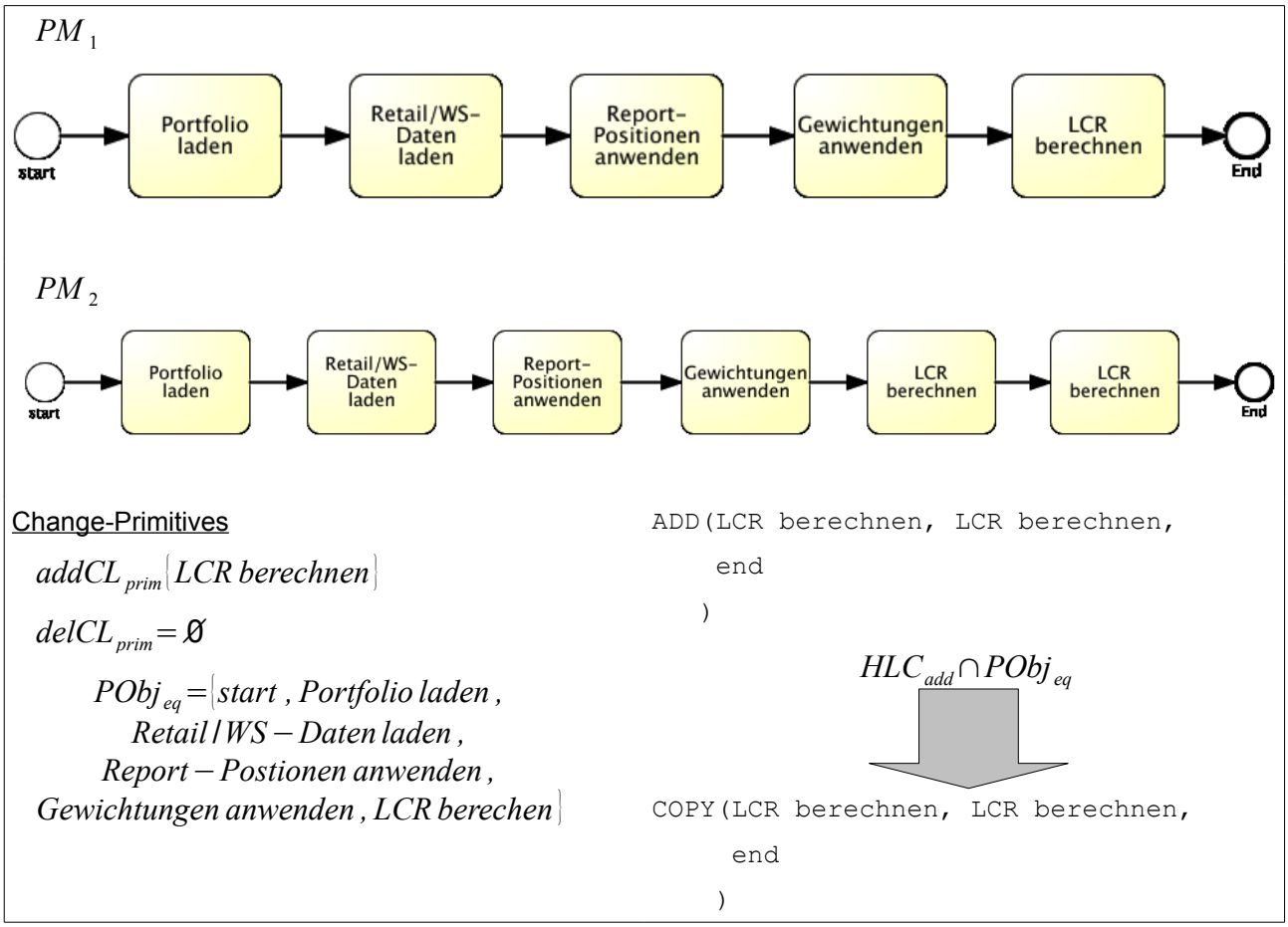


Abbildung 42: Highlevel-Change COPY

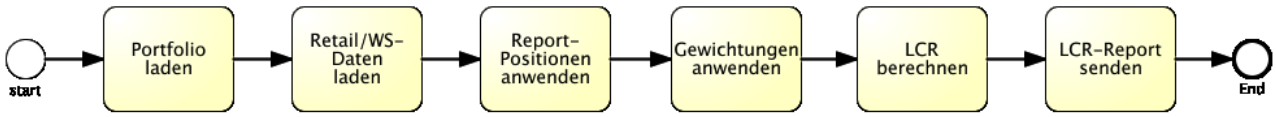
Im Beispiel aus Abbildung 42 wurden, wie schon beschrieben, die Menge der HLC_{add} Objekte abgeleitet. Ein Vergleich der Objekte aus HLC_{add} mit jenen aus $PObj_{eq}$ ergibt eine nicht leere Menge mit dem Prozessobjekt `LCR berechnen`. Demzufolge wurde das Prozessobjekt $LCR\,berechnen \in HLC_{add}$ zwischen $LCR\,berechnen \in POBj_{eq}$ und $end \in PObj_{eq}$ generiert. Nachdem der Highlevel-Change COPY erstellt worden ist, wird das Prozessobjekt `LCR berechnen` aus der Menge HLC_{add} entfernt.

8.2 Bilden von Sequenzen

Existieren Anreihungen von gleichen Highlevel-Changes, kann dies zu langen und unübersichtlichen Change Operationen führen, insbesondere dann wenn es zu Abhängigkeiten in den `predecessor` und `successor` Parametern in Highlevel-Changes kommt [RSRD04]. Weist das Nachfolgerobjekt eines Highlevel-Change die gleiche Operation auf, werden diese zu einer Highlevel-Change-Sequenz zusammengefasst. Somit entsteht eine kürzere Operation, welche über alle angegebenen Prozessobjekte diese Aktion durchführt. Ein weiterer Vorteil beim Zusammenfassen gleichartiger Highlevel-Changes ist es, dass nur noch der Vorgänger des ersten Objektes und der Nachfolger des letzten Objektes in der Operation angegeben werden müssen. Dies ermöglicht eine weitere Komprimierung des Highlevel-Changes und die Korrektheit der Ausführung, da sich die Vorgänger-Nachfolger-Beziehung bereits durch die Anreihung der Objekte innerhalb der Sequenz ergibt.

PM_1

PM_2




```
Editier-Skript:add(LCR berechnen,6); add(LCR-Report,7);
            delete(Report erzeugen, 6);
```

Change-Primitives

$$addCL_{prim} = \{LCR\,berechnen,\ LCR\,Report\,senden\}$$

$$delCL_{prim} = \{Report\,erzeugen\}$$

```
ADD ('LCR berechnen',
     'Gewichtungen anwenden',
     'LCR-Report senden')

ADD ('LCR-Report senden',
     'LCR berechnen', 'end')

DELETE ('Report erzeugen',
        'Gewichtungen anwenden', 'end')
```

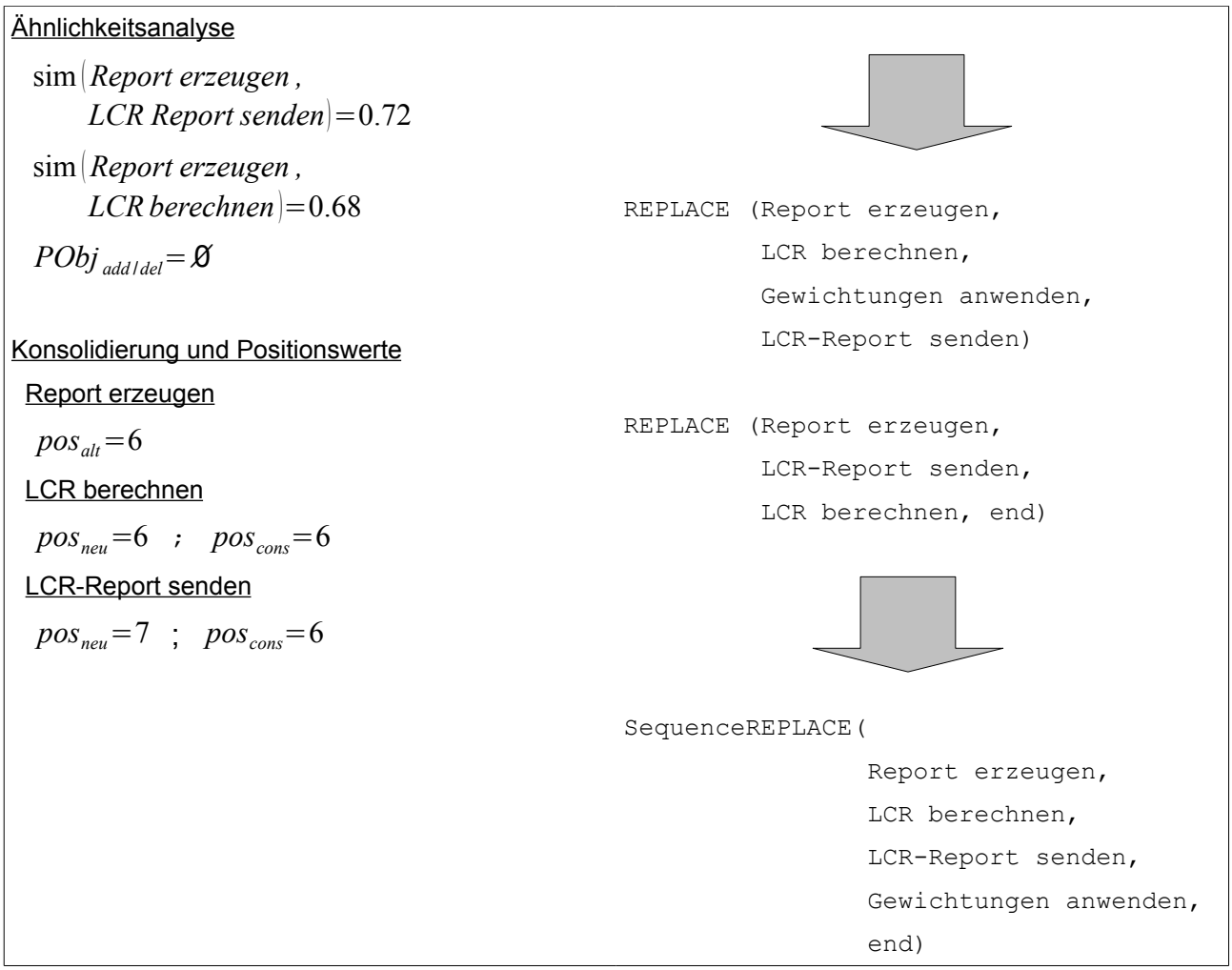


Abbildung 43: Highlevel-Change Sequenzen

Wie Abbildung 43 zeigt, wurden aus den Change-Primitives zwei REPLACE-Operationen abgeleitet. Betrachtet man nun die beiden Highlevel-Changes, hat das Objekt `LCR berechnen` als Nachfolger (Angabe im Parameter `successor`) das Prozessobjekt `LCR-Report senden`. Da das Objekt `LCR-Report senden` die gleiche Operation wie `LCR berechnen` ausweist und dementsprechend auch der Vorgänger von `LCR-Report senden` `LCR berechnen` ist, können die beiden Objekte in einer Sequenz zusammengefasst werden. Daraus entsteht anschließend der Highlevel-Change `SequenceREPLACE`. Dabei ist der erste Parameter das zu ersetzende Objekt, hier `Report erzeugen`. Die nächsten zwei sind jene Objekte die `Report erzeugen` auch tatsächlich ersetzen, gefolgt von den beiden Objekten zwischen welchen der Replace statt gefunden hat. Anzumerken ist, dass sich Sequenzen auf alle Highlevel-Changes abbilden lassen mit Ausnahme von Updates.

8.3 Abhängigkeiten und Reihenfolge von Highlevel-Changes

Zwischen den einzelnen Highlevel-Changes existieren Abhängigkeiten. Erst wenn die Errechnung von bestimmten „Vorgänger“-Highlevel-Changes durchgeführt wurde, können weitere erstellt werden. Die nachstehende Grafik zeigt diese Abhängigkeiten.

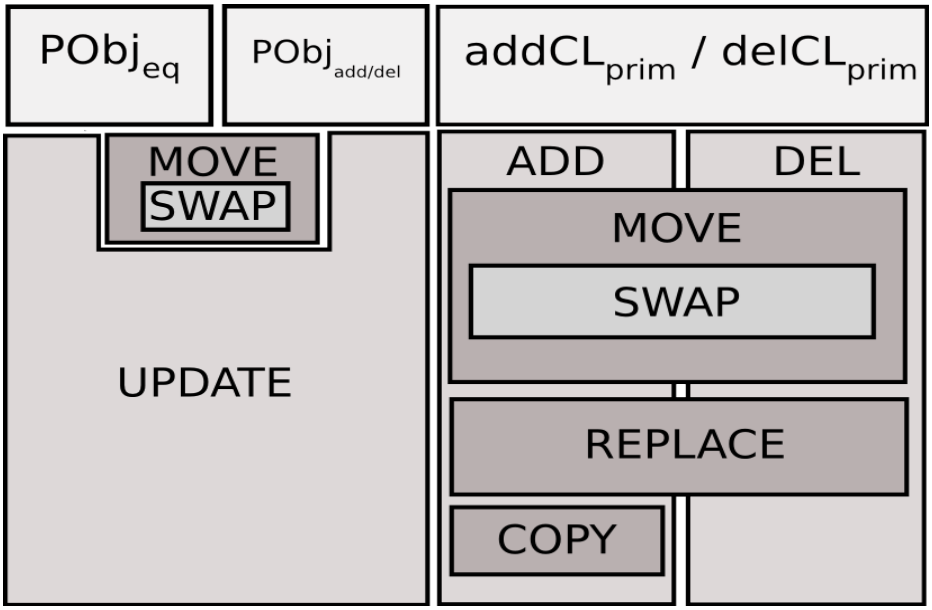


Abbildung 44: Highlevel-Changes-Abhängigkeiten

Aus Abbildung 44 geht hervor, dass das Set der $POBj_{eq}$, also jenes Set das im Editier-Graphen mit Diagonalen gekennzeichnet worden ist, lediglich in den Highlevel-Changes MOVE/SWAPs oder UPDATE aufgehen kann. Anders die $addCL_{prim}$ und $delCL_{prim}$. Je nachdem, ob diese durch die Ähnlichkeitsanalyse als starke Ähnlichkeit erkannt werden, können diese über die Highlevel-Changes ADD und DELETE hinausgehen. Dabei kann eine Kombination aus ADD- und DEL-Positionen durch die Rückrechnung der Positionsnummern in ein REPLACE oder MOVE umgewandelt werden. Bei einem MOVE besteht noch zusätzlich die Möglichkeit ein SWAP zu erstellen, wenn durch zwei MOVES die beiden Prozessobjekte ausgetauscht werden. Ein COPY entsteht grundsätzlich nur durch ADDs, welche auch in den $Pobj_{eq}$ enthalten sind. Mittels Ähnlichkeitsanalyse entstehen add-delete-Paare, welche Teil von $Pobj_{add/del}$ sind. Diese Paare können entweder in einfache Updates oder MOVES und in weiterer Folge auch in SWAPs transformiert werden.

9 Persistierung von Highlevel-Changes

Um die Higlevel-Changes zu einem späteren Zeitpunkt anwenden zu können, insbesondere dann, wenn nur noch eines der beiden verglichenen Prozessmodelle vorhanden ist oder um über mehrere Change-Logs ein früheres Prozessmodell wiederherstellen zu können, werden die Informationen über die Differenzen gespeichert. In der Persistierung werden Informationen über die Änderungen, so wie Zusatzinformationen welche in der Differenzerkennung erzeugt worden sind, mittels eines selbst erzeugten XML-Dokument, gespeichert. Dabei werden die Informationen abgelegt, die man benötigt um einerseits die zuvor präsentierte Form der Highlevel-Changes wieder herstellen zu können, und andererseits auch Zusatzinformationen die für einer Wiederherstellung von Prozessmodellen notwendig sind. Die Summe aller persistierten Highlevel-Changes einer Differenzenerkennung zwischen zwei Prozessmodellen ergibt somit das Change-Log der beiden Prozessmodelle.

Im folgenden Abschnitt werden zu den einzelnen Highlevel-Changes Beschreibungen zur persistenten XML-Struktur gegeben. Des weiteren werden Beispiele zur Verdeutlichung angeführt. Jedes Highlevel-Change wird mit XML-Tags beschrieben. Ein XML-Tag enthält dann weitere Attribute und Kind-Elemente, welche die Informationen über den Change beinhalten.
Aus Platzgründen wird das XML-Schema des Change-Logs mit den einzelnen Highlevel-Changes im Anhang angeführt.

9.1.1 UPDATE

Die Persistierung eines UPDATE fasst eine Gruppe von Update-Operationen über ein Prozessobjekt zusammen. Die unterschiedlichen Update-Operationen (CHANGE, ADD, DELETE) werden in einem XML-Tag mit der Bezeichnung <UPDATE> zusammengefasst. Für jede Update-Operation wird auf das Prozessobjekt ein neues Kind-Element in der Gruppierung erstellt.

```
UPDATE „Gewichtungen bestimmen“
 CHANGE (name,'Gewichtung anwenden',
              'Gewichtung bestimmen')
 DELETE (maxDuration)
```

```
<UPDATE id="12" pos="4" name="Gewichtungen
bestimmen">
  <changeupdate param="name" type ="value">
    <updateto>Gewichtungen
              anwenden</updateto>
    <updatefrom>Gewichtungen
               bestimmen</updatefrom>
  </changeupdate>
  <deleteupdate>
    <param>
      <name>maxDuration</name>
      <valueType>float</valueType>
      <value>3</value>
    </param>
  </deleteupdate>
</UPDATE>
```

Listing 1: Persistente Form des Highlevel-Change UPDATE

In Listing 1 ist ein Update auf das Attribute `name` und die Löschung des Objekt-Parameters `maxDuration` vom Prozessobjekt `Gewichtungen bestimmen` zu sehen. Bei einem `CHANGE`-Update wird im XML-Attribut `param` der geänderte Parametername angegeben und im `type` der Bestandteil des Parameters, dies kann der Wert (`value`) sein oder der Parameter-Type (`type`). Im Beispiel wird der Wert des Namens geändert. Des weiteren wird im XML der ursprüngliche Wert (`updatefrom`) und der neue Wert (`updateto`) des Objekt-Parameters `name` angeführt. Eine Angabe des alten Parameterwert ist notwendig, um bei einer Rückrechnung auf das Vorgängermodell den geänderte Wert (`updatefrom`) wieder zu rekonstruieren.
In diesem Beispiel wurde noch der Parameter `maxDuration` gelöscht. Bei einer Löschung wird der komplette Parameter in der XML-Struktur abgelegt, um die Rückrechnung zu gewähren.

9.1.2 ADD und DELETES

Bei einer Speicherung eines ADD- oder DELETE-Highlevel-Changes wird immer das komplette Prozessobjekt abgelegt. Dies ist vorallem bei DELETEs notwendig um die Rückrechnung in das Vorgängermodell zu ermöglichen. Des weiteren werden das Vorgänger-Objekt (`predecessor`) und das Nachfolger-Objekt (`successor`) aus dem Process-Structure-Tree im XML Tag abgelegt.

In den beiden Tags sind noch zwei Attribute enthalten: `pos` und `mmodeltype`. Im XML-Attribute `pos` wird die Positionsnummer des Objektes aus dem Prozessmodell gespeichert. Die Positionsnummer entspricht dabei der absoluten Postion welche im Process-Structure-Tree abgeleitet wurde. Während `pos` im ADD-Tag die Position im neuen Prozessmodell widerspiegelt, ist die Positionsnummer im DELETE-Tags die Position des Prozessobjektes im Vorgängermodell.
Das Attribute `mmodeltype` speichert den Objekt-Type des Prozessobjektes aus dem Meta-Modell ab.

```
ADD ('LCR berechnen',
     'Gewichtungen anwenden',
     'end')

DELETE ('Report erzeugen',
        'Gewichtungen anwenden',
        'end')
```

```
<ADD id="1" pos="6" mmodeltype="activity">
  <name>LCR berechnen</name>
  <type>task</type>
  <predecessor>Gewichtung anwenden</predecessor>
  <accessor>end</accessor>
</ADD>

<DELETE id="2" pos="6" mmodeltype="activity">
      <name>Report erzeugen</name>
      <type>task</type>
      <predecessor>Gewichtungen
                   anwenden</predecessor>
      <accessor>end</accessor>
</DELETE>
```

Listing 2: Persistente Form der Highlevel-Changes ADD und DELETE

In diesem Listing wird ein neues Objekt `LCR berechnen` hinzugefügt. Das hinzugefügte Objekt ist vom Meta-Model-Type `activity` und wurde an Postion sechs im neuen Prozessmodell gesetzt. In der speziellen Modellierungsnotation ist das Objekt vom Typ (`<type>`) `task` und der Vorgänger ist `Gewichtung anwenden`, der Nachfolger `end`.
Bei der Löschung wurde das Objekt `Report erzeugen` aus dem Modell entfernt. Dieses hat sich im alten Modell an der Postion sechs befunden und lag zwischen `Gewichtung anwenden` und `end`.

9.1.3 Fragmente

Fragmente verhalten sich in der Persistierung ähnlich wie die ADDs und DELETEs vom Meta-Modell-Type `activity`. Der Unterschied liegt jedoch darin, dass bei einem

ADDFragment sowohl ein Split-Knoten als auch ein Join-Knoten explizit erstellt werden. Bei Alternativen und Loops werden noch zusätzlich Attribute zur Speicherung Konditionen (<cterm>) der Alternativ-Kanten und speziell für Loops der Test der Abbruchbedienung (<ctest>) angegeben.

```
ADDFragment(
      Load And-Split,
      start,
      Report-Positionen anwenden
            )

ADDAlternative(
      Load And-Split_alt_0,
      Fragment Load And-Split,
      Portfolio laden
      )

ADDAlternative(
       Load And-Split_alt_1,
       Load And-Split,
       Retail/WS-Daten laden
      )
```

```
<ADD  id="1" pos="2" mmodeltype="split">
  <name>Load And-Split</name>
  <predecessor>start</predecessor>
  <accessor>Report-Positionen anwenden</accessor>
  <type>parallelgateway</type>
</ADD>
<ADD  id="2" pos="7" mmodeltype="join">
  <name>Load And-Join</name>
  <predecessor>start</predecessor>
  <accessor>Report-Positionen anwenden</accessor>
  <type>parallelgateway</type>
</ADD>
<ADD id="3" pos="3" mmodeltype="alternative">
  <name>Load And-Split_alt_0</name>
  <predecessor>Load And-Split</predecessor>
  <accessor>Portfolio laden</accessor>
  <type></type>
  <cterm></cterm>
</ADD>
<ADD id="4" pos="5" mmodeltype="alternative">
  <name>Load And-Split_alt_1</name>
  <predecessor>Load And-Split</predecessor>
  <accessor>Retail/WS-Daten laden</accessor>
  <type></type>
  <cterm></cterm>
</ADD>
```

Listing 3: Persistente Form der Highlevel-Changes ADDFragment mit Alternativ-Kanten

Das Listing zeigt das Hinzufügen einer Parallelisierung aus dem Beispiel im Abschnitt „Von Change-Primitives zu Highlevel-Changes“ Der Highlevel-Change ADDFragment wird dabei in zwei ADD-Tags aufgeteilt. Im Parameter `mmodeltype` ist angegeben um welchen Typ es sich innerhalb des Meta-Modells handelt.

Die Alternativen werden ebenfalls mit einem ADD-Tag abgelegt, allerdings mit `mmodelttype= 'alternative'`. Zu welchem Split diese Alternative gehört, wird über den `accessor`- und `predecessor`-Tag angegeben. Im Tag `<cterm>` befindet sich die

Bedingung der Alternative. Da es sich um eine Parallelisierung handelt, ist dieser Tag leer. Es hat sich herausgestellt, dass es Prozessmodelle gibt (u.a. CPEE-Cockpit), welche den Fragment-Knoten wie Parallelisierungen und Konditionen, keine Namen geben. Dies ist dann problematisch, wenn mehr als ein Parallel-Split im Modell vorhanden ist. Daher wird wie auch schon bei `activity`-ADDs die Positionsnummer angegeben. Über diese Positionsnummer kann dann die Alternative wieder zum richtigen Split Knoten zugerechnet werden.

9.1.4 MOVE

Im Highlevel-Change MOVE werden in der Persistierung zwei Positionen abgelegt: Die Positionsnummer inklusive Vorgänger und Nachfolger sowohl im neuen Prozessmodell als auch im alten. Durch die Ablage der alten und neuen Positionen im MOVE-Tag kann das Objekt wieder auf die ursprüngliche Position im Vorgängermodell verschoben werden.

```
MOVE (Retail/WS-Daten laden,
      Gewichtungen anwenden,
      Report erzeugen
     )
```

```
<MOVE id="5">
  <name>Retail/WS-Daten laden</name>
  <to>
    <pos>5</pos>
    <predecessor>Gewichtungen anwenden</predecessor>
    <accessor>Report erzeugen</accessor>
  </to>
  <from>
    <pos>3</pos>
    <predecessor>Portfolio laden</predecessor>
    <accessor>Report-Positionen anwenden</accessor>
  </from>
</MOVE>
```

Listing 4: Persistente Form der Highlevel-Changes MOVE

Der Move-Tag unterteilt sich dabei in eine `<to>`-Sektion und eine `<from>`-Sektion. Die `<to>`-Sektion gibt dabei die Position im neuen Modell an. In Listing 4 wurde dementsprechend das Prozessobjekt `Retail/WS-Daten laden` auf die Position fünf verschoben. Nun liegt das Objekt zwischen `Gewichtungen anwenden` und `Report erzeugen`. In der `<from>`-Sektion sind die Positionsinformationen über die ursprüngliche Position des Objektes abgelegt. Im Beispiel lag das Objekt ursprünglich an der Position drei.

9.1.5 SWAP

Der SWAP-Tag ist grundsätzlich eine Verbindung aus zwei MOVE-Tags. Wobei die beiden getauschten Elemente nicht nur mit MOVE-Tags dargestellt werden, sondern in einem einzigen Tag zusammengefasst sind. Jedes Objekt, das an einem SWAP beteiligt ist, wird in einem `<obj>`-Tag abgelegt. Die innere Struktur des `<obj>`-Tags folgt dann dem selben Aufbau wie ein MOVE-Tag.

```
SWAP(
    Gewichtung anwenden,
    Retail/WS-Daten laden)
```

```
<SWAP id="6">
  <obj>
    <name>Gewichtung anwenden</name>
    <to>
      <pos>3</pos>
      <predecessor>Portfolio laden</predecessor>
      <accessor>Report-Position anwenden</accessor>
    </to>
    <from>
      <pos>5</pos>
       <predecessor>Report-Position anwenden</predecessor>
        <accessor>Report erzeugen</accessor>
    </from>
  </obj>
  <obj>
    <name>Retail/WS-Daten laden</name>
    <to>
      <pos>5</pos>
      <predecessor>Report-Position anwenden</predecessor>
      <accessor>Report erzeugen</accessor>
    </to>
    <from>
      <pos>3</pos>
      <predecessor>Portfolio laden</predecessor>
      <accessor>Report-Position anwenden</accessor>
    </from>
  </obj>
</SWAP>
```

Listing 5: Persistente Form der Highlevel-Changes SWAP

Das Listing 5 zeigt einen SWAP zwischen den beiden Prozessobjekten `Gewichtung anwenden` und `Retail/WS-Daten laden`. Ersichtlich ist, dass in den `<obj>`-Tags die MOVE-Struktur erkennbar ist, welche sämtliche Positionsinformationen enthält. Im Beispiel

wird das Prozessobjekt `Gewichtung anwenden` auf die dritte Position (`pos`) verschoben. Somit liegt das Objekt nun zwischen `Portfolio laden` und `Report-Postion anwenden`. Diese Information ist aus dem ersten `<to>`-Tag zu erkennen. Das Objekt `Retail/WS-Daten landen` wird auf die Position fünf „getauscht“. Dies ist aus dem zweiten `<to>`-Tag zu entnehmen. Die beiden Ursprungspositionen im alten Prozessmodell sind entsprechend in den `<from>`-Tags zu erkennen. Der SWAP lässt sich auch aus den Positionsnummern der jeweiligen Objekte ableiten.

9.1.6 REPLACE

In einem REPLACE befindet sich die Logik von einem ADD und einem DELETE, welche in einem REPLACE-Tag abgebildet werden. Dabei gibt es für das Add-Objekt ein Kind-Element als auch für das Delete-Objekt.

```
REPLACE (
       Report erzeugen,
       LCR berechnen,
       Gewichtungen anwenden,
       'end'
          )
```

```
<REPLACE id="7">
  <addObj pos="6" mmodeltype="activity">
    <name>LCR berechnen</name>
    <type>task</type>
    <predecessor>Gewichtungen anwenden</predecessor>
    <accessor>end</accessor>
  </addObj>
  <delObj pos="6" mmodeltype="activity">
    <name>Report erzeugen</name>
    <type>task</type>
    <predecessor>Gewichtungen anwenden</predecessor>
    <accessor>end</accessor>
  </delObj>
</REPLACE>
```

Listing 6: Persistente Form der Highlevel-Changes REPLACE

Das Beispiel in Listing 6 zeigt eine Umwandlung eines REPALCE Highlevel-Changes in eine persistente Form. Im Beispiel wird das Prozessobjekt `Report erzeugen` als das Delete-Objekt (`<delObj>`) deklariert, während `LCR berechnen` im ADD-Objekt (`<addObj>`) abgelegt ist. Wie auch schon bei einer einfachen ADD- und DELETE-Persistierung werden wieder alle Informationen über die Objekte angelegt.

9.1.7 COPY

Eine Persistierung des Highlevel-Change COPY wird ähnlich behandelt wie die ADD-Persistierung. Der Unterschied zum ADD-Tag ist jedoch, dass bei einem COPY zusätzlich die `source`, also die Quelle des Objektes angegeben ist. Ohne Anlage einer eigenen ADD-Sektion kann es bei einer späteren Anwendung des COPY Highlevel-Change zu fehlerhaften Transformierungen kommen: Wird ein Objekt zuerst kopiert und unterliegt dann dem Quell-Objekt auch noch ein Update auf einem Parameter, kann dieser Sachverhalt nicht mehr richtig eingespielt werden, da auch für das Ziel-Objekt der geänderte Parameter aus dem Quell-Objekt angewendet werden würde, was im Ziel-Objekt eventuell nicht gewünscht ist, da der Parameter im Ziel-Objekt nicht geändert wurde. Wird das Ziel-Objekt im COPY-Tag in einer eigenen ADD-Sektion abgebildet, wird das Objekt so persistiert, wie es auch im Prozessmodell hinterlegt beziehungsweise kopiert worden ist. Das kopierte Objekt ist somit völlig unabhängig von nachträglichen Updates auf dem Quell-Objekt.

```
COPY(LCR berechnen,
     LCR berechnen,
     end
    )
```

```
<COPY id="4" pos="7"  mmodeltype="activity">
  <source>LCR berechnen</source>
  <addObj>
    <name>LCR berechnen</name>
    <type>task</type>
    <predecessor>LCR berechnen</predecessor>
    <accessor>end</accessor>
  </addAbj>
</COPY>
```

Listing 7: Persistente Form der Highlevel-Changes COPY

In Listing 7wird die Persistierung des Highlevel-Changes COPY dargestellt. Das Objekt `LCR berechnen` (`<addObj>`) ist eine Kopie des gleichnamigen Objektes `LCR berechnen` und wurde auf der Position sieben zwischen LCR berechnen und `end` eingefügt.

9.1.8 Sequenzen

Sequenzen bilden eine Anreihung von gleichartigen und aufeinanderfolgenden Highlevel-Operationen ab. Im Grunde können alle Highlevel-Changes auch in Sequenzen vorkommen. Ausnahme ist jedoch das UPDATE. Wird ein Highlevel-Change mit einer

Sequenz erstellt, werden die entsprechenden Highlevel-Change-Tags in dieser Sequenz gruppiert und in einem <SEQUENCE>-Tag zusammengefasst. Anders als bei einen einzelnen Highlevel-Change-Tag wird nicht für jede Change-Operation eine id vergeben sondern lediglich für die Sequenz. Des weiteren wird auch noch der Sequence-Type, sprich die Änderungsoperation, als Attribut mitgespeichert.

```
SequenceREPLACE(
      Report erzeugen,
      LCR berechnen,
      LCR-Report senden,
      Gewichtungen anwenden,
      end
      )
```

```
<SEQUENCE id="9" type="REPLACE">
  <REPLACE>
    <addObj pos="6" mmodeltype="activity">
      <name>LCR berechnen</name>
      <type>task</type>
      <predecessor>Gewichtungen anwenden</predecessor>
      <accessor>end</accessor>
    </addObj>
    <delObj pos="6" mmodeltype="activity">
      <name>Report erzeugen</name>
      <type>task</type>
      <predecessor>Gewichtungen anwenden</predecessor>
      <accessor>end</accessor>
    </delObj>
  </REPLACE>
  <REPLACE>
    <addObj pos="6" mmodeltype="activity">
      <name>LCR Bericht senden</name>
      <type>task</type>
      <predecessor>Gewichtungen anwenden</predecessor>
      <accessor>end</accessor>
    </addObj>
    <delObj pos="6" mmodeltype="activity">
      <name>Report erzeugen</name>
      <type>task</type>
      <predecessor>Gewichtungen anwenden</predecessor>
      <accessor>end</accessor>
    </delObj>
  </REPLACE>
</SEQUENCE>
```

Listing 8: Persistente Form einer Sequenz

Das Listing 8 zeigt ein Sequence-Beispiel, welches eine REPLACE-Sequenz persistiert. Da diese Replace-Sequenz zwei Replaces beinhaltet, wurden demzufolge auch zwei REPLACE-Tags innerhalb der Sequenz erstellt. Das zu ersetzende Objekt wird in den

<delObj>-Tags angegeben und die neuen Objekte wieder in den <abbObj>-Tags. Im Beispiel ersetzen die beide Objekte LCR berechnen und LCR-Report senden das Objekt Report erzeugen.

10 Versionskontrolle

Da durch Prozessmodell-Anpassungen unterschiedliche Versionen eines Prozessmodells generiert werden, entsteht zwangsläufig die Notwendigkeit diese qualitätszusichern und auditierbar zu machen. Des weiteren müssen Änderungsanwendungen im Sinne von Kosteneffizienz und Transparenz durchführbar sein [REDA00]. Demzufolge verlangt die Evolution von Prozessmodellen nach einer Management-Funktion wie sie in der Software-Entwicklung beschrieben wird [CRWB98]. Änderungen im Geschäftsprozessmodell sollten daher in einer Versionskontrolle abgelegt werden [ZXLC07][JGHO98].

Des weiteren dient die Versionskontrolle allgemein als eine Entwicklungs-Funktion für Prozessentwickler [CRWB98]. Mittels einer Versionskontrolle ist es möglich Aufzeichnungen von Änderungen über mehrere Geschäftsprozess-Versionen abzulegen, unterschiedliche Geschäftsprozesse zu verwalten und zu speichern, Vorgänger-Geschäftsprozessen zu rekonstruieren aber auch das Ableiten (Mergen) von Geschäftsprozessen aus unterschiedlichen Versionen zu einer neuen Version [BCRS13].

Ein weiterer Grund für eine Versionskontrolle im Kontext des Prozess-Engineering ist das Aufzeichnen des Zeitpunkts der Änderungen. So kann durchaus die Situation auftreten, dass mehrere Prozessinstanzen mit unterschiedlichen Prozessmodellversionen gleichzeitig ausgeführt werden. Es ist nicht immer möglich alle langlaufenden Instanzen abzubrechen und die neue Version auszurollen. Die Migration von laufenden Instanzen ist, wenn überhaupt, nur unter bestimmten Bedingungen beziehungsweise Zeitpunkten möglich. [GRRA06] verwendet für die Adaptierung von aktiven Instanzen die Change-Informationen zwischen den zwei Prozessmodellen. Werden laufende Instanzen migriert, ist es wichtig, dass die Korrektheit der migrierten Instanz weiterhin gegeben ist [ECKR95]. Demzufolge müssen Change-Informationen präzise abgeleitet werden und dies ermöglicht ebenfalls in einem Versionskontrollsystem möglich.

Intensive Untersuchungen bezüglich der Versionskontrolle in flexiblen und agilen Systemen wurden bereits in der Software-Entwicklung betrieben [CRWB98]. Hierbei haben sich verschiedene Begriffe herausgebildet, welche auch für die Versionskontrolle für Prozessmodelle von Interesse sein können:

Repository: Unter einem Repository versteht man ein geteiltes Depot von Artefakten welche in ihren Lebenszyklus weiterentwickelt werden [BPDU94]. Dabei hält das Depot das Artefakt selbst sowie die Meta-Daten über die Weiterentwicklung des Objektes.

Checkin/Checkout: Mit Checkin/Checkout werden die Transaktionen zwischen dem Benutzer und dem Repository bezeichnet. Bei einen Checkin wird eine neue Version in eines Objektes in das Repository geladen. Mit einen Checkout werden Daten aus dem Repository dem Benutzer zur Verfügung gestellt [BPDU94].

Revision/Version: Unter einer Revision oder Version versteht man ein Objekt, bei welchen beabsichtigt wird, dass dieses den Vorgänger ersetzt. Revisionen werden dann erstellt, wenn ältere Revisionen weiterhin gewartet werden müssen. Die kann unter anderem dann der Fall sein, wenn laufende Instanzen noch nicht auf neuere Prozessmodelle migriert werden können [CRWB98].

Variante: Sind Versionen, welche mit anderen Versionen koexistieren [CRWB98]. Dies ist dann der Fall wenn unterschiedliche Versionen eines Prozessmodells gleichzeitig zum Einsatz kommen.

Branch: Eine Branch ist eine Abzweigung von der zeitlichen Hauptentwicklungslinie eines Objektes. In einer Branch können weitere Prozessmodelle entstehen, wenn Objekte als unabhängig von der zeitlichen Hauptentwicklungslinie entwickelt werden sollen.

Untersuchungen hinsichtlich Versisonsmanagement im Kontext von Prozessmodellen wurden bereits getätigt [JGHO98][KMGA99][ZXLC07][BHCB07]. Die Autoren schlagen Versionskontroll-Konzepte und Modelle für ein Repository vor, welche für Prozessmodelle geeignet sind. Dabei werden zur Repräsentation der Versionen Versionsgraphen eingesetzt und die Änderungen zwischen den einzelnen Versionen werden mittels Deltas abgebildet. Eine Integration von zuvor abgeleiteten Highlevel-Changes in eines der vorgeschlagenen Konzepte existiert noch nicht, häufig sind die Untersuchungen auf Add/Delete/Update-Operationen beschränkt. Dieser Abschnitt beschäftigt sich daher mit einer geeigneten Versionskontroll-Struktur des Repositorys hinsichtlich der temporalen Abbildung von Prozessmodell-Evolutionen, dem Branching sowie dem Checkout von Prozessmodellen aus dem Repository über mehrere Modellversionen hinweg unter dem

Aspekt von Deltas mit Highlevel-Changes welche in einem Change-Log abgelegt wurden.

10.1 Repository-Aufbau

In der Software-Entwicklung existieren verschiedene Konzepte für die Struktur eines Repositorys. Dort haben sich die Versionsgraphen durchgesetzt. [JGHO98] verwendet für die Abbildung der Versionen von Prozessmodellen ebenfalls einen Versionsgraphen. Allerdings können die Ansätze aus der Software-Entwicklung nicht direkt übernommen werden, daher sollte nach [ZXLC07] und [ERHF11] das Modell für ein Repository von Prozessmodellen modifiziert werden.

Prozessmodellversionen entwickeln sich auf einem Produktivsystem nicht linear mit der letzten Version im Repository sondern unterliegen auch temporären Änderungen, weshalb das Konzept von Branches vermehrt in den Mittelpunkt rückt [ERHF11]. Des weiteren ist eine gleichzeitige Ausführung von unterschiedlichen Prozessversionen in einem Workflow-Tool eine wirtschaftliche Notwendigkeit. Ein Checkout einer Version aus dem Repository kann daher häufiger durchgeführt werden als bei einem Repository in der Software-Entwicklung.

[ZXLC07] unterteilt daher die Versionsgraphen in eine Hauptentwicklungslinie und in mehrere Nebenäste (Branches). Die Hauptentwicklungslinie stellt Änderungen dar, welche einen starken Einfluss auf das Prozessmodell haben. Dies könnten zum Beispiel Technologie-Änderungen sein oder Änderungen, welche langfristig im Modell bestehen bleiben. Ausgehend von diesen Änderungen werden Branches erstellt welche sich aus temporären beziehungsweise Änderungen durch spezielle Situationen wie zum Beispiel im Bankensektor für regulatorische Anforderungen in einzelnen Ländern ergeben. Somit entstehen zwei Achsen: Eine für permanente Abänderungen welche Prozessverbesserungen abbilden. Diese Achse bildet die Entwicklung über die Zeit hinweg ab. Die zweite Achse repräsentiert Varianten des Prozessmodells in einer bestimmten Version und somit kleinen Änderungen.

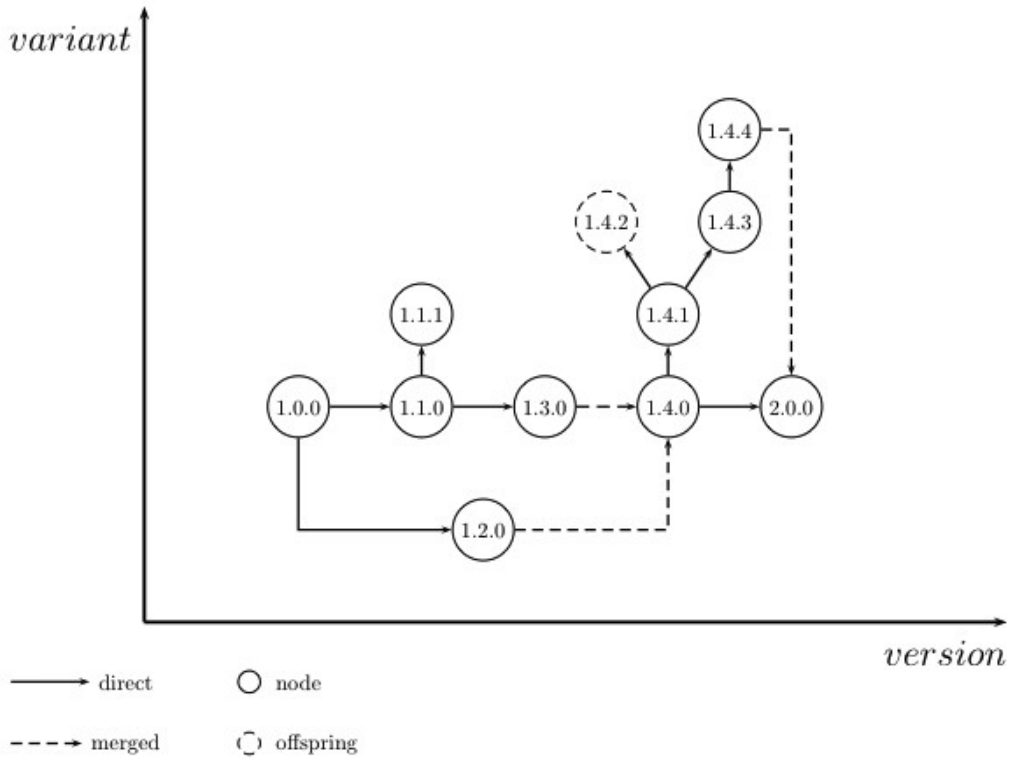


Abbildung 45: Versionsgraph "LCR-Kalkulation"

Quelle: in Anlehnung an [ZXLC07]

In Abbildung 45 ist ein Versionsgraph für den Basel-III-Ablauf zur LCR-Berechnung zu sehen. Um die Versionen zu unterscheiden, wird ein Nummerierungssystem mit drei Stellen verwendet [ZXLC07]. Das System folgt dem Schema `x.y.z.`. Die z-Stelle steht für eine fortlaufende Nummer, einer Variante aus einer Version. Die y-Position ist eine fortlaufende Versionsnummer auf der $versions$ -Achse. Die x-Postion ist ebenfalls eine Versionsnummer auf der $versions$ -Achse. Sie wird aber im Unterschied zur y-Postion nicht automatisch bei jeden Checkin erhöht, sondern manuell durch den den Benutzer manuell um erhöht werden.

Abhängig vom Umfang des Changes, wird eine Version auf der $version$ -Achse angelegt, oder auf der $variant$ -Achse. Wird eine Version auf der $version$ -Achse angelegt, wird die zweite Stelle des Nummerierungssystems um $max(y)+1$ abhängig von x erhöht und eine Verbindung zwischen der Vorgänger-Version und dem CheckIn-Prozessmodell erstellt. In Abbildung 45 wurde aus der Version 1.0.0 die Version 1.1.0 erstellt, sowie die Version 1.2.0. Später wurde aus der Version 1.1.0 eine weitere Version abgeleitet. Für alle Versionen mit $x=1$ erhält man durch $max(y)+1=2+1=3$ als neue Versionsnummer. Dies entspricht der neuen dreistelligen Versionsnummer 1.3.0.

Die Version 1.4.0 ist durch einen Merge zwischen der Version 1.2.0 und der Version 1.3.0

entstanden. Bei einem Merge werden die Gemeinsamkeiten und Differenzen zweier Prozessmodelle miteinander verbunden [BCRS13]. Auf Version 1.4.0 folgt die Version 2.0.0. Hier wurde die x-Postion manuell vom Benutzer um eins erhöht. In dieser Version konnten so umfangreiche Änderungen gemacht worden sein, dass zukünftige Prozessmodelle nur noch aufbauend auf der Version 2.0.0 modifiziert werden sollen. Aus Abbildung 45 geht hervor, dass die Version 2.0.0 durch einen Merge zwischen Version 1.4.0 und 1.4.4 hervorgegangen ist.

Varianten existieren in Version 1.1.0 und 1.4.0. Eine Variante von 1.1.0 ist 1.1.1. Eine Variante erhöht dabei die z-Position im Nummerierungssystem in Abhängigkeit von y um $max(z)+1$. Anhand von Version 1.4.0 ergibt sich daher die Variante 1.4.1. Aus dieser wiederum sind zwei weitere Varianten hervorgegangen. Ähnlich wie auch schon bei Version 1.0.0 können aus einer Vorgänger-Version mehrere Varianten beziehungsweise Versionen erstellt werden. Die Umrandung in Version 1.4.2 kennzeichnet einen „Offspring“ [CRWB98]. Weiterentwicklungen dieser Variante werden in einem gesonderten Versionsgraphen abgebildet und sind unabhängig von der ursprünglichen Entwicklung. Dies kann zum Beispiel dann der Fall sein, wenn ein Produkt oder eine Dienstleistung einen auf sich zugeschnittenen Prozess benötigt, weitere Änderungen in der ursprünglichen Hauptentwicklungslinie jedoch für andere Prozesse nicht relevant sind. Modelle welche in einem neuen Versionsgraphen abgebildet werden, beginnen wieder mit der Versionsnummer 1.0.0.

Im Versionsgraphen wird nicht für jede Version oder Variante das vollständige Prozessmodell abgebildet, sondern nur die Differenzen zum Vorgängermodell. Einerseits können so Problematiken wie zum Beispiel Speicherplatz Verbrauch bei der Abspeicherung von Deltas umgangen werden [JGHO98]. Des weiteren kann durch Deltas eine einfachere Verbindung zur Disziplin des Change Minings [GRRA06] aufgebaut werden. Vollständige Prozessmodelle inklusive Change-Logs werden lediglich in den Versionen, beziehungsweise Varianten gespeichert, welche keinen Nachfolger besitzen. Abbildung 45 enthält zum Beispiel die Variante 1.1.1, 1.4.2, 1.4.4 und die Version 2.0.0 ein vollständiges Prozessmodell.

10.2 Persistierung des Versionsgraphen

Mittels der Persistierung des Versionsgraphen werden die Vorgänger-Beziehungen zwischen einzelnen Versionen und Varianten abgebildet. Wie aus Abbildung 45 ersichtlich, entstehen neben der Hauptentwicklungslinie in einem Versionsgraphen innerhalb der Varianten-Dimension Subgraphen. Einzelne Varianten können mittels Merge wieder in die Hauptentwicklungslinie zurückgeführt werden. Somit entsteht mit dem gewählten Versionskontrollmodell ein gerichteter azyklischer Graph. Modellversionen welche durch einen Merge entstanden sind werden in der Persistierung besonders behandelt, da bei einem Checkin des Prozessmodells abhängig von der Anzahl der involvierten Prozessmodelle entsprechend viele Change-Logs erstellt werden. Ist eine Version beziehungsweise eine Variante ein Offspring wird in der Persistierung eine Verlinkung zwischen den beiden Versionsgraphen hergestellt.

Im Nachfolgenden werden Persistierungen des Versionsgraphen aus Abbildung 45 gezeigt. Der vollständig persistierte Versionsgraph und ein XML-Schema werden im Anhang angeführt.

10.2.1 Hauptentwicklungslinie und Varianten-Persistierung

Für die Persistierung der Hauptentwicklungslinie und Varianten werden aus Abbildung 45 die Versionen 1.0.0 bis 1.3.0 sowie die Variante 1.1.1 exemplarisch herangezogen:

```
<vg name="LCR-Kalkulation">
      <v type="version" vid="1.0.0">
            <cl>Delta_1.0.0.xml</cl>
      </v>
      <v type="version" vid="1.1.0">
            <cl predecssor="1.0.0">Delta_1.1.0.xml</cl>
            <v type="variant" vid="1.1.1">
                  <cl predecessor="1.1.0">Delta_1.1.1.xml</cl>
            </v>
      </v>
      <v type="version" vid="1.2.0">
            <cl predecssor="1.0.0">Delta_1.2.0.xml</cl>
      </v>
      <!-- ... -->
      <!-- ... -->
</vg>
```

Listing 9: Hauptentwicklungslinie- und Varianten-Persistierung

Eine Versionsgraph beginnt mit dem `<vg>`-Tag und dem Namen des Prozesses. Dieser

Tag ist der Elternknoten für alle Versionen auf der Hauptentwicklungslinie. Jede Version und Variante erhält innerhalb des <vg>-Tags einen Kindknoten <v>. Zwischen einer Version und einer Variante kann über das Attribut type differenziert werden. Jeder <v>-Tag enthält im Attribut vid die Versionsnummer im Versionsgraphen. Jeder <v>-Tag vom type 'version' ist ein direktes Kind vom <vg>-Tag. Die Reihenfolge der <v>-Elemente spiegelt die zeitliche Komponente im Versionsgraphen wider.
Jeder <v>-Tag, unabhängig von type, besitzt mindestens ein weiteres Kind-Element. Verpflichtend dabei ist der <cl>-Tag. Im <cl>-Element wird die Verlinkung zum zuvor errechneten Change-Log hergestellt. Das Change-Log ergibt sich dabei aus dem direkten Vorgänger. Dieser Vorgänger wird im <cl>-Tag im Attribut predecessor angegeben. Wie aus Listing 9 und den predecessor Attributen der Versionen mit der vid 1.1.0 und 1.2.0 entwickelten sich diese aus der Version 1.0.0. Die jeweiligen Change-Logs der beiden Prozessmodelle 1.1.0 und 1.2.0 wurden daher aus der Version 1.0.0 errechnet.

Einen Spezialfall bildet das Chang-Log der ersten Version (1.0.0). Da dieses keinen Vorgänger hat (das Attribut predecessor ist hier nicht gesetzt) wird ein Change-Log Delta_1_0_0.xml nur mit Add-Operationen erstellt. Dies ist einerseits notwendig, da für alle Prozessmodelle welche einen Nachfolger haben nur die Abspeicherung durch Change-Logs erfolgt und nicht das vollständige Prozessmodell, anderseits kann bei einem Checkout der Version 1.0.0 keine vollständige Rückrechnung auf Grund eines Informationsverlustes bei unveränderten Prozessobjekten erfolgen.

Wurde aus einer Version eine Variante abgeleitet, wie es in Abbildung 45 mit der Version 1.1.0 gemacht worden ist, wird die Variante als ein Kind dieser Version hinzugefügt. Wie in Abbildung zwei gezeigt, folgt der <v>-Tag für die Variante der gleichen Logik wie ein <v>-Tag aus für eine Version. Der Unterschied besteht lediglich darin, dass das Attribut 'type' den Wert variant zugewiesen bekommt.

10.2.2 Merge-Persistierung

Ein Merge ist ein weiterer Spezialfall bei der Persistierung des Versionsgraphen. Eine gemergte Version im Versionsgraphen weist mehrere Vorgänger auf. Hier ergeben sich zwei Probleme: Erstens, die Vorgänger-Version aus welcher das Change-Log erstellt wird

kann nicht eindeutig bestimmt werden (es kommen beide Vorgänger in Frage). Zweitens, wird ein Change-Log aus einer Vorgänger-Version v_1 bestimmt und möchte man die Vorgänger-Versions v_2 aus dem Repository auschecken, kann v_2 aus dem zuvor erstellten Change-Log von v_1 nicht mehr errechnet werden.
Ausschlaggebend hierfür sind die unterschiedliche Highlevel-Changes in den Change-Logs der beiden Vorgänger-Version.

Wie dennoch die beiden Konflikte gelöst werden können, zeigt Listing 10

```
<vg name="LCR-Kalkulation">
      <!-- ... -->
      <!-- ... -->
      <v type="version" vid="1.4.0">
            <cl predecssor="1.3.0">Delta_1.4.0_merged_1.3.0.xml</cl>
            <cl predecssor="1.2.0">Delta_1.4.0_merged_1.2.0.xml</cl>
      </v>
      <!-- ... -->
      <!-- ... -->
</vg>
```

Listing 10: Merge-Persisiterung

In Listing 10 wird der <v>-Tag für die Version 1.4.0 gezeigt. Wird ein Merge-Algorithmus wie in [BCRS13] beschrieben verwendet, entsteht durch den Merge von zwei Prozessmodellen ein neues Prozessmodell. Dieses gemergte Modell kann, wie in Abbildung 45 dargestellt, in das Repository eingecheckt werden und für jede Vorgänger-Version, die am Merge beteiligt war, erhält der Versionsgraph eine Verlinkung zu den Change-Logs mit Highlevel-Changes der involvierten Prozessmodellen. Die so generierten Change-Logs werden dann wie in Listing 10 gezeigt, in <cl>-Tags abgelegt. Jeder <cl>-Tag beinhaltet dabei den Vorgänger und den Verweis auf das Change-Log. Wird für jede Vorgänger-Version ein Change-Log erstellt, können die beiden eingangs erwähnten Probleme gelöst werden. Eine Rückrechnung auf eine der beiden Vorgänger-Version ist dann möglich, da für diese die Change-Logs vorhanden sind.

10.2.3 Varianten-Subgraph-Persitierung

Entsteht aus einer Variante weitere Varianten wie es in Abbildung 45 in Version 1.4.0 dargestellt ist, kann zu jedem Elternknoten ein weiteres <v>-Kind-Element hinzugefügt werden. Wie auch schon bei der Persistierung der Hauptentwicklungslinie bildet die

Reihenfolge der hinzugefügten Kinder eine zeitliche Komponente ab.

```
<vg name="LCR-Kalkulation">
<!-- ... -->
<!-- ... -->
      <v type="version" vid="1.4.0">
            <!-- ... -->
            <!-- ... -->
            <v type="variant" vid="1.4.1">
                  <cl predecessor="1.4.0">Delta_1.4.1.xml</cl>
                  <v type="variant" vid="1.4.2">
                        <cl predecessor="1.4.1">Delta_1.4.2.xml</cl>
                  </v>
                  <v type="variant" vid="1.4.3">
                        <cl predecessor="1.4.1">Delta_1.4.3.xml</cl>
                        <v type="variant" vid="1.4.4">
                              <cl predecessor="1.4.3">Delta_1.4.4.xml</cl>
                        </v>
                  </v>
            </v>
      </v>
<!-- ... →
<!-- ... -->
</vg>
```

Listing 11: Varianten-Subgraph-Persistierung

In Listing 11 wird der Subgraph der Version 1.4.0 gezeigt. Die Version 1.4.0 besitzt eine Variante 1.4.1, welche als Kindknoten der Version 1.4.0 in einem weiteren `<v>`-Tag vom Typ `variant` hinzugefügt wurde. Da die Varianten 1.4.2 und 1.4.3 aus der Variante 1.4.1 entstanden sind, sind diese ebenfalls als Kindknoten in der Variante 1.4.1 aufgenommen worden. Durch die Verschachtelung der Kind-Elemente kann so der Subgraph der Varianten abgebildet werden.

10.2.4 Offsping-Persistierung

Wird eine Variante oder eine Version als ein Offsping markiert, kann durch eine Referenz der unabhängige Versionsgraph für diese Variante verknüpft werden.

```
<vg name="LCR-Kalkulation">
<!-- ... →
<!-- ... -->
      <v type="variant" vid="1.4.2">
            <cl predecessor="1.4.1">Delta_1.4.2.xml</cl>
            <vgref>"LCR-Kalkulation_MainCurrency"</vgref>
      </v>
<!-- ... -->
</vg>
```

Listing 12: Offsping-Persistierung

Das Listing zeigt die Persistierung eines Offsping. Grundsätzlich wird die Variante beziehungsweise die Version wie üblich mit einem `<v>`-Tag hinzugefügt. Der `<vgref>`-Tag erstellt die Verbindung zum unabhängigen Versionsgraphen. In Listing 12 zeigt der `<vgref>`-Tag auf einen neuen Versionsgraphen `LCR-Kalkulation_MainCurrency`.

10.3 Metainformationen in Change-Logs

Metainformationen in Change-Logs enthalten zusätzliche Informationen über die Änderungen in einem Prozessmodell. Änderungen können so nachvollziehbarer gestaltet werden. Zu dem wird es mit Metainformationen möglich statistische Auswertungen zu erheben. In der Software-Entwicklung haben sich bestimmte Metainformationen in Versionskontrollsysteme als nützlich erwiesen [ATDA98]. In Tabelle 1 sind in der ersten Spalte Tags aufgelistet. Diese Tags werden zu jeden `<v>`-Tag als ein Kind-Element hinzugefügt. In der zweiten Spalte wird die Bedeutung jedes Tags angegeben.

Tag	**Bedeutung**
`<cuser>`	Welcher Entwickler hat den Change zwischen zwei Versionen vollzogen?
`<ctime>`	Wann wurde die Version in das Repository eingecheckt?
`<comment>`	Kommentar des Entwicklers über den Change. Warum wurde der Change gemacht?

Tabelle 1: Metainformationen

Der vollständig gezeigte Versionsgraph im Anhang beinhaltet auch die eben gelisteten Metainformationen.

10.4 Rückrechnung von Prozessmodellen

In den letzten Kapiteln wurde bislang nur die Weiterentwicklung von Prozessmodellen von einem Zeitpunkt t nach $t+1$ behandelt. Im Zuge einer Versionskontrolle wird es jedoch notwendig Prozessmodelle von einem Zeitpunkt t nach $t-1$ beziehungsweise noch weiter zurück in die Vergangenheit ($t-n$) zu bestimmen. Dies ist dann der Fall, wenn ältere Modelle wieder zur Anwendung kommen sollen Weiterentwicklungen eines alten Prozessmodells angestrebt werden, oder unterschiedliche Prozessmodellversionen in einem Workflow-Tool lauffähig sein müssen.

In der vorgeschlagenen Struktur eines Prozessmodell-Repositorys werden nicht die kompletten Prozessmodelle der Vorgängerversion abgespeichert, sondern nur die erzeugten Deltas. Die generierten Change-Logs beinhalten daher nur die Änderungen in das $t+1$ -Modell. Sprich, es sind die Operationen abgelegt, welche notwendig sind um das Modell aus $t-1$ nach t zu migrieren. Soll nun ein Vorgängermodell abgeleitet werden, müssen die Änderungsoperationen wieder „rückgängig“ gemacht werden. Dies kann erreicht werden, indem die Highlevel-Changes aus dem Change-Log in ihrer inversen Form ausgeführt werden [KMGA99].
Formal gesprochen, lässt sich ein Prozessmodell in die Vorgängerversion wie folgt übertragen: $PM_t \lceil \Delta^{-1} \rangle PM_{t-1}$. Wobei PM_t ein vollständiges Prozessmodell ist und Δ^{-1} das inverse Change-Log aus der Differenzenerkennung. Durch Anwendung von Δ^{-1} auf PM_t , entsteht dann PM_{t-1} .

Die nachfolgende Tabelle listet zu jedem Highlevel-Change die inverse Variante auf. Zusätzlich wird zu jeder inversen Form eine Erläuterung gegeben und ein Beispiel gezeigt. Die Beispiele wurden aus den vorhergehenden Kapiteln entnommen.

Highlevel-Change	Inverser Highlevel-Change
ADD(Obj, predecessor$_t$, accessor$_t$)	DELETE(Obj, predecessor$_t$, accessor$_t$)
DELETE(Obj, predecessor$_{t-1}$, accessor$_{t-1}$)	ADD(Obj, predecessor$_{t-1}$, accessor$_{t-1}$)
MOVE(Obj, predecessor$_t$, accessor$_t$)	MOVE(Obj, predecessor$_{t-1}$, accessor$_{t-1}$)
REPLACE(Obj$_t$, ReplacedO$_{t-1}$, predecessor$_t$, accessor$_t$)	REPLACE(Ob$_{t-1}$, ReplacedObj$_t$, predecessor$_{t-1}$, accessor$_{t-1}$)
SWAP(Obj1, Obj2)	SWAP(Obj2, Obj1)
COPY(Obj, predecessor$_t$, accessor$_t$)	DELETE(Obj, predecessor$_{t-1}$, accessor$_{t-1}$)
UPDATE(Obj) ADD UPDATE(Obj) DELETE UPDATE(Obj) CHANGE	UPDATE(Obj) DELETE UPDATE(Obj) ADD UPDATE(Obj) CHANGE

Tabelle 2: Inverse Form der Highlevel-Changes

Die komplette Liste der inversen Operationen inklusive Fragment-Operationen und Sequenzen befinden sich im Anhang.

10.4.1 Inverser Add-Change

Ein Add-Change wird durch ein Delete rückgängig gemacht, indem das hinzugefügte Objekt zwischen $predecessor_t$ und $accessor_t$ entfernt wird.

```
ADD ('LCR berechnen',
     'Gewichtungen anwenden',
     'end')
```

```
DELETE ('LCR berechnen',
     'Gewichtungen anwenden',
     'end')
```

Listing 13: Inverser Add-Change

10.4.2 Inverser Delete-Change

Ein Delete-Change wird durch ein Add rückgängig gemacht, indem das gelöschte Objekt zwischen $predecessor_{t-1}$ und $accessor_{t-1}$ in $PM_t - 1$ wieder hinzugefügt wird.

```
DELETE(
    'Report erzeugen',
    'Gewichtungen anwenden',
    'end')
```

```
ADD('Report erzeugen',
    'Gewichtungen anwenden',
    'end')
```

Listing 14: Inverser Delete-Change

10.4.3 Inverser Move-Change

Der Move-Highlevel-Change lässt sich durch eine entgegengesetzte Move-Operation wieder rückgängig machen. Das Objekt wird an die ursprüngliche Position verschoben. Die ursprüngliche Position ist jene im $t-1$ -Modell. Daher wird das Objekt zwischen $predecessor_{t-1}$ und $accessor_{t-1}$ gesetzt. $predecessor_{t-1}$ und $accessor_{t-1}$ können aus den persistierten Move-Changes über den <from>-Tag abgeleitet werden.

```
MOVE (                                  MOVE (
 Retail/WS-Daten laden,                  Retail/WS-Daten laden,
 Gewichtungen anwenden,                  Portfolio laden,
 Report erzeugen                         Report-Position anwenden
     )                                       )
```

Listing 15: Inverser Move-Change

10.4.4 Inverser Replace-Change

Der inverse Replace-Change verhält sich ähnlich wie zwei inverse Delete und Add Changes. Hierbei wird das Objekt, welches im t -Modell hinzugefügt wurde (Obj_t) als das zu ersetzende Objekt $ReplacedObj_t$ behandelt. Das gelöschte, beziehungsweise ersetzte Objekt im $t-1$ -Modell wird wieder zum Modell zwischen $predecessor_{t-1}$ und $accessor_{t-1}$ hinzugefügt.

```
REPLACE (                               REPLACE (
      Report erzeugen,                     LCR berechnen,
      LCR berechnen,                       Report erzeugen,
      Gewichtungen anwenden,               Gewichtung anwenden,
      'end'                                end
         )                                 )
```

Listing 16: Inverser Replace-Change

10.4.5 Inverser Swap-Change

Um einen Swap wieder rückgängig zu machen, werden die beiden getauschten Objekte an die Position im $t-1$ -Modell zurückverschoben. Ein inverser Swap verhält sich wie zwei inverse Moves.

```
SWAP(                                  | SWAP(
   Gewichtung anwenden,                |    Retail/WS-Daten laden,
   Retail/WS-Daten laden)              |    Gewichtung anwenden)
```

Listing 17: Inverser Swap-Change

10.4.6 Inverser Copy-Change

Da der Copy-Change grundsätzlich eine Abwandlung des Add-Changes ist, wird der Copy-Change mit einem Delete-Change invertiert. Dabei wird das Prozessobjekt aus dem t -Modell nicht in $t-1$ eingefügt.

```
COPY(LCR berechnen,                    | DELETE(LCR berechnen,
     LCR berechnen,                    |      LCR berechnen,
     end                               |      end
    )                                  |     )
```

Listing 18: Inverser Copy-Change

10.4.7 Inverses Update

Je nach Update-Typ kommt eine andere inverse Variante zu tragen. Wird ein Attribut zu einem Prozessobjekt hinzugefügt, dann wird dies durch ein Update-Delete wieder rückgängig gemacht. Das Attribut befindet sich nicht mehr im $t-1$ -Modell. Im Falle eines Update-Deletes wird das Attribut welches im t -Modell entfernt wurde, im $t-1$ -Modell hinzugefügt. Bei einem Update-Change wird der alte Wert wiederhergestellt.

Für Sequenzen wird abhängig von der Sequenz-Art die inverse Form gebildet. Gleiches gilt auch für die Operationen welche Fragmente und Alternativen betreffen. Auch hier existiert zu jeder Operation eine inverse Form.

10.4.8 Anwendung inverser Operationen

Ausgehend von einem Prozessmodell PM_t zum Zeitpunkt t , kann durch Anwendung der inversen Operationen das Modell PM_{t-1} zum Zeitpunkt $t-1$ rückgerechnet werden. Dabei ist das Modell, auf welches die inversen Operationen Δ^{-1} angewendet werden, vollständig vorhanden. Sprich, aus dem Modell PM_t kann ein Process-Structure-Tree PST_t abgeleitet werden.

Ziel ist es, aus PST_t und den inversen Operationen Δ^{-1} die flache Struktur $fPST_{t-1}$ von PM_{t-1} zu erzeugen. In weiterer Folge kann dann aus $fPST_{t-1}$ der Process-Structure-Tree PST_{t-1} für das rückgerechnete Modell PM_{t-1} erstellt werden. Zur Generierung von $fPST_{t-1}$ kann wie folgt vorgegangen werden: Beginnend beim Start-Knoten von PST_t wird dieser wieder mittels Depth-First-Ansatz bis zum End-Knoten traversiert. Während PST_t durchlaufen wird, kann gleichzeitig und unter Anwendung von fünf Regeln $fPST_{t-1}$ aufgebaut werden. Je nachdem welche Regel zur Anwendung kommt, wird im PST_t weiter traversiert oder in $fPST_{t-1}$ Operationen durchgeführt, sowie die aktuell zu behandelnde Position in $fPST_{t-1}$ verschoben.

Regel 1: Ist die aktuell zu behandelnde Position i in $fPST_{t-1}$ nicht leer, dann wird i um eins erhöht. Regel 1 kommt dann zur Anwendung, wenn durch eine Positionsänderung, zum Beispiel durch eine Move-Operation, ein Objekt an einer Position j eingefügt wurde, an der gilt $j > i$. In diesem Fall wird die Position übersprungen, da bereits ein Objekt an dieser Stelle vorhanden ist. Im PST_t erfolgt keine Positions-Änderung.

Regel 2: Existiert für die aktuell zu behandelnde Position i in $fPST_{t-1}$ eine Add-Operation in Δ^{-1}, wird dieses Objekt auf dieser Position in $fPST_{t-1}$ eingefügt. Diese Regel kommt dann zur Anwendung, wenn ein Objekt aus PM_{t-1} gelöscht wurde um PM_t zu erstellen. Das Objekt muss daher wieder auf diese Position gesetzt werden. Innerhalb PST_t erfolgt keine Traversierung. Nachdem das Objekt zu $fPST_{t-1}$ hinzugefügt worden ist, wird i ums eins erhöht.

Regel 3: Ausgehend vom aktuellen Knoten n im PST_t wird geprüft, ob gilt $n \notin \Delta^{-1}$. Ist dies der Fall, wird das Prozessobjekt aus n auf der Position i in $fPST_{t-1}$ eingefügt. Regel drei kommt dann zu tragen, wenn das Prozessobjekt im Prozessmodell PM_t nicht geändert wurde. Nach Durchführung dieser Regel, wird i um eins erhöht und die Traversierung im PST_t schreitet um einen Knoten weiter.

Regel 4: Befindet sich der aktuelle Knoten n im PST_t in Δ^{-1} und gleichzeitig eine Delete-Operation, wird der Knoten n nicht in $fPST_{t-1}$ aufgenommen. Anders

ausgedrückt wird Regel vier angewandt, wenn n erst in PM_t hinzugefügt wurde – der Knoten wird im PST_t übersprungen. Demzufolge wird die Position i in $fPST_{t-1}$ nicht erhöht, jedoch wird im PST_t zum nächsten Knoten gesprungen.

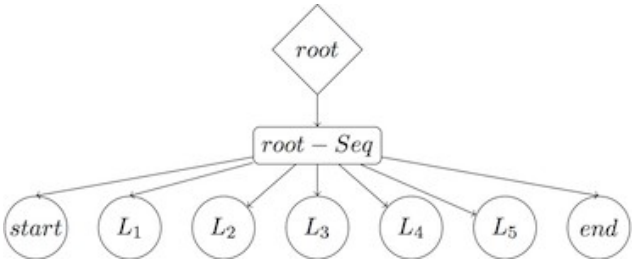
Regel 5: Gilt $n \in \Delta^{-1}$ und ist die Highlevel-Change-Operation kein Add, wird die entsprechende Operation durchgeführt. Da es sich bei Regel 5 unter anderem auch um Positionsänderungen handelt, wird das Prozessobjekt an der entsprechenden Position j in $fPST_{t-1}$ hinzugefügt. In der Position in welcher das Prozessobjekt eingefügt wird, darf $i \neq j$ gelten. Ist $i > j$ dann wird das Objekt zwischen zwei bestehenden Objekten in $fPST_{t-1}$ eingefügt. Gilt $j > i$ wird die flache Struktur erweitert und das Prozessobjekt an Position j gesetzt. Durch eine Erweiterung der flachen Struktur um $j-i$ Positionen, entstehen zwar Lücken in $fPST_{t-1}$, diese werden jedoch im Laufe der Process-Structure-Tree-Traversierung unter Anwendung der fünf Regeln mit anderen Prozessobjekten welche in PM_{t-1} vorhanden sind aufgefüllt. Bei $i = j$ wird das Objekt auf Position i gesetzt. Dies gilt vor allem für Replace-Operationen. Ein Spezialfall sind Swaps, da hier zwei Objekte gleichzeitig verschoben werden. Um sich später eine Prüfung zu ersparen, ob eine Swap-Anweisung angewandt wurde, wird der Move im Swap nur für das aktuell zu behandelnde Objekt durchgeführt und nicht für beide gleichzeitig.
Nachdem Regel fünf angewendet wurde, wird die Position i in $fPST_{t-1}$ nur dann erhöht, wenn gilt $i = j$. Im PST_t wird um ein Objekt weitergegangen.

Beispiel 16: Gegeben ist das Prozessmodell PM_2 , welches auf das Prozessmodell PM_1 zurückgerechnet werden soll.

PM_1

PM_2

PST_2 von PM_2



Deltas

Δ_1 = ADD('LCR berechnen',
'Gewichtung anwenden','end');
DELETE('Report erzeugen',
'Gewichtung anwenden', 'end');

Δ_1^{-1} = DELETE('LCR berechnen',
'Gewichtung anwenden', 'end');
ADD('Report erzeugen',
'Gewichtung anwenden','end');

Rückrechnung $fPST_1$:

1: **s**
2: s – **Portfolio laden**
3: s – Portfolio laden – **Retail/WS-Daten laden**
4: s – Portfolio laden – Retail/WS-Daten laden – **Report-Pos anwenden**
5: s – Portfolio laden – Retail/WS-Daten laden – Report-Pos anwenden – **Gewichtungen anwenden**
6: s – Portfolio laden – Retail/WS-Daten laden – Report-Pos anwenden – Gewichtungen anwenden – **Report erzeugen**
7: s – Portfolio laden – Retail/WS-Daten laden – Report-Pos anwenden – Gewichtungen anwenden – Report erzeugen
8: s – Portfolio laden – Retail/WS-Daten laden – Report-Pos anwenden – Gewichtungen anwenden – Report erzeugen – **e**

Abbildung 46: Rückrechnung einfaches Modell

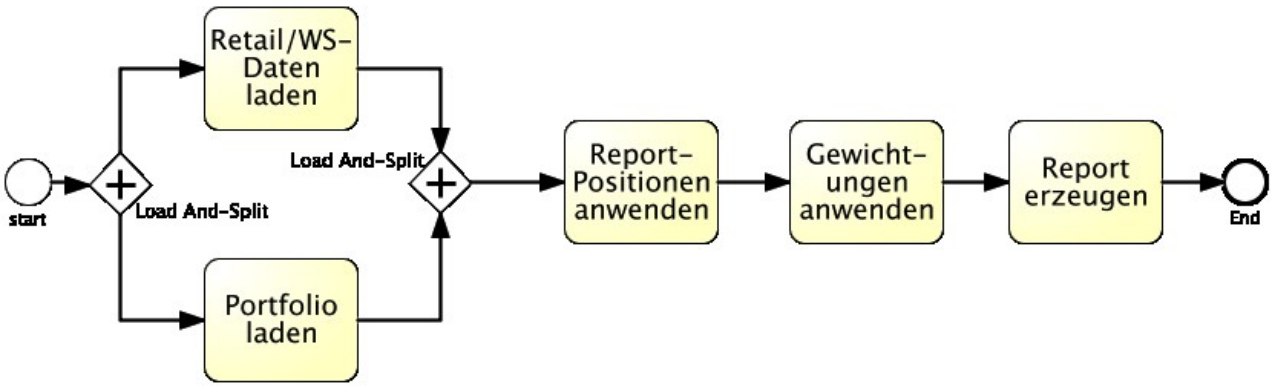
Abbildung 46 zeigt eine Rückrechnung des Prozessmodells PM_2 auf PM_1 . In einem vorgelagerten Schritt wurde Δ_1 , wie schon in der Differenzenerkennung beschrieben, generiert. Bei einer Rückrechnung wird Δ_1 nach Δ_1^{-1} transformiert, womit die inversen Change-Operationen für Δ_1 entstehen. In diesem Beispiel wäre das eine Delete-Operation für das Hinzufügen von `LCR berechnen` und eine Add-Operation für die

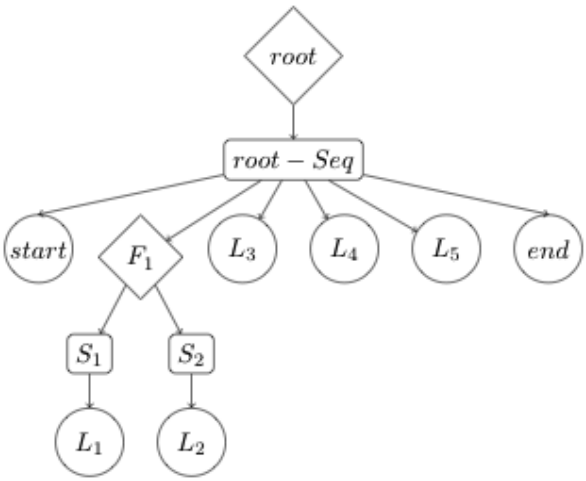
entsprechende Lösch-Aktion von Objekt `Report erzeugen`. Um eine Rückrechnung zu starten, wird der Process-Structure-Tree vom vollständigen Prozessmodell PM_2 benötigt. Im obigen Beispiel sind im Prozessmodell nochmal die einzelnen Fragmente des Prozessmodells angezeigt. Die umrandenden Linien sind zur Verdeutlichung des Process-Structure-Trees eingezeichnet. Anschließend kann die Rückrechnung unter Anwendung der fünf Regeln durchgeführt werden. In der ersten Iteration wird der Knoten `s` in PST_2 analysiert. Da es zu `s` keine Change-Operation in Δ_1^{-1} gibt und auch keine Add-Operation an der Position eins für $fPST_1$, wurde die Regel 3 angewandt und `s` auf Position 1 in $fPST_1$ gesetzt. Die aktuell zu analysierende Position in $fPST_1$ wird um eins erhöht und der nächste Knoten in PST_2 ist `Portfolio laden` (= `L1`). Auch zu `Portfolio laden` gibt es keine Einträge in Δ_1^{-1} , somit wird wieder Regel 3 angewandt. Regel 3 kann bis inklusive Iteration 5 verwendet werden. Nun ist bei Iteration sechs die aktuelle Position in $fPST_1$ sechs und in PST_2 ebenfalls. Es wird wieder geprüft, ob es für diese Position eine Add-Operation gibt. Das Change-Log Δ_1^{-1} enthält dieses mal eine Add-Operation, nämlich `Report erzeugen`. Es muss also die Regel 2 angewendet werden. Diese fügt bei einer positiven Evaluierung auf eine Add-Operation das Prozessobjekt zu $fPST_1$ an der aktuellen Position ein. Des weiteren verlangt diese Regel, dass $fPST_1$ um eine Position – auf sieben – erhöht wird. Der PST_2 wird durch diese Regel nicht weiter traversiert. Der „Pointer“ befindet sich in PST_2 daher weiterhin auf `LCR berechnen` (=`L5`). Für Position sieben existiert in $fPST_1$ keine Add-Operation, jedoch für das Prozessobjekt auf der Position sechs in PST_2 im Change-Log mit einer Delete-Operation. Da es sich um eine Delete-Operation bei einer Rückrechnung handelt, wird das Objekt `LCR berechnen` nicht in $fPST_1$ aufgenommen – das Objekt wird also gelöscht. Hier wurde nun Regel vier angewendet. Weiters wird, wenn Regel vier feuert, die aktuelle Position sieben in $fPST_1$ beibehalten, da es für diese Position keine Operation gab, in PST_2 wird jedoch einen Knoten weitergegangen. Nun befindet man sich am letzten Knoten in PST_2 , weil es keine weiteren Einträge mehr in Δ_1^{-1} gibt, wird wieder Regel 3 angewendet und `e` wird zu $fPST_1$ an Position sieben eingefügt.

Beispiel 17: Gegeben ist das Prozessmodell PM_3 , welches auf das Prozessmodell PM_1 zurückgerechnet werden soll.

PM_1

PM_3





Δ_1 =

```
ADDFragment(Load And-Split,...);
ADDAlternative(Load And-Split_alt_0,...);
ADDAlternative(Load And-Split_alt_1,...);
MOVE(Portfolio laden...);
MOVE(Retail/WS-Daten laden,...);
```

Δ_1^{-1} =

```
DELETEFragment(Load And-Split,...);
DELETEAlternative(Load And-Split_alt_0,...);
DELETEAlternative(Load And-Split_alt_1,...);
MOVE(Portfolio laden
     , start
     , Retail/WS-Daten laden
    );
MOVE(Retail/WS-Daten laden,
    , Portfolio laden
    , Report Positionen anwenden
    );
```

Fortsetzung nächste Seite

Rückrechnung $fPST_1$:

1: **s**
2: s
3: s
4: s - ␣ - **Retail/WS-Daten laden**
5: s - ␣ - Retail/WS-Daten laden
6: s – **Portfolio laden** – Retail/WS-Daten laden
7: s – Portfolio laden – Retail/WS-Daten laden
8: s – Portfolio laden – Retail/WS-Daten laden
9: s – Portfolio laden – Retail/WS-Daten laden – **Reporting Positionen anwenden**
10: s – Portfolio laden – Retail/WS-Daten laden – Reporting Positionen anwenden – **Gewichtungen anwenden**
11: s – Portfolio laden – Retail/WS-Daten laden – Reporting Positionen anwenden – Gewichtungen anwenden – **Report erzeugen**
12: s – Portfolio laden – Retail/WS-Daten laden – Reporting Positionen anwenden – Gewichtungen anwenden – Report erzeugen - **e**

Abbildung 47: Rückrechnung komplexes Modell

In Abbildung 47 ist eine weitere Rückrechnung von zwei Prozessmodellen PM_1 und PM_3 zu sehen. Wieder ist der Process-Structure-Tree PST_3 gegeben und Δ_1^{-1} beinhaltet die inversen Change-Operationen aus Δ_1 . Da bei einer Transformation von PM_1 auf PM_3 das Fragment `Load-And-Split` inklusive zwei Alternativ-Kanten hinzugefügt worden ist, muss dieses bei der Rückrechnung wieder entfernt werden. Die beiden Objekte `Retail/WS-Daten laden` und `Portfolio laden` wurden in die entsprechenden Alternativen verschoben, diese werden bei einer Rückrechnung wieder an ihre ursprüngliche Position in PM_1 zurück verschoben. Unter Anwendung der fünf Regeln wird nun gezeigt, wie sich die flache Struktur $fPST_1$ aus der Abbildung erzeugen lässt. In der ersten Iteration wird der Knoten `s` aus PST_3 analysiert. Es existiert noch kein Eintrag in $fPST_1$ und auch keine Operation für diese Position in $fPST_1$, dementsprechend wird die Regel 3 angewendet und das Objekt `s` wird an die erste Position in $fPST_1$ gesetzt. Nach Regel drei wird in $fPST_1$ und PST_3 um eine Stelle weitergegangen. Der nächste Knoten in PST_3 ist der Fragment-Knoten. Da der Fragment-Knoten das Split- und das Join-Objekt aus PM_3 enthält, werden die beiden Objekte gleichzeitig behandelt und gegebenenfalls auf die entsprechenden Positionen in $fPST_1$ gesetzt. Da sich jedoch eine Delete-Operation für dieses Fragment in Δ_1^{-1} befindet, kommt Regel vier zum Zug. Die beiden Prozessobjekte `Load-And-Split` und

`Load-And-Join` werden also nicht in $fPST_1$ aufgenommen. Der Regel entsprechend wird in PST_3 um einen Knoten weitergegangen, dies ist dann der Knoten `S1`. In $fPST_1$ wird die Position zwei als aktuelle Position beibehalten. Für den Knoten `S1`, welcher die Alternative repräsentiert, existiert ebenfalls eine Delete-Operation, wodurch wieder die Regel vier angewendet wird. Der nächste Knoten welcher in PST_3 analysiert wird ist `Retail/WS-Daten laden` (=`L1`). Für diesen Knoten existiert ebenfalls eine Operation in Δ_1^{-1} . Hierbei handelt es sich um eine Move-Operation, daher wird Regel fünf angewendet. Aus der persistierten Form[9] der Move-Operation kann entnommen werden, dass das Objekt `Retail/WS-Daten laden` in PM_1 auf Position drei lag. Da jedoch die aktuelle Position in $fPST_1$ noch immer die Zwei ist, wird $fPST_1$ um eine Stelle expandiert und das Prozessobjekt auf diese Position „verschoben“. Bei einer Erweiterung gilt, dass die aktuelle Position sich von der tatsächlich eingefügten Position vom Prozessobjekt unterscheidet ($2 \neq 3$), daher wird in $fPST_1$ die aktuelle Position nicht erhöht. Die Iteration vier in der Abbildung 47 zeigt die Erweiterung von $fPST_1$ und das Setzen des Objektes `Retail/WS-Daten laden`. Da es zu keiner Operation auf Position zwei kommt, sondern nur auf der erweiterten Position drei, kommt es zu einer Lücke in $fPST_1$ welche mit einem '␣' gekennzeichnet ist. Gemäß des Depth-First-Search-Ansatzes ist der nächste zu behandelnde Knoten `S2`, welcher der zweiten Alternative im Fragment entspricht. Auch für diese Alternative existiert ein Eintrag in Δ_1^{-1} , daher wird die Delete-Operation mit der Regel vier angewendet. Wie die Iteration fünf zeigt, ändert sich in $fPSt_1$ nichts. Die aktuelle Position bleibt weiterhin die Zwei. Als nächstes wird in PST_3 der Knoten `L2` (= `Portfolio laden`) analysiert. Eine Prüfung gegen Δ_1^{-1} ergibt eine Move-Operation für diesen Knoten. Aus der persistenten Form lässt sich ableiten, dass das Prozessobjekt `Portfolio laden` im Prozessmodell PM_1 ursprünglich an Position zwei lag. Die inverse Move-Operation verschiebt somit das Prozessobjekt von Position sechs in PM_3 , zurück auf Position zwei in $fPST_1$. Iteration sechs schließt daher die Lücke, welche in Iteration vier entstanden ist. In den nächsten beiden Iterationen sieben und acht, kommt jeweils Regel eins zum Zug - die aktuelle Position in $fPST_1$ ist dann die Position vier. In PST_3 ist der nächste Knoten `Report-Positionen anwenden`. Für diesen Knoten wird die Regel drei angewendet, da es zu diesem Prozessobjekt keinen Eintrag in Δ_1^{-1} gibt. Wie auch schon im

9 Die persitierten Changes befindet sich im Anhang.

vorhergehenden Beispiel gezeigt, wird das Objekt, welches in PST_3 auf Position acht eingetragen ist, auf Position vier in $fPST_1$ gesetzt. Diese Position entspricht dann auch wieder der ursprünglichen Position in PM_1 . Die restlichen Iterationen folgen der gleichen Regel wie das Prozessobjekt `Report-Positionen anwenden` aus Iteration neun. Nachdem alle 12 Iterationen durchgeführt wurden, konnte die vollständige flache Struktur $fPST_1$ für PM_1 abgeleitet werden. Aus $fPST_1$ lässt sich dann PST_1 und in weiterer Folge auch PM_1 erstellen.

10.5 Rückrechnung über mehrere Prozessmodelle

Soll nicht nur eine Version zurückgerechnet werden sondern über mehrere Versionen eines Prozessmodells, müssen alle Change-Logs bis zu dieser Version miteinbezogen werden. Dabei müssen die Change-Logs der involvierten Versionen chronologisch angewendet werden:

- $PM\left[\sigma\right\rangle PM'$ nur dann wenn $\exists PM_t, PM_{t-1}, PM_{t-2}, \ldots PM_{t-n}$ mit $PM = PM_t$, $PM' = PM_{t-n}$ und $PM_i\left[\Delta_{i-1}^{-1}\right\rangle PM_{i-1}$ wobei $i = \{t, t-1, \ldots, t-n+1\}$.

Dabei ist $\sigma = \{\Delta_{t-1}^{-1}, \Delta_{t-2}^{-1}, \ldots, \Delta_{t-n+1}^{-1}\}$ eine Sequenz von Change-Logs und Δ_n^{-1} sind die Change-Logs mit den inversen Operationen, welche zur Rückrechnung benötigt werden. Wendet man nun Δ_{t-1}^{-1} auf PM an, so erhält man PM_{t-1} . Auf PM_{t-1} wird dann Δ_{t-2}^{-1} angewendet. Daraus entsteht PM_{t-2} . Dies wird solange iteriert, bis alle Change-Logs abgearbeitet wurden Mit der Anwendung von Δ_{t-n+1}^{-1} entsteht dann PM' .

Angenommen, es existiert das vollständige Prozessmodell PM_3 aus den vorhergehenden Beispielen und man möchte aus diesem das Prozessmodell PM_1 erzeugen. Zwischen PM_3 und PM_1 liegt noch das Prozessmodell PM_2 . Durch

eine Differenzerkennung in der Evolution von PM_1 zu PM_3 über PM_2 wurden zwei Deltas generiert, welche in ihrer inversen Form wie folgt aussehen:

$PM_3[\sigma\rangle PM_1$, wobei $\sigma=\{\Delta_2^{-1},\Delta_1^{-1}\}$ und

Δ_2^{-1} = `DELETEFragment(Load And-Split,...); DELETEAlternative(Load And-Split_alt_0,...);`
`DELETEAlternative(Load And-Split_alt_1,...); MOVE(Portfolio laden...);`
`MOVE(Retail/WS-Daten laden,...);`

Δ_1^{-1} = `DELETE('LCR berechnen', ...); DELETE('Report erzeugen',...)`

Abbildung 48: Change-Log Sequenz

Die Abbildung zeigt die beiden inversen Deltas. Wird auf das vollständige Prozessmodell PM_3 Δ_2^{-1} angewendet, erhält man PM_2 . Wird anschließend auf PM_2 Δ_1^{-1} angewendet, erhält man das Prozessmodell PM_1 .

Werden Prozessmodelle mit dieser Vorgehensweise rückgerechnet, entstehen, um PM' zu generieren, $n-1$ Prozessmodelle, welche nicht benötigt werden. Des Weiteren werden bei der Rückrechnung Highlevel-Changes angewendet, welche bei einer Transformation von $PM[\sigma\rangle PM'$ keine Effekte mit sich ziehen [RRJK06][SRJR07]. In [RRJK06] werden zwischen drei Typen die sogenannten „Stör“-Operationen unterschieden:

<u>Compensating Change:</u> Compensating Changes entstehen dann, wenn sich Änderungsoperationen gegenseitig aufheben. Dies kann dann der Fall sein, wenn es zwei inverse Change-Logs gibt, wobei in Δ_2^{-1} ein Add von einem Prozessobjekt enthalten ist und in Δ_1^{-1} das selbe Prozessobjekt gelöscht wird. Bei Anwendung von Δ_2^{-1} und Δ_1^{-1} gibt es somit keinen Effekt auf das rückzurechnende Prozessmodell. Die Add- und Delete-Operationen brauchen daher nicht durchgeführt werden, um das Prozessmodell korrekt darzustellen.

Δ_2^{-1} = `ADD('LCR Bericht senden', 'LCR berechnen', 'end'), SWAP(`
`Gewichtung anwenden,`
`Retail/WS-Daten laden)`

Δ_1^{-1} = `DELETE('LCR Bericht senden' 'LCR berechnen', 'end')`

Abbildung 49: Compensating Change

Abbildung 49 zeigt einen Compensating Change. In Δ_2^{-1} wird das Prozessobjekt `LCR berechnen` in das Prozessmodell hinzugefügt und zwischen `LCR berechnen` und `end` eingefügt. Wird nun wie in Δ_1^{-1} angegeben, das Objekt aus dem Modell gelöscht um PM' abzuleiten, braucht man diese zwei Operationen nicht durchführen, da `LCR berechnen` im Prozessmodell PM' nicht mehr vorkommt. Die einzige Operation, welche zwischen PM und PM' durchgeführt werden muss, ist der Swap aus Δ_1 . Die Copy/Delete-Kombination ist daher ein Compensating Change. Weitere Compensating Changes nach [SRJR07] sind:

- ADD/DELETE-Operationen.
- DELETE/ADD-Operationen, wenn die Attribute des wieder hinzugefügte Objektes den Attributen inklusive deren Werten dem des gelöschten Objektes gleichen.
- UPDATE/UPDATE-Operationen, wenn das letzte Update auf einen Attribut-Wert genau dem Wert aus dem t-Model entspricht.

Hidden Change: Wird ein Prozessobjekt aus dem Modell gelöscht und in späteren Versionen wieder eingefügt, kann bei einer Anwendung mehrerer Change-Logs keine Delete/Add-Operation entstehen, sondern ein Move.

Δ_2^{-1} = DELETE('Retail/WS-Daten laden', 'Portfolio laden', 'Report-Positionen anwenden')

Δ_1^{-1} = ADD('Retail/WS-Daten laden', 'Gewichtungen anwenden', 'LCR berechnen')

Abbildung 50: Hidden Change

Das Beispiel aus Abbildung 50 zeigt einen Hidden Change. Zuerst wird das Objekt `Retail/WS-Daten laden` gelöscht und eine Prozessmodellversion später wieder zwischen `Gewichtungen anwenden` und `LCR berechnen` hinzugefügt. Aus diesen beiden Deltas kann ein Move abgeleitet werden: Da man nicht an der Ableitung von PM_2 (welches nur durch Δ_2^{-1} entsteht) interessiert ist, sondern in PM' kann die Delete/Add Kombination zwischen PM und PM' wie ein Kombinationspaar zur Move-Erkennung behandelt werden: Zuerst wird in PM das Objekt `Retail/WS-Daten laden` an der Position zwischen `Portfolio laden` und `Report-Positionen`

anwenden gelöscht, um es dann wieder zwischen Gewichtungen anwenden und LCR berechnen einzufügen. Da nun ein Move abgeleitet ist, muss nur noch eine Change-Operation durchgeführt werden und nicht mehr zwei Operationen. Hidden Changes entstehen auch, wenn eine DELETE/ADD-Operation auftritt und sich das Delete- und Add-Objekt lediglich in den Attribute-Werten unterscheidet. In diesem Fall kann eine Update Operation vom Typ Change durchgeführt werden [SRJR07].

Overriding Change: Entsteht dann, wenn auf einem Prozessobjekt mehrere Change-Operationen durchgeführt werden und jede dieser Change-Operationen die zuvor durchgeführte Operation überschreibt. Die kann zum Beispiel dann entstehen, wenn über mehrere Deltas hinweg ein Attribut Update-Operationen unterliegt.

Δ_2^{-1} = UPDATE „Gewichtungen bestimmen" CHANGE (maxDuration, value, 500, 600)

Δ_1^{-1} = UPDATE „Gewichtungen bestimmen" CHANGE (maxDuration, value, 600, 700)

Abbildung 51: Overriding Change

In Abbildung 51 wird ein Overriding Change auf das Objekt Gewichtungen bestimmen für den Parameter maxDuration gezeigt. Beide Updates aktualisieren den Wert von diesem Attribut.
Jedes dieser Update-Operationen überschreibt den Attribut-Wert. Bei Overriding Changes sind für das zu generierende Prozessmodell nur die letzten Operationen auf dieses Objekt von Interesse. In diesem Fall wird nur der Wert 700 in den Attribut-Wert geschrieben.
Eine weitere Kombination für einen Overriding Change entsteht bei einer MOVE/MOVE-Kombination. Dabei überschreibt der jeweils letzte Move die Position des zuvor ausgeführten Moves. Ebenso wie bei einer UPDATE/UPDATE-Kombination vom Type Change, wird bei einer MOVE/MOVE-Kombination nur der letzte Positionswert übernommen.
Weitere Analysen von [SRJR07] führen noch zusätzlich zwei Kombinationen auf:

- UPDATE/DELETE: Es muss nur das Delete ausgeführt werden.
- ADD/UPDATE: Bei dieser Kombination wir nur das ADD durchgeführt, wobei das Update in ADD inkludiert wird. Demzufolge ist nur noch eine Change-Operation notwendig.

Die beiden Varianten lassen sich statt mit einem Update, auch mit einem Move anwenden. Bei einer MOVE/DELETE-Aktion wird das Objekt gelöscht, bei einer ADD/MOVE-Kombination lediglich das ADD ausgeführt. Jedoch wird das Objekt nicht an die Position des ADDs gesetzt sondern an die Position im MOVE.

Werden diese Stör-Operationen erkannt und ausgeglichen, kann ein minimales Set an Änderungs-Operationen erstellt werden [RRJK06]. Dieses minimale Change-Set σ' enthält nur noch all jene Operationen, welche tatsächlich notwendig sind um aus PM PM' zu generieren. Wird σ' angewendet anstatt σ, kann PM' direkt aus PM mittels eines einzigen inversen Change-Logs abgeleitet werden und es entsteht nur das Prozessmodell PM' in der Rückrechnung.

10.6 Change-Log Bereinigung

Soll nun σ bereinigt werden um dann σ' auf PM anzuwenden, werden die enthaltenen Deltas aus σ nach ihrer chronologischen Anordnung ausgewertet. Werden Stör-Operationen gefunden, wird deren Effekt errechnet und das Ergebnis aus dieser Berechnung als Basis für das Auffinden von Stör-Operationen in den nachfolgenden Deltas herangezogen.

In der Bereinigung von σ werden die Highlevel-Changes Swap und Replace in die jeweiligen Suboperationen aufgeteilt. Dies hat den Vorteil, dass die einzelnen Bestandteile des Highlevel-Changes einfacher untersucht und Effekte auf die enthaltenen Objekte in einem Swap oder Replace unabhängig des anderen Prozessobjektes errechnet werden können. Ein Swap wird dabei in zwei Move-Operationen aufgeteilt und der Replace in eine Add- und eine Delete-Operation.

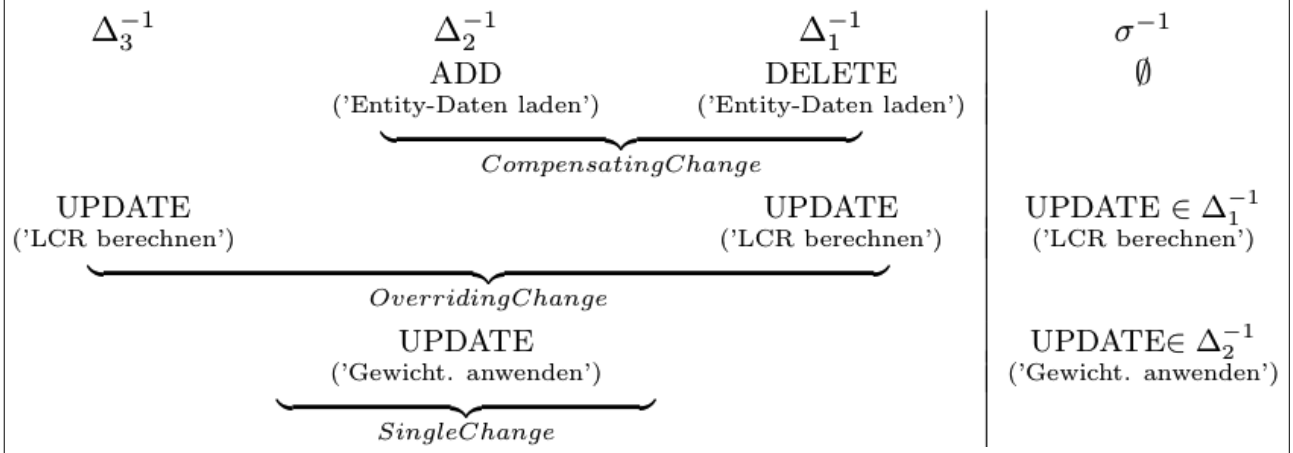
Beispiel 18: Angenommen es existiert eine Sequenz von Change-Logs $\sigma = \Delta_{1,} \Delta_{2,} \Delta_3$ aus welcher das Prozessmodell PM_4 entstanden ist.

Chage-Log-Sequenz $\sigma = \Delta_{1,} \Delta_{2,} \Delta_3$, wobei

$\Delta_1 = ADD('Entity - Daten\ laden'), Update('LCR\ berechnen')\ CHANGE\ maxDuration$

$\Delta_2 = Delete('Entity - Daten\ laden'), Update('Gewichtungen\ anwenden')\ ADD(procCall)$

$\Delta_3 = Update('LCR\ berechnen')\ CHANGE$ maxDuration



$\sigma' = UPDATE('LCR\ berechnen') \in \Delta_{1,}^{-1}\ UPDATE(y) \in \Delta_2^{-1}$

Abbildung 52: Einfache Rückrechnung mehrerer Prozessmodelle.

Die Abbildung 52 zeigt eine Rückrechnung des Prozessmodells PM_4 auf das Prozessmodell PM_1. Eine Rückrechnung ist möglich, da alle Deltas von PM_1 bis PM_4 vorhanden sind. Zuerst werden wieder die inversen Deltas gebildet: Für Δ_1 entsteht

$$\Delta_1^{-1} = DELETE('Entiy - Daten\ laden'), UPDATE('LCR\ berechnen')\ CHANGE\ "maxDuration"$$

Aus dem Change-Log Δ_2 wird das inverse Change-Log

$$\Delta_2^{-1} = ADD('Entity - Daten\ laden'), UPDATE('Gewichtungen\ anwenden')\ DELETE\ "procCall"$$

und aus Δ_3 $\Delta_3^{-1} = UPDATE('LCR\ berechnen')\ CHANGE\ "maxDuration"$.

Die inversen Operationen für jedes inverse Delta sind in den Spalten der Tabelle in Abbildung 52 gelistet. Für jedes Objekt in den Change-Logs wird dabei eine eigene Zeile erstellt. Anschließend kann, beginnend bei Δ_3^{-1}, untersucht werden, ob Stör-Operationen existieren:

Das Prozessobjekt `Entity-Daten laden` hat in Δ_2^{-1} eine Add-Operation und in Δ_1^{-1} eine Delete-Operation. Da eine Add/Delete-Kombination ein Compensating Change ist, werden die beiden Operationen nicht durchgeführt – sie haben keinen Effekt bei der Rückrechnung von PM_4 nach PM_1.

Auf dem Prozessobjekt `LCR berechnen` wurden zwei Updates auf das Attribut `maxDuration` durchgeführt. In Δ_3^{-1} wird das Attribut zurück auf den Wert von PM_3 gesetzt und in Δ_1^{-1} auf den ursprünglich gesetzten. Werden zwei Updates auf das gleiche Objekt und dem gleichen Attribut durchgeführt, überschreibt das zuletzt durchgeführt Update – welches in Abbildung 52 Δ_1^{-1} ist – alle vorhergehenden Updates. In diesem Fall entsteht ein Overriding Change. Hier muss nur das Update aus Δ_1^{-1} durchgeführt werden.

Für das Objekt `Gewichtungen anwenden` gibt es in dieser Sequenz nur eine Change-Operation in Δ_2^{-1}. Demzufolge hat diese eine Operation eine Auswirkung auf das Prozessmodell PM_1 und kann daher nach σ' direkt übernommen werden.

Wurden alle Zeilen in der Tabelle aus Abbildung 52 auf Stör-Operationen untersucht, erhält man σ'. Wird nun $PM_4[\sigma' \rangle PM_1$ angewendet, werden nur noch zwei Change-Operationen ausgeführt anstatt fünf. Aus σ' ist zu entnehmen, dass nur diese zwei Update-Operationen einen Effekt bei der Rückrechnung auf PM_1 haben.

Beispiel 19: Angenommen es existiert eine Sequenz von Change-Logs $\sigma = \Delta_{1,} \Delta_{2,} \Delta_{3,} \Delta_4$ aus welchen ein Prozessmodell PM_5 entstanden ist. In diesem Beispiel soll ein Prozessmodell PM_1 abgeleitet werden.

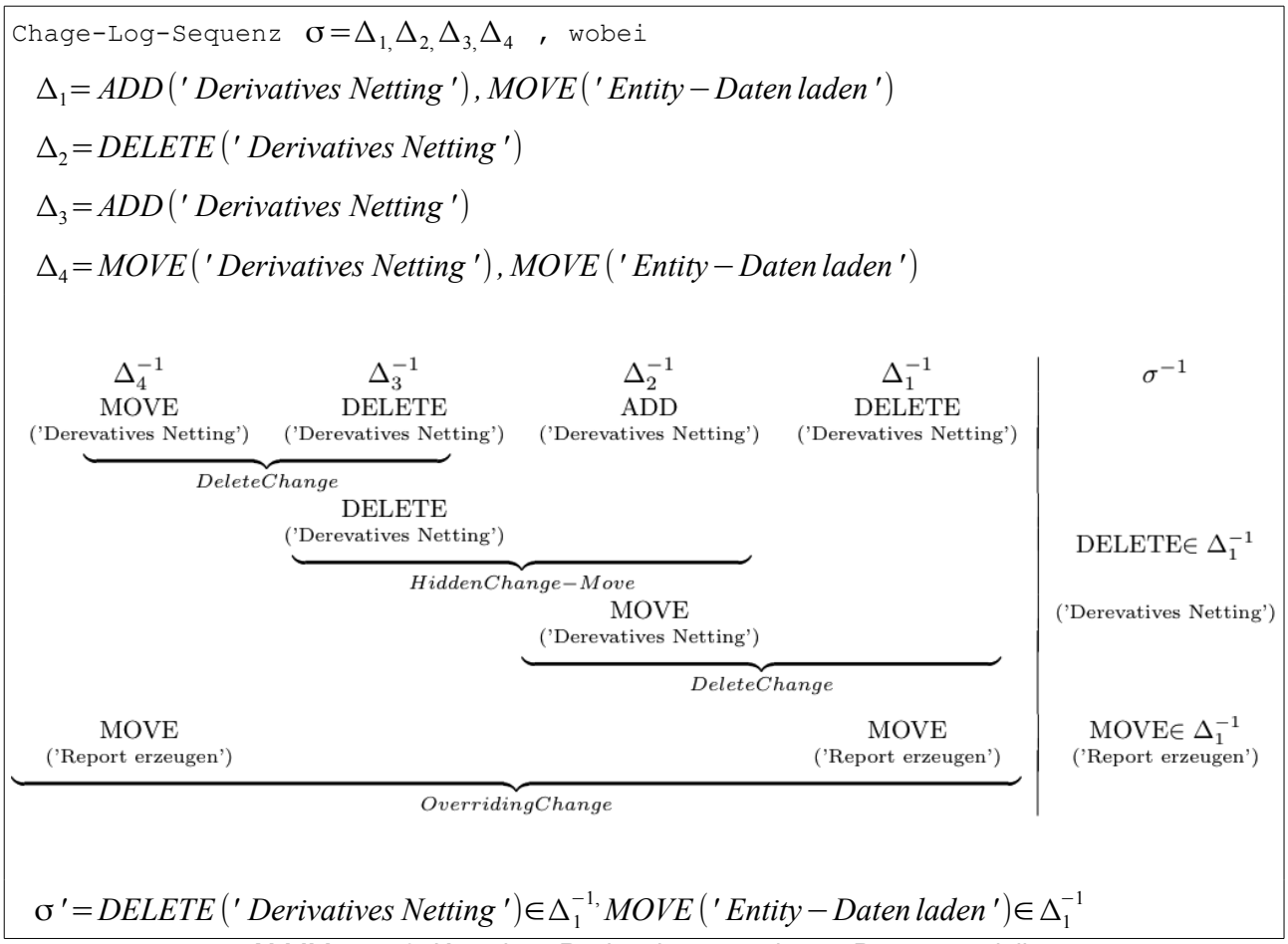


Abbildung 53: Komplexe Rückrechnung mehrerer Prozessmodells

In Abbildung 53 ist eine Bereinigung einer Rückrechnung zu sehen in der, abhängig von der zuletzt errechneten Bereinigung zwischen zwei Deltas, das Ergebnis der nachfolgenden Bereinigung beeinflusst wird.

Wieder werden die inversen Deltas erstellt und die Change-Operationen aus allen Change-Logs für ein Objekt in einer Zeile zusammengefasst. Die erste Zeile in der Tabelle aus Abbildung 53 behandelt das Prozessobjekt `Derivatives Netting`. Um aus PM_5 wieder PM_1 zu erstellen, müssen daher vier Operationen durchgeführt werden. Nach einer Bereinigung braucht nur noch eine Operation durchgeführt werden. Dabei beginnt die Analyse mit dem Move aus Δ_4^{-1} und dem Delete aus Δ_3^{-1} . Eine Move/Delete-Kombination ergibt einen Delete-Change, da der zuvor durchgeführte Move

nicht durchgeführt werden muss, wenn das Objekt anschließend gelöscht wird. Das Ergebnis wird dann mit dem Add aus Δ_2^{-1} konsolidiert. Wird ein Objekt zuerst aus dem Prozessmodell gelöscht, um es dann wieder hinzuzufügen, kann aus dieser Kombination ein Update oder ein Move errechnet werden. In Abbildung 53 wurde ein Move errechnet, da sich die Positionen geändert haben (hierfür siehe Anhang). Aufbauend auf dem Hidden Change – Move wird noch Δ_1^{-1} herangezogen. Hier muss noch ein Delete durchgeführt werden. Da nun wieder eine Move/Delete-Kombination entstanden ist, braucht man, wie schon in der ersten Analyse zu Prozessobjekt `Derivatives Netting`, nur das Delete übernehmen. Diese Delete-Operation wird dann auch nach σ' übernommen.

Weiters existieren in Abbildung 53noch zwei Move-Operationen in Δ_4^{-1} und Δ_1^{-1} für das Prozessobjekt `Report erzeugen`. Zwei aufeinander folgende Move-Operationen werden als Overriding Changes behandelt, weshalb nur die Move-Operation aus Δ_1^{-1} angewendet.

In Abbildung 53 konnte daher, durch eine Bereinigung der Change-Logs $\sigma' = DELETE(Derivatives\ Netting), MOVE(Report\ erzeugen)$ erstellt werden. Bei der Durchführung von $PM_5[\sigma' \rangle PM_1$ brauchen daher nur noch zwei Operationen ausgeführt werden, anstatt sechs bei $PM_5[\sigma \rangle PM_1$.

Beispiel 20: Angenommen es existiert eine Change-Log-Sequenz $\sigma = \Delta_{1,}\Delta_2$ und ein Prozessmodell PM_3 . Aus der Change-Log-Sequenz σ wurde PM_3 erstellt. Nun soll wieder PM_1 erstellt werden.

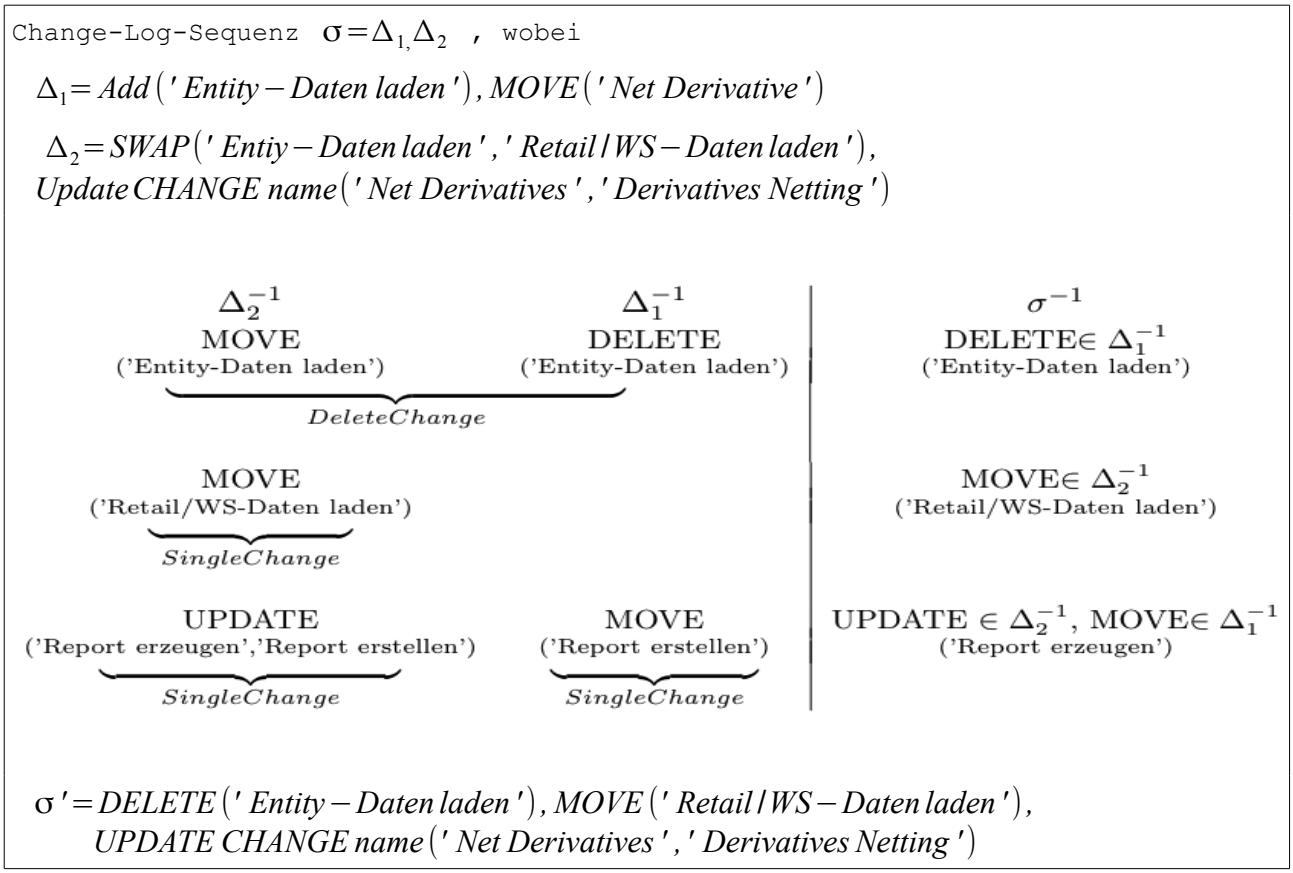


Abbildung 54: Aufteilung eines Swaps und Name-Update bei einer Rückrechnung

Die Abbildung 54 zeigt die Aufteilung eines Swaps in zwei Move-Operationen und ein Update auf den Namen eines Prozessobjektes.

In Δ_2^{-1} existiert ein Swap zwischen den beiden Prozess-Objekten `Entity-Daten laden` und `Retail/WS-Daten laden`. Da einem Swap zwei Move-Operationen zu Grunde liegen, wird dieser in die zwei Moves aufgeteilt. Wie schon zuvor beschrieben, zieht dies eine Erleichterung in der Change-Log-Bereinigung mit sich.

Nachdem die Aufteilung erfolgt, kann nun das Prozessobjekt `Entity-Daten laden` analysiert werden. Aus der Bereinigung zwischen Δ_2^{-1} und Δ_1^{-1} für Prozessobjekte `Entity-Daten laden` ergibt sich eine Delete-Operation.

Da Prozessobjekt `Retail/WS-Daten laden` in der Change-Log-Sequenz nur in der Swap-Operation und im weiteren Sinne in der Move-Operation vorhanden ist, kann der

Move für die Rückrechnung auf PM_1 direkt übernommen werden.

Einen Spezialfall bildet eine Update-Operation vom Typ Change, wenn der Name des Prozessobjektes geändert wurde, da das Prozessobjekt auf Grund der Namensänderung nicht mehr der entsprechenden Zeile zugeordnet werden kann. Um dennoch das Prozessobjekt richtig zuzuordnen, wird bei einer Namensänderung, mit dem ursprünglichen Namen des Prozessobjektes in den Change-Logs weitergesucht. Da die Namensänderung im persistenten Update-Change abgelegt ist, können über diese Verbindung nachfolgende Changes für dieses Objekt gefunden werden. In Abbildung 54 ist dieser Sachverhalt abgebildet. Hier wird in Δ_2^{-1} das Prozessobjekt `Net Derivatives` auf `Derivatives Netting` umbenannt. Wird über diese Verbindung im Change-Log Δ_1^{-1} mit dem Namen `Report erzeugen` für das Prozessobjekt 'Report erzeugen' gesucht, kann eine weitere Move-Operation gefunden werden. Nachdem die Zuordnung erfolgte, kann eine Bereinigung durchgeführt werden. Da ein Update und eine Move-Operation einen Effekt auf das Prozessmodell PM_1 haben, werden beide Operationen bei einer Rückrechnung durchgeführt.

11 Evaluierung

11.1 Evaluierung der Differenzenerkennung

Die Konzepte der Differenzenerkennung wurden zur Gänze in der Programmiersprache Java implementiert. Um die Unabhängigkeit der Algorithmen zu testen, wurden zwei unterschiedliche Modellierungsnotationen verwendet. Die untersuchten Geschäftsprozesse stammen aus dem Bankensektor und dienen zur Berechnung einer Basel-III relevanten Kennzahl. Um alle Konzepte der Differenzenerkennung zu testen, mussten jedoch manche Modelle geringfügig erweitert werden. Da, in den zur Verfügung gestellten Prozessmodellen, sonst nicht alle Konzepte und Highlevel-Changes getestet werden konnten.

Es zeigte sich, dass bei einer Ähnlichkeitsanalyse, welche nur auf der Semantik der Prozessobjekte beruht, es zu einer besseren Highlevel-Change Bestimmung kommen kann, je genauer ein Prozessobjekt im Modell beschrieben wird. Eine Einbeziehung der Attributmenge als auch Attributwerte eines Prozessobjektes hatte daher positive Auswirkungen auf die Ähnlichkeitsanalyse. Werden „Default“-Werte in Attributen entdeckt und entsprechend behandelt, beeinflussen diese die Berechnung ebenfalls positiv.

Die Erkennung der Change-Primitives mit einem anfänglich textbasierten Ansatz zeigte sich als vorteilhaft, da die Ergebnisse als eine Art „Fallback“ herangezogen werden konnten, wenn Prozessobjekte zu wenig Informationsgehalt aufwiesen um eine starke Ähnlichkeit zu bestimmen. Existierten solche Prozessobjekte konnte dann, aufbauen auf den Ergebnissen der LCS, zumindest die Highlevel-Changes ADD und DELETE abgeleitet werden. Diese Vorgehensweise garantiert somit, dass, selbst bei wenig Semantik der Prozessobjekte, alle Differenzen zwischen zwei Prozessmodellen gefunden werden.

Der mengenorientierte Ansatz zur Bestimmung der Highlevel-Changes baut auf vier Grundelemente auf: ADD, DELETE, UPDATE und MOVE. Diese Änderungsoperationen finden sich in allen weiteren Highlevel-Changes wieder. In der Persistierung wird dies besonders deutlich, da die Highlevel-Changes SWAP und REPLACE lediglich als eine Art Aggregation aus diesen Grundelementen dargestellt werden. Diese Sichtweise der

Highlevel-Changes erwies sich bei der Erstellung eines minimalen Change-Sets als nützlich.

Zusätzlich wird zu jeder Änderung eines Prozessobjektes die Positionsnummer im Change-Log abgelegt. Die Berücksichtigung der Positionsnummern in den Change-Logs hatte den Vorteil, dass bei einer Rückrechnung eines Prozessmodells auf eine Vorgängerversion, keine Vorgänger-Nachfolger-Beziehungen berücksichtigt werden mussten, sondern die Postion im Prozessmodell. Wurden die Vorgänger-Versionen mit Positionsnummern abgeleitet, konnten Abhängigkeiten zwischen Change-Operationen umgangen werden und eine Änderungsoperation konnte unabhängig von allen anderen Änderungsoperationen durchgeführt werden.

Die durchzuführenden Highlevel-Changes können sich jedoch unterscheiden, wenn ein Prozess in zwei verschiedenen Prozessmodellnotationen erstellt worden ist. Dies liegt jedoch an der internen Repräsentation der einzelnen Sprachen und wie Elemente, insbesondere in Daten- und Ressourcen-Perspektive, einem Prozessobjekt in der Kontrollfluss-Perspektive zugeordnet werden. Unterschiede entstehen dann, wenn die Prozessmodellierungssprache eigene Objekte für die Daten- und Ressourcen-Perspektive unterstützt oder ob diese Perspektiven in den Attributen der Prozessobjekte angeführt werden. Ebenfalls einstanden Abweichungen in den erkannten Highlevel-Changes wenn Modellierungssprachen mit einer unterschiedlichen Granularität die Prozessobjekte beschreiben. So konnte zum Beispiel in einer Modellierungssprache ein UPDATE auf ein Objekt erkannt werden, während in der zweiten Modellierungssprache eine ADD- und DELETE-Aktion bei selbigen Objekt durchgeführt wurde. In der zweiten Modellierungssprache konnte kein UPDATE erkannt werden, da nicht genügend Attribute vorhanden waren um entsprechende Ähnlichkeit zu errechnen.

Arbeiten aus diesem Bereich unterstellen für die Prozessobjekte eine eindeutige ID der Prozessobjekte um Unterschiede zwischen zwei Prozessmodellen zu erkennen. Dies schränkt eine allgemein gültige Differenzenerkennung stark ein. Mit den erstellten Methoden konnte diese Beschränkung aufgehoben werden. Erstmals können also auch Differenzen in Prozessmodellen errechnet werden, auch wenn die interne Repräsentation der Prozessmodelle keine eindeutigen IDs aufweisen.

[KGFE08] verwenden zur Move-Erkennung Fragmentnamen um Verschiebungen zwischen den Fragmenten zu erkennen. Auch hier stellt sich wieder ein Problem der automatischen Erkennung der IDs der Fragmente beziehungsweise wie diese vergeben werden. Mittels der Pfadkonsolidierung im Process-Structure-Tree werden keine Fragment-IDs mehr benötigt. Die Rückrechnung erfolgt über einen komplett neuen Ansatz, nämlich über die Positionsnummern. Es werden die Pfade des Process-Structure-Trees so weit verändert, wie es bis zu dem zu untersuchenden Prozessobjekt Änderungsoperationen gegeben hat. Dabei werden – um die Pfade zu konsolidieren – die inversen Änderungsoperationen angegeben. Über die Anwendung der inversen Operationen kann so der Vorgänger-Teilbaum bis zu diesem Prozessobjekt komplett rekonstruiert werden. Anschließend konnten dann die konsolidierte Position mit der alten Position verglichen werden. Stimmten die beiden Positionsnummern nicht überein, konnte so ein Move erkannt werden, da sich, selbst durch das „Rückgängig machen“ aller Änderungsoperationen, das Prozessobjekt an einer anderen Position befindet.

Insgesamt konnten mit den entwickelten Konzepten in dieser Master-Thesis 10 von 14 Struktur-Patterns aus [WBRR08] abgeleitet werden, wobei sich zusätzlich einige Highlevel-Changes auch zu Fragmenten zusammenfassen lassen.

11.1.1 Programmdarstellung

Der folgende Abschnitt zeigt die Umsetzung des Prototyps zur Differenzenerkennung. Die Implementierung beinhaltet die Verarbeitung des Meta-Modells in den Process-Structure-Tree, die Erstellung der flachen Struktur des Process-Structure-Trees, die Erkennung der Change-Primitives mittels der LCS, die Ähnlichkeitsanalyse zwischen Prozessobjekten, sowie die Transformation der Change-Primitives in Highlevel-Changes und die Serialisierung der Highlevel-Changes.
Für die Eingabe der zu vergleichenden Modelle, der Ausgabe von Informationen aus der Ähnlichkeitsanalyse als auch die Transformation von Change-Primitives zu Highlevel-Changes wird die Standard-Textkonsole verwendet. Um die Differenzen und Ähnlichkeiten zu errechnen, benötigt man lediglich zwei Prozessmodelle, welche bereits in das vorgeschlagenen Meta-Modell transformiert worden sind. Die komplette Abarbeitung erfolgt dann völlig automatisch.
Die folgende Abbildung zeigt eine Ausgabe der Ähnlichkeitsanalyse. Dabei werden die einzelnen Ähnlichkeiten zwischen den Attributen ausgegeben sowie die Gesamtähnlichkeit

der beiden Prozessobjekte „Report senden“ und „Report versenden“. Die genaue Berechnung der Ähnlichkeitswerte ist in Abbildung 33 auf Seite 81 aus dem Kapitel „Ähnlichkeitsanalyse“ angeführt.

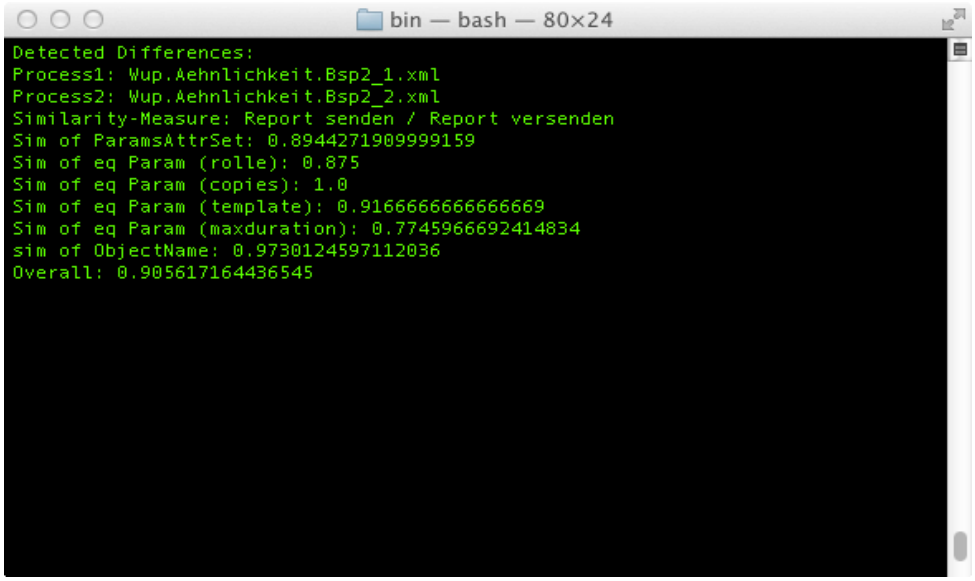


Abbildung 55: Programmdarstellung - Ähnlichkeitsanalyse

In der nächsten Abbildung sind die Highlevel-Changes der Differenzenerkennung zwischen zwei Prozessmodellen zu sehen. Die Ausgabe der Highlevel-Changes stammt aus der Berechnung des Beispiels in Abbildung 39 auf Seite 97 aus dem Kapitel „Von Change-Primitives zu Highlevel-Changes“.

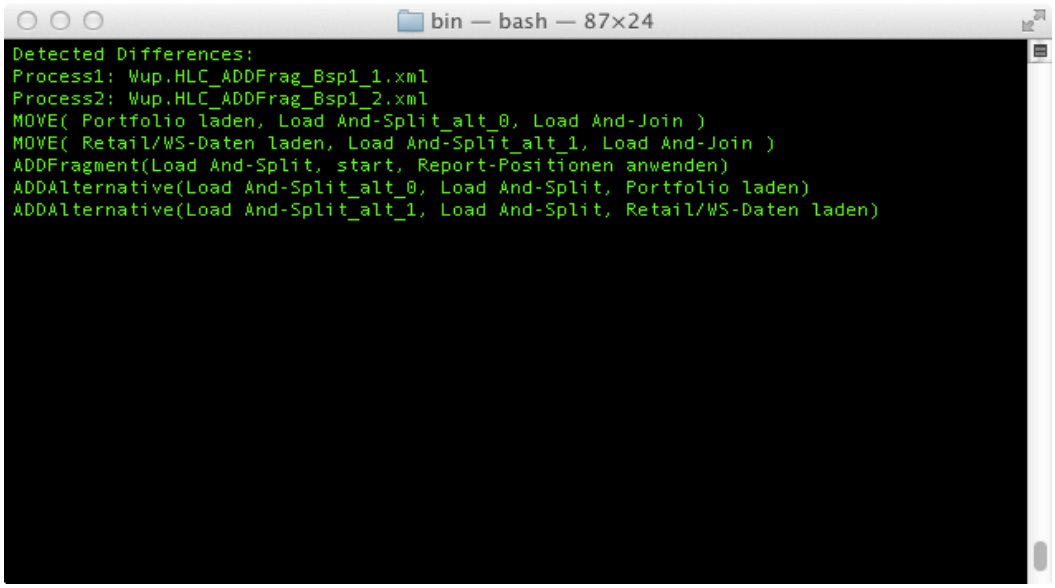


Abbildung 56: Programmdarstellung - Highlevel-Changes

11.2 Evaluierung der Versionskontrolle

Betrachtet man Arbeiten zum Thema Versionskontrolle wie [ZXLC07] oder [BHC07], dann zeigt sich, dass sich diese lediglich auf ein beschränktes Set an Änderungsoperationen in den Change-Logs beziehen. Zwar ließen sich die in dieser Arbeit erstellten Highlevel-Changes in die von den Autoren vorgeschlagenen Änderungs-Operationen überführen, werden jedoch die abgeleiteten Highlevel-Changes im Versionskontrollsystem angewendet, entfallen Konvertierungen in dieses Set.

In der Differenzenerkennung wurden die Operationen SWAP und REPLACE speziell behandelt, indem in der Persistierung für SWAPS zwei MOVES und für REPLACE ein ADD und ein DELETE zusammengefasst wurden. Diese Zusammenfassung stellte sich bei der Rückrechnung von Prozessmodellen auf deren Vorgängerversionen als nützlich heraus, da ein SWAP und REPLACE einfach auf die entsprechenden einzelnen Operationen aufgebrochen werden konnten. Man benötigte bei der Errechnung des minimalen Change-Sets keine komplexe Abarbeitung der verknüpften Objekte. Jedes Prozessobjekt konnte dadurch autonom berechnet werden.

Auch wenn bei der Aufteilung der REPALCE und SWAP-Operationen kurzfristig mehr Highlevel-Changes erzeugt wurden, konnte in dieser Arbeit gezeigt werden, dass mit dieser Vorgehensweise dennoch ein minimales Change-Set abgeleitet werden kann.

Des weiteren wurde durch das Einführen von „Offsprings“ im Versionsbaum eine Möglichkeit geschaffen, Prozessmodelle aus ihrer Hauptentwicklungslinie herauszulösen und völlig unabhängig von dieser weiterzuentwickeln. Dies bietet einerseits mehr Flexibilität im Customizing von Geschäftsprozessen andererseits können auch so genannte experimentelle Branches, wie sie in Versionsverwaltungssystemen wie SVN und Git existieren, erstellt werden.

Um nicht nur die Vorgänger-Beziehungen von einer Version auf die nächste Version abzubilden, sondern auch die Art der Entstehung des Prozessmodells zu zeigen. Diese unterschiedlichen Repräsentationen werden auch in der Softwareentwicklung eingesetzt [CRWB98]. Hierfür wurden zwei unterschiedliche Kantentypen eingeführt. Ein Katentyp zeigt, dass ein Prozessmodell ein direkter Nachfolger von einem Prozessmodell ist und sich somit direkt aus dem Vorgänger entwickelt hat. Der andere Kantentyp zeigt, dass ein

Prozessmodell durch eine Zusammenführung aus mehreren Prozessmodellen durch einen sogenannten Merge, wie ihn [BCRS13] beschreibt, entstanden ist. Durch Einführung von Merge-Kanten müssen zwar genau so viele Vergleiche gerechnet werden wie Vorgänger-Prozessmodelle an diesem Merge beteiligt waren, jedoch garantiert dies eine vollständige Rückrechnung zu allen Vorgänger-Modellen.

12 Schlussfolgerung

Im nachfolgenden werden die Ergebnisse dieser Master-Thesis zusammengefasst. Anschließend folgen eine Diskussion über die verwendeten Konzepte sowie ein Ausblick auf weitere Forschungsmöglichkeiten hinsichtlich eines Versionskontrollsystems von Geschäftsprozessen. Die zu Beginn gestellten Forschungsfragen konnten alle beantwortet werden und somit einen wichtigen Beitrag zur Konstruktion eines Versionsverwaltungssystems für Geschäftsprozesse liefern. In dieser Arbeit konnten – kurzgefasst – die folgenden Punkte erreicht werden:

- Eine automatische Differenzenerkennung von blockstrukturierten Geschäftsprozessmodellen. Diese wurden implementiert, um die Korrektheit der Konzepte zu prüfen.
- Es konnte eine Lösung gefunden werden, welche auch die Ressourcen und Daten-Perspektive berücksichtigt.
- Es wurde eine Ähnlichkeitsanalyse entwickelt, welche auf einer Analyse der Semantik der Prozessobjekte beruht.
- Es wurde ein geeignetes Versionskontrollsystem-Modell gefunden, sowie eine passende Form um die Highlevel-Changes so ablegen zu können, dass diese zur Wiederherstellung der Vorgänger-Prozessmodellen eingesetzt werden können.
- Es konnte ein minimales Change-Set zwischen Prozessmodellen errechnet werden, wobei die Errechnung des Change-Sets alle abgeleiteten Highlevel-Changes beinhalten kann.
- Um autonom von der Prozessmodellnotation zu sein und um eine Datenstruktur erstellen zu können, welche die Differenzenerkennung und Rückrechnung von Prozessmodellen unterstützt, wurde ein Meta-Modell erstellt. Das Meta-Model selbst entstand vornehmlich aus Analysen der Modellierungsnotationen BPNM, EPK sowie aus den Workflow-Engines CPEE-Cockpit, UC4 und AristaFlow.
- Durch die vollständige Automatisierung der Differenzenerkennung, kann dennoch garantiert werden, dass aufgrund eines hybriden Ansatzes, bei welchem textbasierte als auch mengenorientierte Verfahren angewendet werden, alle Änderungen zwischen zwei Prozessmodellen gefunden werden.

12.1 Ausblick

In dieser Diplomarbeit wurden neue Methodiken zur Differenzenerkennung entwickelt und innerhalb des Abschnittes für die Versionskontrolle von Geschäftsprozessmodellen wurden bestehende Ansätze erweitert und mit den Ergebnissen aus der Differenzenerkennung verknüpft. Der Fokus lag dabei auf einer Hersteller-unabhängigen Konstruktion der Konzepte. Dennoch ergeben sich aus dieser Arbeit weitere Anknüpfungspunkte, um das vorgeschlagene Gesamtsystem weiterzuentwickeln. Daher werden in diesem Abschnitt Anregungen zu weiteren Forschungen gegeben.

In Kapitel „Vereinheitlichung von Prozessmodellen“ wird ein Meta-Modell vorgestellt. Dieses Modell beruht auf Analysen von ausgewählten Prozessnotationen als auch Workflow-Engines. Eine interessante Fragestellung wäre es, inwieweit das Meta-Modell für andere Modellierungssprachen genutzt werden kann. Zu denken wäre hier an Petri-Netze, Aktivitäten-Diagramme oder an Prozessmodelle welche keine Blockstrukturierung aufweisen.

Der Process-Structure-Tree lässt sich in dieser Arbeit einstweilen nur mit blockstrukturierten Geschäftsmodellen erstellen. [VJVK09] beschreiben einen Ansatz wie nicht-blockstrukturierte Prozessmodelle in einen Process-Structure-Tree überführt werden können. Eine Berücksichtigung von nicht-blockstrukturierten Prozessmodellen würde das Spektrum der Versionskontrolle wesentlich erweitern.

Die textbasierte Erkennung der Change-Primitives im Kapitel vier beruht auf dem Konzept der Erkennung der Longest-Common-Subsequence einer Zeichenkette, wobei die Zeichenkette die Anreihung der Prozessobjekte der beiden zu vergleichenden Prozessmodelle darstellt. Es zeigte sich, dass durch die gewählte Vorgehensweise nicht immer das minimalste Set an Change-Primitives erstellt werden konnte, wodurch mehr Vergleiche zwischen Add- und Delete-Operationen im Nachgang durchgeführt werden müssen. Hier wäre eine Optimierungsmöglichkeit gegeben in dem der Editier-Graph ähnlich wie bei [MYER86] erstellt wird.

Im Abschnitt zur Ähnlichkeitsanalyse wird eine vereinfachte Variante der Kosinus-Simularität verwendet. Obwohl dieses Verfahren für die Ähnlichkeitsanalyse von Prozessobjekten gute Resultate erzielte, könnte das Verfahren noch verfeinert werden,

indem zum Beispiel eine gewichtete Kosinus-Simularität, wie sie [LBHL13] vorschlagen, berücksichtigt wird. Des weiteren werden nur semantische Aspekte berücksichtigt. Eine interessante Fragestellung wäre es, wie diese semantische Evaluierung mit strukturbasierten Ansätzen, wie eine Nachbarschaftsanalyse, erweitert werden könnte und welche Resultate mit einer Vereinigung der beiden Ansätze erzielt werden können.

In Kapitel sechs wird die Herleitung der Highlevel-Changes aus Change-Primitives beschrieben. In dieser Arbeit wurde auch die Daten- und Ressourcen-Perspektive berücksichtigt. Jedoch beschränken sich die Highlevel-Changes in diesen zwei Perspektiven nur auf Add-, Delete und Update-Operationen. Um eine stärkere Aussagekraft über die Änderungen in diesen Perspektiven zu erzielen, könnten auch hier weitere Highlevel-Changes wie sie in der Kontrollfluss-Perspektive beschrieben sind definiert und umgesetzt werden.

Die Highlevel-Changes werden in den Change-Logs sequenziell abgelegt. Auch hier könnte die Qualität erhöht werden, in dem diese Changes in den Change-Logs zu Gruppen zusammengefasst werden um die Änderungen in einzelnen Fragmenten stärker herauszuheben. Zur Erstellung von Gruppen von Änderungsoperationen könnte der bereits verwendete Process-Structure-Tree herangezogen werden [KGFE08].

Im Versionsgraphen wird in den verschiedenen Ästen immer die letzte Version eines Prozessmodells vollständig abgelegt. Hier wird, vor allem bei größeren Geschäftsprozessen, noch viel Speicherplatz verbraucht. Es könnten Anstrengungen getätigt werden, um auch für das jeweils letzte Prozessmodell eines Astes nur die Deltas zu speichern und lediglich das letzte Prozessmodell in der Hauptentwicklungslinie zu speichern.

In dieser Arbeit wurde lediglich auf die Erzeugung der Highlevel-Changes sowie das Ein- und Auschecken eines Prozessmodells eingegangen. Ein weiterer logischer Schritt zur Vervollständigung wäre es, eine Visualisierung der Versionsgraphen zu erstellen. Im Zuge dessen, könnten auch die Änderungen zwischen zwei Prozessmodellen grafisch unterlegt werden [KFKF12]. Ebenso notwendig wäre es, das vorgeschlagene Versionskontrollsystem mit einer Prozessmodellvereinigung (Merging) zu implementieren wie es [BCSR13] vorschlägt. Sowohl das Merging von Prozessmodellen und die Visualisierung innerhalb des Versionskontrollsystems ist bereits ein Standard-Feature in

Versionsverwaltungssystemen wie SVN oder Git, welche hauptsächlich zur Verwaltung von Sourcecode dienen.

Insbesondere beim Mapping der Prozessmodelle in das Meta-Modell wäre es sinnvoll von einer manuellen Mapping-Erstellung wegzukommen und hin zu einer Automatisierung eines solchen Mappings. Es muss noch viel manueller Aufwand betrieben werden um die spezielle Modellierungsnotation in das Meta-Modell zu überführen. Eine solche Lösung könnte dabei den Prozess vom Einchecken des tatsächlichen Geschäftsprozesses bis hin zu einem späteren Auschecken vollständig automatisieren, was in weiterer Folge Fehler durch manuelle Eingriffe reduzieren würde.

13 Literaturverzeichnis

[AADM00]: Agostini, A. & De Michelis, G. (2000): *Improving flexibility of workflow management systems*. In: Business Process Management, LNCS 1806, pp. 218-234. Springer-Verlag.

[AALS98]: van der Aalst, W. (1998): *The application of Petri nets to workflow management*. In: Journal of circuits, systems, and computer, Vol. 8, No. 1, pp. 21-66. World Scientific.

[AEHR05]: Russel, N. & van der Aalst, W. & ter Hofstede, A. & Edmond, D. (2005): *Workflow resource patterns: Identification, representation and tool support.* In: Advanced Information Systems Engineering, LNCS 3520, pp. 216-232 Springer-Verlag.

[AHKB03]: van der Aalst, W. & ter Hofstede, A. & Kiepuszewski, B. & Barros, A. (2003): *Workflow Patterns*. In: Distributed and parallel databases. Vol. 14. No. 1. pp. 5-51. Springer Verlag.

[ATDA98]: Atkins, D. (1998): *Version sensitive editing: Change history as a programming tool*. In: System Configuration Management, LNCS 1439, pp. 146-157. Springer-Verlag.

[BAAO10]: Alex, B. & Onysko, A. (2010): *Zum Erkennen von Anglizismen im Deutschen: der Vergleich einer automatischen und einer manuellen Erhebung.* In: Strategien der Integration und Isolation nicht-nativer Einheiten und Strukturen, pp. 223ff, De Gruyter.

[BCBS14]: Basel Comittee on Banking Supervision (2014): Liquidity coverage ratio disclosure standards. http://www.bis.org/publ/bcbs272.pdf . Zuletzt aufgerufen: 17. April 2014. Bank for International Settlements.

[BCRS13]: Böhmer, K. & Rinderle-Ma, S. (2013): *Difference-Preserving Process Merge*. In: OTM 2013 Workshop, LNCS 8186, pp.718-721. Springer-Verlag.

[BHCB07]: Bae, H. & Cho, E. & Bae, J. (2007): *A version management of business process models in BPMS*. In: Advances in Web and Network Technologies, and Information Management, LNCS 4537, pp. 534-539, Springer-Verlag.

[BLHR00]: Bergroth, L. & Hakonen, H. & Raita, T. (2000): *A survey of longest common subsequence algorithms.* In Proc. of 7th International Symposium on String Processing and Information Retrieval. SPIRE 2000. pp. 39-48. IEEE Computer Society.

[BPDU94]: Bernstein, P. & Dayal, U. (1994): *An overview of repository technology*. In: VLDB '94. pp. 705-713. ACM.

[BUMO06]: Bunescu, R. & Mooney, R. (2006): *Subsequence kernels for relation extraction*. In: Advances in neural information processing systems, Vol. 18, pp. 171–179. MIT-Press.

[CBKO92]: Curtis, B. & Kellner, M. & Over, J. (1992): *Process modeling*. In: Communications of the ACM, Vol. 35, No. 9, pp. 75-90. ACM.

[CLRS09]: Cormen, T. & Leiserson, C. & Rivest, R. & Stein, C. (2009): *Introduction to Algorithms*. 3. Auflage. MIT Press.

[CRWB98]: Conradi, R. & Westfechtel, B. (1998): *Version models for software configuration management*. In: ACM Computing Surveys (CSUR), Vol. 30, No. 2, pp. 232-282. ACM.

[DDDK11]: Dijkman, R. & Dumas, R. & v.Dongen, B. & Käärik, R. & Mendling, J. (2011): *Similarity of business process models: Metrics and evaluation*. In: Information System, Vol. 36, No. 2, pp. 498 – 516. Elsevier Ltd.

[DELP03]: BPM Delphi (2003): *Market Milestone Report*. Delphi.

[ECKR95]: Ellis, C. & Keddara, K. & Rozenberg, G. (1995): *Dynamic change within workflow systems.* In Proc.: of Conference on Organizational computing systems. pp. 10-21. ACM.

[ERHF11]: Ekanayake, C. & La Rosa, M. & ter Hofstede, A. & Fauvet, M. (2011): *Fragment-based version management for repositories of business process models*. In: On the Move to Meaningful Internet Systems, OTM 201, LNCS 7044, pp. 20-37. Springer-Verlag.

[GBLU08]: García-Bañuelos, L. (2008): Pattern *Identification and Classification in the Translation from BPMN to BPEL*. In: On the Move to Meaningful Internet Systems, OTM 2008, LNCS 5331, pp. 436-444. Springer-Verlag.

[GRRA06]: Günther, C. & Rinderle, S. & Reichert, M. & Van Der Aalst, W. (2006): *Change mining in adaptive process management systems*. In: On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, LNCS 4275, Vol. 4275, pp. 309 – 326. Springer-Verlag.

[GTKW08]: Gwschind, T. & Köhler, J. & Wong, J. (2008): *Applying patterns during business process modeling*. In: Business Process Management, LNCS 5240, pp. 4-19. Springer Verlag.

[GTKW08]: Gschwind, T. & Koehler, J. & Wong, J. (2008): *Applying patterns during business process modeling*. In: Business Process Management. LNCS 5250, pp. 4-19, Springer-Verlag.

[GWCY92]: Gale, W. & Church, K. & Yarowsky, D. (1992): *One sense per discourse*. In: Proc.: The workshop on Speech and Natural Language, pp. 233 – 237, Association for Computational Linguistics.

[HFKV06]: Hauser, R. & Friess, M. & Kuster, J. & Vanhatalo, J. (2006): *Combining Analysis of Unstructured Workflows with Transformation to Structured Workflows*. In: Enterprise Distributed Object Computing Conference, 2006 (EDOC '06). pp 129-140. IEEE Computer Society.

[HIDA77]: Hirschberg, D. (1977): *Algorithms for the longest common subsequence problem*. In: Journal of the ACM. Vol. 24. No. 4. pp. 664-675. ACM.

[HLBB13]: Heidari, F. & Loucopoulos, P. & Brazier, F. & Barjis, J. (2013): *A Meta-Meta-Model for Seven Business Process Modeling Languages*. Inprocedings of 15th Conference on Business Informatics (CBI). pp. 216-221. IEEE Computer Society.

[JBLO68]: Lovins, J. (1968): *Development of a stemming algorithmen*. In: Mechanical Translation and Computational Linguistics, Vol. 11, No. 2, pp. 22 – 31. Association of Computational Linguistics (ACL).

[JGHO98]: Joeris, G. & Herzog, O. (1998): *Managing evolving workflow specifications*. In Proc. of: 3rd IFCIS International Conference on Cooperative Information Systems, pp. 310-319. IEEE Computer Society.

[JJNJ14]: Jeston, J. & Nelis, J. (2014): *Business Process Management*. 3. Auflage. Routledge.

[JRPP94]: Johnson, R. & Pearson, D. & Pingali. K. (1994): *The Program Structure Tree: Computing Control Regions in Linear Time*. In: SIGPLAN Notification, Vol. 29, No. 6, pp. 171-185. ACM.

[KAOH12]: Keshk, A. & Ossman, M. & Lamina, H. (2012): *Fast Longest Common Subsequences for Bioinformatics Dynamic Programming.* In: International Journal of Computer Applications, Vol. 57, No. 22, pp. 12-18. Foundation of Computer Science, New York.

[KBHB00]: Kiepuszewski, B. & ter Hofstede, A. & Bussler, C. (2000): *On structured workflow modelling.* In: Advanced Information Systems Engineering, LNCS 1789, pp. 431-445. Springer-Verlag.

[KFKF12] Kabicher-Fuchs, S. & Kriglstein, S. & Figl, K. (2012): T*imeline Visualization for Documenting Process Model Change*. In Proc. of 5th Int'l Workshop on Enterprise Modelling and Information Systems Architectures, Austria

[KGFE08]: Küster, J. & Gerth, C. & Förster, A. & Engels, G. (2008): *Detecting and Resolving Process Model Differences in the Absence of a Change Log*. In Proc. of: 6th International Conference on Business Process Management (BPM'08), LNCS 5240, pp. 244 – 260. Springer-Verlag.

[KMGA99]: Kradolfer, M. & Geppert, A. (1999): *Dynamic workflow schema evolution based on workflow type versioning and workflow migration*. In Proc. of: CoopIS'99. IFCIS International Conference on Cooperative Information Systems. pp. 104-114. IEEE Computer Society.

[KRLL09]: Ko, R. & Lee, S. & Lee, E. (2009): *Business process management (BPM) standards: a survey.* In: Business Process Management Journal, Vol. 15, No. 5, pp.744-791. Emerald Group Publishing Limited.

[LBHL13] Li, B. & Han, L. (2013): *Distance Weighted Cosine Similarity Measure for Text Classification.* In: Intelligent Data Engineering and Automated Learning – IDEAL 2013., LNCS: 8206, pp. 611-618. Springer-Verlag

[LBKB06]: List, B. & Korherr, B. (2006): *An evaluation of conceptual business process modelling languages. In: Proceedings of the 2006 ACM symposium on Applied computing. pp. 1532 – 1539. ACM.*

[LCRW06]: Li, C.& Reichert, M. & Wombacher, A. (2006): *On measuring process model similarity based on high-level change operations*. In: Conceptual Modeling-ER 2008, LNCS 5321, pp. 248-264, Springer-Verlag.

[MADA78]: Maier, D. (1978): *The Complexity of Some Problems on Subsequences and Supersequences*. In: Journal of ACM, Vol. 25, No. 2, pp. 322-336. ACM.

[MEKO07]: Ehrig, M. & Koschmider, A. & Oberweis, A. (2007): *Measuring similarity between semantic business process models*. In Proc. of: APCCM'07, The 4th Asia-Pacific conference on Comceptual modelling, Vol. 67, pp. 71–80, Australian Computer Society, Inc.

[MJNA07]: Mendling, J. & Neumann, G. & van der Aalst, W (2007): *Understanding the Occurrence of Errors in Process Models Based on Metrics*. In Proc. of: The OTM Conference on Cooperative information Systems (CoopIS 2007), LNCS 4803, pp. 113-130, Springer-Verlag.

[MJNN05]: Mendling, J. & Neumann, G. & Nüttgens, M. (2005): *A comparison of XML interchange formats for business process modelling* . In: Workflow handbook, pp. 185-198. Workflow Management Coalition.

[MJRC07]: Mendling, J. & Reijers, H. & Cardoso, J (2007): *What makes process models understandable*. In: International Conference on Business Process Management (BPM 2007), LNCS 4717, pp. 48-63, Springer-Verlag.

[MWME85]: Miller, W. & Mayer, E. (1985): *A file comparison programm*. In: Software: Practice and Experience, Vol. 15, No. 11, pp. 1025-1040. Wiley Online Library.

[MYER86]: Myer, E. (1986): *An O(ND) difference algorithm and its variations*. In: Algorithmica, Vol. 1, No. 1-4, pp- 251-266. Springer-Verlag.

[NSWC70]: Needleman, S. & Wunsch, D. (1970): *A general method applicable to the search for similarities in the amino acid sequence of two proteins*. In: Journal of molecular biology, Vol. 48, No. 3, pp. 443-453. Elsevier.

[RAEH05]: Russell, N. & ter Hofstede, A. & Edmond, D. & van der Aalst, W. (2005): *Workflow data patterns: Identification, representation and tool support*. In: Conceptual Modeling, LNCS 3716, pp. 353-368. Springer-Verlag.

[RDRJ09]: Rinderle-Ma, S. & Reichert, R. & Jurisch, M. (2009): *Equivalence of Web Services in Process-Aware Service Compositions*. In Proc. of: 7th Int'l Conference on Web Services (ICWS'09), pp. 501 – 508. IEEE Computer Society Press.

[REDA00]: Reichert, M. & Dadam, P. (2000): *Geschäftsprozessmodellierung und Workflow-Management-Konzepte, Systeme und deren Anwendung.* In: Industrie Management, Vol. 16, No. 3, pp- 23-27. GITO-Verlag.

[REMA00]: Reichert, M. (2000): *Dynamische Ablaufänderungen in Workflow-Management-Systemen*. PhD-Thesis, Universität Ulm.

[RIMM92]: Richter, M. (1992): *Classification and learning of similarity measures*. In: Studies in Classification, Data Analysis and Knowledge Organization, pp. 323-334. Springer-Verlag.

[RIRD04]: Rinderle, S. & Reichert, M. & Dadam, P. (2004): *Correctness criteria for dynamic changes in workflow systems - a survey*. In: Data & Knowledge Engineering, Vol. 50, No. 1, pp. 9-34. Elsevier.

[RMDP98]: Reichert, M. & Dadam, P. (1998): *ADEPTflex-Supporting dynamic changes of workflows without losing control*. In: Journal of Intelligent Information Systems, Vol. 10, No. 2, pp.93-129. Springer-Verlag.

[RNAH06]: Russell, N. & van der Aalst, W. & ter Hofstede, A. (2006): *Workflow Exception Patterns*. In: Advanced Information Systems Engineering, LNCS 4001, pp.288-302. Springer-Verlag.

[RNHM06]: Russel, N. & ter Hofstede, A. & Mulyar, N. (2006): *Workflow ControlFlow Patterns: A Revised View*. In: BPM Center Report BPM-06-22, BPMcenter.org.

[RRJK06]: Rinderle-Ma, S. & Reichert, R. & Jurisch, M. & Kreher, U. (2006): *On representing, purging, and utilizing change logs in process management systems*. In Proc. of: The 4th International Conference on Business Process Management (BPM'06), LNCS 4102,. pp. 241–256. Springer-Verlag.

[RSRD04]: Rinderle, S. & Reichert, M. & Dadam, P. (2004): *Disjoint and overlapping process changes: Challenges, solutions, applications*. In : On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE. LNCS 3290. pp. 101-120. Springer Verlag.

[RWMH11]: La Rosa, M. & Wohed, P. & Mendling, J. & ter Hofstede, A. & Reijers, H.A. & van der Aalst, W. (2006): *Managing Process Model Complexity Via Abstract Syntax Modifications*. In: IEEE Transactions on Industrial Informatics, Vol. 7, Issue: 4, pp. 614-629. IEEE Computer Society.

[SABU88]: Salton, G. & Buckley, C. (1988): *Term-weighting approaches in automatic text retrieval.* In: Information Processing & Management, Vol. 24, No. 5, pp. 513 – 523. Pergamon Press.

[SAJA06]: Savoy, J. (2006): *Light stemming approaches for the French, Portuguese, German and Hungarian languages*. In Proc. of: The 2006 ACM symposium on Applied computing (SAC'06). pp. 1031–1035. ACM.

[SAMG86]: Salton, G. & McGill, M. (1986): *Introduction to Modern Information Retrieval*. McGraw-Hill Inc.

[SANK72]: Sankoff, D. (1972): *Matching sequences under deletion/insertion constraints*. In Proc. of: The National Academy of Sciences, Vol. 69, No. 1. pp. 4-6. National Acad Sciences.

[SGFH83]: Salton G. & Fox, E. And Wu, H. (1983): *Extended Boolean information retrieval*. In: Communication of the ACM, Vol. 26, No. 11, pp. 1022–1036. ACM.

[SGWY75]: Salton, G. & Wong, A. & Yang, C. (1975): *A vector space model for automatic indexing*. In: Communications of the ACM, Vol. 18, No. 11, pp. 613–620. ACM.

[SRJR07]: Rinderle, S. & Jurisch, M. & Reichert, M. (2007): *On Deriving Net Change Information From Change Logs-The DELTALAYER-Algorithm*. In Proc. of: The 12th GI-Conference on Database Systems in Business, Technology and Web (BTW'07), LNI-103, pp. 364-381. Koellen Verlag.

[TWTH07] Wilde, T. & Hess, T. (2007): Forschungsmethoden der Wirtschafts-informatik. In: Wirtschaftsinformatik,Vol. 49, No. 4, pp. 280–287. Srpringer-Verlag

[VAHV02]: van der Aalst, W. & Hirnschall, A. & Verbeek, H. (2002): *An alternative way to analyze workflow graphs*. In: Advanced Information Systems Engineering, LNCS 2348, pp. 535-552. Springer-Verlag.

[VDAW04]: van der Aalst, W. (2004): *Business process management demystified: A tutorial on models, systems and standards for workflow management*. In: Lectures on concurrency and Petri nets, LNCS 3098, pp. 1-65. Springer Verlag.

[VJVK09]: Vanhatalo, J. & Völzer, H. & Köhler, J. (2009): *The refined process structure tree*. In: Data & Knowledge Engineering, Vol. 68, No. 9, pp. 793-818. Elsevier.

[VVLO07]: Vanhatalo, J. & Völzer, H. & Leymann, F. (2007): *Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition.* In Proc. of: The 5th International Conference on Service-Oriented Computing (ICSOC '07), LNCS 4749, pp. 43-55, Springer-Verlag.

[WBRR08]: Weber, B. & Reichert, M. & Rinderle-Ma, S. (2008): *Change patterns and change support features-enhancing flexibility in process-aware information systems*. In: Data & knowledge engineering, Vol. 66, No. 3, pp. 438-466. Elsevier.

[WESK07]: Wekse, M. (2007): *Business Process Management – Concepts, Langauges, Architectures*. Springer-Verlag.

[WfMC98]: Workflow Management Coalition (1998): *Interface 1: Process Definition Interchange Process Model (TC-1016-P),* The Workflow Management Coalition.

[WSMO00]: Sadiq, W. & Orlowska M. (2000): *Analyzing process models using graph reduction techniques*. In : Information Systems, Vol. 25, No. 2, pp 117-134. Elsevier.

[ZMUM04]: zur Muehlen, M. (2004): *Workflow-Based Process Controlling: Foundation, Design, and Application of Workflow-Driven Process Information Systems*. Advances in Information Systems and Management Science (Buch 6). Logos, Berlin.

[ZXLC07]: Zhao, X. & Liu, C. (2007): Version management in the business process change context. In: Business Process Management. LNCS 4714. pp. 198-213. Springer-Verlag.

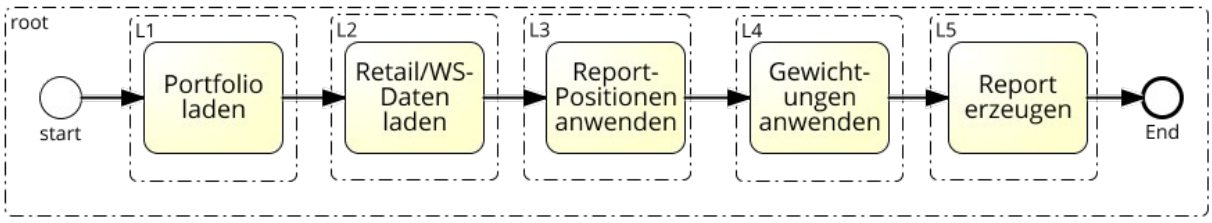
14 Anhang

14.1 Anhang zu Kapitel „Vereinheitlichung von Prozessmodellen“

Mappings

Der folgende Anhang zeigt Prozessmodelle, welche mit den speziellen Modellerungssprachen BPNM und CPEE-Cockpit modelliert und in das vorgeschlagene Meta-Modell transformiert worden sind.

Einfaches, sequenzielles Prozessmodell in BPMN



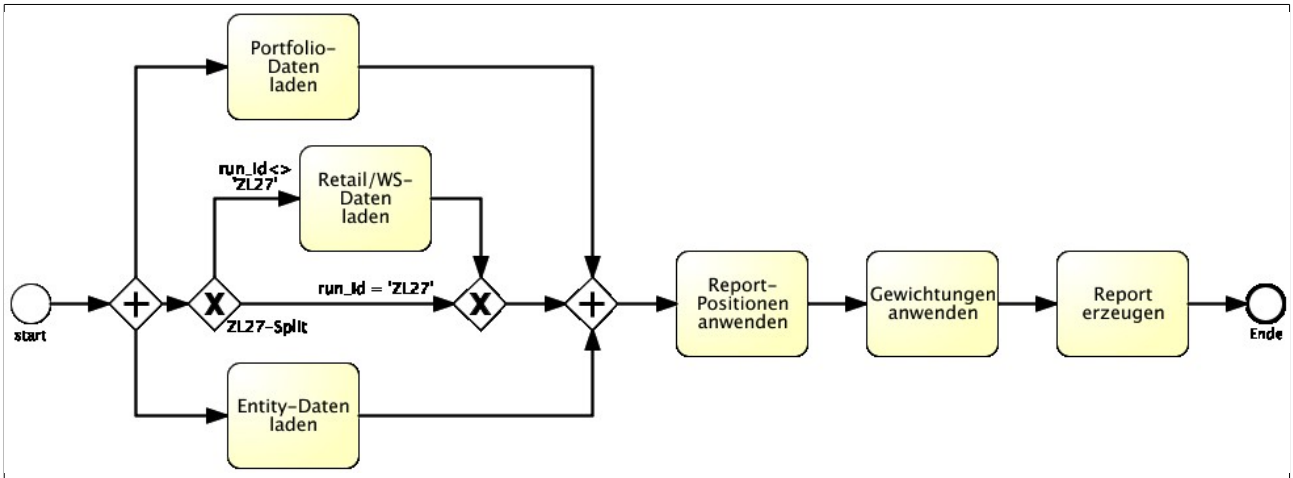

```
<processmodel name="BIII-Main_Process">
      <start name="start" type="startEvent"/>
      <activity name="Portfolio laden" type="task" />
      <activity name="Retail/WS-Daten laden" type="task" />
      <activity name="Report-Positionen anwenden" type="task" />
      <activity name="Gewichtungen anwenden" type="task" />
      <activity name="Report erzeugen" type="task" />
      <end name = "end" type="endEvent" />
</processmodel>
```

Einfaches, sequenzielles Prozessmodell in CPEE

```
001: call :a1, :, { :label => "Portfolio laden", :method => "post", :parameters => nil }
002: call :a2, :, { :label => "Retail/WS-Daten laden", :method => "post", :parameters => nil }
003: call :a3, :, { :label => "Report-Positionen anwenden", :method => "post", :parameters => nil }
004: call :a4, :, { :label => "Gewichtungen anwenden", :method => "post", :parameters => nil }
005: call :a5, :, { :label => "Report erzeugen", :method => "post", :parameters => nil }
```

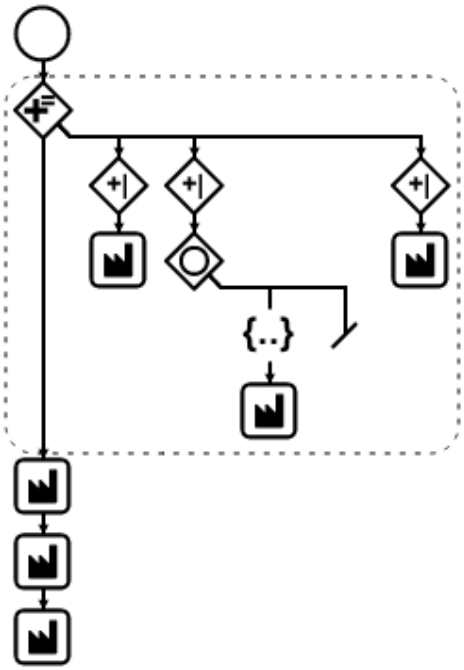
```
<processmodel name="BIII-Main_Process">
      <start name="start" type="" />
      <activity name="Portfolio laden" type="call" />
      <activity name="Retail/WS-Daten laden" type="call" />
      <activity name="Report-Positionen anwenden" type="call" />
      <activity name="Gewichtungen anwenden" type="call" />
      <activity name="Report erzeugen" type="call" />
      <end name = "end" type="" />
</processmodel>
```

Komplexes Prozessmodell mit Split/Join-Knoten in BPMN




```
<processmodel name="BIII-Main_Process_with_Splits">
	<start name="start" type="startEvent" />
	<split name = "LoadParallelSplit" type="parallelGateway">
		<alternative>
			<activity name="Portfolio laden" type="task" />
		</alternative>
		<alternative>
		<split name = "ZL27?Split" type="exclusiveGateway">
			<alternative cterm="rund_id='ZL27'">
				<activity name="Retail/WS-Daten laden" type="task" />
				</alternative>
				<alternative cterm="rund_id<>'ZL27'" />
		    </split>
		<join name = "ZL27?Join" type="exclusiveGateway"></join>
		</alternative>
		<alternative>
			<activity name="Entity-Daten laden" type="task" />
		</alternative>
	</split>
	<join name = "LoadParallelJoin" type="parallelGateway"></join>
	<activity name="Report-Positionen anwenden" type="task" />
	<activity name="Gewichtungen anwenden" type="task" />
	<activity name="Report erzeugen" type="task" />
	<end name = "end" type="endEvent" />
</processmodel>
```

Komplexes Prozessmodell mit Split/Join-Knoten in CPEE

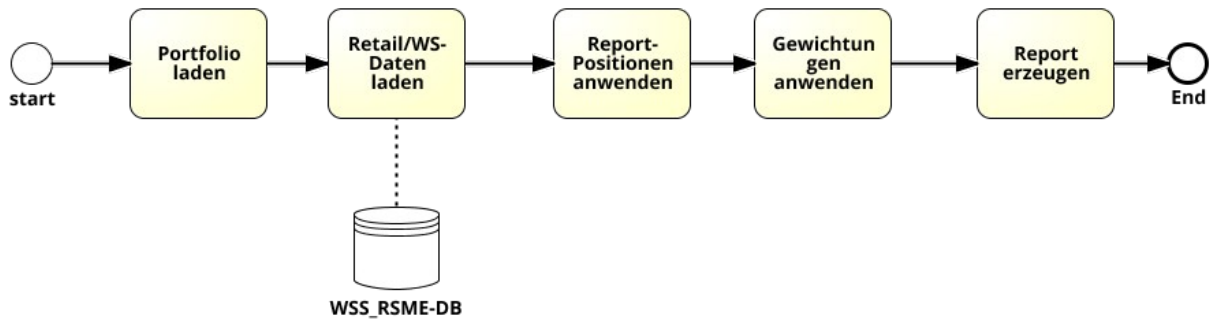


```
001: parallel do
002:   parallel_branch do
003:     call :a1, :, { :label => "Entity-Daten laden", :method => "post", :parameters => nil }
004:   end
005:   parallel_branch do
006:     choose do
007:       alternative run_id = 'ZL27' do
008:         call :a6, :, { :label => "Retail/WS-Daten laden", :method => "post", :parameters => nil }
009:       end
010:       otherwise do
011:       end
012:     end
013:   end
014:   parallel_branch do
015:     call :a2, :, { :label => "Portfolio Daten laden", :method => "post", :parameters => nil }
016:   end
017: end
018: call :a3, :, { :label => "Report-Postionen anwenden", :method => "post", :parameters => nil }
019: call :a4, :, { :label => "Gewichtungen anwenden", :method => "post", :parameters => nil }
020: call :a5, :, { :label => "Report erzeugen", :method => "post", :parameters => nil }
```

```
<processmodel name="BIII-Main_Process_with_Splits">
      <start name="start" type="" />
      <split name = "parallel" type="parallel do">
            <alternative>
                  <activity name="Portfolio lDaten laden" type="call" />
            </alternative>
            <alternative>
            <split name = "choose" type="choose do">
                  <alternative cterm="rund_id='ZL27'">
                        <activity name="Retail/WS-Daten laden" type="call" />
                  </alternative>
                  <alternative cterm="otherwise" />
               </split>
            <join name = "choose" type=""></join>
            </alternative>
            <alternative>
```

```
                    <activity name="Entity-Daten laden" type="call" />
                </alternative>
        </split>
        <join name = "parallel" type=""></join>
        <activity name="Report-Positionen anwenden" type="call" />
        <activity name="Gewichtungen anwenden" type="call" />
        <activity name="Report erzeugen" type="call /> 
        <end name = "end" type="" />
</processmodel>
```

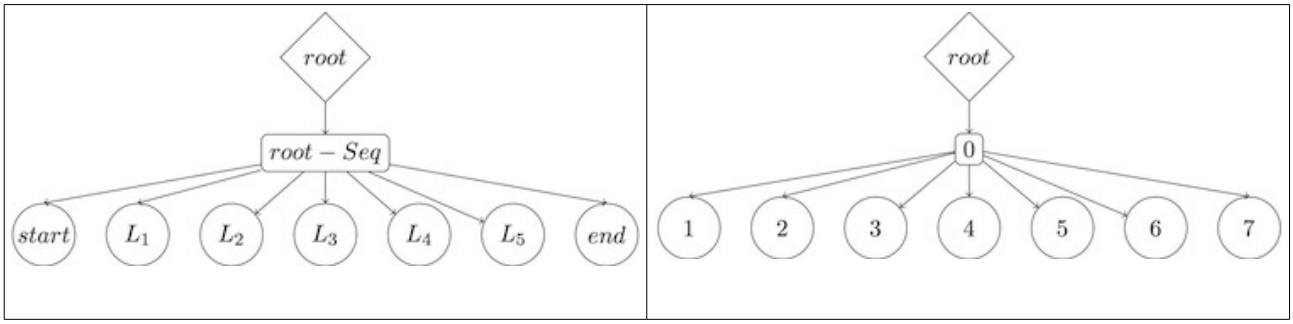
Prozessmodell mit Ressourcen-Ansicht in BPNM



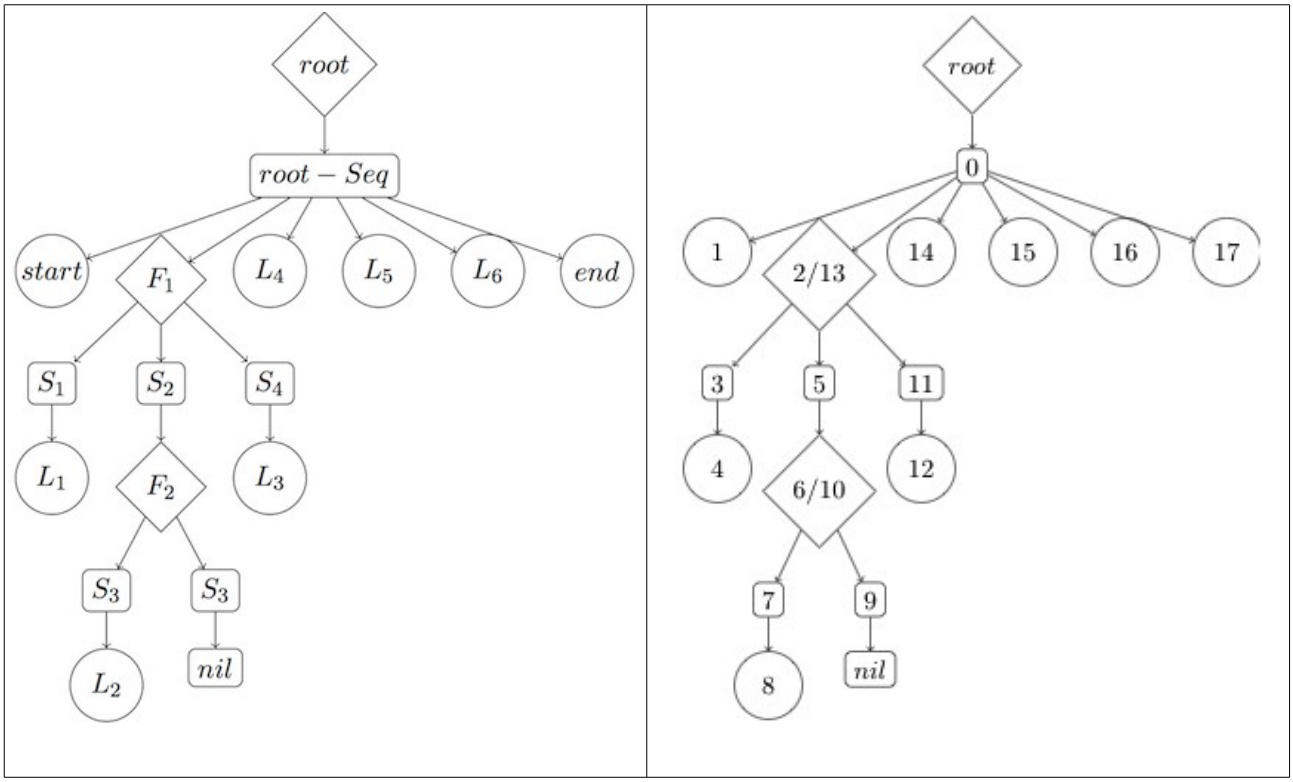

```
<processmodel name="BIII-Main_Process_with_Splits">
        <start name="start" type="startEvent" />
        <activity name="Portfolio laden" type="task" />
        <activity name="Retail/WS-Daten laden" type="task" />
        <activity name="Report-Positionen anwenden" type="task" />
        <activity name="Gewichtungen anwenden" type="task" />
        <activity name="Report erzeugen" type="task" />
        <end name = "end" type="endEvent"  />

        <!-- Beginn des ProcessSupportObjektes -->
        <processsupport name="WSS_RSME-DB" type="dataStore" />
        <processsupport name="AssocArc" type="association">
                <param>
                        <name>associationDirection</name>
                        <valuetype>String</valuetype>
                        <value>None</value>
                </param>
                <param>
                        <name>sourceRef</name>
                        <valuetype>String</valuetype>
                        <value>WSS_RSME-DB</value>
                </param>
                <param>
                        <name>targetRef</name>
                        <valuetype>String</valuetype>
                        <value>Retail/WS-Daten laden</value>
                </param>
        </processsupport>
</processmodel>
```

14.2 Anhang zu Kapitel „Process-Structure-Tree“

Positionsnummer PST aus Abbildung 20



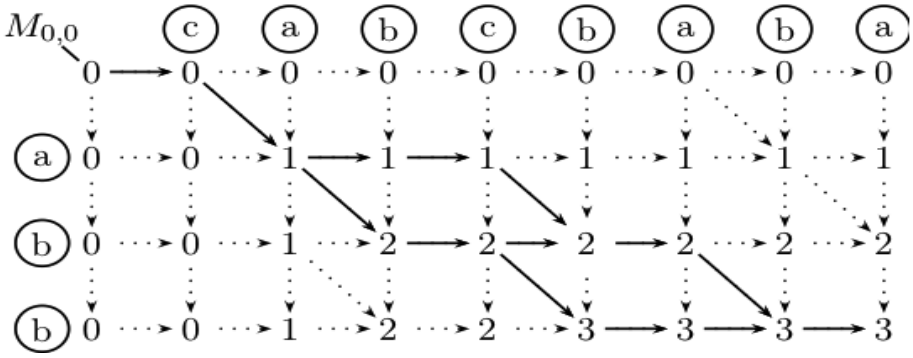
Positionsnummer PST aus Abbildung 21



In der Abbildung ist ein Process-Structure-Tree zu sehen, welcher auch Fragment-Knoten beinhaltet. Diese sind durch ein Diamanten-Symbol gekennzeichnet. Innerhalb der Fragment-Knoten existieren zwei Positionsnummern. Die erste Positionsnummer gibt den Split-Knoten an, die zweite Nummer den Join-Knoten. Eine weitere Besonderheit in diesem Process-Structure-Tree stellt die Sequenz S_3 dar: Diese Sequenz beinhaltet keinen Aktivitäten.

14.3 Anhang zu Kapitel „Differenzen Erkennung zwischen zwei Prozessmodellen“

LCS-Matrix der Subzeichenkette $A' = abb$ **und** $B' = cabcbaba$



14.4 Anhang zu Kapitel „Persistierung von Highlevel-Changes“

XML-Schema des Change-Logs

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

<!--ADD -->
<xs:element name="ADD">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="name"/>
				<xs:element ref="type"/>
				<xs:element ref="predecessor"/>
				<xs:element ref="accessor"/>
				<xs:element ref="cterm" minOccurs="0" maxOccurs="1"/>
			</xs:sequence>
			<xs:attribute name="id" use="required" />
			<xs:attribute name="pos" use="required" />
			<xs:attribute name="mmodeltype" use="required" />
		</xs:complexType>
</xs:element>

<!--DELETE -->
<xs:element name="DELETE">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="name"/>
				<xs:element ref="type"/>
				<xs:element ref="predecessor"/>
				<xs:element ref="accessor"/>
				<xs:element ref="cterm" minOccurs="0" maxOccurs="1"/>
			</xs:sequence>
			<xs:attribute name="id" use="required" />
			<xs:attribute name="pos" use="required" />
			<xs:attribute name="mmodeltype" use="required" />
		</xs:complexType>
</xs:element>

<!-- UPDATE -->
	<xs:element name="UPDATE">
		<xs:complexType>
			<xs:choice minOccurs="0" maxOccurs="unbounded">
				<xs:element ref="changeupdate"/>
				<xs:element ref="deleteupdate"/>
				<xs:element ref="addupdate" />
			</xs:choice>
			<xs:attribute name="id" use="required" />
			<xs:attribute name="pos" use="required" />
			<xs:attribute name="name" use="required" />
		</xs:complexType>
	</xs:element>
```

```
	<!-- UPDATE add-->
	<xs:element name="addupdate">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="param" />
			</xs:sequence>
		</xs:complexType>
	</xs:element>
	<!-- UPDATE delete-->
	<xs:element name="deleteupdate">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="param" />
			</xs:sequence>
		</xs:complexType>
	</xs:element>
	<!-- UPDATE change-->
	<xs:element name="changeupdate">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="updateto"/>
				<xs:element ref="updatefrom"/>
			</xs:sequence>
			<xs:attribute name="type" use="required">
				<xs:simpleType>
					<xs:restriction base="xs:string">
						<xs:enumeration value="value"/>
						<xs:enumeration value="type"/>
						<xs:enumeration value="name"/>
					</xs:restriction>
				</xs:simpleType>
			</xs:attribute>
			<xs:attribute name="param" use="required" />
		</xs:complexType>
	</xs:element>
	<xs:element name="updateto"/>
	<xs:element name="updatefrom"/>

<!-- MOVE -->
	<xs:element name="MOVE">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="name"/>
				<xs:element ref="to"/>
				<xs:element ref="from"/>
			</xs:sequence>
			<xs:attribute name="id" use="required" />
		</xs:complexType>
	</xs:element>

<!-- SWAP -->
	<xs:element name="SWAP">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="obj" maxOccurs="2"/>
			</xs:sequence>
```

```
                  <xs:attribute name="id" use="required" />
            </xs:complexType>
      </xs:element>

<!-- COPY -->
      <xs:element name="COPY">
            <xs:complexType>
                  <xs:sequence>
                        <xs:element ref="source"/>
                        <xs:element ref="addObj"/>
                  </xs:sequence>
                  <xs:attribute name="id" use="required" />
                  <xs:attribute name="pos" use="required" />
                  <xs:attribute name="mmodeltype" use="required" />
            </xs:complexType>
      </xs:element>

<!-- REPLACE -->
      <xs:element name="REPLACE">
            <xs:complexType>
                  <xs:sequence>
                        <xs:element ref="addObj"/>
                        <xs:element ref="delObj"/>
                  </xs:sequence>
                  <xs:attribute name="id"/>
            </xs:complexType>
      </xs:element>



      <xs:element name="to">
            <xs:complexType>
                  <xs:sequence minOccurs="0">
                        <xs:element ref="pos"/>
                        <xs:element ref="predecessor"/>
                        <xs:element ref="accessor"/>
                  </xs:sequence>
            </xs:complexType>
      </xs:element>

      <xs:element name="from">
            <xs:complexType>
                  <xs:sequence minOccurs="0">
                        <xs:element ref="pos"/>
                        <xs:element ref="predecessor"/>
                        <xs:element ref="accessor"/>
                  </xs:sequence>
            </xs:complexType>
      </xs:element>



      <xs:element name="obj">
            <xs:complexType>
                  <xs:sequence>
                        <xs:element ref="name"/>
                        <xs:element ref="to"/>
```

```
                        <xs:element ref="from"/>
                    </xs:sequence>
            </xs:complexType>
    </xs:element>


    <xs:element name="param">
            <xs:complexType>
                    <xs:sequence>
                            <xs:element ref="name"/>
                            <xs:element ref="valueType"/>
                            <xs:element ref="value"/>
                    </xs:sequence>
            </xs:complexType>
    </xs:element>




    <xs:element name="addObj">
            <xs:complexType>
                    <xs:sequence>
                            <xs:element ref="name"/>
                            <xs:element ref="type"/>
                            <xs:element ref="predecessor"/>
                            <xs:element ref="accessor"/>
                    </xs:sequence>
                    <xs:attribute name="pos" />
                    <xs:attribute name="mmodeltype" />
            </xs:complexType>
    </xs:element>

    <xs:element name="delObj">
            <xs:complexType>
                    <xs:sequence>
                            <xs:element ref="name"/>
                            <xs:element ref="type"/>
                            <xs:element ref="predecessor"/>
                            <xs:element ref="accessor"/>
                    </xs:sequence>
                    <xs:attribute name="pos" use="required" />
                    <xs:attribute name="mmodeltype" use="required" />
            </xs:complexType>
    </xs:element>



    <xs:element name="SEQUENCE">
            <xs:complexType>
                    <xs:choice minOccurs="1" maxOccurs="1">
                            <xs:element ref="REPLACE" maxOccurs="unbounded"/>
                            <xs:element ref="COPY" maxOccurs="unbounded"/>
                            <xs:element ref="ADD" maxOccurs="unbounded"/>
                            <xs:element ref="DELETE" maxOccurs="unbounded"/>
                            <xs:element ref="UPDATE" maxOccurs="unbounded"/>
                            <xs:element ref="MOVE" maxOccurs="unbounded"/>
                            <xs:element ref="SWAP" maxOccurs="unbounded"/>
```

```
                </xs:choice>
                <xs:attribute name="id" use="required" />
                <xs:attribute name="type" use="required">
                        <xs:simpleType>
                                <xs:restriction base="xs:string">
                                        <xs:enumeration value="REPLACE"/>
                                        <xs:enumeration value="COPY"/>
                                        <xs:enumeration value="ADD"/>
                                        <xs:enumeration value="DELETE"/>
                                        <xs:enumeration value="UPDATE"/>
                                        <xs:enumeration value="MOVE"/>
                                        <xs:enumeration value="SWAP"/>
                                </xs:restriction>
                        </xs:simpleType>
                </xs:attribute>
        </xs:complexType>
    </xs:element>

    <xs:element name="changelog">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="UPDATE"/>
                        <xs:element ref="ADD"/>
                        <xs:element ref="DELETE"/>
                        <xs:element ref="MOVE"/>
                        <xs:element ref="SWAP"/>
                        <xs:element ref="REPLACE"/>
                        <xs:element ref="COPY"/>
                        <xs:element ref="SEQUENCE"/>
                </xs:sequence>
        </xs:complexType>
    </xs:element>

    <!-- Common simple elements -->
    <xs:element name="pos" />
    <xs:element name="name" />
    <xs:element name="type" />
    <xs:element name="value"  />
    <xs:element name="source" />
    <xs:element name="valueType" />
    <xs:element name="predecessor" />
    <xs:element name="accessor" />
    <xs:element name="cterm" />

</xs:schema>
```

14.5 Anhang zu Kapitel „Rückrechnung von Prozessmodellen“

Vollständige Liste der inversen Highlevel-Changes

Highlevel-Change	Inverser Highlevel-Change
ADD(Obj, $predecessor_t$, $accessor_t$)	DELETE(Obj, $predecessor_t$, $accessor_t$)
DELETE(Obj, $predecessor_{t-1}$, $accessor_{t-1}$)	ADD(Obj, $predecessor_{t-1}$, $accessor_{t-1}$)
MOVE(Obj, $predecessor_t$, $accessor_t$)	MOVE(Obj, $predecessor_{t-1}$, $accessor_{t-1}$)
REPLACE(Obj_t, $ReplacedO_{t-1}$, $predecessor_t$, $accessor_t$)	REPLACE(Ob_{t-1},$ReplacedObj_t$, $predecessor_{t-1}$, $accessor_{t-1}$)
SWAP(Obj1, Obj2)	SWAP(Obj2, Obj1)
COPY(Obj, $predecessor_t$, $accessor_t$):	DELETE(Obj, $predecessor_{t-1}$, $accessor_{t-1}$)
UPDATE(Obj) ADD UPDATE(Obj) DELETE UPDATE(Obj) CHANGE	UPDATE(Obj) DELETE UPDATE(Obj) ADD UPDATE(Obj) CHANGE
ADDFragement(Obj, $predecessor_t$, $accessor_t$)	DELETEFragment(Obj, $predecessor_t$, $accessor_t$)
DELETEFragment(Obj, $predecessor_{t-1}$, $accessor_{t-1}$)	ADDFragement(Obj, $predecessor_{t-1}$, $accessor_{t-1}$)
ADDAlternative(Obj, $predecessor_t$, $accessor_t$)	DELETEAlternative(Obj, $predecessor_t$, $accessor_t$)
DELETEAlternative(Obj, $predecessor_{t-1}$, $accessor_{t-1}$)	ADDAlternative(Obj, $predecessor_{t-1}$, $accessor_{t-1}$)
SEQUENCEADD(AddObj1,...,AddObjn ,$predecessor_t$, $accessor_t$)	SEQUENCEDELETE(AddObj1,...,AddObjn ,$predecessor_t$, $accessor_t$)
SEQUENCEDELETE(DelObj1,...,DelObjn ,$predecessor_{t-1}$, $accessor_{t-1}$)	SEQUENCEADD(DelObj1,...,DelObjn, $predecessor_{t-1}$, $accessor_{t-1}$)
SEQUENCESWAP(SwapObjs1, SwapObjs2)	SEQUENCESWAP(SwapObj2, SwapObjs1)
SEQUENCEREPLACE(AddObjs, DeleteObjs ,$predecessor_t$, $accessor_t$)	SEQUENCEREPLACE(DeleteObjs, AddObj1 ,$predecessor_{t-1}$, $accessor_{t-1}$)
SEQUENCEMOVE(MoveObj1,...MoveObj2 ,$predecessor_t$, $accessor_t$)	SEQUENCEMOVE(MoveObj1,...MoveObj2 ,$predecessor_{t-1}$, $accessor_{t-1}$)

14.6 Anhang zu Kapitel „Versionskontrolle“

XML Schema des Versionsgraphen

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
	<xs:element name="vg">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="v" maxOccurs="unbounded"/>
			</xs:sequence>
			<xs:attribute name="name" use="required"/>
		</xs:complexType>
	</xs:element>
	<xs:element name="v">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="cuser" maxOccurs="1" />
				<xs:element ref="ctime" maxOccurs="1" />
				<xs:element ref="comment" maxOccurs="1" />
				<xs:element ref="cl" maxOccurs="unbounded"/>
				<xs:choice minOccurs="0">
					<xs:element ref="vgref"/>
					<xs:element ref="v" maxOccurs="unbounded"/>
				</xs:choice>
			</xs:sequence>
			<xs:attribute name="vid" use="required"/>
			<xs:attribute name="type" use="required">
				<xs:simpleType>
					<xs:restriction base="xs:string">
						<xs:enumeration value="variant"/>
						<xs:enumeration value="version"/>
					</xs:restriction>
				</xs:simpleType>
			</xs:attribute>
		</xs:complexType>
	</xs:element>
	<xs:element name="cl">
		<xs:complexType>
			<xs:simpleContent>
				<xs:extension base="xs:string">
					<xs:attribute name="predecessor" />
				</xs:extension>
			</xs:simpleContent>
		</xs:complexType>
	</xs:element>
	<xs:element name="vgref"/>
	<xs:element name="cuser"/>
	<xs:element name="ctime"/>
	<xs:element name="comment"/>
</xs:schema>
```

Versionsgraph komplett

```
<vg name="LCR-Kalkulation">
      <v type="version" vid="1.0.0">
            <cuser>wzbsxs</cuser>
            <ctime>2013/10/29 15:05:30</ctime>
            <comment>initial Version LCR-Kalkulation</comment>
            <cl>Delta_1.0.0.xml</cl>
      </v>
      <v type="version" vid="1.1.0">
            <cuser>wzbsxs</cuser>
            <ctime>2013/10/31 09:10:28</ctime>
            <comment>Laengere Processing-Time fuer Task 'LCR berechnen'</comment>
            <cl predecssor="1.0.0">Delta_1.1.0.xml</cl>
            <v type="variant" vid="1.1.1">
                  <cuser>wzbdmk</cuser>
                  <ctime>2013/12/12 10:00:13</ctime>
                  <comment>Hinzufügen einer Whole/Retail-Ausnahme
                        fuer unit ZL27</comment>
                  <cl predecessor="1.1.0">Delta_1.1.1.xml</cl>
            </v>
      </v>
      <v type="version" vid="1.2.0">
          <cuser>wzbdmk</cuser>
            <ctime>2013/11/12 10:10:47</ctime>
            <comment>Einbau Parallelisierung Whole/Retail-Daten
                  laden und Portfolio-Daten laden</comment>
            <cl predecssor="1.0.0">Delta_1.2.0.xml</cl>
      </v>
      <v type="version" vid="1.3.0">
          <cuser>wzbdmk</cuser>
            <ctime>2013/11/12 10:10:47</ctime>
            <comment>Einbau eines neuen Procedure-Aufrufs in
                  Reporting-Positionen bestimmen: regulatorische
                  Ausnahme für unit ZL19</comment>
            <cl predecssor="1.1.0">Delta_1.3.0.xml</cl>
      </v>
      <v type="version" vid="1.4.0">
          <cuser>wzbdmk</cuser>
            <ctime>2013/11/12 10:10:47</ctime>
            <comment>Paralleliserung bringt Effizienz Steigerung,
                  daher Merge mit unit ZL19 wegen regulatorischer
                  Anforderungen muss die Ausnahme in zukünftigen
                  Prozessen mit verwendet werden.</comment>
            <cl predecssor="1.3.0">Delta_1.4.0_merged_1.3.0.xml</cl>
            <cl predecssor="1.2.0">Delta_1.4.0_merged_1.2.0.xml</cl>
            <v type="variant" vid="1.4.1">
                  <cl predecessor="1.4.0">Delta_1.4.1.xml</cl>
                  <v type="variant" vid="1.4.2">
                        <cuser>wzbsxs</cuser>
                        <ctime>2013/01/25 12:25:34</ctime>
                        <comment>Einbau eines neuen Tasks zu
                              Main-Currency-Berechnung. Wird in
                              einen neuen Versionsbaum ausgelagert.
                              Da hier eventuell nach dem
                              CRR-Framework neue Anforderungen kommen.</comment>
                        <cl predecessor="1.4.1">Delta_1.4.2.xml</cl>
                        <vgref>"LCR-Kalkulation_HO"</vgref>
```

```
                    </v>
                    <v type="variant" vid="1.4.3">
                          <cuser>wzbsxs</cuser>
                          <ctime>2013/01/25 12:25:34</ctime>
                          <comment>'Reportpositionen setzen' verwendet
                                   neue Funktion für das Setzen der Reporting
                                   Positions</comment>
                          <cl predecessor="1.4.1">Delta_1.4.3.xml</cl>
                          <v type="variant" vid="1.4.4">
                              <cuser>wzbdmk</cuser>
                                 <ctime>2013/01/30 09:05:14</ctime>
                                 <comment>Added new CRR-Template to LCR-berechnen
                                          for new Reporting Positions</comment>
                                 <cl predecessor="1.4.3">Delta_1.4.4.xml</cl>
                          </v>
                    </v>
              </v>
       </v>
       <v type="version" vid="2.0.0">
              <cuser>wzbsxs</cuser>
              <ctime>2014/02/03 18:00:00</ctime>
              <comment>Released new Workflow-Version 2.0 - start developing
                       from this version. New CRR-Template</comment>
              <cl predecssor="1.4.0">Delta_2.0.0_merged_1.4.0.xml</cl>
              <cl predecssor="1.4.4">Delta_2.0.0_merged_1.4.4.xml</cl>
       </v>
</vg>
```

14.7 Anhang zu Kapitel „Rückrechnung von Prozessmodellen“

Persistente Highlevel-Changes aus Abbildung 47

```
<ADD  id="1" pos="2" mmodeltype="split">
  <name>Load And-Split</name>
  <predecessor>start</predecessor>
  <accessor>Report-Positionen anwenden</accessor>
  <type>parallelgateway</type>
</ADD>
<ADD  id="2" pos="7" mmodeltype="join">
  <name>Load And-Join</name>
  <predecessor>start</predecessor>
  <accessor>Report-Positionen anwenden</accessor>
  <type>parallelgateway</type>
</ADD>
<ADD id="3" pos="3" mmodeltype="alternative">
  <name>Load And-Split_alt_0</name>
  <predecessor>Load And-Split</predecessor>
  <accessor>Retail/WS-Daten laden</accessor>
  <type></type>
  <cterm></cterm>
</ADD>
<ADD id="4" pos="5" mmodeltype="alternative">
  <name>Load And-Split_alt_1</name>
  <predecessor>Load And-Split</predecessor>
  <accessor>Portfolio laden</accessor>
  <type></type>
  <cterm></cterm>
</ADD>
<MOVE id="5">
  <name>Portfolio laden</name>
  <to>
    <pos>4</pos>
    <predecessor>Load And-Split_alt_0</predecessor>
    <accessor>Load And-Join</accessor>
  </to>
  <from>
    <pos>2</pos>
    <predecessor>start</predecessor>
    <accessor>Retail/WS-Daten laden</accessor>
  </from>
</MOVE>
<MOVE id="6">
  <name>Retail/WS-Daten laden</name>
  <to>
    <pos>6</pos>
    <predecessor>Load And-Split_alt_1</predecessor>
    <accessor>Load And-Join</accessor>
  </to>
  <from>
    <pos>3</pos>
    <predecessor>Portfolio laden</predecessor>
    <accessor>Report-Positionen anwenden</accessor>
  </from>
</MOVE>
```

14.8 Anhang zu Kapitel „Change-Log Bereinigung“

Persistente Highlevel-Changes aus Abbildung 52

```
<!-- Delta_1 -->
<ADD id="1" pos="3" mmodeltype="activitiy">
       <name>Entity-Daten laden</name>
       <type>task</type>
       <predecessor>Portfolio-Daten laden</predecessor>
       <accessor>Retail/WS-Daten laden</accessor>
</ADD>
<UPDATE id="2" pos="7" name="LCR berechnen">
       <changeupdate param="maxDuration" type="value">
              <changeto>310</changeto>
              <changefrom>300</changefrom>
       </changeupdate>
</UPDATE>

<!-- Delta_2 -->
<DELETE id="1" pos="3" mmodeltype="activitiy">
       <name>Entity-Daten laden</name>
       <type>task</type>
       <predecessor>Portfolio-Daten laden</predecessor>
       <accessor>Retail/WS-Daten laden</accessor>
</DELETE>
<UPDATE id="2" pos="5" name="Gewichtungen anwenden">
<changeupdate param="maxDuration" type="value">
              <changeadd>
                     <param>
                            <name>procCall</name>
                            <valueType>plsql-call</valueType>
                            <value>exec pack_cust_biii.exec_weightings</value>
                     </param>
              </changeadd>
       </changeupdate>
</UPDATE>


<!-- Delta_3 -->
<UPDATE id="1" pos="6" name="LCR berechnen">
       <changeupdate param="maxDuration" type="value">
              <changeto>350</changeto>
              <changefrom>310</changefrom>
       </changeupdate>
</UPDATE>
```

Persistente Highlevel-Changes aus Abbildung 53

```
<!-- Delta_1 -->
<ADD id="1" pos="17" mmodeltype="activitiy">
       <name>Derivatives Netting</name>
       <type>task</type>
       <predecessor>Gewichtungen anwenden</predecessor>
       <accessor>Report erzeugen</accessor>
</ADD>
<MOVE id="2">
       <name>Entity-Daten laden</name>
       <to>
              <pos>13</pos>
              <predecessor>Ratings laden</predecessor>
              <accessor>AND-Load Join</accessor>
       </to>
       <from>
              <pos>5</pos>
              <predecessor>Portfolio laden</predecessor>
              <accessor>AND-Load Join</accessor>
       </from>
</MOVE>

<!-- Delta_2 -->
<DELETE id="1" pos="17" mmodeltype="activitiy">
       <name>Derivatives Netting</name>
       <type>task</type>
       <predecessor>Gewichtungen anwenden</predecessor>
       <accessor>Report erzeugen</accessor>
</DELETE>

<!-- Delta_3 -->
<ADD id="1" pos="15" mmodeltype="activitiy">
       <name>Derivatives Netting</name>
       <type>task</type>
       <predecessor>AND-Load Join</predecessor>
       <accessor>Report-Positionen anwenden</accessor>
</ADD>

<!-- Delta_4 -->
<MOVE id="1">
       <name>Derivatives Netting</name>
       <to>
              <pos>6</pos>
              <predecessor>Portfolio-Daten laden</predecessor>
              <accessor>AND-Load Join</accessor>
       </to>
       <from>
              <pos>15</pos>
              <predecessor>AND-Load Join</predecessor>
              <accessor>Report-Positionen anwenden</accessor
       </from>
</MOVE>
<MOVE id="2">
       <name>Entity-Daten laden</name>
       <to>
              <pos>2</pos>
              <predecessor>start</predecessor>
```

```
            <accessor>AND-Load Split</accessor>
      </to>
      <from>
            <pos>13</pos>
            <predecessor>Ratings laden</predecessor>
            <accessor>AND-Load Join</accessor>
      </from>
</MOVE>
```

Persistente Highlevel-Changes aus Abbildung 54

```
<!-- Delta_1 -->
<ADD id="1" pos="1" mmodeltype="activitiy">
      <name>Entity-Daten laden</name>
      <type>task</type>
      <predecessor>Portfolio-Daten laden</predecessor>
      <accessor>Retail/WS-Daten laden</accessor>
</ADD>
<MOVE id="2" >
      <name>Net Deivatives</name>
      <to>
            <pos>7</pos>
            <predecessor>Gewichtungen anwenden</predecessor>
            <accessor>Report erzeugen</accessor>
      </to>
      <from>
            <pos>5</pos>
            <predecessor>Retail/WS-Daten laden</predecessor>
            <accessor>Report-Positionen anwenden</accessor>
      </from>
</MOVE>

<!-- Delta_2 -->
<SWAP id="1">
<obj>
    <name>Entity-Daten laden</name>
    <to>
      <pos>4</pos>
      <predecessor>Portfolio-Daten laden</predecessor>
      <accessor>Report-Positionen anwenden</accessor>
    </to>
    <from>
      <pos>1</pos>
      <predecessor>start</predecessor>
      <accessor>Portfolio-Daten laden</accessor>
    </from>
  </obj>
  <obj>
    <name>Retail/WS-Daten laden</name>
    <to>
      <pos>1</pos>
      <predecessor>start</predecessor>
      <accessor>Portfolio-Daten laden</accessor>
    </to>
    <from>
```

```
      <pos>4</pos>
      <predecessor>Portfolio-Daten laden</predecessor>
      <accessor>Report-Positionen anwenden</accessor>
    </from>
  </obj>
</SWAP>
<UPDATE id="2" pos="7" name="Derivatives Netting">
       <changeupdate param="name" type="value">
              <changeto>Derivatives Netting</changeto>
              <changefrom>Net Derivatives</changefrom>
       </changeupdate>
</UPDATE>
```

14.9 Kurzfassung

In dieser Masterarbeit werden Ansätze und Methoden zur Erkennung von Differenzen zwischen Geschäftsprozessmodellen erstellt, beschrieben und implementiert. Weiteres wird der Aufbau eines Versionskontrollsystems für Geschäftsprozesse behandelt und konzeptioniert. Zuerst wird ein theoretischer Überblick über den Aufbau von Geschäftsprozessen, unterschiedlicher Patterns in der Modellierung von Prozessmodellen, grundsätzliche Änderungsoperationen sowie semantisch höherwertige Änderungsoperationen und die Anwendung von Change-Logs gegeben. Um ein Versionskontrollsystem und eine Differenzenerkennung unabhängig von der verwendeten Prozessnotation zu erstellen, wird in Kapitel zwei ein Meta-Modell vorgestellt, welches den Ansprüchen blockstrukturierter Prozessmodelle entspricht. Dieses minimale Superset ist die Grundlage für die Ablage der Prozessmodelle innerhalb des Versionskontrollsystems. Zur Erkennung der Differenzen zwischen zwei Prozessmodellen wird ein hybrider Ansatz vorgestellt. Dieser Ansatz beinhaltet Methodiken zur Erkennung von Differenzen wie sie auch in der Bioinformatik und in der Plagiatserkennung angewendet werden. Zusätzlich wird zur Erkennung von höherwertigen Änderungsoperationen ein mengen-theoretischer Ansatz verfolgt. Es wird ein gänzlich neues Verfahren zur Erkennung von Verschiebungen von Prozessobjekten vorgestellt und hinsichtlich der Ähnlichkeitsanalyse liegt der Fokus auf dem semantischen Charakter eines Prozessobjektes. Der Abschnitt über die Differenzenerkennung wird mit der Persistierung der erkannten Änderungen in Change-Logs abgerundet. Das letzte Kapitel widmend sich dem Aufbau des Versionskontrollsystems. Hier werden Konzepte aus der Literatur zusammengefasst und erweitert. So wird ein Konzept zum Branching von Prozessmodell-Versionen vorgeschlagen. Die Rückrechnung von Prozessmodellen auf ältere Versionen erfolgt über ein minimales Set an Änderungs-Operationen und berücksichtig dabei die höherwertigen Änderungs-Operationen. Die Evaluierungen zeigen, dass durch die eingesetzten Verfahren sich aus einfachen Änderungs-Operationen höherwertige Operationen ableiten lassen und dabei unabhängig von der Prozessmodellnotation zu sein. Ebenso wird verdeutlicht, dass Versionskontrollsysteme für Geschäftsprozesse mit semantisch höherwertigen Änderungsoperationen unterlegt und mit diesen auch grundlegende Funktionalitäten erreicht werden können.

Schlagwörter: Differenzenerkennung, Versionskontrolle, Ähnlichkeitsanalyse, Änderungs-Operationen

14.10 Abstract

In this thesis, approaches and methods for the detection of differences between business process models are created, described and implemented. Additionally, the composition of a version control system and the versioning of business process models will be conceptualized. First, a theoretical overview of the structure of business processes, different patterns in the modeling of process models, basic change operations, and semantically higher change operations and the application of change logs are presented. In order to create a version control system and difference detection independently of the used business process model notation, a meta-model will be established which meets the requirements of block-structured process models. This minimal superset is also the basis for the storage of process models within the version control system.

A hybrid approach is presented for identifying differences between two process models. This approach includes methodologies for the detection of differences as they are also used in bioinformatics and in the detection of plagiarism. In addition, a quantitative theoretical approach is used for the recognition of high-level change operations. An entirely new method for detecting shifts of process objects is presented. The underlying similarity analysis focuses on the semantic nature of process objects. The section about detection of differences is completed by the serialization of the detected changes in change logs.

The last chapter is devoted to building the version control system. Here different concepts from the relevant specialized literature are summarized and extended. A concept for branching of process model versions is proposed. The back propagation of changes of process models to older versions uses a minimal set of change operations, and includes thereby high-level changes. The evaluations show that with the developed methods and concepts it is possible to derive high-level changes from change-primitives, independent of the process model notation. Likewise, it is shown that version control systems for business process models can hold high-level changes without losing the control over previous model versions.

Tags: differences detection, version control, meta-model, similarity analysis, Change Operations

14.11 Lebenslauf

Lebenslauf mit Fokus auf die akademische Laufbahn

Persönliche Daten

Name:	Schreier
Vorname:	Lukas
Geburtsdatum:	19.05.1985
Geburtsort:	Wien
Staatsbürgerschaft:	Österreich
Email:	lukas.schreier@chello.at

Ausbildung

1991 – 1995	Private Volksschule der Schulbrüder-Marianum
1995 – 1999	Private Hauptschule der Schulbrüder-Währung
1999 – 2004	Schule für Informatikkaufleute
2005 – 2006	Studium der Medizinischen Information an der TU Wien
2006 – 2010	Studium der Wirtschaftsinformatik an der WU-Wien
2010	Abschluss Studium der Wirtschaftsinformatik an der WU-Wien als BSc (WU)
2011 – 2014	Masterstudium der Wirtschaftsinformatik an der Universität Wien
2014	Abschluss Masterstudium der Wirtschaftsinformatik an der Universität Wien

Wien, Juli 2014