Diplomarbeit an der Universität Ulm
- Fakultät für Informatik -
Abteilung Datenbanken und Informationssysteme

Konzeption eines Rahmenwerkes zur Erstellung und Modifikation von Prozessvorlagen und -instanzen

vorgelegt von: Martin Jurisch

Erstgutachter:	Prof. Dr. Peter Dadam
Zweitgutachter:	Dr. Stefanie Rinderle
Betreuer:	Ulrich Kreher, Markus Lauer

Ulm, März 2006

Kurzfassung

Der Einsatz von Prozess-Management-Systemen (PMS) zur rechnerbasierten Verwaltung und Steuerung von Geschäftsprozessen gewinnt zunehmend an Bedeutung. Damit ein PMS in umfassender Weise einsetzbar ist, müssen die von ihm verwalteten Prozessschemata und -instanzen bei Bedarf rasch geändert werden können. Dies betrifft neben den einfachen Änderungen von Prozessschemata und -instanzen auch die Fähigkeit, Schemaänderungen auf laufende Instanzen übertragen zu können (Schemaevolution). Weiterhin muss ein flexibles PMS auch das Zusammenspiel zwischen Schema- und Instanzänderungen unterstützen.
Die heutigen auf dem Markt verfügbaren PMS bieten aber entweder gar keine Möglichkeit bestehende Prozesse und Instanzen anzupassen oder die angebotenen Mechanismen sind derart unzureichend implementiert, dass es in der Folge einer Änderung zu Inkonsistenzen oder gar Systemabstürzen kommen kann.
Um dieses Problem zu lösen wurden in der Forschung einige konzeptionelle Ansätze entwickelt. Diese besitzen alle den Nachteil, dass sie sich immer nur mit einem Teil der geforderten Änderungsmechanismen befassen und meist von solch theoretischer Natur sind, dass eine effiziente Implementierung nicht möglich ist.

Ausgehend von dieser Problemstellung ist es das Ziel dieser Arbeit, ein Änderungsrahmenwerk zu erstellen, das alle Arten von Änderungen unterstützt, die Korrektheit geänderter Prozesse und Instanzen in jedem Fall gewährleistet und in Form einer Änderungskomponente direkt und effizient implementiert werden kann.

Die Ausgangsbasis hierfür bilden die konzeptionellen Arbeiten zum Thema Flexibilität in PMS. Anhand dieser werden die Anforderungen an ein Änderungsrahmenwerk herausgearbeitet und die Grundkonzepte zur Unterstützung von Flexibilität erläutert. Die Darstellung der Konzepte erfolgt dabei in integrierter Form, d.h. die Teilaspekte aus den konzeptionellen Arbeiten werden zu einem Ansatz vereint. Um eine strukturelle Änderung umfassend unterstützen zu können, werden auf konzeptioneller Basis bereits bestehende Änderungsoperationen um konkrete Vor- und Nachbedingungen erweitert und beschrieben, wie diese in das Rahmenwerk eingebunden werden. Für die Grundfunktionalität des Änderungsrahmenwerkes werden implementierungsnahe Konzepte und Algorithmen vorgestellt, mit denen ein Prozess erstellt und geändert werden kann. Hierbei werden insbesondere softwaretypische Aspekte wie Benutzerfreundlichkeit und Erweiterbarkeit berücksichtigt. Zur Unterstützung der Schemaevolution werden die Konzepte aus den theoretischen Ansätzen beschrieben und deren Grenzen aufgezeigt. Darauf aufbauend stellt diese Arbeit vor, wie unter Berücksichtigung implementierungstechnischer Details, wie der internen Repräsentation von Prozessen, die Konzepte in optimierter und implementierungsnaher Form umgesetzt werden können.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die fortschreitende Globalisierung und die damit verbundene deregulierte Öffnung der Märkte [WiGl06], verlangt von den Unternehmen eine effiziente Nutzung der vorhandenen Kapazitäten, um die Stellung am Markt behaupten zu können. Dieser Umstand führte, geprägt durch den Begriff *Business Reengineering* [HaCh94], in vielen Unternehmen dazu, dass sich die Sichtweise auf die wertschöpfenden Abläufe grundlegend geändert hat. Sie konzentrieren sich nun nicht mehr auf die funktionalen Abläufe, sondern vielmehr auf die aus Kundensicht wertschöpfenden Prozesse. Der damit einhergehende Paradigmenwechsel, von einer funktionalen zu einer prozessorientierten Organisationsstruktur, führt zu fundamental veränderten Anforderungen an die Umsetzung betrieblicher Prozesse in den Informationssystemen eines Unternehmens [Öste95]. Anwendungsprogramme und die zugehörigen Daten müssen nun nicht in einzelnen funktionalen Einheiten sondern über Abteilungs- oder sogar Unternehmensgrenzen hinweg verfügbar sein. Hier wurden bereits entsprechende Systeme wie *Enterprise-Ressource-Planning* (ERP) und *Enterprise-Application-Integration* (EAI) entwickelt. Für sich betrachtet bieten diese für den Anwender allerdings keine aktive Unterstützung bei der Ausführung eines Geschäftsprozesses. Vielmehr geschieht die prozessorientierte Verknüpfung der benötigten Anwendungsfunktionen bei diesen rein funktions- und datenzentrierten Systemen weitgehend in den Köpfen der Mitarbeiter oder „hart verdrahtet“ in den Anwendungsprogrammen selbst [DaRe04]. Dies ist weder hinsichtlich einer optimalen Ausnutzung der effizienzsteigernden Effekte des prozessorientierten Paradigmas noch in Bezug auf eine schnelle Änderbarkeit bestehender Prozesse akzeptabel. Deshalb wird zunehmend sowohl von Unternehmens- als auch von Anwenderseite gefordert, den Ablauf eines Geschäftsprozesses gesamtheitlich zu unterstützen. Dabei soll ein geeignetes prozessorientiertes Informationssystem die Durchführung unternehmensweiter und -übergreifender Abläufe koordinieren, Anwendungskomponenten prozessorientiert integrieren, Benutzer ablaufbezogen unterstützen, den Fortgang der Prozesse überwachen und ihren realen Verlauf möglichst lückenlos dokumentieren [JBSc99, Ober96].

Als viel versprechender Ansatz erweisen sich hierfür Prozess-Management-Systeme (PMS). Durch deren charakteristisches Merkmal, die Prozesslogik vom Anwendungscode zu trennen (vgl. Abbildung 1.1), ist es möglich, die Ablauflogik eines Arbeitsprozesses, dem PMS explizit durch (graphische) Modellierung bekannt zu machen und somit eine „versteckte“ Darstellung im Anwendungscode zu verhindern.

Dazu wird in einem PMS für jeden Geschäftsprozess-Typ ein zugehöriges Prozessschema modelliert und im System hinterlegt. In einem solchen Schema werden die verschiedenen Aspekte eines Geschäftsprozesses wie Kontroll- und Datenflüsse, Bearbeiterzuordnungen oder Ausnahmebehandlungen beschrieben. Weiterhin können den Prozessschritten (Aktivitäten) Anwendungskomponenten zugeordnet werden, die während der Bearbeitung des Geschäftsprozesses zur Ausführung kommen. Auf Basis eines solchen Prozessschemas können dann neue Instanzen erzeugt werden, für die das PMS die Durchführung von Aktivitäten koordiniert, anstehende Aktivitäten den Anwendern über Arbeitslisten anbietet, deren fristgerechte Durchführung überwacht, die zugehörigen Anwendungsprogramme mit den benötigten Daten aufruft [RReD02] und den Verlauf in einer Ausführungshistorie speichert.

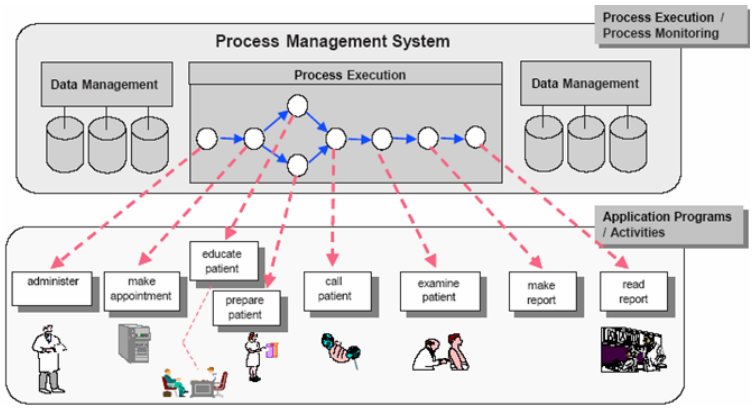


Abbildung 1.1 Trennung von Prozesslogik und Anwendungscode in einem Prozess-Management-System [Rind04]

Der Anwender erhält mit einem PMS unbestritten eine umfassende Unterstützung bei der Ausführung eines Geschäftsprozesses. Nebenbei entstehen durch den Einsatz eines PMS zusätzliche Vorteile. So ist es beispielsweise durch die explizite Übernahme der Prozesslogik durch das PMS nicht mehr notwendig diese Logik in den Anwendungsprogrammen zu implementieren. Dies erleichtert die Entwicklung dieser Programme maßgeblich, garantiert die Unabhängigkeit von einem bestimmt Prozess und ermöglicht somit deren Einsatz in beliebigen Geschäftsprozessen.

1.2 Flexibilität: Das Kernkriterium von Prozess-Management-Systemen

Trotz der beschriebenen Vorteile bei der Verwendung eines PMS, konnten sich diese in der Praxis bisher nicht durchsetzen. Dies begründet sich u.a. damit, dass ein solch klar beschriebener Arbeitsablauf, wie er in einem Schema definiert wird, in der Realität meist von kurzer Dauer ist. So zwingt beispielsweise eine Gesetzesänderung oder eine neue Marktsituation ein Unternehmen, seine bisherigen Prozesse an die neuen Gegebenheiten anzupassen. Heutige auf dem Markt verfügbare PMS erlauben es jedoch entweder gar nicht die bestehenden Prozesse anzupassen, oder aber eine Änderung kann in der Folge zu Inkonsistenzen oder gar Systemabstürzen führen [Rind04].

Sollen PMS aber in umfassender Weise für die rechnerbasierte Verwaltung und Steuerung von Geschäftsprozessen einsetzbar sein, müssen die von ihnen verwalteten Prozessschemata und -instanzen bei Bedarf rasch angepasst werden können. Dabei gilt es grundsätzlich zwei Arten von Änderungen zu unterscheiden: Zum einen Änderungen auf Prozess-Typ-Ebene (Schema) und zum anderen Änderungen auf Instanzebene.

Eine Änderung auf Prozess-Typ-Ebene wird immer dann notwendig, wenn auf Grund von Gesetzesänderungen, geänderter Marktsituation oder einfach durch Optimierung eines vorhandenen Geschäftsprozesses das bisher geltende Schema nicht mehr den gewünschten Ablauf repräsentiert. Wird ein bestehender Prozess-Typ geändert, so muss ein PMS zumindest gewährleisten, dass diejenigen Prozessinstanzen, die auf dem ursprünglichen Prozess-Typ-Schema gestartet wurden, ohne Störung beendet werden können. Dies lässt sich z.B. durch geeignete Versionskonzepte erreichen. Für Prozesse kurzer Dauer ist ein solches Konzept eventuell noch ausreichend. Für Prozesse mit langer Laufzeit, wie z.B. der Abwicklung von Leasingverträgen oder Hypothekendarlehen, führt dieses Verfahren aber zu erheblichen Problemen. Einerseits kann es durch die evtl. lange Koexistenz von Instanzen unterschiedlicher Schemaversionen eines Prozess-Typs zu einem Durcheinander bei der Prozessabwicklung kommen. Andererseits ist es möglich, dass z.B. auf Grund einer Gesetzesänderung ein Fortführen von Instanzen auf dem „alten“ Schema gar nicht zulässig ist. Deshalb besteht von

Anwenderseite die Forderung, dass die Prozess-Typ-Änderungen – wo sinnvoll und möglich – auch auf die darauf laufenden Instanzen übertragen werden können. Man spricht in diesem Zusammenhang auch von der Propagation einer Prozess-Typ-Änderung auf laufende Prozessinstanzen bzw. von der Migration verträglicher Prozessinstanzen auf das geänderte Prozessschema. Dabei muss bei einem solchen Vorgang gewährleistet sein, dass es in der Folge nicht zu Inkonsistenzen oder Fehlern bei der Ausführung von migrierten Instanzen kommt.
Zusätzlich zu diesen Änderungen auf Prozess-Typ-Ebene muss ein „flexibles“ PMS auch die Anpassung von Instanzen zur Laufzeit erlauben, um so auf spezielle Kundenwünsche oder Ausnahmefälle reagieren zu können.
Kommt es nach einer solchen instanzspezifischen Änderung zusätzlich zu einer Änderung auf Prozess-Typ-Ebene, so muss ein PMS auch in der Lage sein, die bereits individuell modifizierten Prozessinstanzen auf das geänderte Prozessschema migrieren zu können. Ein flexibles Prozess-Management-System muss also auch das Zusammenspiel von Prozess-Typ- und Prozessinstanz-Änderungen in geeigneter Weise unterstützen.

Diese Funktionalität bieten die kommerziellen PMS noch nicht einmal annähernd. Dies ist im Hinblick auf die stetig steigende Notwendigkeit, besonders im betrieblichen Umfeld, auf geänderte Situation schnellstmöglich reagieren zu können, völlig indiskutabel.
Hinzu kommt, dass auch Ansätze aus der Forschung in vielerlei Hinsicht die Problematik nur unzureichend behandeln. So existiert kein Ansatz, der bei vollständigem Metamodell[1] alle Änderungsarten, also Prozessschema und -instanz-Änderungen, sowie die Kombination von beiden Arten unterstützt und dabei zusätzlich die Korrektheit gewährleistet. Selbst wenn ein solcher Ansatz gewisse Aspekte vollständig behandelt, so geschieht dies meist auf einer rein konzeptionellen Basis. Dadurch wird nicht ersichtlich, ob sich die entwickelten Mechanismen, unter Berücksichtigung der an eine Softwarekomponente gestellten Anforderungen, wie Benutzerfreundlichkeit, Effizienz und Erweiterbarkeit, überhaupt implementieren und in die Gesamtstruktur eines PMS integrieren lassen.

1.3 Ziele und Aufgabenstellung der Arbeit

Ziel dieser Arbeit ist es, ein implementierungsnahes Konzept für ein Änderungsrahmenwerk zu entwickeln, dass direkt in Form einer Änderungskomponente in eine prototypische Implementierung eines Prozess-Management-Systems (ADEPT2) integriert werden kann [ADEPT].
Aus funktionaler Sicht soll dieses Änderungsrahmenwerk neben der Erstellung und Änderung von Prozessschemata sowie der Modifikation von Prozessinstanzen auch die Funktionalität anbieten, Schemaänderungen auf laufende Instanzen übertragen zu können (Schemaevolution). Hier muss insbesondere auch das Zusammenspiel zwischen Schema- und Instanzänderungen beachtet und durch geeignete Mechanismen unterstützt werden.

Als Ausgangsbasis sind bestehende konzeptionelle Ansätze zur Behandlung von Flexibilität in PMS zu untersuchen und zu prüfen, in wie weit enthaltende Konzepte als Grundlage für das Änderungsrahmenwerk geeignet sind. Unter dem Aspekt, dass es bisher keinen Ansatz gibt, der alle für ein Änderungsrahmenwerk notwendigen Mechanismen behandelt, ist es eines der Hauptziele, die Erkenntnisse aus den verschiedenen konzeptionellen Arbeiten zu vereinen und in ein ganzheitliches Konzept zu integrieren. Dabei muss geprüft werden, an welchen Stellen sich einerseits

[1] Man spricht von einem vollständigen Metamodell, wenn die Menge der zur Verfügung stehenden Modellierungskonstrukte nicht eingeschränkt ist (z.B. durch Verbieten von Schleifen oder nicht Beachtung von Datenflusskonstrukten).

Gemeinsamkeiten und andererseits Unterschiede zwischen den Konzepten zur Instanzänderung und zur Schemaevolution ergeben. Anhand dessen ist im Hinblick auf die Vermeidung von Redundanzen zu untersuchen, welche Konzepte mit den gleichen Mechanismen umgesetzt werden können und welche eine separate Vorgehensweise erfordern.

Da es sich bei den zu betrachtenden Ansätzen um rein logische und somit implementierungsunabhängige Konzepte handelt, ist zu prüfen, wie diese für den Einsatz im Änderungsrahmenwerk optimiert werden können. Hierbei sind besonders implementierungsspezifische Details wie die interne Repräsentation von Schema- und Instanzdaten auszunutzen.
Weiterhin ist das Änderungsrahmenwerk in die bestehende Architektur des ADEPT2-PMS einzugliedern und zu beschreiben, mit welchen Komponenten kommuniziert und auf welche Daten zugegriffen werden kann. In diesem Zusammenhang gilt es das vorhandene ADEPT2-Datenmodell so zu erweitern, dass die Mechanismen des Änderungsrahmenwerkes optimal unterstützt werden können.

Die Grundlage für die Durchführung einer Änderung bilden die für ADEPT2 definierten Änderungsoperationen. Diese sollen durch geeignete Vor- und Nachbedingungen garantieren, dass auch nach deren Anwendung die Korrektheit eines Schemas bzw. einer Instanz erhalten bleibt. Dazu sind bereits vorhandene strukturelle Vorbedingungen zu evaluieren, gegebenenfalls anzupassen und in das Änderungsrahmenwerk einzubinden. Zusätzlich müssen für die Änderungsoperationen zustandsbasierte Vorbedingungen, Nachbedingungen und klare Regeln zur Zustandsneubewertung betroffener Aktivitäten definiert werden.

Weiterhin sind bei der Konzeption des Rahmenwerkes auch softwarespezifische Nebenbedingungen zu berücksichtigen. Dazu gehören Aspekte wie Erweiterbarkeit, Flexibilität und Benutzerfreundlichkeit. So soll das Änderungsrahmenwerk eine Plug-In-Schnittstelle bereitstellen, mit deren Hilfe man neue Änderungsoperationen in das System einbringen kann. Weitere Aspekte umfassen eine *UNDO*-Funktion und einen Aufrufbarkeitstest für Änderungsoperationen. Zusätzlich soll die Funktionalität des Änderungsrahmenwerkes für die Verwendung in einer graphischen Schnittstelle (GUI) optimiert sein, gleichzeitig aber auch für ein komponentenunabhängiges Ansprechen über eine API zur Verfügung stehen.
Aus diesen implementierungsnahen Anforderungen ergeben sich Wechselwirkungen mit den Mechanismen aus den konzeptionellen Ansätzen, die unter Umständen eine direkte Implementierung der jeweiligen Funktion verhindern. Solche Konflikte sind aufzudecken und die Mechanismen so zu ändern, dass diese bei einer späteren Implementierung der Änderungskomponente direkt umgesetzt werden können.

1.4 Aufbau

Die Arbeit ist folgendermaßen aufgebaut: Kapitel 2 befasst sich mit konzeptionellen Ansätzen zur Unterstützung von Flexibilität in PMS. Diese Ansätze werden untersucht und in Kapitel 3 die für das Änderungsrahmenwerk benötigten Grundlagen und -konzepte vorgestellt. Basierend auf den daraus gewonnenen Informationen und den in der Aufgabenstellung geforderten Eigenschaften des Änderungsrahmenwerkes werden in Kapitel 4 konkrete funktionale und nicht-funktionale Anforderungen an das Änderungsrahmenwerk definiert. In Kapitel 5 wird das Änderungsrahmenwerk in die bestehende Architektur des ADEPT2-PMS eingegliedert. Dabei wird unter Betrachtung der Kommunikation zwischen den Komponenten beschrieben, auf welche Daten das Änderungsrahmenwerk zurückgreifen kann. Weiterhin wird das Rahmenwerk entlang der Anforderungen aus

Kapitel 4 in einzelne Komponenten unterteilt. Kapitel 6 beschreibt die Mechanismen zum Erstellen und Ändern eines Prozessschemas sowie für die Manipulation einer Instanz. Dabei wird anhand einiger Änderungsoperationen detailliert erläutert, welche strukturellen und zustandsbasierten Vorbedingungen vor einer Änderung zu prüfen sind, wie die strukturellen Änderungen materialisiert werden und welche Aktivitätenzustände nach der Durchführung angepasst werden müssen. Ein weiterer Schwerpunkt in Kapitel 6 ist die Beschreibung der implementierungsspezifischen Konzepte wie das Plug-In-Interface, die *UNDO*-Funktionalität und das Prüfen der kontextabhängigen Aufrufbarkeit einer Änderungsoperation. Kapitel 7 befasst sich umfassend mit den Mechanismen zur Durchführung einer Schemaevolution. Dabei werden unter Ausnutzung der internen Repräsentation von Schema- und Instanzdaten, bestehende konzeptionelle Ansätze dahingehend angepasst, dass sie unter Berücksichtigung der in Kapitel 6 beschriebenen implementierungsspezifischen Funktionen, direkt und mit maximaler Effizienz implementiert werden können. In Kapitel 8 werden verwandte Arbeiten auf ihre Fähigkeit hin untersucht, die Anforderungen an ein flexibles PMS zu erfüllen. Kapitel 9 gibt einen Ausblick auf Aspekte, die im Anschluss an diese Arbeit weiterverfolgt werden können. In Kapitel 10 werden die Ergebnisse der Arbeit zusammengefasst.

2 Konzeptionelle Ansätze zur Unterstützung von Flexibilität in Prozess-Management-Systemen

Die Ausgangsbasis für die Erstellung eines umfassenden Änderungsrahmenwerkes bilden konzeptionelle Ansätze zum Thema Flexibilität in Prozess-Management-Systemen. In diesem Kapitel werden existierende Ansätze untersucht und geprüft, welcher Ansatz bzw. welche Konzepte sich am besten als Grundlage für ein umfassendes Änderungsrahmenwerk eignen. Dazu werden in Abschnitt 2.1 die Anforderungen an einen idealen Ansatz aufgestellt. Anhand dieser Anforderungen werden in Abschnitt 2.2 die Ansätze untersucht und die Ergebnisse in 2.3 zusammengefasst.

2.1 Anforderungen an konzeptionelle Ansätze

Eine sinnvolle Untersuchung konzeptioneller Ansätze erfordert die Aufstellung von Kriterien, nach denen die Ansätze beurteilt werden können. Diese Kriterien sind hier so zu wählen, dass sie exakt den Kriterien eines idealen Ansatzes zur Unterstützung von Flexibilität in PMS entsprechen. Die zu erfüllenden Anforderungen definieren sich folgendermaßen:

1. **Vollständigkeit**: Der Prozessmodellierer wird weder im Hinblick auf das angebotene Metamodell noch bei den zur Verfügung gestellten Änderungsoperationen eingeschränkt. Die Verwendung eines Metamodells mit einem vollständigen Satz an Konstrukten für die Modellierung und Änderung des Kontroll- und Datenflusses ist dabei Grundvoraussetzung. Weiterhin ist es insbesondere im Hinblick auf die umfassende Unterstützung aller Änderungsarten (Instanzänderungen, Schemaänderungen, etc.) notwendig, dass auch das Zusammenspiel zwischen dauerhaften Änderungen eines Prozesses (Schemaänderung) und instanzspezifischen Änderungen berücksichtigt wird.
2. **Korrektheit**: Bei allen vorgestellten Mechanismen ist die Korrektheit der neu modellierten und der laufenden Prozesse gewährleistet. Das bedeutet insbesondere im Hinblick auf Adaptivität, dass bei allen durchgeführten Änderungen die Korrektheit eines Schemas bzw. einer Instanz erhalten bleibt. Dabei ist es unbedeutend, ob es sich um instanzspezifische Änderungen, dauerhafte Änderungen oder um die Übertragung von Schemaänderungen auf laufende Instanzen handelt.
3. **Effizienz**: Die zur Umsetzung eines vorgestellten Konzeptes notwendigen Algorithmen befinden sich bezüglich ihrer Komplexität in einem praktisch umsetzbaren Rahmen. Dies ist insbesondere wichtig für Mechanismen, die der Anpassung von laufenden Instanzen auf ein geändertes Schema dienen, da hier erwartungsgemäß die höchste Wiederholungsrate auftritt und somit die algorithmische Komplexität möglichst gering zu halten ist.
4. **Benutzerfreundlichkeit/Automation**: Es wird versucht, die für die Flexibilität notwendigen Mechanismen soweit wie möglich zu automatisieren und damit den Anwender zu entlasten.

Erfüllt ein konzeptioneller Ansatz alle der aufgestellten Kriterien umfassend, so können die darin enthaltenen Mechanismen unter Berücksichtigung implementierungstechnischer Details direkt im Änderungsrahmenwerk umgesetzt werden.

2.2 Untersuchung konzeptioneller Ansätze

In der Forschung existieren zahlreiche Ansätze zum Thema Flexibilität in PMS. Diese unterscheiden sich zum Teil bedeutend in Art und Umfang der unterstützten Modellierungskonstrukte, Änderungsarten und Korrektheitsmechanismen. Als Folge davon eignen sich die Ansätze bzw. die

darin enthaltenen Konzepte unterschiedlich gut als Grundlage für ein umfassendes Änderungsrahmenwerk. Die Aufgabe dieses Abschnitts ist es, die existierenden Ansätze zu untersuchen und zu identifizieren, welche Ansätze bzw. Konzepte sich für eine weitere Betrachtung am besten eignen. Dabei konzentrieren sich die Beschreibungen ausschließlich auf den Umfang, in welchem die Konzepte in der Lage sind, die im vorherigen Abschnitt aufgestellten Anforderungen zu erfüllen. Eine umfassende Behandlung der Grundkonzepte findet sich in [RReD04]. Im Folgenden wird entlang des Metamodells in Ansätze basierend auf Petri-Netzen und Ansätzen mit Graph-/Aktivitäten-basiertem Metamodell unterschieden.

2.2.1 Petri-Netz-basierte Ansätze

Als Vertreter mit einem Petri-Netz-basierten Metamodell werden die konzeptionellen Ansätze *WF Nets* [AaBa02, AWWi03], *Flow Nets* [ElKe00, EKRo95, ElMa97] und *MILANO* [AgDe00a, AGDe00b] genauer betrachtet:

WF Nets: Bei *WF Nets* wird ein Prozessschema über ein beschriftetes Stellen-/Transitionsnetz repräsentiert. Das Metamodell erlaubt es Sequenzen, parallele und alternative Verzweigungen, sowie Schleifen zu modellieren. Die strukturelle Korrektheit eines Prozessschemas lässt sich über die bei Petri-Netzen übliche Erreichbarkeitsanalyse bestimmen. Bezüglich Kontrollfluss sind *WF Nets* vollständig. Das Metamodell hat jedoch den gravierenden Nachteil, dass Datenflussaspekte nicht berücksichtigt werden.
Bezüglich der Änderung bereits modellierter Prozesse bzw. Instanzen, bieten *WF Nets* die Möglichkeit, Sequenzen, Verzweigungen und Schleifen hinzuzufügen und zu löschen. Das Verschieben bestehender Konstrukte ist hingegen nicht erlaubt.
Für die Korrektheit bei der Durchführung einer Änderung werden in [AaBa02] formale Kriterien definiert. Ob eine Änderung auf einen bestehenden Prozess angewendet werden kann, lässt sich über Vererbungsbeziehungen entscheiden. Dabei wird das geänderte Petri-Netz *S'* mit dem originalen Netz *S* verglichen. Handelt es sich bei *S* um ein Sub-Netz von *S'* oder ist dies umgekehrt der Fall, so ist die Änderung verträglich und somit das resultierende Prozessschema bzw. die resultierende Instanz korrekt (siehe hierzu [AaBa02, RRD04]). Ob *S* und ein beliebig geändertes Netz *S'* in einer Vererbungsbeziehung zueinander stehen, ist in der Praxis jedoch schwierig zu entscheiden, da das Problem PSPACE-vollständig ist.
Wie sich bei der Migration einer geänderten Instanz auf ein geändertes Schema eine Sub-Netz-Beziehung erkennen lässt wird nicht beschrieben. Weiterhin werden zur Prüfung der Verträglichkeit keine Laufzeitinformationen, wie beispielsweise der Ausführungszustand eines Netzes verwendet. Dadurch ist es möglich bereits durchlaufende Abschnitte des Schemas zu ändern. Da das Metamodell von *WF Nets* keine Datenflussaspekte berücksichtigt, ist dies nicht weiter problematisch. Für ein umfassendes Änderungsrahmenwerk (inkl. Datenfluss) ist das Konzept jedoch unbrauchbar. Ein Ändern bereits durchlaufener Abschnitte könnte dazu führen, dass nachfolgende Aktivitäten/Transitionen nicht mit Daten versorgt werden (vgl. Abbildung 2.1).

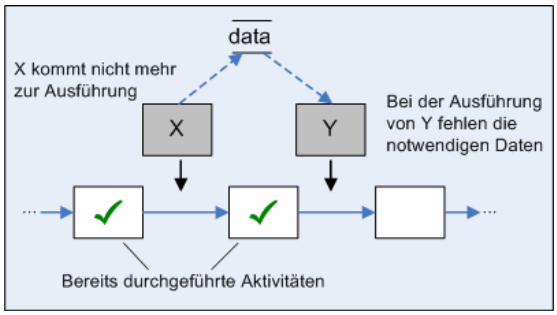


Abbildung 2.1 Fehlende Datenversorgung durch Änderung bereits durchlaufener Abschnitte [RRD04]

Flow Nets: Das Metamodell bei *Flow Nets* ist bezüglich Kontrollfluss und Ausführungssemantik mit dem bei *WF Nets* verwendeten Modell vergleichbar. Der entscheidende Unterschied besteht darin, dass in diesem Fall mehr als ein Token pro Stelle möglich ist. Datenflussaspekte werden somit nicht von vornherein ausgeklammert. Weiterhin ermöglichen *Flow Nets* neben dem Einfügen und Löschen von Kontrollkonstrukten auch Reihenfolgeänderungen. Die Umsetzung der Änderungen findet dabei über ein Substituieren des von den Änderungen betroffenen Netz-Bereichs (*change region*) statt (vgl. Abbildung 2.2). Wie sich der betroffene Bereich allerdings berechnen lässt, wird nicht beschrieben.

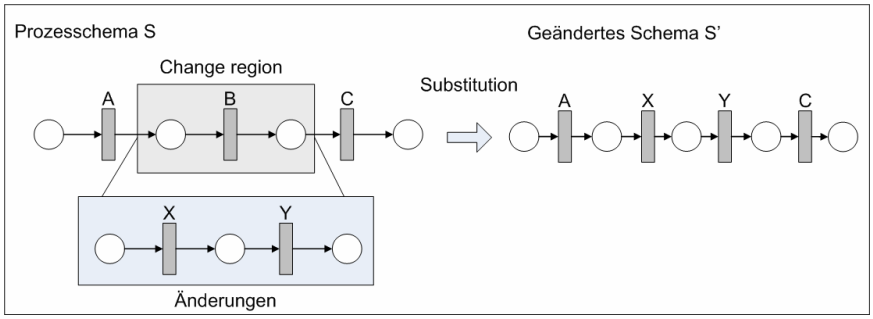


Abbildung 2.2 Änderung durch Substitution bei *Flow Nets*

Die Kriterien ob eine Instanz *I* bzw. dessen Ausführungszustand mit einem geänderten Schema *S'* verträglich ist, beruhen auf der Überprüfung der Ausführungsgleichheit (*trace-equivalence*). [EKRo95] vergleicht hierzu die auf dem ursprünglichen Schema *S* erzeugte Instanzmarkierung *m* mit der aus der Migration von *I* auf *S'* resultierenden Markierung *m'*. Wie *m'* berechnet werden kann, wird wiederum nicht beschrieben. Bei dem auf *Flow Nets* basierendem Prototyp *Chautauqua* [ElMa97] muss deshalb der Benutzer „von Hand" eine korrekte Instanzmarkierung erzeugen. In Anbetracht der im Normalfall großen Anzahl zu migrierender Instanzen, ist dieses Konzept für die Praxis ungeeignet. Eine weitere Betrachtung des Ansatzes erübrigt sich.

MILANO: Das Metamodell des *MILANO*-Ansatzes basiert ebenfalls auf Petri-Netzen. Im Gegensatz zu *WF Nets* und *Flow Nets* ist die Ausdrucksmächtigkeit beschränkt. Das Modellieren von Schleifen ist nicht möglich. Weiterhin werden auch Datenflussaspekte nicht explizit behandelt. Die Möglichkeiten zur Änderung modellierter Prozesse bzw. Instanzen sind ebenfalls sehr begrenzt. Es können lediglich vorhandene Aktivitäten parallelisiert, sequenzialisiert oder in ihrer Reihenfolge vertauscht werden. Konzepte zum Einfügen neuer oder Löschen bestehender Aktivitäten werden nicht beschrieben.
Die starke Vereinfachung bezüglich der angebotenen Konstrukte und Änderungsmöglichkeiten erleichtert im Gegenzug die Definition eines Verträglichkeitskriteriums: Eine Instanz *I* ist mit einem geänderten Schema *S'* verträglich, wenn der momentane Ausführungszustand von *I* auf *S* auch in *S'* vorhanden ist. Die möglichen Zustände von *S'* lassen sich über den Erreichbarkeitsgraphen des

geänderten Schemas bestimmen. Die Berechnung eines solchen Erreichbarkeitsgraphen ist relativ einfach und wird im Umfeld von Petri-Netzen häufig angewendet.
Trotz der einfachen Verträglichkeitsberechnung ist der Ansatz insgesamt zu restriktiv.

Alle Petri-Netz-basierten Ansätze besitzen noch zusätzliche Nachteile, die aus den Eigenschaften der Petri-Netze resultieren. Es ist beispielsweise keine explizite Trennung zwischen Kontroll- und Datenfluss möglich. Durch die implizite Schleifen-Modellierung werden Korrektheitsüberprüfungen wie Verklemmungs-Tests erschwert. Weiterhin kann bei Aktivitäten lediglich zwischen aktiviert (*activated*) und nicht aktiviert (*not-activated*) unterschieden werden. Dadurch ist beim Löschen einer Aktivität nicht ersichtlich, ob bereits ein Teil der bei einer Aktivität auszuführenden Arbeit durchgeführt wurde. Dies ist insofern problematisch, da unklar ist, wie mit teilweise ausgeführter Arbeit verfahren wird.

2.2.2 Ansätze mit Graph-/Aktivitäten-basiertem Metamodell

Die Ansätze in [CCPP98, KrGe99, Wesk00, Sadi00, Reic00, ReDa98] verwenden zur Darstellung und Ausführung eines Prozesses ein Graph-/Aktivitäten-basiertes Metamodell. Bei diesem Modell können neben aktiviert und nicht-aktiviert weitere Zustände unterschieden werden. Durch eine klare Abgrenzung zwischen aktiviert und laufend (*running*) kann das Problem durch teilweise ausgeführte Arbeit vermieden werden.

WIDE [CCPP98]: Das Metamodell von *WIDE* ist bezüglich den angebotenen Konstrukten zur Modellierung von Kontroll- und Datenfluss vollständig. Weiterhin werden in [CCPP98] Änderungsoperationen beschrieben, mit denen ein Schema *S* in *S'* geändert werden kann. Sowohl für das Modellieren als auch für das Ändern eines Graphen existieren klare Kriterien bei deren Erfüllung die Korrektheit gewährleistet ist. Die Ausführung einer *WIDE*-Graph-Instanz *I* wird in einer Ausführungshistorie mitprotokolliert.
Die Überprüfung, ob *I* korrekt auf ein geändertes Graphschema *S'* migriert werden kann, beruht auf *trace-equivalence*. Bei *WIDE* ist *I* genau dann mit *S'* verträglich, wenn die Ausführungshistorie von *I* auch auf dem geänderten Schema *S'* erzeugbar gewesen wäre. Wie dies geprüft werden kann wird in [CCPP98] nicht angegeben. [RReD04] geht davon aus, dass zur Überprüfung der Verträglichkeit die gesamte Ausführungshistorie von *I* auf *S'* „nachgespielt" werden muss. Dies hat den Vorteil, dass nach der Migration automatisch eine korrekte Zustandsmarkierung für *I* resultiert. Ein sonst übliches Anpassen von Aktivitätenzuständen entfällt. Das Nachspielen ist jedoch sehr ineffizient, da Historiendaten aufgrund ihrer Größe im Normalfall im Sekundärspeicher gehalten werden müssen [KrGe99]. Weiterhin werden viele Aktivitäten neu bewertet, die gar nicht von einer Änderung betroffen sind. Das Verfahren ist somit besonders bei einer großen Anzahl zu migrierender Instanzen zu langsam. Hinzu kommt, dass in der Historie nur die Ereignisse bei Beendigung der Aktivitäten (Aktivitäten-End-Ereignisse) gespeichert werden. Eine Entscheidung zwischen aktiviert und laufend ist somit wie bei Petri-Netzen nicht möglich. Weiterhin ist die Betrachtung der gesamten Historie bzgl. Schleifen zu restriktiv, d.h. das Verträglichkeitskriterium ist schleifenintollerant. In Abbildung 2.3 wird die Instanz *I* als unverträglich erkannt, da die Historie Π_I nicht auf *S'* produziert werden kann. Tatsächlich entstehen aber keine Probleme, wenn *I* auf *S'* weiter ausgeführt wird.
Insgesamt ist *WIDE* besonders bei der Überprüfung der Verträglichkeit zu ineffizient.

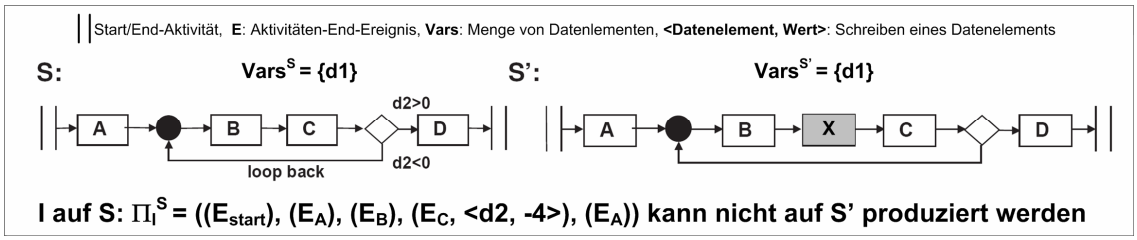


Abbildung 2.3 Schleifenintolleranz bei der Verträglichkeitsprüfung [CCPP98]

TRAMs [KrGe99]: Im Gegensatz zu den anderen graph-/aktivitätenbasierten Ansätzen wird in *TRAMs* der Datenfluss explizit modelliert. Der Kontrollfluss wird stattdessen deklarativ durch Definition klarer Aktivitäten-Start- und End-Bedingungen beschrieben. Schleifen werden nicht unterstützt. Durch die fehlende Modellierung von Kontrollkanten ist ein Ändern von Prozessen schwierig. So erfordert beispielsweise das Einfügen von Aktivitäten, die Angabe umfassender Start- und End-Bedingungen durch Benutzereingriff. Das Kriterium der Benutzerfreundlichkeit bzw. Automation erfüllt der Ansatz somit nicht.
Zur Prüfung der Verträglichkeit einer Instanz *I* mit einem geänderten Schema *S'*, wird bei *TRAMs* dasselbe Kriterium verwendet wie bei *WIDE*. Interessanterweise wird in [KrGe99] nicht die gesamte Historie nachgespielt. Es werden stattdessen für jede Änderungsart (z.B. Einfügen, Löschen einer Aktivität oder einer Datenkante) Kriterien aufgestellt, mit Hilfe derer die Verträglichkeit von *I* bestimmt werden kann.

WASA₂ [Wesk00, Wesk01]: Kontroll- und Datenfluss werden explizit modelliert. Die strukturelle Korrektheit eines Schemas *S* ist dann gewährleistet, wenn der modellierte Graph azyklisch ist. Ein separates Schleifenkonstrukt existiert nicht. Somit ist es nicht möglich iterative Prozesse abzubilden. Bei der Darstellung von Instanzen verwendet *WASA₂* eine modell-inhärente Aktivitätenmarkierung, d.h. die Zustände der in einem Instanzschema enthaltenen Aktivitäten werden direkt im Modell bzw. im Instanzschema gespeichert. Dadurch kann der Ausführungszustand über ein gekürztes Instanzschema dargestellt werden. Dieses Schema enthält nur diejenigen Aktivitäten (inkl. Kanten), die bereits ausgeführt wurden oder sich gerade in Ausführung befinden. Ein Verträglichkeitstest wird damit folgendermaßen definiert:
Eine Instanz *I* ist dann mit einem geänderten Schema *S'* verträglich, wenn jede Aktivität und jede Kante des gekürzten Instanzschemas auch in *S'* enthalten ist. Wie dieser Test effizient durchgeführt werden kann, und wie verträgliche Instanzen an das geänderte Schema *S'* angepasst werden können, wird nicht beschrieben. Weiterhin verhält sich das Verträglichkeitskriterium bzgl. Verschiebeoperationen zu restriktiv. Die Instanz *I* aus Abbildung 2.4 wäre beispielsweise ohne weiteres migrierbar. Das Verträglichkeitskriterium verbietet dies jedoch, weil die im gekürzten Instanzschema vorhanden Kante (*B*, *C*) nicht in *S'* enthalten ist.

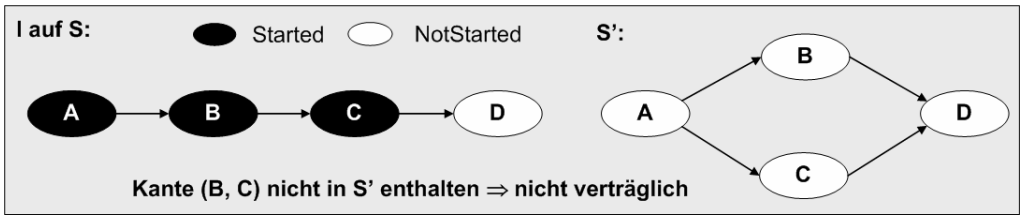


Abbildung 2.4 Parallelisierung von B und C

Breeze [Sadi00, SMOr00]: Ein Prozessschema wird bei *Breeze* als gerichteter azyklischer Graph dargestellt. Es können Sequenzen, parallele und alternative Verzweigungen, sowie komplexe

Aktivitäten modelliert werden. Schleifen sind nicht zulässig. Ein Schema ist strukturell korrekt, wenn alle Knoten auf einem Pfad zwischen eindeutigem Start- und Endknoten liegen. Neben dem Kontrollfluss wird auch der Datenfluss explizit dargestellt. *Breeze* gewährleistet die Korrektheit des Datenflusses durch Überprüfung der Datenversorgung entsprechender Aktivitäten.
Bezüglich der Migration laufender Instanzen wird in *Breeze* kein Verträglichkeitskriterium angegeben. Stattdessen beschreibt der Ansatz ein *Rollback*-Konzept, mit dem die Ausführung unverträglicher Instanzen soweit rückgängig gemacht werden kann, dass eine Migration auf das geänderte Schema möglich ist. Wie bei dieser Rücksetzung Aktivitäten kompensiert werden und wie sich der Datenfluss verhält, wird nicht erläutert. Weiterhin existieren keine Angaben wie die Instanzen auf das geänderte Schema zu migrieren sind.

WSM-Nets [Reic00, ReDa98]: Bei *WSM-Nets* handelt es sich um wohlstrukturierte azyklische Graphen. Das Metamodell ist bezüglich den angebotenen Konstrukten zur Modellierung von Kontroll- und Datenfluss vollständig. Die Modellierung von Schleifen erfolgt über explizite Schleifenkonstrukte. Dies ermöglicht einen einfachen Korrektheitstest. Ein Schema ist dann korrekt, wenn es abgesehen von den expliziten Schleifen azyklisch ist. Instanzen besitzen eine modell-inhärente Zustandsmarkierung. Zusätzlich werden sowohl Start- als auch Endereignisse in einer Historie gespeichert.
Für die Änderung bereits bestehender Schemata *S* und Instanzen *I* werden in [Reic00] konkrete Änderungsoperationen beschrieben, bei deren Anwendung die Korrektheit von *S* bzw. *I* automatisch gewährleistet wird. Die Migration von Instanzen auf ein geändertes Schema wird in [Reic00] nicht berücksichtigt.
Mit [Rind04] existiert allerdings ein Ansatz, bei dem basierend auf *WSM-Nets* Migrationskonzepte beschrieben werden. Dieser behandelt neben der Definition von Verträglichkeitskriterien und der Vorstellung von Konzepten zur korrekten Migration unveränderter Instanzen auch die Migration bereits veränderter Instanzen.

2.3 Zusammenfassung und Fazit

Um eine geeignete konzeptionelle Grundlage für das Änderungsrahmenwerk zu finden, wurden in diesem Kapitel existierende Ansätze zum Thema Flexibilität in PMS untersucht. Die Ergebnisse werden im Folgenden zusammengefasst:
Die Petri-Netz-basierten Ansätze *WF Nets*, *Flow Nets* und *MILANO* besitzen die prinzipbedingten Schwächen der Petri-Netze. *WF Nets* und *MILANO* ermöglichen deshalb keine Datenflussmodellierung. Schleifen existieren weder in *Flow Nets* noch in *MILANO*. Bezüglich der Änderung bereits bestehender Schemata und der Fähigkeit Instanzen auf ein geändertes Schema zu migrieren, verhalten sich die Ansätze unterschiedlich. MILANO definiert klare und praktisch berechenbare Kriterien, beschränkt die Änderungsmöglichkeiten aber auf simple Sequenzialisierung, Parallelisierung oder Reihenfolgeänderung bereits vorhandener Aktivitäten. *WF Nets* und *Flow Nets* ermöglichen umfassendere Änderungen, die Korrektheitskriterien und Migrationskonzepte sind in der Praxis aber nicht umsetzbar. Als konzeptionelle Grundlage für ein umfassendes Änderungsrahmenwerk sind die Petri-Netz-basierten Ansätze deshalb unbrauchbar.
Ein Graph-/Aktivitäten-basiertes Metamodell ist für die Prozessmodellierung besser geeignet. *WIDE* bietet einen vollständigen Satz an Modellierungs- und Änderungskonstrukten sowohl für Kontroll- als auch für Datenfluss. Die Überprüfung der Verträglichkeit erfordert jedoch das Nachspielen der kompletten Änderungshistorie von *I* auf *S'*. Dies ist für eine große Anzahl an Instanzen zu ineffizient. *$WASA_2$* umgeht dieses Problem durch eine modell-inhärente Zustandsmarkierung und einem damit

erzeugbaren gekürzten Instanzschema. Wie die darauf basierenden Verträglichkeitskriterien getestet werden, wird nicht angegeben. *Breeze* beschreibt überhaupt keine Konzepte zur Verträglichkeitsüberprüfung und zur Migration von Instanzen. Stattdessen wird erläutert, wie unverträgliche Instanzen in einen verträglichen Zustand zurückgesetzt werden können. Zusätzlich werden weder bei *WASA*$_2$ noch bei *Breeze* Schleifen berücksichtigt. *TRAMs* modelliert anstelle des Kontrollflusses explizit den Datenfluss. Dadurch wird die Änderung des Kontrollflusses massiv erschwert. Weiterhin bleibt abzuwarten, ob das datengetriebene Modell in der Praxis eine sinnvolle Unterstützung des Anwenders ermöglicht. Als konzeptionelle Grundlage für das Änderungsrahmenwerk ist *TRAMs* deshalb nicht geeignet. *WSM-Nets* bieten – wie *WIDE* – einen vollständigen Satz an Modellierungs- und Änderungskonstrukten an. Dabei ist die Korrektheit einer Instanz oder eines Schemas bei der Anwendung der in [Reic00] beschriebenen Änderungsoperationen in jedem Fall gewährleistet. Zusammen mit den Konzepten zur Migration von Instanzen aus [Rind04] eignet sich der Ansatz am besten als Grundlage für ein umfassendes Änderungsrahmenwerk. Insbesondere auch deshalb, weil kein anderer Ansatz die Kombination aus geänderten Instanzen und geändertem Schema berücksichtigt.

3 Grundlagen und Änderungskonzepte

In diesem Kapitel werden die konzeptionellen Grundlagen für die Entwicklung des Änderungsrahmenwerkes vorgestellt. Ausgangsbasis bilden die im vorherigen Kapitel als geeignet identifizierten Ansätze aus [Reic00] und [Rind04]. Da diese aber jeweils nur einen Teil der geforderten Änderungsfunktionalität behandeln, werden die darin enthaltenen Konzepte integriert dargestellt. Gleichzeitig wird der vorgesehene Einsatzzweck des Änderungsrahmenwerkes als Komponente des ADEPT2-PMS [ADEPT] berücksichtigt, indem die vorgestellten Konzepte direkt an speziell für ADEPT2 entwickelte Änderungsoperationen angepasst werden.

In den Abschnitten 3.1 und 3.2 werden als Grundlage für die weiteren Beschreibungen das Prozess-Metamodell und die zur Manipulation notwendigen Änderungsoperationen vorgestellt. Darauf aufbauend befasst sich der Abschnitt 3.3 mit den Konzepten zur Durchführung einer Instanzänderung. Abschnitt 3.4 beschreibt Schemaänderungen und die notwendigen Schritte um laufende Instanzen korrekt auf ein geändertes Schema migrieren zu können. In Abschnitt 3.5 werden die für eine praktische Umsetzung entscheidenden Konzepte zur internen Repräsentation von Schema- und Instanzänderungen vorgestellt.

3.1 Das Prozess-Metamodell

Die Ansätze aus [Reic00] und [Rind04] verwenden als Grundlage für ihre Überlegungen bezüglich Flexibilität wohlstrukturierte azyklischen Netze (*WSM-Nets*). Da die Eigenschaften dieses Prozess-Modells für das Verständnis der Änderungskonzepte von elementarer Bedeutung sind, werden die Modellierungs- und Laufzeitaspekte des Modells in den Abschnitten 3.1.1 und 3.1.2 vorgestellt.

3.1.1 Modellierungsaspekte – Wohlstrukturierte azyklische Netze

Bei wohlstrukturierten azyklischen Netzen (kurz: WA-Netz) handelt es sich um attributierte, serien-parallele Graphen mit zusätzlichen Synchronisationskanten. Sie umfassen alle für die Prozessmodellierung relevanten Aspekte wie beispielsweise Kontrollfluss, Datenfluss, Zeit und Ressourcen. Für das Änderungsrahmenwerk sind allerdings hauptsächlich der Kontroll- und Datenfluss (Abschnitte 3.1.1.1 und 3.1.1.2) sowie die Kriterien zu deren Korrektheit (Abschnitt 3.1.1.3) relevant.

3.1.1.1 Kontrollfluss

Die Darstellung des Kontrollflusses in einem WA-Netz geschieht über ein blockstrukturiertes Prozess-Modell. Es können sowohl Sequenzen, parallele und alternative Verzweigungen als auch Schleifen modelliert werden.

Der in Abbildung 3.1 modellierte Prozess zeigt alle für eine weitere Betrachtung relevanten Konstrukte: Der Kontrollfluss des modellierten Prozesses enthält neben einfachen Sequenzen (z.B. *C*, *D*, *E*) auch Verzweigungen und ein durch den Schleifenstartknoten L_S und den Schleifenendknoten L_E begrenztes Schleifenkonstrukt (*Loop-Edge*). Die Verzweigungen lassen sich untergliedern in parallele und alternative Verzweigungen. So handelt es sich bei dem durch den *AND-Split*-Knoten *A* und den *AND-Join*-Knoten *T* umfassten Block um eine parallele Verzweigung, also eine Verzweigung bei der immer beide Teilzweige zur Ausführung kommen. Knoten *M* und *S* hingegen definieren eine alternative Verzweigung, bei der abhängig von einem festgelegten Entscheidungskriterium nur derjenige Teilzweig zur Ausführung kommt, dessen *Selection Code* (Sc) das Kriterium erfüllt. Bei dem durch *F* und *J* umfassten Konstrukt handelt es sich um einen Spezialfall einer parallelen

Verzweigung. Bei dieser so genannten parallelen Verzweigung mit finaler Auswahl kommen zuerst beide Teilzweige zur Ausführung. Am zugehörigen *OR-Join*-Knoten wird dann entschieden, welcher Teilzweig gültig ist und welcher zurückgesetzt wird. Zwischen den Knoten *J* und *D* befindet sich eine Synchronisationskante. Diese ermöglicht die Abbildung einer Ausführungsabhängigkeit zwischen Aktivitäten unterschiedlicher Teilzweige. Das bedeutet, dass die Aktivität *D* erst dann zur Ausführung kommen kann, wenn entweder Aktivität *J* erfolgreich ausgeführt wurde oder wenn klar ist, dass *J* nicht zur Ausführung kommt. Dieser Kantentyp ist notwendig, da die strikte Blockstrukturierung die Abbildung einer Ausführungsabhängigkeit zwischen Knoten unterschiedlicher Teilzweige mit den sonst üblichen Kontrollkanten verbietet.

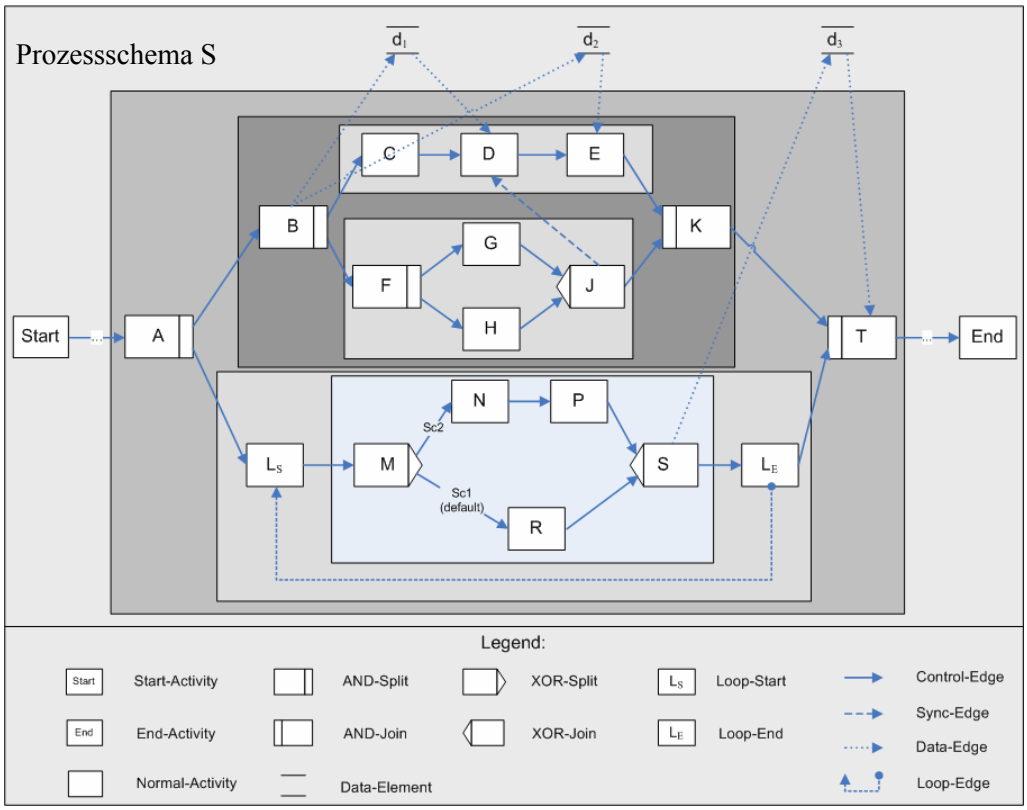


Abbildung 3.1 Darstellung eines Prozessschemas durch einen wohlstrukturierten azyklischen Netzgraph

Die mit unterschiedlichen Grautönen hinterlegten Rechtecke verdeutlichen die regelmäßige Blockstruktur. Dies bedeutet, dass Kontrollstrukturen, wie Sequenzen, Verzweigungen und Schleifen auf eindeutige Start- und Endaktivitäten abgebildet werden. Solche, als Kontrollblöcke bezeichneten Strukturen, können beliebig ineinander verschachtelt sein, dürfen sich aber nicht überlappen. Durch die dadurch erzeugte „symmetrische“ Struktur lassen sich Verzweigungs- und Vereinigungsknoten gegenseitig eindeutig zuordnen. Diese Art der Strukturierung bringt zahlreiche Vorteile, da Prozessabläufe übersichtlicher modelliert und zahlreiche Fehler (z.B. falsches Zusammenführen von Teilzweigen) bereits per Konstruktion vermieden werden. So ist es beispielsweise möglich, syntaxgesteuerte Graphikeditoren zu erstellen, die die Zulässigkeit bestimmter Konstrukte bereits im Vorfeld prüfen und somit den Anwender vor der Modellierung inkorrekter Graphen bewahren. Eine solche Umsetzung findet sich im ADEPT-Prototyp [ADEPT]. Weiterhin ermöglicht das Prozess-Metamodell eine wesentlich effizientere Analyse und Verifikation von Kontroll- und Datenflussschemata, was auch im Hinblick auf die Durchführung einer Schemaevolution von entscheidender Bedeutung ist.

3.1.1.2 Datenfluss

Der Datenfluss wird über die Definition globaler Datenelemente modelliert. Dabei werden die Datenelemente über Datenlese- und Datenschreibkante mit den entsprechenden Aktivitäten[2] verbunden, wobei eine Lese-/Schreibkante einen Lese-/Schreibzugriff einer Aktivität auf das entsprechende Datenelement darstellt. Der Datenfluss des in Abbildung 3.1 modellierten Prozesses vollzieht sich über die globalen Datenelemente d_1 bis d_3. Aktivität *B* schreibt beispielsweise das Datenelement d_1, was mit Hilfe einer Datenschreibkante (*Data-Edge*) zwischen *B* und d_1 ausgedrückt wird. Gelesen wird das geschriebene Datum von Aktivität *D* über eine Datenlesekante zwischen d_1 und *D*.

3.1.1.3 Korrektheit

Die bisher vorgestellten Modellierungskonstrukte alleine garantieren allerdings noch kein korrektes Laufzeitverhalten des modellierten Prozesses. Es lassen sich beispielsweise noch mit Hilfe von Synchronisationskanten *Deadlock*-verursachende Zyklen[3] modellieren. Weiterhin kann das korrekte Versorgen von Aktivitäten mit Eingabeparametern noch nicht allein durch die Blockstruktur gewährleistet werden.
Um ein solches Verhalten auszuschließen werden in [Rind04] Einschränkungen definiert, bei deren Einhaltung die Korrektheit eines Netzes garantiert werden kann.

Definition 1 (Strukturelle Korrektheit eines WA-Netzes)
Ein WA-Netz ist genau dann korrekt, wenn es die folgenden Kriterien erfüllt:

1. Der Netzgraph hat genau einen eindeutigen Start- und Endknoten.
2. Abgesehen vom Start- und Endknoten hat jeder Knoten mindestens eine ein- und eine ausgehende Kontrollkante.
3. Der Netzgraph besitzt eine strikte Blockstruktur.
4. Das Netz ist abgesehen von den expliziten Schleifenkonstrukten azyklisch, d.h. die Verwendung von Kontroll- und Sync-Kanten führt nicht zu Zyklen.
5. Synchronisationskanten dürfen weder von Knoten innerhalb eines Schleifenkonstruktes ausgehen, noch dürfen sie zu einem solchen Knoten führen.
6. Für eine Aktivität *A*, bei der ein obligater Eingabeparameter mit einem Datenelement *d* verbunden ist, muss garantiert werden, dass *d* zur Laufzeit unabhängig vom gewählten Ausführungspfad in jedem Fall von einer im Kontrollfluss vor *A* liegenden Aktivität geschrieben wird.
7. Es gibt keine parallelen Schreibzugriffe auf Datenelemente $\Rightarrow$ es gibt keine „*lost updates*“.

Mit den vorgestellten Modellierungskonstrukten und Korrektheitskriterien erhält man ein ausdruckstarkes und leicht zu handhabendes Mittel zur Prozessschemabeschreibung, das im Folgenden als Basis für die Vorstellung der Laufzeitaspekte dient.

---

[2] Wir verwenden Knoten und Aktivitäten (= Knoten mit Aktivitätenvorlage) analog, da eine Unterscheidung hier nicht notwendig ist.

[3] Als *Deadlock*-verursachende Zyklen bezeichnet man bei WA-Netzeb ein Konstrukt, welches zur Laufzeit die erfolgreiche Ausführung blockiert. Dies ist immer dann der Fall, wenn jeweils zwei Knoten auf die erfolgreiche Beendigung des jeweils anderen Knoten warten.

3.1.2 Laufzeitaspekte – Prozessinstanzen und deren Ausführungszustände

Auf Grundlage eines korrekten WA-Netzes S lassen sich Prozessinstanzen erzeugen und ausführen. Diese Instanzen I sind logisch gesehen im Zeitpunkt ihrer Erzeugung exakte Abbilder einer Prozessvorlage (Schema). Die statische Struktur einer Instanz entspricht also der Struktur des von ihr instanziierten WA-Netzes. Man spricht von einem instanzspezifischen Schema S_I. Im Gegensatz zu den Schemata enthalten Instanzen allerdings noch Markierungen, die ihren Ausführungszustand signalisieren. Eine solche instanzspezifische Markierung definiert sich über die Funktion $M^I = (NS^I, ES^I)$, wobei jeder Aktivität von I ihr aktueller Zustand $NS(n)$ und jeder Kontroll-, Synchronisations- und Schleifenrücksprungkante ihre jeweilige Markierung $ES(e)$ zugewiesen wird. Dabei werden anders als zum Beispiel bei Petri-Netzen auch die Markierungen bereits ausgeführter Graphabschnitte und nicht gewählter Teilzweige gespeichert. Der dadurch erhöhte Speicherbedarf wird durch die Möglichkeit kompensiert, Algorithmen – insbesondere bei der Anwendung dynamischer Änderungen – effizienter durchzuführen. Wie eine solche Markierung zustande kommt und welche Ausführungssemantik den in Abschnitt 3.1.1 vorgestellten Modellierungskonstrukten zugrunde liegt, zeigt Abbildung 3.2.

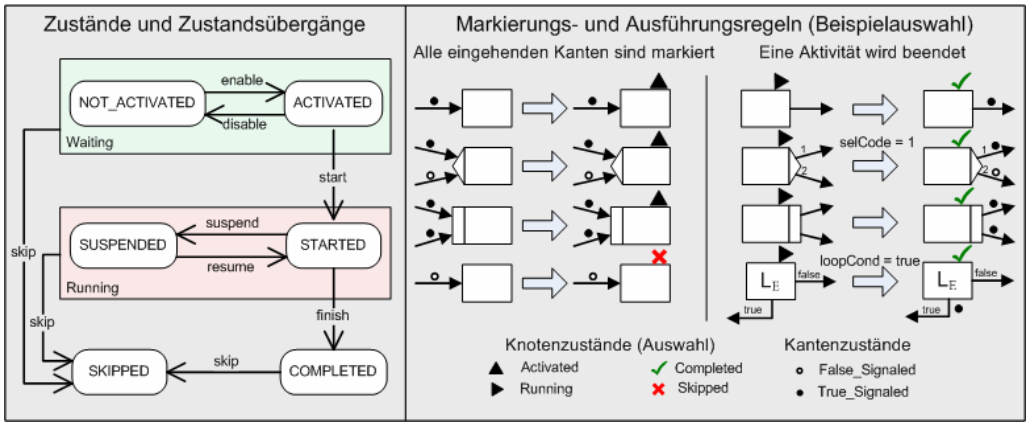


Abbildung 3.2 Zustandsübergangsdiagramm und Markierungs-/Ausführungsregeln [Rind04]

Jede Aktivität einer Instanz befindet sich anfänglich im Zustand *NOT_ACTIVATED*. Sind für deren Ausführung alle Vorbedingungen erfüllt, d.h. alle eingehenden Kanten in Abhängigkeit von der Eingangssemantik der jeweiligen Aktivität entsprechend markiert, so ändert sich der Zustand in *ACTIVATED*. Trifft dies zu, so wird die Aktivität als ausführbarer Arbeitsschritt in die Arbeitslisten berechtigter Benutzer eingetragen. Wählt ein Benutzer einen solchen Arbeitsschritt aus seiner Liste, wird dieser in den Zustand *RUNNING* versetzt und eine mit dieser Aktivität verknüpfte Anwendungskomponente gestartet. Gleichzeitig wird der Arbeitsschritt aus den Arbeitslisten anderer Benutzer entfernt. Wird der Arbeitsschritt erfolgreich abgeschlossen, so ändert sich der Zustand in *COMPLETED*.

Der Zustand *SKIPPED* nimmt eine Sonderstellung ein. Er signalisiert, dass eine Aktivität nicht mehr zur Ausführung kommt. Er kann sowohl aus den übergeordneten Zuständen *WAITING* und *RUNNING* als auch aus dem Zustand *COMPLETED* erreicht werden. Der Übergang von *WAITING* nach *SKIPPED* erfolgt bei einer alternativen Verzweigung für alle Knoten, die sich in einem anderen als dem von der Verzweigungsbedingung ausgewählten Teilzweig befinden (*Dead-Path-Elimination*[4]).

[4] Siehe hierzu [LeRo99]

Bei alternativen Verzweigungen mit finaler Auswahl kann der Zustand *SKIPPED* auch aus den Zuständen *RUNNING* oder *COMPLETED* erreicht werden.
Der Status einer Kante hängt vom Ausführungsstatus ihrer Quellaktivitäten und bei alternativen Verzweigungen und Schleifen zusätzlich von der Auswahlentscheidung- bzw. Schleifenbedingung ab. Initial befindet sich eine Kante im Zustand *NOT_SIGNALED*. Im Laufe der Prozessausführung ändert sich der Zustand einer Kante entweder in *TRUE_SIGNALED* oder *FALSE_SIGNALED* (vgl. Abbildung 3.2).
Evaluiert eine Schleifenbedingung zu „*true*" wird die Schleifenrücksprungkante entsprechend markiert und alle in der Schleife befindlichen Aktivitäten und Kanten werden in ihren initialen Zustand *NOT_ACTIVATED* zurückversetzt. Evaluiert die Bedingung zu „*false*", so bleiben die aktuellen Markierungen und Zustände erhalten und die Schleife wird verlassen.

Mit diesen Markierungs- und Ausführungsregeln lässt sich die Korrektheit einer instanzspezifischen Markierung folgendermaßen bestimmen:

Definition 2 (Korrektheit einer Instanzmarkierung)
Eine instanzspezifische Markierung M^I ist genau dann korrekt, wenn sie ausgehend von der Anfangsmarkierung[5] M^I_0 unter Anwendung der Markierungs- und Ausführungsregeln erreicht werden kann.

Zusammen mit den Anforderungen an die strukturelle Korrektheit aus Definition 1, ergibt sich für Instanzen die folgende Korrektheitsdefinition:

Definition 3 (Korrektheit einer Instanz)
Eine auf dem Schema S basierende Instanz I ist genau dann korrekt, wenn
1. ihr Instanzschema S_I ein nach Definition 1 korrektes WA-Netz ist und
2. ihre Markierung M^I Definition 2 erfüllt.

Durch Definition 3 wird sichergestellt, dass einerseits kein ungewolltes Systemverhalten durch *Deadlock* verursachende Zyklen oder fehlende Eingabeparameter auftritt und andererseits zur Laufzeit keine unerwünschten Zustände eingenommen werden.

Neben den Auswirkungen auf den Kontrollfluss beeinflussen die Laufzeitaspekte auch den Datenfluss. So können Datenelemente zur Laufzeit nicht nur mehrmals gelesen sondern auch mehrmals geschrieben werden. Dies tritt zum Beispiel ein, wenn aus einer Schleife heraus ein Datenelement geschrieben wird. Um dabei zu verhindern, dass Daten von einem erneuten Schreibzugriff überschrieben werden, wird bei jedem schreibenden Zugriff eine neue Version des Datenelements erstellt (*Datenhistorie*). Dadurch ist zu jeder Zeit ein kontextabhängiges Lesen der Daten möglich.

Analog zur Situation beim mehrmaligen Schreiben eines Datenelements kommt es im Falle von Schleifen zu einer mehrmaligen Bewertung von Aktivitäten und Kanten. Da aber immer nur die aktuellen Zustände und Markierungen im Prozessgraph ersichtlich sind, würde der vorhergehende Status im Falle einer erneuten Bewertung (z.B. im Zuge eines erneuten Schleifendurchlaufs) ohne weitere Maßnahmen verloren gehen. Um dies zu verhindern existiert für jede Instanz eine Ausführungshistorie, in der relevante Ereignisse (i.W. Informationen zum Start und zur Beendigung

[5] Die Anfangsmarkierung M^I_0 entspricht dem Zustand, in dem der Start-Knoten aktiviert ist.

von Aktivitäten) des Prozessablaufs mitprotokolliert werden. Aktionen, wie das Markieren nicht mehr ausführbarer Aktivitäten (*SKIPPED*) werden hingegen nicht protokolliert. Weiterhin kann auch auf das Eintragen von Kantenmarkierungen verzichtet werden, da diese aus den Zuständen ihres jeweiligen Quellknotens rekonstruiert werden können. Neben den reinen Start- und Endereignissen werden zusätzlich für jede gestartete Aktivität die Werte der gelesenen und für jede beendete Aktivität die Werte der geschriebenen Datenelemente in der Historie gespeichert. So ist mit Hilfe eines ebenfalls gespeicherten Schleifenzählers eine direkte Zuordnung auch im Falle mehrmaliger Schleifendurchläufe problemlos möglich.

Abbildung 3.3 zeigt die vorgestellten Laufzeitaspekte anhand eines konkreten Beispiels.

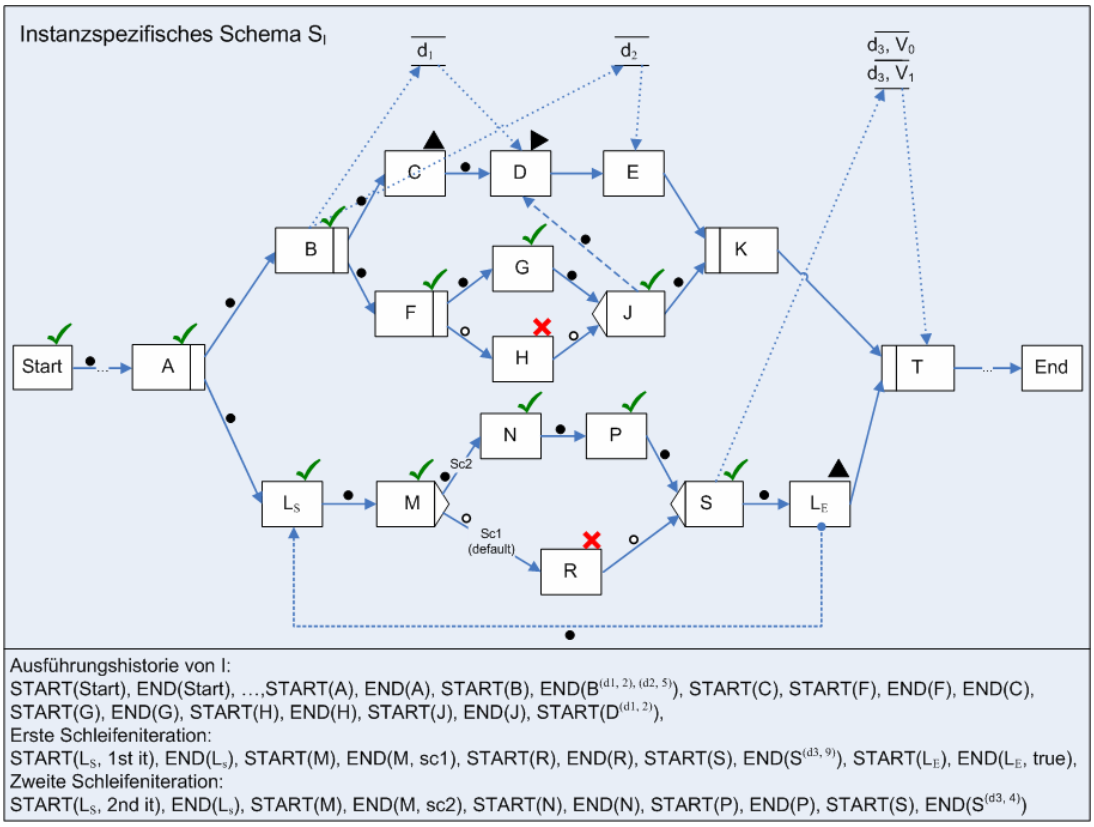


Abbildung 3.3 Instanzspezifisches Schema mit zugehöriger Ausführungshistorie

Zusammen mit den Modellierungskonstrukten und Korrektheitskriterien, kann man mit dem in diesem Abschnitt beschriebenen Laufzeitmodell, beliebige Prozesse modellieren und Instanzen darauf ausführen. Wie bereits bestehende Prozesse und Instanzen geändert werden können, wird im nächsten Abschnitt behandelt.

3.2 Änderungsoperationen

Voraussetzung für die Durchführung beliebiger Prozessänderungen, ist die Bereitstellung einer vollständigen Menge an Änderungsoperationen. Die in [Reic00] erstmals vorgestellten und in [KrAc04] für ADEPT2 überarbeiteten Operationen lassen sich dabei in unterschiedliche semantische Ebenen einteilen (vgl. Abbildung 3.4).

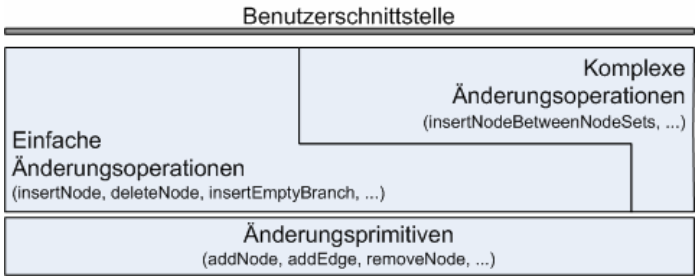


Abbildung 3.4 Semantische Ebenen von Änderungsoperationen [KrAc04]

3.2.1 Änderungsprimitiven

Das Fundament aller Änderungsoperationen bilden die so genannten Änderungsprimitiven. Mit diesen Primitiven können die Mengen der Knoten, Kanten und Datenelemente eines Prozesses direkt manipuliert werden. Beispiele für Änderungsprimitiven sind das Einfügen eines Knotens, ohne diesen in den Prozesskontext einzubinden, das Einfügen und Löschen einzelner Kontrollkanten oder das Setzen von Knotenattributen.

Da wir für das Prozessmodell fordern, dass es bei der Anwendung einer Änderungsoperation von einem nach Definition 1 (bzw. auf Instanzebene nach Definition 3) korrekten Zustand wieder in einen korrekten Zustand überführt wird, dürfen die Änderungsprimitiven allerdings nicht von der Benutzerschnittstelle aus aufrufbar sein. Die Anforderungen an den Benutzer wären immens und damit die Fehleranfälligkeit hoch.

3.2.2 Semantisch höhere Operationen

Um Fehler bei der Anwendung von Änderungsoperationen zu verhindern und die Benutzerfreundlichkeit zu verbessern, werden in [KrAc04] die semantisch höheren Operationen definiert. Diese einfachen und komplexen Änderungsoperationen bauen auf den Änderungsprimitiven auf und können unmittelbar vom Benutzer aufgerufen werden. Für die notwendige Korrektheit des Graphen sorgen dabei die Änderungsoperationen selbst. Dadurch liegt die Verantwortung für die korrekte Durchführung einer Änderung nicht mehr beim Anwender sondern wird direkt vom System übernommen. Dieses Konzept erfordert allerdings das Hinterlegen klarer Ausführungsregeln bei jeder Änderungsoperation. Zu diesem Zweck werden in [KrAc04] strukturelle Vorbedingungen für jede Änderungsoperation angegeben. Diese Vorbedingungen allein reichen aber nicht aus, um insbesondere bei Verschiebeoperationen die strukturelle Korrektheit eines Graphen zu gewährleisten. Dies hängt damit zusammen, dass beispielsweise der Datenfluss erst nach der Anwendung einer strukturellen Änderung überprüft werden kann. Um auch in diesen Fällen die Korrektheit gewährleisten zu können, wurden die Vorbedingungen im Rahmen dieser Arbeit angepasst und um geeignete Nachbedingungen erweitert. Tabelle 3.1 zeigt exemplarisch, wie die für die Verschiebeoperation *moveNodes(first, last, pred, succ)* notwendigen strukturellen Vor- und Nachbedingungen nach dieser Erweiterung aussehen. Eine Auflistung der Vor- und Nachbedingungen aller ADEPT2-Änderungsoperationen findet sich in Anhang A (Änderungsoperationen).

	Funktionale Beschreibung	
moveNodes(first, last, pred, succ)	Verschiebt die durch first und last spezifizierte Knotenmenge (Menge aller Knoten zwischen first und last (inkl.)) zwischen pred und succ; ggf. bleibt ein leerer Teilzweig zurück.	
	Strukturelle Vorbedingung	**Nachbedingung**
moveNodes(first, last, pred, succ)	- pred, succ, first und last sind im Schema enthaltene Knoten - pred und succ sind durch eine Kontrollkante miteinander verbunden (dazu zählen auch Verzweigungs- und Vereinigungsknoten, die durch einen leeren Zweig verbunden sind) - first und last bilden einen gültigen Kontrollblock, d.h. sie befinden sich auf derselben Blockebene	- pred(first) und succ(last) sind mit einer Kontrollkante verbunden, die die gleichen Attribute besitzt, wie sie die ehemalige Kante (pred(first), first) besaß. - der Block (first, last) wurde zwischen (pred, succ) erfolgreich eingefügt, ggf. mit korrektem Auswahlcode - topologische Sortierung (falls vorhanden) wurde aktualisiert - Es existieren keine Zyklen durch Sync-Kanten - Der Datenfluss bleibt korrekt, d.h. die von den verschobenen Knoten benötigten Daten werden alle noch vor Ausführung der Knoten geschrieben und alle verschobenen Knoten schreiben ihre Daten noch rechtzeitig.

Tabelle 3.1 Strukturelle Vor- und Nachbedingung bei moveNodes

Wie in der Tabelle beschrieben, ist das Ausführen der Verschiebeoperation *moveNodes(first, last, pred, succ)* nur dann zulässig, wenn die durch die Knoten *pred* und *succ* festgelegte Zielstelle aus direkt aufeinanderfolgenden Knoten besteht. Weiterhin müssen sich die Knoten *first* und *last*, die die zu verschiebende Knotenmenge begrenzen, auf der gleichen Blockebene befinden (vgl. Abschnitt 3.1.1). Die Nachbedingungen garantieren, dass es durch die Anwendung von *moveNodes* nicht zu Zyklen durch diejenigen Sync-Kanten kommt, die aus den verschobenen Knoten ein- oder ausgehen. Zusätzlich wird auch die Korrektheit des Datenflusses berücksichtigt.
Mit Hilfe dieser Vor- und Nachbedingungen lässt sich nun exakt bestimmen, ob die Anwendung der Änderungsoperation *moveNodes* in einem gegebenen Kontext zu einem korrekten Graphen führt.

3.2.3 Einfache vs. komplexe Änderungsoperationen

Die Unterscheidung zwischen einfachen und komplexen Änderungsoperationen ergibt sich aus der Zusammensetzung der jeweiligen Operationen. Baut eine Operation lediglich auf Änderungsprimitiven auf, so gehört sie zur Gruppe der einfachen Änderungsoperationen. Setzt sie sich hingegen aus einfachen oder wiederum aus komplexen Änderungsoperationen zusammen, so spricht man von einer komplexen Änderungsoperation. Die Ausführbarkeit einer komplexen Änderungsoperation ergibt sich dabei direkt aus der Ausführbarkeit derjenigen Operationen aus denen diese zusammengesetzt ist. Dies gilt unabhängig von der Tatsache, dass auf praktischer Ebene aus Effizienzgründen an Stelle von einfachen Änderungsoperationen direkt Primitiven verwenden werden können.

Abbildung 3.5 zeigt exemplarisch eine einfache und eine komplexe Änderungsoperation inklusive den Operationen, die von diesen verwendet werden (siehe hierzu auch Anhang A (Änderungsoperationen)). Bei der einfachen Änderungsoperation *insertNode* wird ein Knoten *X* automatisch zwischen die beiden Knoten *A* und *B* eingefügt. Die komplexe Änderungsoperation *createSurroundingBlock(type, first, last)* gestaltet sich etwas aufwendiger. Hier wird unter Anwendung einer Folge von Änderungsoperationen eine Verzweigung vom Typ *type* um den durch *first* und *last* definierten Kontrollblock erzeugt.

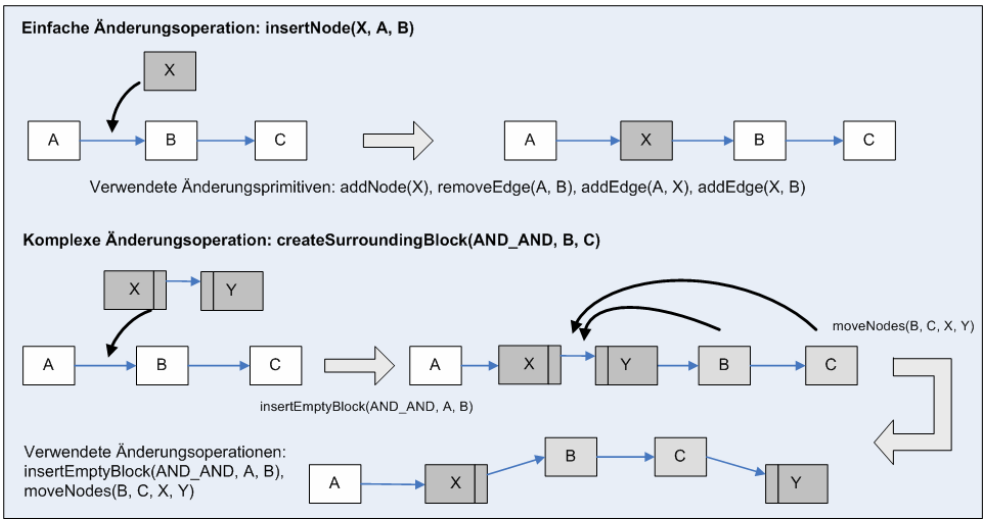


Abbildung 3.5 Beispiel einer einfachen und einer komplexen Änderungsoperation

Man erhält mit dem vorgestellten Konzept einen vollständigen Satz an strukturellen Änderungsmöglichkeiten, bei deren Anwendung die Korrektheit eines Prozessgraphen gewährleistet bleibt. Wann und wie dieser Satz an Änderungsoperationen verwendet wird und welche Erweiterungen im Einzelfall notwendig sind, zeigen die folgenden zwei Abschnitte.

3.3 Instanzspezifische Änderungen

Für die Akzeptanz eines Prozess-Management-Systems ist es zwingend erforderlich, zur Laufzeit Änderungen an einzelnen Instanzen eines Prozesses zu ermöglichen. Eine solche Änderung wird im weiteren Verlauf als instanzspezifische Änderung oder kurz Instanzänderung bezeichnet. Als Folge einer solchen instanzspezifischen Änderung ergibt sich ein vom ursprünglichen Prozessschema S abweichender instanzspezifischer Prozessgraph $S_I := S + \Delta_I$. Δ_I wird dabei als *Bias* bezeichnet und beschreibt die Abweichungen von S_I zum ursprünglichen Schema S.
Es wird (vorerst) für Δ_I die folgende Darstellung verwendet: Δ_I enthält alle bis zu diesem Zeitpunkt auf der Instanz I ausgeführten Änderungsoperationen $op^I_1, \ldots, op^I_n$ (vgl. Abbildung 3.6).

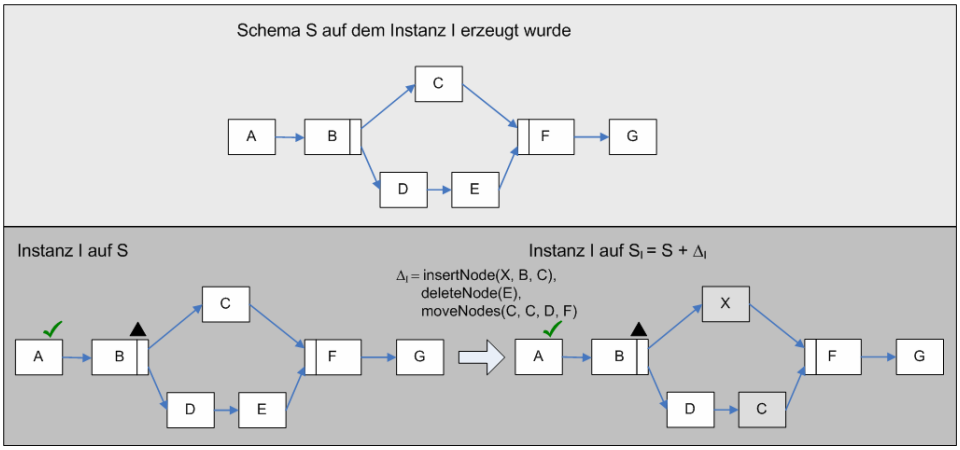


Abbildung 3.6 Instanzspezifische Änderung

Die Durchführung einer solchen Änderung beruht im Wesentlichen auf der Anwendung der bereits vorgestellten Änderungsoperationen. Allerdings gilt es neben den für jede Änderungsoperation definierten strukturellen Vor- und Nachbedingungen auch Laufzeitaspekte zu berücksichtigen. So

kann die Anwendung einer Änderungsoperation zwar aus struktureller Sicht erlaubt sein, der momentane Ausführungszustand einer Instanz kann dies allerdings verbieten. Wie Abbildung 3.7 zeigt, ist die Anwendung der Änderungsoperation *insertNode* für Instanz I_1 zulässig für Instanz I_2 hingegen nicht. Dies liegt daran, dass Instanz I_1 in ihrer Ausführung erst so weit fortgeschritten ist, dass ein Aktivieren des neu eingefügten Knotens *X* nach den Markierungs- und Ausführungsregeln noch möglich ist. Bei I_2 trifft dies nicht zu. Hier befindet sich der Knoten *C* bereits im Zustand *RUNNING*, was ein Einfügen von *X* vor diesem Knoten verbietet. Dies ist auch aus logischer Sicht leicht nachvollziehbar, da es nicht sinnvoll ist, in die laufende Instanz eines Geschäftsprozesses Aktivitäten einzufügen, die nicht mehr zur Ausführung kommen können.

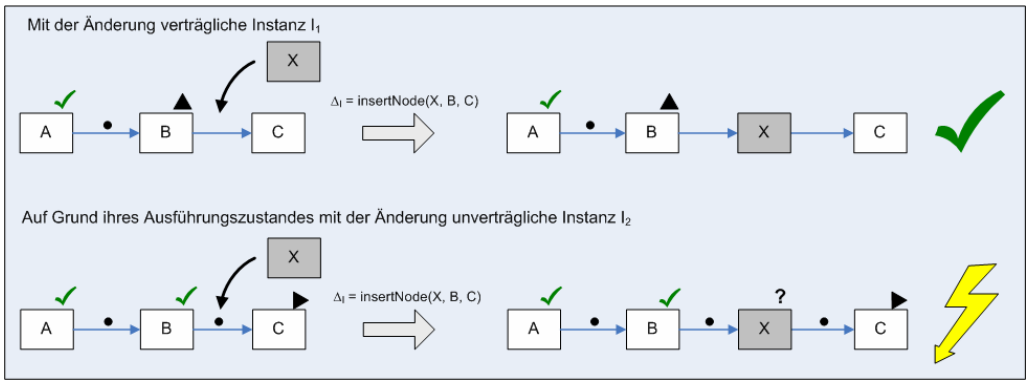


Abbildung 3.7 Zustandsabhängige Verträglichkeit

Um diesem Umstand Rechnung zu tragen, wurden die ADEPT2-Änderungsoperationen im Rahmen dieser Arbeit um zustandsbasierte Vorbedingungen erweitert. Tabelle 3.2 zeigt diese anhand einiger ausgewählter Änderungsoperationen. Die zustandsbasierten Vorbedingungen aller Änderungsoperationen finden sich in Anhang A (Änderungsoperationen).

Änderungsoperation	Zustandsbasierte Vorbedingung
insertNode(X, pred, succ)	Die Aktivität (succ), vor der eingefügt werden soll, darf sich nur in den Zuständen NOT_ACTIVATED oder ACTIVATED befinden
insertSyncEdge(src, dest)	Das Kantenziel (dest) befindet sich in einem der Zustände NOT_ACTIVATED oder ACTIVATED
createSurroundingBlock(LOOP, first, last)	Die erste Aktivität des Blockes, der von der Schleife (LOOP) umschlossen werden soll (first), befindet sich in einem der Zustände NOT_ACTIVATED oder ACTIVATED
deleteNode(X)	Die Aktivität befindet sich in einem der Zustände NOT_ACTIVATED oder ACTIVATED
deleteDataElement(D)	Jede Aktivität die schreibend auf das Datenelement D zugreift befindet sich in einem der Zustände NOT_ACTIVATED oder ACTIVATED.
createDataEdge(node, dataElement, type)	type = READ: der lesende Knoten besitzt einen der Zustände Activated oder Not_Activated type = WRITE: der schreibende Knoten besitzt einen der Zustände Activated oder Not_Activated

Tabelle 3.2 zustandsbasierte Vorbedingungen ausgewählter Änderungsoperationen

Damit lässt sich die Anwendbarkeit einer Änderungsoperation im Zuge einer instanzspezifischen Änderung folgendermaßen definieren:

Definition 4 (Zulässigkeit einer Änderungsoperation auf Instanzebene)

Das Anwenden einer Änderungsoperation *op* auf eine Instanz *I* im Zuge einer instanzspezifischen Änderung ist genau dann zulässig, wenn

1. alle zustandsbasierten Vorbedingungen und

2. alle strukturellen Vor- und Nachbedingungen

der Änderungsoperation erfüllt sind.

Betrachtet man Tabelle 3.2 genauer, so fällt auf, dass die zustandsbasierten Vorbedingungen auch erfüllt werden, wenn sich beteiligte Knoten bereits im Zustand *ACTIVATED* befinden. Dies erfordert einen zusätzlichen Mechanismus, der nach der Ausführung einer zulässigen Operation, die Zustandsmarkierung entsprechend anpasst. Dies kann je nach Änderungsoperation mehr oder weniger aufwendig ausfallen (vgl. Tabelle 0.6 Anhang A (Änderungsoperationen)). So hat das Einfügen eines Knotens *X* vor einem bereits als *ACTIVATED* markierten Knoten *Y* zur Folge, dass die Knotenmarkierung von *Y* zurückgenommen und stattdessen *X* mit *ACTIVATED* markiert werden muss (vgl. Abbildung 3.8). Das Löschen einer Datenkante erfordert hingegen keinerlei Zustandsanpassung.

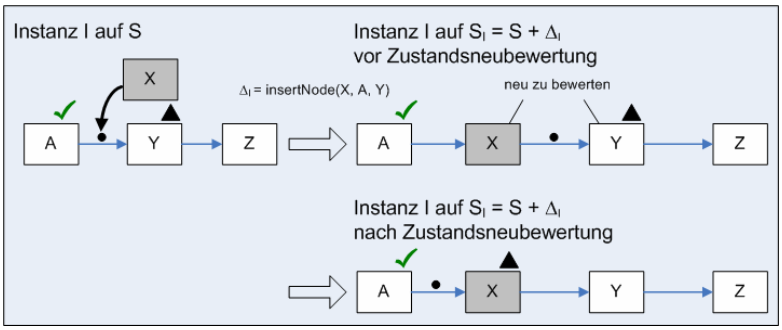


Abbildung 3.8 Zustandsneubewertung bei *insertNode*

Zusätzlich zu dieser Zustandsneubewertung sind in einigen Fällen Interaktionen mit anderen Komponenten des PMS erforderlich. So erfordert das beschriebene Einfügen des Knotens *X*, dass der mit *Y* verknüpfte Arbeitsauftrag aus den Arbeitslisten ausführungsberechtigter Anwender entfernt werden muss und stattdessen der Knoten *X* als Arbeitsschritt in die Listen einzufügen ist.

3.4 Schemaevolution

Bei einer Schemaevolution muss in einem ersten Schritt das Prozesschema in die gewünschte Form gebracht werden. Wie dies durchzuführen ist, wird in Abschnitt 3.4.1 beschrieben. Zu einer Schemaevolution gehört aber nicht nur die reine Schemaänderung, sondern auch die Übertragung dieser Änderungen auf (laufende) Instanzen. Dabei hängen die bei einer solchen Migration notwendigen Mechanismen davon ab, ob es sich bei der zu migrierenden Instanz um eine instanzspezifisch-geänderte oder eine unveränderte Instanz handelt. Im Falle einer geänderten Instanz lassen sich noch weitere Abstufungen vornehmen, nach denen sich eine zu migrierende Instanz kategorisieren lässt. Dies wird in Abschnitt 3.4.2 beschrieben. Aufbauend auf dieser Einteilung werden die Mechanismen zur Migration der einzelnen Instanzen auf konzeptioneller Ebene erläutert (Abschnitte 3.4.3 und 3.4.4).

3.4.1 Schemaänderung

Bei einer Schemaänderung wird im Gegensatz zu einer Instanzänderung nicht eine einzelne Instanz geändert, sondern das Schema auf dem die Instanzen beruhen. Dies wird, wie in Abschnitt 1.2 beschrieben, immer dann notwendig, wenn der Prozessablauf auf Grund geänderter Voraussetzungen angepasst werden muss. Die Änderung des Prozessschemas erfolgt durch Anwendung der in Abschnitt 3.1.2 vorgestellten einfachen und komplexen Änderungsoperationen. Die strukturelle Korrektheit ist

somit durch die strukturellen Vorbedingungen der angewendeten Änderungsoperationen implizit gewährleistet. Es ergibt sich – analog zu einer Instanzänderung – das vom ursprünglichen Schema *S* abweichende Prozessschema *S'* := *S* + Δ_S (vgl. Abbildung 3.9). Laufzeitaspekte sind bei einer Schemaänderung nicht zu berücksichtigen.

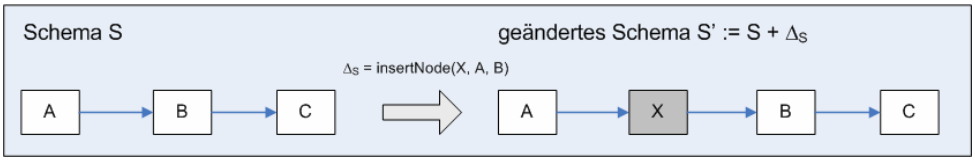


Abbildung 3.9 Schemaänderung *insertNode*

3.4.2 Kategorisierung zu migrierender Instanzen

Bei neu zu startenden Prozessinstanzen ergeben sich durch eine Schemaänderung keine Probleme. Die Prozessinstanzen referenzieren das geänderte Schema und übernehmen somit automatisch dessen Struktur. Komplexer wird die Situation erst, wenn bereits gestartete oder sogar instanzspezifisch-geänderte Instanzen ebenfalls an das geänderte Schema angepasst werden sollen. Hier muss überprüft werden, ob die zu migrierenden Instanzen mit den durchgeführten Schemaänderungen „verträglich" (*compliant*) sind. Dabei bezeichnet man eine Instanz *I* immer dann als verträglich, wenn die Schemaänderungen so auf *I* übertragen werden können, dass die Korrektheit nach Definition 3 erhalten bleibt. Welche Verträglichkeitstests und welche Migrationstrategie bei einer bestimmten Instanz anzuwenden sind, ist von der Art (unverändert, geändert) der zu migrierenden Instanz abhängig. Um hier gezielt Verträglichkeitstests und Migrationsstrategien definieren zu können, müssen die Instanzen kategorisiert werden. In [Rind04] werden hierbei die folgenden Fälle bzw. Unterkategorien unterschieden:

- Die zu migrierende Prozessinstanz wurde nicht geändert (*unbiased*).
- Die zu migrierende Prozessinstanz wurde geändert (*biased*).
  - *Disjoint*: Die Auswirkungen der instanzspezifischen Änderung und die Auswirkungen der Schemaänderung auf das Prozessschema sind disjunkt, d.h. sie betreffen unterschiedliche Bereiche des Schemas.
  - *Overlapping*: Die Auswirkungen überlappen sich.
    - *equivalent*: Die Änderungen haben exakt die gleichen Auswirkungen
    - *subsumption equivalent*: Die Auswirkungen der Instanzänderung sind eine Teilmenge der Auswirkungen der Schemaänderung oder umgekehrt.
    - *partially equivalent*: Die Auswirkungen beider Änderungen sind teilweise identisch, doch haben beide Änderungen zusätzliche Auswirkungen, die die jeweils andere nicht besitzt.

Abbildung 3.10 zeigt jeweils einen Vertreter der Kategorie/Klasse *Disjoint* und *Overlapping*. Vertreter der Unterklassen finden sich in Abschnitt 3.4.4.2 (*Overlapping*).

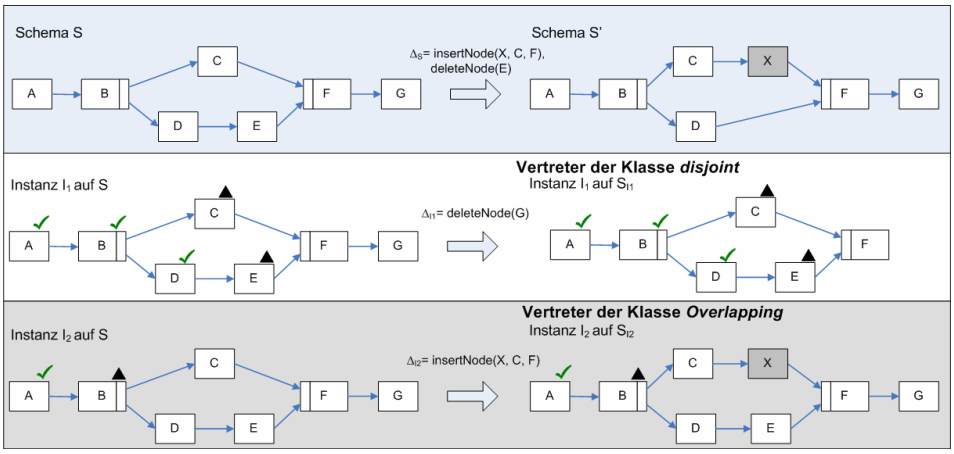


Abbildung 3.10 Disjoint vs. Overlapping

Wie aus Abbildung 3.10 ersichtlich ist, betrifft die Instanzänderung Δ_{I1} einen anderen Bereich wie die Schemaänderung Δ_S. Dies gilt sowohl für neu eingefügte und gelöschte Knoten als auch für betroffene Kanten. Die Auswirkungen der Instanzänderungen von I_1 sind folglich zu den Auswirkungen der Schemaänderungen von S' disjunkt und damit I_1 zur Klasse *Disjoint* zuzuordnen. Bei I_2 hingegen überlappen sich die Auswirkungen von Δ_{I2} und Δ_S. So sind beispielsweise die gegenüber S neu hinzugekommenen Kanten (C, X) und (X, F) und der Knoten X sowohl bei S' als auch bei I_2 gleich. Es handelt sich also um einen Vertreter der Klasse *Overlapping*; im Speziellen um eine Instanz der Klasse *subsumption equivalent*.
Bereits dieses Beispiel zeigt, dass sich bei Instanzen unterschiedlicher Klassen, eine völlig andere Ausgangssituation für entsprechende Verträglichkeitstests und Migrationsstrategien ergibt. Wie diese Tests und Strategien im Einzelnen aussehen, wird im Folgenden beschrieben.

3.4.3 Migration unveränderter Prozessinstanzen (*unbiased*)

Die Migration einer unveränderten Instanz I beginnt mit einer Verträglichkeitsprüfung. Dabei definiert sich die Verträglichkeit einer unveränderten Instanz mit einem geänderten Schema folgendermaßen:

Kriterium 1 (Verträglichkeit instanzspezifisch unveränderter Instanzen)

Eine unveränderte Prozessinstanz ist dann mit einem geänderten Schema verträglich, wenn die Ausführungshistorie der Instanz auch auf dem geänderten Schema erzeugbar ist.

Die Limitierungen durch dieses Kriterium hängen allerdings stark von der verwendeten Ausführungshistorie ab (vgl. Abschnitt 2.2.2). In [Rind04] wird hierfür die für eine Schemaevolution besonders geeignete schleifentolerante und datenflusskonsistente Sicht auf die Historie verwendet. Man erhält diese reduzierte Ausführungshistorie (Π_{Ired}), indem man die ursprüngliche Historie um diejenigen Einträge bereinigt, die sich in einem anderen als dem aktuellen Schleifendurchlauf befinden. Betrachtet man hierzu die Ausführungshistorie von Abbildung 3.3, so erkennt man, dass sich die Schleife im zweiten Durchlauf befindet. Für die Schemaevolution relevant sind allerdings nur die Markierungen des aktuellen (hier des zweiten) Durchlaufs. Folglich erhält man die reduzierte Ausführungshistorie indem man alle Einträge zwischen START(L_s, *1st it*) und END(L_E, *true*) (inklusive) nicht in Π_{Ired} übernimmt.

Mit Π_{Ired} lässt sich nun die Verträglichkeit einer *unbiased* Instanz am einfachsten testen, indem man versucht, die komplette reduzierte Ausführungshistorie von I auf S' „nachzuspielen“. Kann auf diese

Weise auf *S'* die gleiche Ausführungshistorie erzeugt werden, so ist die Instanz verträglich. Für eine praktische Umsetzung ist dieses Nachspielen auf Grund des hohen Aufwands allerdings nicht geeignet (vgl. Abschnitt 2.2).
Ein wesentlich effizienterer Test ergibt sich unter Ausnutzung der in den Änderungsoperationen enthaltenen Semantik. Betrachtet man beispielsweise das Löschen eines Knotens *X* auf Schemaebene, so ist eine Instanz dann mit dem geänderten Schema verträglich, wenn die reduzierte Ausführungshistorie von *I* keinen START-Eintrag für *X* enthält. Die Verträglichkeit einer Instanz lässt sich also mit einem kleinen Teil der in $\Pi_{I_{red}}$ enthaltenen Information testen. Stellt man diese Überlegung für alle Änderungsoperationen an, so zeigt sich, dass sich die notwendigen Informationen immer auf die Zustände einzelner Knoten beschränken. Diese Zustände finden sich bei dem in Abschnitt 3.1 vorgestellten Prozess-Metamodell nicht nur in der Ausführungshistorie, sondern auch in der Instanzmarkierung M^I. Dies bringt den Vorteil, dass auf Grund der geringen Größe von M^I und der damit verbundenen Möglichkeit diese Daten im Primärspeicher zu halten, „teure" Zugriffe auf die Ausführungshistorie entfallen. Für einen Verträglichkeitstest ist es also lediglich notwendig, auf die Zustände bestimmter in der Instanzmarkierung enthaltener Knoten zuzugreifen, um deren Verträglichkeit mit einer bestimmten Änderungsoperation zu testen. Von welchen Knoten dabei die Zustände überprüft werden müssen, hängt von den angewendeten Änderungsoperationen ab. Hier zeigen sich deutliche Parallelen zur zustandsbasierten Verträglichkeitsprüfung bei instanzspezifischen Änderungen. Auch dort werden Zustände geprüft, um die Zulässigkeit einer Änderungsoperation zu bestimmen. Lediglich das Ergebnis einer solchen Prüfung führt zu unterschiedlichen Reaktionen. Ist ein Test nicht erfolgreich, so führt dies auf Instanzebene zu einem Abbruch der entsprechenden Änderungsoperation. Bei einer Schemaevolution hingegen wird die getestete Instanz als nicht verträglich markiert und kann somit nicht auf das neue Schema migriert werden. Das Vorgehen ist aber in weiten Teilen identisch, was ein Verwenden der in Abschnitt 3.3 für Änderungsoperationen beschriebenen zustandsbasierten Vorbedingungen ermöglicht. Lediglich eine kleine Änderung ist zu berücksichtigen. Während bei den zustandsbasierten Vorbedingungen im Rahmen einer Instanzänderung größtenteils nur die Zustände *ACTIVATED* und *NOT_ACTIVATED* erlaubt sind, ist im Rahmen einer Schemaevolution auch der Zustand *SKIPPED* möglich (vgl. Abbildung 3.11). Dies liegt darin begründet, dass auf Instanzebene Graphmanipulationen in Zweigen, die nicht mehr zur Ausführung kommen, unsinnig sind. Im Zuge einer Schemaevolution steht aber weniger die Ausführbarkeit einzelner Aktivitäten im Mittelpunkt, sondern vielmehr das Ziel, möglichst viele Instanzen auf das geänderte Schema zu migrieren.

Neben der Überprüfung der zustandsbasierten Verträglichkeit, sind bei *unbiased* Instanzen keine weiteren Tests notwendig. Die vom Zustand verträglichen Instanzen werden einfach auf das geänderte Schema „umgehängt", wodurch die strukturelle Korrektheit der Instanzen implizit gewährleistet ist.

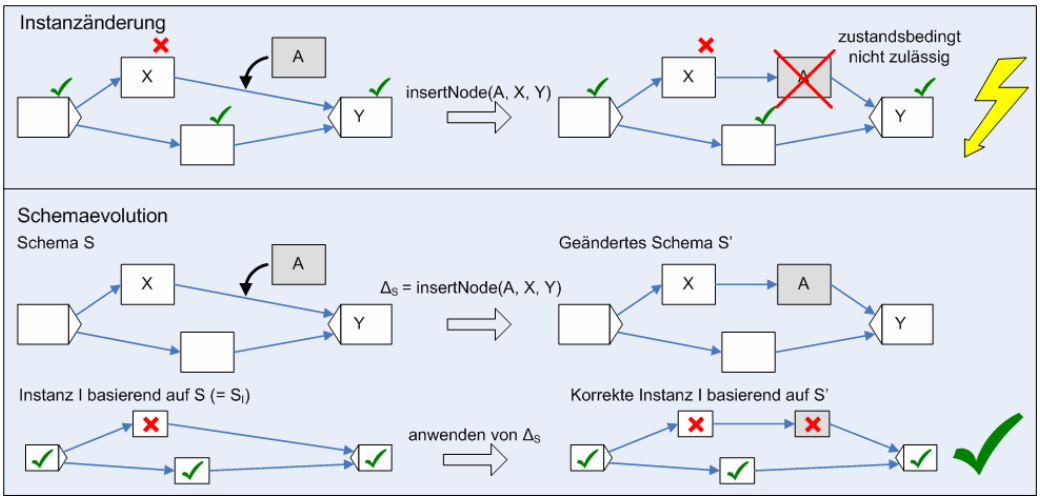


Abbildung 3.11 Einfügen in einen SKIPPED Zweig

Nach Umhängen der Instanzen ist es – analog zu instanzspezifischen Änderungen – notwendig, die Zustände betroffener Graphelemente anzupassen. Im Gegensatz zu Instanzänderungen können die Elemente allerdings nicht sofort nach der Anwendung jeder einzelnen Änderungsoperation neu bewertet werden. Dies liegt daran, dass die dazu notwendige Instanzmarkierung erst bei der Migration der Instanzen zur Verfügung steht, d.h. zu einem Zeitpunkt, an dem das Schema bereits durch Anwendung meist mehrerer Änderungsoperationen in die gewünschte Form gebracht wurde. Ein Bewerten der Zustände ist folglich erst dann möglich, wenn bereits alle Änderungen abgeschlossen sind. Dies stellt sich allerdings nicht als Nachteil heraus, da sich durch geschicktes Optimieren ein unnötiges Anpassen von Kanten- und Knotenzuständen vermeiden lässt (vgl. Abbildung 3.12).

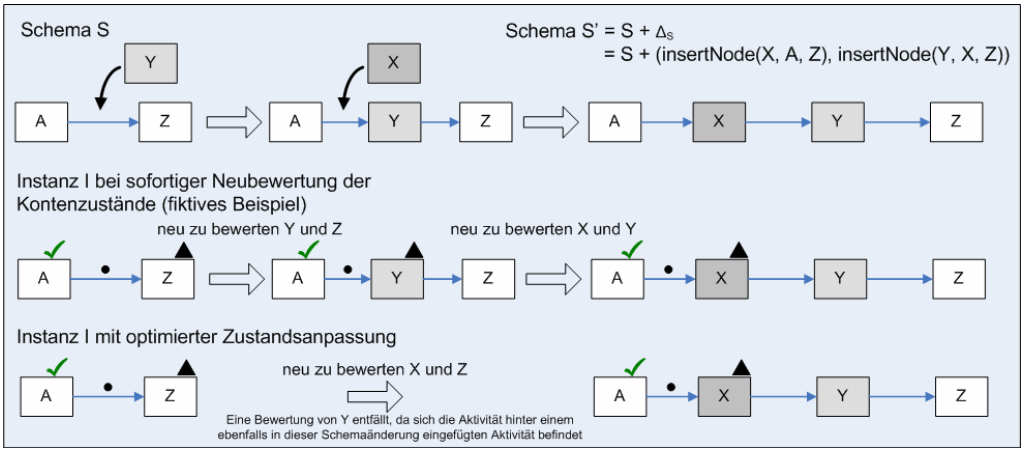


Abbildung 3.12 Vermeidung unnötiger Neubewertung durch optimierte Zustandsanpassung

Wie Abbildung 3.12 zeigt, wären bei einer separaten Betrachtung der angewendeten Änderungsoperationen die Zustände der Aktivitäten *X*, *Y* und *Z* neu zu bewerten. Betrachtet man allerdings die Änderungen insgesamt, so genügt es, lediglich die Zustände von *X* und *Z* anzupassen, um eine nach Definition 3 korrekte Instanz zu erhalten. Dies gilt analog für die Neubewertung von Kanten.

3.4.4 Migration geänderter Prozessinstanzen (*biased*)

Bei der Migration geänderter Instanzen stellen besonders die Wechselwirkungen zwischen der Schemaänderung Δ_S und der Instanzänderung Δ_I eine große Herausforderung dar. Dies kommt daher, dass die jeweiligen Änderungen – jeweils separat betrachtet – durch die Vorbedingungen der angewendeten Änderungsoperationen korrekt sind, im Zusammenspiel allerdings zu inkorrekten Graphen führen können. Weiterhin gestaltet es sich in einigen Fällen schwierig, nach einer erfolgreichen Propagierung der Schemaänderungen auf eine geänderte Instanz *I*, die resultierenden Abweichungen (*Bias*) zwischen *I* und dem geänderten Schema *S'* neu zu bestimmen.
Zur weiteren Differenzierung der Problematik werden in Abschnitt 3.4.4.1 die Instanzen mit disjunkten (*disjoint*) und in Abschnitt 3.4.4.2 die Instanzen mit überlappenden Änderungen (*overlapping*) betrachtet.

3.4.4.1 *Disjoint*

Zur Klasse disjunkt (*disjoint*) gehören diejenigen Instanzen, bei denen die Auswirkungen der instanzspezifischen Änderung Δ_I nicht mit den Auswirkungen der Schemaänderung Δ_S überlappen. Formal: $\Delta_I \cap \Delta_S = \varnothing$. Ob eine solche Instanz auf ein geändertes Schema migriert werden kann, definiert sich folgendermaßen:

Kriterium 2 (Verträglichkeit geänderter Prozessinstanzen mit disjunkten Änderungen)
Instanz *I* ist verträglich mit dem geänderten Schema *S'*, wenn folgende Kriterien erfüllt sind:

1. *Strukturelle Korrektheit*: Die Propagierung der Schemaänderungen Δ_S auf die geänderte Instanz *I* resultiert in einem nach Definition 1 korrekten Schema, d.h. Δ_S kann korrekt auf $(S + \Delta_I)$ angewendet werden.
2. *Zustandsbasierte Korrektheit*: Die reduzierte Ausführungshistorie Π_{Ired} von Instanz *I* kann auch auf $(S + \Delta_S) + \Delta_I$ erzeugt werden.

Vergleicht man Kriterium 2 mit den Verträglichkeitskriterien bei *unbiased* Instanzen, so fällt auf, dass in diesem Fall auch die strukturelle Korrektheit explizit gefordert wird. Dies hängt damit zusammen, dass eine vom Zustand verträgliche, geänderte Instanz nicht ohne weitere Tests auf das geänderte Schema umgehängt werden kann. Würde man dies trotzdem tun, so könnte durch Wechselwirkungen zwischen Δ_S und Δ_I ein inkorrekter Graph entstehen.
In [Rind04] werden die in diesem Zusammenhang auftretenden Konfliktsituationen detailliert beschrieben, weshalb dies hier nur graphisch verdeutlicht wird (siehe Abbildung 3.13).
Zur Aufdeckung solcher Konflikte ist es am einfachsten, den resultierenden Graph $(S + \Delta_I) + \Delta_S$ zu materialisieren und entsprechende Korrektheitstests durchzuführen. Da dies für jede auf dem Schema laufende *biased* Instanz getan werden müsste, ist dieser Weg zu ineffizient. Bei den in [Rind04] entwickelten Konflikttests wird deshalb wiederum die Semantik der angewendeten Änderungsoperationen aus Δ_S und Δ_I herangezogen. So lässt sich wesentlich effizienter feststellen, ob ein potentieller Konflikt vorliegt oder nicht. Wir werden hierauf in Abschnitt 7.3 (Strukturelle Konfliktbestimmung) in stark abgewandelter Form zurückkommen.

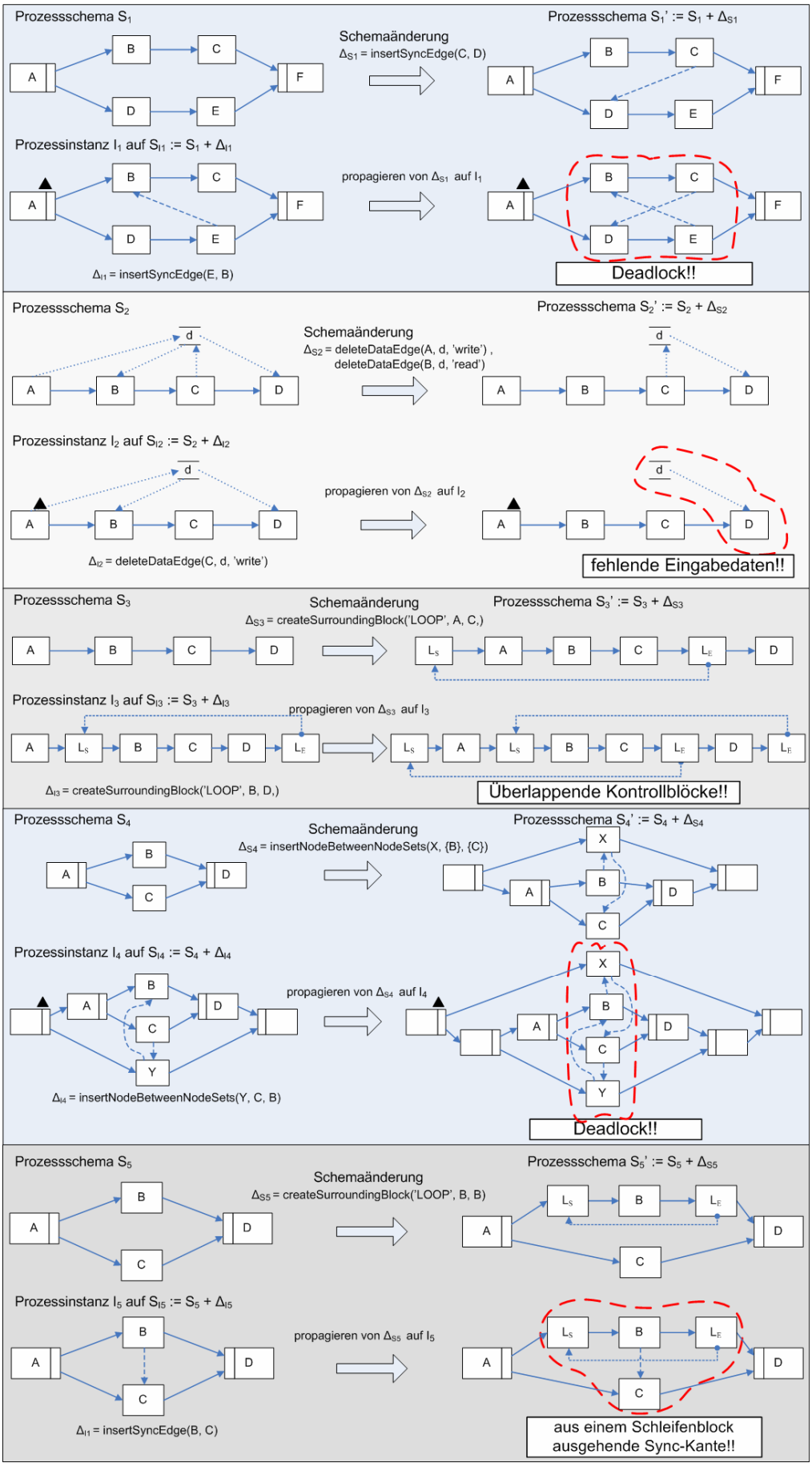


Abbildung 3.13 Konflikte durch konkurrierende Anwendung von Änderungsoperationen

Neben der strukturellen Korrektheit muss auch die zustandsbasierte Korrektheit gewährleistet werden. Eine separate Definition von zustandsbasierten Korrektheitskriterien für *biased disjoint* Instanzen ist allerdings nicht erforderlich, da exakt die gleichen Mechanismen wie bei *unbiased* Instanzen verwendet werden können. Dies liegt daran, dass die Anwendung der zustandsbasierten Verträglichkeitskriterien unabhängig vom jeweiligen instanzspezifischen Schema ist, d.h. eine Unterscheidung zwischen instanzspezifisch-geänderten und unveränderten Instanzen erübrigt sich.

Ein Migrieren verträglicher Instanzen der Klasse *disjoint* ist relativ einfach. Es genügt die verträglichen Instanzen auf das geänderte Schema umzuhängen. Die instanzspezifischen Änderungen Δ_I können dabei unverändert übernommen werden, da die Änderungen von Δ_S und Δ_I disjunkt sind und daher Ausführungsgleichheit zwischen $(S + \Delta_I) + \Delta_S$ und $(S + \Delta_S) + \Delta_I$ gewährleistet ist (Kommutativität). Als Ergebnis erhält man $S' + \Delta_I$. Auch hier müssen noch von den Schemaänderungen betroffene Graphelemente bzgl. ihres Zustandes neu bewertet werden. Hierzu kann der gleiche Algorithmus verwendet werden, der auch die Zustände bei Instanzen der Klasse *unbiased* neu bewertet.

3.4.4.2 *Overlapping*

Zur Klasse *overlapping* gehören alle Instanzen, bei denen die Auswirkungen der instanzspezifischen Änderung Δ_I vollständig oder teilweise mit den Auswirkungen der Schemaänderung Δ_S überlappen. Formal: $\Delta_I \cap \Delta_S \neq \emptyset$. Die Instanzen dieser Klasse lassen sich wie bereits beschrieben nach ihrem Grad der Überlappung in die Klassen *equivalent*, *subsumption equivalent* und *partially equivalent* weiter unterteilen. Abbildung 3.14 zeigt jeweils ein einfaches Beispiel.

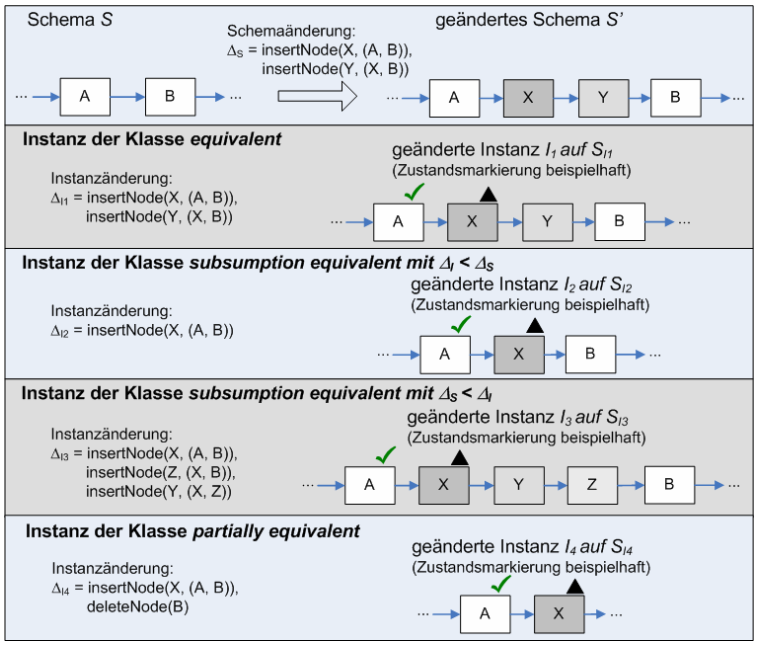


Abbildung 3.14 Instanzen der Klasse *overlapping*

Wie für die einzelnen Klassen die Verträglichkeit definiert wird und wie diese korrekt migriert werden, wird in den Abschnitten 3.4.4.2.1 - 3.4.4.2.3 erläutert.

3.4.4.2.1 Equivalent

Zur Klasse äquivalent (*equivalent*) gehören die Instanzen, bei denen die angewendeten instanzspezifischen Änderungen Δ_I exakt die gleichen Auswirkungen auf das Schema *S* besitzen, wie die im Zuge der Schemaänderung auf *S* angewendeten Änderungsoperationen Δ_S. In Abbildung 3.14 ist dies für die Instanz I_1 der Fall. Hier ist der resultierende Effekt der auf I_1 angewendeten Änderungsoperationen *insertNode*(*X*, (*A*, *B*)) und *insertNode*(*Y*, (*X*, *B*)) identisch mit den Auswirkungen der auf *S* angewendeten Änderungsoperationen. Weiterhin wurden weder bei der Änderung des Schemas noch bei der Instanzänderung weitere Änderungsoperationen angewendet. Folglich muss das geänderte Schema *S'* und das geänderte instanzspezifische Schema S_{I1} gleich sein.
Dadurch ist es verhältnismäßig einfach, für Instanzen der Klasse *equivalent* ein Verträglichkeitskriterium und eine entsprechende Migrationsstrategie zu definieren.

Kriterium 3 (Verträglichkeit instanzspezifisch-geänderter Prozessinstanzen mit äquivalenten Änderungen)

Bei der Migration einer Instanz *I*, bei der die Auswirkungen einer Instanzänderung Δ_I mit den Auswirkungen der Schemaänderung Δ_S äquivalent sind, ist die strukturelle und zustandsbasierte Korrektheit immer gewährleistet.

Die Korrektheit erklärt sich folgendermaßen: Da durch die Anwendung der Schemaänderungen Δ_S auf *S* ein mit dem geänderten instanzspezifischen Schema S_I deckungsgleicher Graph resultiert, ist es nicht notwendig, dass Δ_S auf *I* propagiert wird. Eine nicht notwendige Propagation der Schemaänderungen bedeutet gleichzeitig einen Wegfall von strukturellen und zustandsbasierten Verträglichkeitstests. Eine Instanz *I* der Klasse *equivalent* kann folglich auf das geänderte Schema *S'* migriert werden, wenn *I* selbst ein nach Definition 3 korrektes instanzspezifisches Schema ist. Da dies bereits durch die strukturellen und zustandsbasierten Verträglichkeitskriterien der bei der Instanzänderung angewandten Änderungsoperationen gewährleistet ist, erübrigen sich weitere Verträglichkeitstests.
Aus der unnötigen Propagation resultiert ein weiterer positiver Effekt: Es können sogar Instanzen migriert werden, bei denen die zustandsbasierten Verträglichkeitskriterien der auf *I* propagierten Schemaänderungen – separat betrachtet – nicht mehr erfüllbar wären. Instanz I_1 aus Abbildung 3.14 kann beispielsweise selbst dann noch auf das Schema *S'* migriert werden, wenn sich *B* bereits im Zustand *COMPLETED* befindet, was eigentlich nach den zustandsbasierten Verträglichkeitskriterien von *insertNode* nicht zulässig ist (vgl. Tabelle 0.3 Anhang A (Änderungsoperationen)).

Zur Migration solcher Instanzen reicht es aus, eine Instanz *I* auf das geänderte Schema *S'* umzuhängen und Δ_I zu löschen. Zustandsanpassungen sind nicht notwendig.

3.4.4.2.2 Subsumption equivalent

Die Klasse *subsumption equivalent* präsentiert sich zweigeteilt (vgl. Instanzen I_2 und I_3 in Abbildung 3.14): Der eine Teil umfasst Instanzen, bei denen die Schemaänderungen aus Δ_S die gleichen Auswirkungen besitzen wie ein Teil der Instanzänderungen aus Δ_I, Δ_I aber noch zusätzliche Änderungen besitzt, die nicht in Δ_S enthalten sind. Es handelt sich somit bei den Auswirkungen von Δ_S quasi um eine Teilmenge der Auswirkungen von Δ_I. Dies wird mit $\Delta_S < \Delta_I$ bezeichnet. Bei dem anderen Teil der *subsumption equivalent* Instanzen verhält sich die Teilmengenbeziehung genau umkehrt. Hier haben die Schemaänderungen aus Δ_S gegenüber den Instanzänderungen aus Δ_I noch zusätzliche Auswirkungen auf *S*. Dies wird analog mit $\Delta_I < \Delta_S$ bezeichnet.

Die beiden Arten verhalten sich bzgl. Verträglichkeitskriterien und anzuwendender Migrationsstrategie gänzlich unterschiedlich, weshalb diese im Folgenden getrennt betrachtet werden.

Instanzen mit $\Delta_S < \Delta_I$

Die Verträglichkeit für eine Instanz dieser Klasse ist mit der Verträglichkeitsdefinition für Instanzen der Klasse *equivalent* identisch. Dies ist leicht nachvollziehbar, da sich auch hier eine Propagation der Schemaänderungen Δ_S auf das geänderte instanzspezifische Schema S_I erübrigt. Der Unterschied zwischen Instanzen dieser Klasse und denen der Klasse *equivalent* besteht darin, dass hier noch zusätzliche Auswirkungen durch Δ_I entstehen. Da diese Auswirkungen aber durch Änderungsoperationen verursacht werden, die im Zuge der instanzspezifischen Änderung zum Einsatz kamen, spielen diese bei der Verträglichkeitsprüfung keine Rolle. Es ergibt sich die folgende Definition:

Kriterium 4 (Verträglichkeit instanzspezifisch-geänderter Prozessinstanzen der Klasse *subsumption equivalent* ($\Delta_S < \Delta_I$))

Bei der Migration einer Instanz I, bei der die Auswirkungen der Schemaänderung Δ_S eine Teilmenge der Auswirkungen der instanzspezifischen Änderungen Δ_I sind, ist die strukturelle und zustandsbasierte Korrektheit immer gewährleistet.

Während die zusätzlichen Auswirkungen von Δ_I bei der Definition der Verträglichkeit noch nicht von Bedeutung waren, müssen sie bei einer Migrationsstrategie sehr wohl beachtet werden. Dies wird klar, wenn man berücksichtigt, dass Δ_I die angewendeten Änderungsoperationen und somit die Abweichungen zum referenzierten Schema beinhaltet. Bei den hier behandelten Instanzen gilt bekanntermaßen $\Delta_S < \Delta_I$. Dies bedeutet, dass eine solche Instanz selbst nach der Migration auf das geänderte Schema S' noch von diesem abweicht. Die Abweichungen lassen sich folgendermaßen formalisieren: $\Delta_I(S') := \Delta_I \setminus \Delta_S$. Damit ergibt sich für die Migration einer Instanz I vom Typ *subsumption equivalent* mit $\Delta_S < \Delta_I$ der folgende Ablauf: Die Instanz wird ohne Propagation der Schemaänderungen Δ_S auf das geänderte Schema S' umgehängt und das resultierende *Bias* $\Delta_I(S')$ berechnet.

Dass sich die Berechnung von $\Delta_I(S')$ als keinesfalls trivial herausstellt, zeigt ein entsprechender Algorithmus in [Rind04], S. 260f. Die nachfolgende Abbildung zeigt beispielhaft die Migration der Instanz I_3 aus Abbildung 3.14 auf das geänderte Schema S' und das daraus resultierende *Bias* $\Delta_I(S')$.

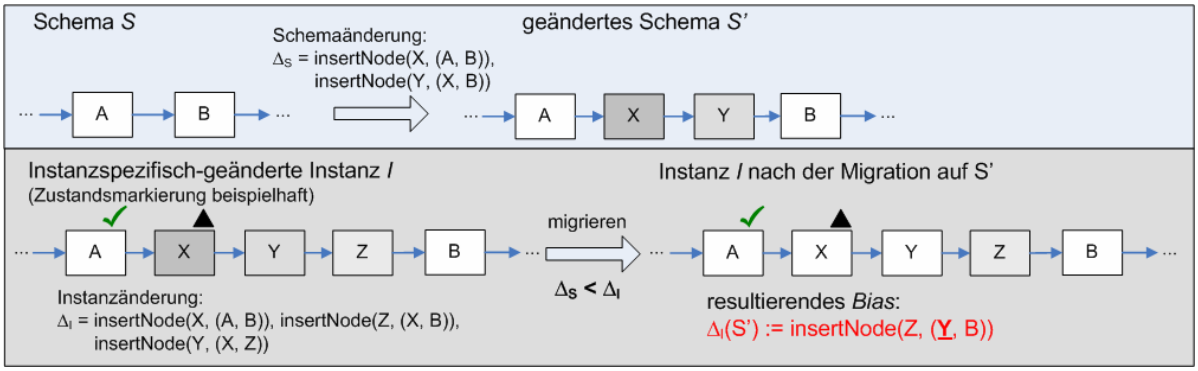


Abbildung 3.15 Migration einer Instanz der Klasse *subsumption equivalent* mit $\Delta_S < \Delta_I$

Es lässt sich hier bereits erkennen, dass es für die Berechnung von $\Delta_I(S')$ im Normalfall nicht ausreicht, einfach die Differenz zwischen den angewendeten Änderungsoperationen zu bilden. Dies ist bei der Umsetzung dieses Konzepts im Änderungsrahmenwerk entsprechend zu berücksichtigen.

Instanzen mit $\Delta_I < \Delta_S$

Für Instanzen mit $\Delta_I < \Delta_S$ ist besonders der Teil der Schemaänderungen von Bedeutung, der lediglich auf Schemaebene nicht aber auf Instanzebene angewendet wurde. Formal: $\Delta_S \setminus \Delta_I$. Mit Hilfe dieser Änderungen lässt sich die Verträglichkeit der hier betrachteten Instanzen bzgl. eines geänderten Schemas folgendermaßen beschreiben:

Kriterium 5 (Verträglichkeit instanzspezifisch-geänderter Prozessinstanzen der Klasse *subsumption equivalent* ($\Delta_I < \Delta_S$))

Eine Instanz I, bei der die Auswirkungen der instanzspezifischen Änderung Δ_I eine Teilmenge der Auswirkungen der Schemaänderung Δ_S sind, ist mit dem geänderten Schema S' verträglich, wenn $\Delta_S \setminus \Delta_I$ ohne Verletzung der zustandsbasierten Korrektheit auf I propagiert werden kann.

Wie Kriterium 5 zeigt, ist die zustandsbasierte Verträglichkeit nur für die Menge $\Delta_S \setminus \Delta_I$ zu prüfen. Dieses Ausschließen derjenigen Änderungen die sowohl auf Schema als auch auf Instanzebene durchgeführt wurden ($\Delta_{S \cap I}$) ist zwingend erforderlich. Nur so lässt sich verhindern, dass eigentlich verträgliche Instanzen als nicht verträglich erkannt werden. Dies begründet sich damit, dass sich im hier betrachteten Fall die Änderungen aus $\Delta_{S \cap I}$ analog zu den äquivalenten Änderungen bei Instanzen der Klasse *equivalent* verhalten, d.h. die Schemaänderungen die ebenfalls auf Instanzebene durchgeführt wurden, müssen bzw. dürfen nicht auf die Instanz propagiert werden. Die zustandsbasierte Korrektheit ist somit für diesen Teil der Änderungen automatisch gewährleistet.

Ein Überprüfen der strukturellen Korrektheit ist nicht erforderlich. Dies ist leicht nachvollziehbar, da die auf Instanzebene durchgeführten Änderungen alle auch auf Schemaebene durchgeführt wurden. Ein bei der Propagation auftretender struktureller Fehler müsste somit bereits in S' vorhanden sein. Dies wird aber durch die strukturellen Vorbedingungen (vgl. Tabelle 3.2 Anhang A (Änderungsoperationen)) der in Δ_S angewendeten Änderungsoperationen verhindert und ist somit von vornherein ausgeschlossen.

Bei der Migration einer verträglichen Instanz dieser Klasse, wird diese in einem ersten Schritt auf das geänderte Schema S' umgehängt. Da S' alle Änderungen von Δ_I enthält, weicht das resultierende instanzspezifische Schema nicht von diesem ab. Das *Bias* von I ($= \Delta_I$) kann gelöscht werden. In einem zweiten Schritt müssen die Zustände der von $\Delta_S \setminus \Delta_I$ betroffenen Knoten neu bewertet werden. Es handelt sich dabei um die Knoten, die von denjenigen Änderungen betroffen sind, die lediglich auf Schemaebene durchgeführt wurden. Dadurch ergibt sich eine analoge Situation wie bei Instanzen der Klasse *unbiased*. Zur Zustandsneubewertung können somit exakt die gleichen Mechanismen angewendet werden.

3.4.4.2.3 Partially equivalent

Zur Klasse *partially equivalent* gehören alle Instanzen, bei denen einerseits ein Teil der Instanzänderungen Δ_I mit den Schemaänderungen Δ_S übereinstimmen, andererseits aber sowohl Δ_I als auch Δ_S noch weitere Änderungen besitzen, die im jeweils anderen nicht enthalten sind (vgl.

Abbildung 3.14 Instanz I_4). Die Verträglichkeit einer Instanz mit dem geänderten Schema definiert sich für diese Klasse folgendermaßen:

Kriterium 6 (Verträglichkeit instanzspezifisch-geänderter Prozessinstanzen der Klasse *partially equivalent*)

Instanz I ist verträglich mit dem geänderten Schema S' wenn folgende Kriterien erfüllt sind:

1. *Strukturelle Korrektheit*: Die Propagierung des Teils der Schemaänderungen, der nicht mit Δ_I übereinstimmt, führt mit dem Teil der Instanzänderungen, der nicht in Δ_S enthalten ist, zu keinem strukturellen Konflikt.
2. *Zustandsbasierte Korrektheit*: Durch Propagieren der Änderungen $\Delta_S \setminus \Delta_I$ kommt es zu keiner Verletzung der zustandsbasierten Korrektheit von I.

Im ersten Kriterium müssen lediglich die Änderungen betrachtet werden, die im jeweils anderen Δ nicht enthalten sind. Formal: $(\Delta_S \cup \Delta_I) \setminus (\Delta_S \cap \Delta_I)$. Dies liegt daran, dass der Teil der Änderungen, der sowohl bei Δ_I als auch bei Δ_S vorhanden ist, keine Konflikte verursachen kann (vgl. *equivalent*). Für die anderen Änderungen ergibt sich eine mit den Instanzen der Klasse *disjoint* analoge Situation. Folglich muss für diesen Teil geprüft werden, ob es im resultierenden instanzspezifischen Schema zu einem strukturellen Konflikt kommt.
Das zweite Kriterium schränkt die zu prüfende Menge auf diejenigen Änderungen ein, die nur auf Schemaebene zum Einsatz gekommen sind. Dies hängt damit zusammen, dass es durch die Änderungen, die auch auf Instanzebene durchgeführt wurden, nicht zu einem zustandsbasierten Konflikt kommen kann. Die hier betrachteten Instanzen verhalten sich diesbezüglich analog zu den Instanzen der Klasse *subsumption equivalent* ($\Delta_I < \Delta_S$). Die dort angegebene Begründung ist somit auch hier anwendbar.

Bei der Migration verträglicher Instanzen muss beachtet werden, dass die instanzspezifischen Änderungen sehr stark variieren können. Diese reichen vom Einfügen zweier unterschiedlicher Knoten (bzw. Aktivitäten) in denselben Zielkontext auf Instanz- und auf Schemaebene, bis hin zu Schema- und Instanzänderungen, die annähernd die gleichen Auswirkungen auf das ursprüngliche Schema S besitzen [Rind04]. Weiterhin ist es in einigen Fällen überhaupt nicht möglich, eine Instanz ohne Eingriff des Benutzers auf ein geändertes Schema zu migrieren. Als Konsequenz daraus, kann für Instanzen der Klasse *partially equivalent* keine einheitliche Migrationsstrategie definiert werden.
Es ist allerdings möglich, die angewandten Änderungen entlang ihres Typs weiter zu unterteilen. Dies wird als Änderungsprojektion (*change projection*) bezeichnet. Die Idee für diese Unterteilung beruht auf der Beobachtung, dass bei vielen der hier betrachteten Instanzen die Instanzänderungen Δ_I nur insgesamt gesehen mit den Schemaänderungen Δ_S teilweise äquivalent (*partially equivalent*) sind. Gruppiert man jedoch die Änderungen von Δ_I und Δ_S jeweils nach ihrem Typ und vergleicht die resultierenden Gruppen miteinander, so lassen sich diese oft so behandeln, wie die Änderungen der Klassen *disjoint*, *equivalent* oder *subsumption equivalent*. Da für diese Klassen automatische Migrationsstrategien vorhanden sind, ist es in einem solchen Fall möglich, auch Instanzen der Klasse *partially equivalent* automatisch zu migrieren. Anhand der folgenden Abbildung wird dies näher erläutert:

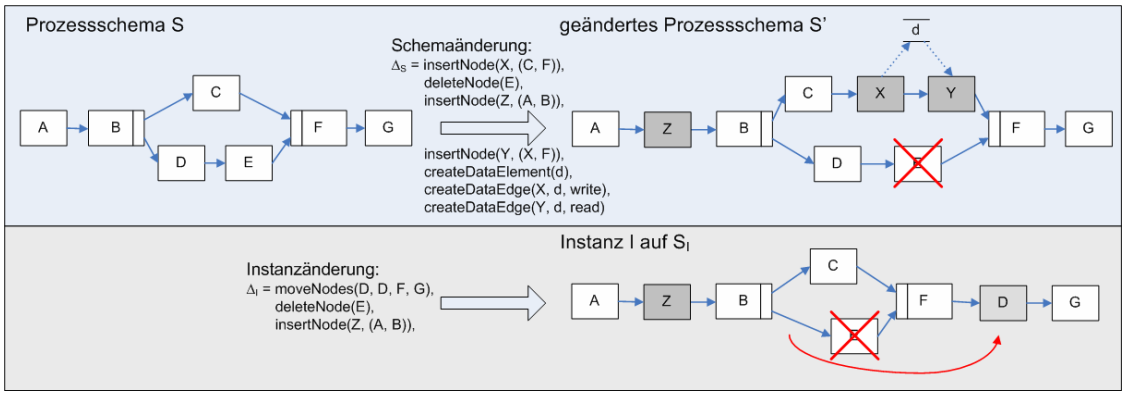


Abbildung 3.16 Vergleich zweier Änderungen Δ_I und Δ_S

Vergleicht man die Änderungen, so zeigt sich, dass eine Instanz mit den abgebildeten instanzspezifischen Änderungen Δ_I zur Klasse *partially equivalent* gehört. Ohne weitere Überlegungen wäre es nun nicht möglich diese Instanz auf das geänderte Schema zu migrieren. Vergleicht man die Änderungen jedoch gruppiert nach ihrem Typ, so ergibt sich das folgende Bild:

Schemaänderung Δ_S:	Instanzänderung Δ_I:
Einfügeoperationen:	Einfügeoperationen:
insertNode(X, (C, F)),	*insertNode(Z, (A, B)),*
insertNode(Z, (A, B)),	
insertNode(Y, (X, F)),	
Löschoperationen:	Löschoperationen:
deleteNode(E),	*deleteNode(E)*
Verschiebeoperationen:	Verschiebeoperationen:
-	*moveNodes(D, D, F, G)*
Datenflussänderungen:	Datenflussänderungen:
createDataElement(d),	-
createDataEdge(X, d, write),	
createDataEdge(Y, d, read)	

Abbildung 3.17 Gruppieren der Änderungsoperationen

Die auf Instanzebene angewendeten Einfügeoperationen sind eine Teilmenge der Einfügeoperationen von Δ_S. Die Einfügeoperationen von Δ_I sind somit *subsumption equivalent* zu den Einfügeoperationen von Δ_S. Die Löschoperationen sind sogar bei beiden Änderungen identisch. Folglich ist Δ_I und Δ_S diesbezüglich *equivalent*. Weiterhin ist Δ_I bzgl. Datenflussänderungen eine Teilmenge von Δ_S (*subsumption equivalent* $\Delta_I < \Delta_S$) und Δ_S eine Teilmenge[6] von Δ_I bzgl. Verschiebeoperationen (*subsumption equivalent* $\Delta_S < \Delta_I$). Durch diese Einteilung kann die Instanz automatisch auf das geänderte Schema migriert werden, da alle Änderungen mit Hilfe der Mechanismen aus *equivalent* und *subsumption equivalent* behandelt werden können.

Trotz der Projektionen ist es nicht möglich, alle Instanzen der Klasse *partially equivalent* automatisch zu migrieren. Dies ist z.B. dann der Fall, wenn beim Vergleich der Änderungen von Δ_S und Δ_I einer der folgenden Konflikte erkannt wird:

- Konkurrierender Zielkontext (*conflicting target context*): Eine Änderung aus $\Delta_S \setminus \Delta_I$ vom Typ Einfüge- oder Verschiebeoperation (*insert* bzw. *move*) verwendet denselben Zielkontext wie eine *insert-* oder *move-* Operation aus $\Delta_I \setminus \Delta_S$. Diese Situation erfordert den Eingriff des Benutzers. Nur dieser kann entscheiden, welche Reihenfolge die eingefügten bzw.

[6] Es wird immer die Teilmengenbeziehung verwendet, obwohl die Teilmenge wie im hier gezeigten Fall auch „leer“ sein kann. Dies hängt damit zusammen, dass keine Migrationsstrategie für Änderungen existiert, die ausschließlich auf Instanzebene und nicht auf Schemaebene durchgeführt wurden.

verschobenen Knoten im resultierenden instanzspezifischen Schema einnehmen sollen. Das folgende Beispiel zeigt eine solche Situation.

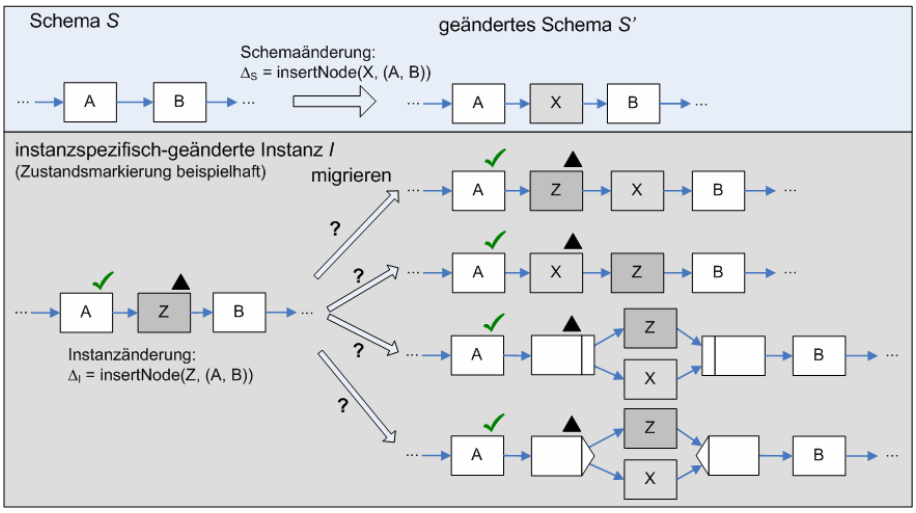


Abbildung 3.18 Konkurrierender Zielkontext

In beiden Fällen wird im Zuge der jeweiligen Änderungen ein neuer Knoten in den gleichen Kontext eingefügt. Während es sich bei der Schemaänderung um den Knoten *X* handelt, wird auf Instanzebene der Knoten *Z* eingefügt. Wie man sieht, besitzen beide Knoten als Zielkontext die Knoten *A* und *B*. Dies hat zur Folge, dass ohne Benutzereingriff nicht entschieden werden kann, welche Form das aus der Migration resultierende instanzspezifische Schema in Abbildung 3.18 besitzen soll.

- Zerstörung des Zielkontextes (*context destroying change*): Dieser Konflikt tritt immer dann auf, wenn entweder eine Änderungsoperation aus Δ_S den Zielkontext einer Änderungsoperation aus Δ_I zerstört oder dies umgekehrt der Fall ist. Es können die folgenden 5 Situationen unterschieden werden:
  1. Kontext wird verschoben (*moving context away*): Δ_I verschiebt einen Knoten, der für eine Einfüge- oder Verschiebeoperation aus Δ_S als Teil des Zielkontextes fungiert. Gleiches gilt, wenn Δ_I einen Quell- oder Zielknoten verschiebt, der in Δ_S von einer *createSyncEdge* oder einer *createDataEdge* Operation benötigt wird. Beides gilt analog mit vertauschtem Δ_I und Δ_S.
  2. Kontext wird gelöscht (*deleting context*): Die Situation ist grundsätzlich mit derjenigen aus Punkt 1 vergleichbar. Der Unterschied besteht jedoch darin, dass hier der Zielkontext nicht verschoben sondern gelöscht wird.
  3. Aufheben einer Knotenattributänderung (*overriding activity attribute change*): Δ_S bzw. Δ_I löscht einen Knoten, bei dem eine Operation aus Δ_I bzw. Δ_S ein Attribut ändert.
  4. Aufheben einer Kantenattributänderung (*overriding edge attribute change*): Δ_S bzw. Δ_I löscht einen Knoten, der bei einer Kantenattributänderung in Δ_I bzw. Δ_S als Quell- oder Zielknoten fungiert.
  5. Datenkontext wird gelöscht (*deleting data context*): Δ_S bzw. Δ_I löscht ein Datenelement, das bei einer *createDataEdge* Operation aus Δ_I bzw. Δ_S verwendet wird.

Es wird an dieser Stelle nicht näher erläutert, wie Instanzen mit solchen Konflikten zu behandeln sind. Hier genügt die Erkenntnis, dass sich zwar nicht alle Instanzen der Klasse *partially equivalent* ohne

Eingriff des Benutzers migrieren lassen, die Konflikte, die dies verhindern, aber zuverlässig erkannt werden können. Daraus resultiert der Vorteil, dass der Benutzer bei der manuellen Migration besser unterstützt werden kann. Welche Mechanismen dazu vom Änderungsrahmenwerk bereitgestellt werden, wird in Abschnitt 7.7 (Migration von *partially equivalent* Instanzen) detailliert erläutert.

3.5 Repräsentation von Schema- und Instanzänderungen

In den Abschnitten 3.3 und 3.4 wurde aus einer eher funktionalen Sicht beschrieben, wie Schemata und Instanzen auf konzeptioneller Ebene geändert werden. Von der Art und Weise, wie diese Änderungen intern repräsentiert werden, wurde abstrahiert. Für die Umsetzung in einem Änderungsrahmenwerk ist die Repräsentation von Schema- und Instanzänderungen aber von entscheidender Bedeutung. Die Art der Darstellung beeinflusst maßgeblich die Möglichkeiten, wie die Änderungskonzepte umgesetzt bzw. implementiert werden können. Dies betrifft sowohl Effizienz- als auch Speicherbedarfsaspekte. In Abschnitt 3.5.1 werden deshalb Konzepte zur Repräsentation von Instanzänderungen vorgestellt und das geeignetste Konzept beschrieben. Schemaänderungen werden in Abschnitt 3.5.2 behandelt.

3.5.1 Instanzen

In den existierenden Prozess-Management-Systemen werden die Instanzen eines Prozessschemas in den meisten Fällen auf eine der folgenden beiden Arten verwaltet bzw. erzeugt:

1. Für jede Instanz wird eine komplette Kopie des Prozessschemas erzeugt (Schemakopie) oder
2. Jede Instanz referenziert das Schema auf dem es erzeugt wurde (Schemareferenz).

Die erste Variante findet sich beispielsweise bei *MQSeries Workflow* [IBM04], während die zweite in *Staffware* [Staff04] zum Einsatz kommt. Beide Arten haben bezüglich Änderungen Vor- und Nachteile [Laue04]. So ermöglicht Variante 1 zwar ein einfaches Ändern von Instanzen, verhindert aber, dass Schemaänderungen auf laufende Instanzen propagiert werden können, da aufgrund der individuellen Schemakopie der Bezug zum ursprünglichen Schema verloren geht. Variante 2 verhält sich genau umgekehrt. Jede Schemaänderung ändert implizit auch alle Instanzen die dieses Schema referenzieren. Eine individuelle Instanzänderung ist dadurch aber nicht möglich.
Beide Varianten sind für die gleichzeitige Unterstützung von Instanz- und Schemaänderungen sowie für die Migration instanzspezifisch-geänderter Instanzen gänzlich ungeeignet. Eine wesentlich bessere Lösung ist die Speicherung von Instanzen bzw. Instanzänderungen mit Hilfe einer Deltaschicht. Bei diesem, in [LRRe04] veröffentlichten und bereits in [JMSW04] umgesetzten Konzept, referenziert eine Instanz bei ihrer Erzeugung – analog zu Variante 2 – das Schema, auf dem sie beruht. Wird eine solche Instanz im Zuge einer Instanzänderung modifiziert, so wird eine Deltaschicht erzeugt, in der exakt die Abweichungen zum referenzierten Schema gespeichert werden. Die geänderte Instanz referenziert nun nicht mehr das ursprüngliche Schema, sondern die Deltaschicht, die ihrerseits eine Referenz auf das ursprüngliche Schema besitzt. Abbildung 3.19 verdeutlicht die beschriebene Situation.

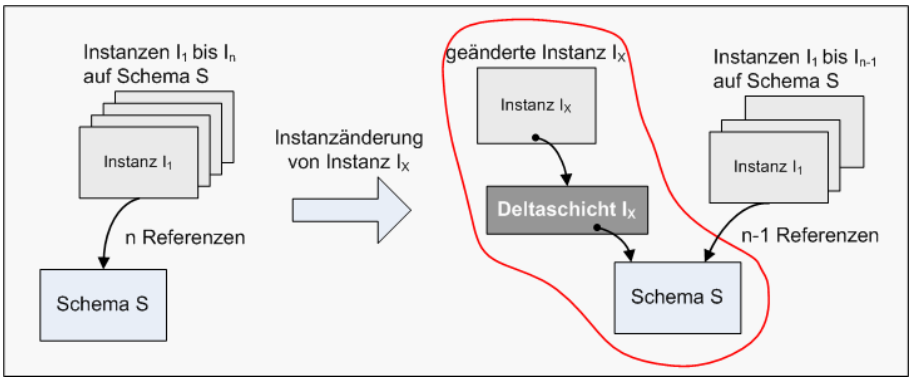


Abbildung 3.19 Geänderte Instanz mit Deltaschicht-Konzept

Da die Deltaschicht die gleichen Schnittstellen und somit dieselben Operationen wie ein Prozessschema bietet, ist es von außen irrelevant, ob eine Operation auf einer instanzspezifisch-geänderten oder einer unveränderten Instanz durchgeführt wird. Lediglich intern ergibt sich der Unterschied, dass bei modifizierten Instanzen eine Anfrage zuerst an die Deltaschicht und nicht wie bei unveränderten Instanzen direkt an das referenzierte Schema gerichtet wird.
Die Vorteile dieses Konzepts liegen auf der Hand. Einerseits ermöglicht die Verwendung von Schemareferenzen eine Migration verträglicher Instanzen durch einfaches Umbiegen der Referenz auf das geänderte Schema. Andererseits verhindert die Deltaschicht, dass bei einer Instanzänderung eine Schemakopie erzeugt und damit die für eine Schemaevolution notwendige Referenz verloren geht.

3.5.2 Schemata

Im Gegensatz zu Instanzänderungen werden Schemaänderungen direkt materialisiert. Das bedeutet, dass bei der Anwendung einer Änderungsoperation die interne Repräsentation eines Schemas direkt verändert wird. Die Änderungsoperation *insertNode*(*X*, (*A*, *B*)) aus Abbildung 3.20 bewirkt beispielsweise, dass die vorhandene Kante (*A*, *B*) gelöscht und dafür die Kanten (*A*, *X*), (*X*, *B*) und der Knoten *X* direkt in die interne Repräsentation eingefügt werden.
Um durch diese direkte Veränderung des Schemas das Weiterlaufen der bereits gestarteten Instanzen nicht zu behindern, werden die Materialisierungen auf einer Kopie der originalen Schema-Version durchgeführt.

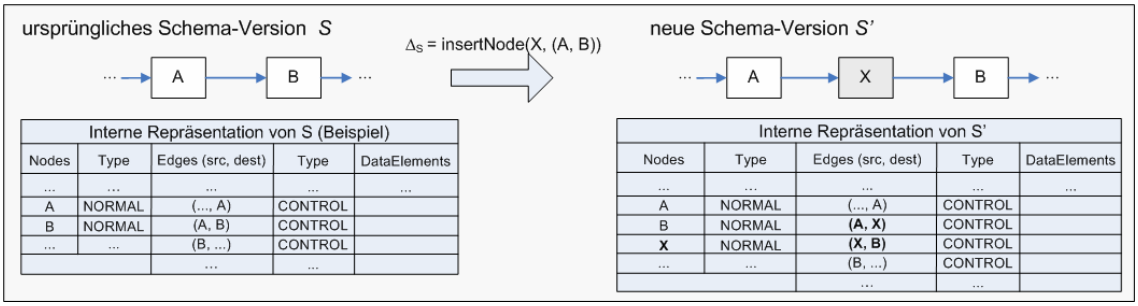


Abbildung 3.20 Materialisierung der Schemaänderungen

Auf konzeptioneller Ebene genügt die alleinige Materialisierung der Änderungen. Für die Umsetzung der Konzepte zur Schemaevolution ist es aber vorteilhaft, wenn auch auf Schemaebene die Abweichungen zum ursprünglichen Schema direkt vorliegen. Es bietet sich also an, auch hier das Deltaschicht-Konzept zu verwenden. Dadurch kann der im Rahmen einer Migration häufig vorkommende Vergleich zwischen Änderungen auf Schema- und Änderungen auf Instanzebene ohne aufwendige Berechnungen und ohne Zugriff auf eine Änderungshistorie durchgeführt werden.

Eine ausschließliche Verwendung des Deltaschicht-Konzeptes kann auf Schemaebene aber nicht angewendet werden. Dies hat seine Ursache im Zusammenspiel zwischen der Repräsentation von Instanzänderungen und der Änderung von Schemata wie folgendes Beispiel verdeutlicht:

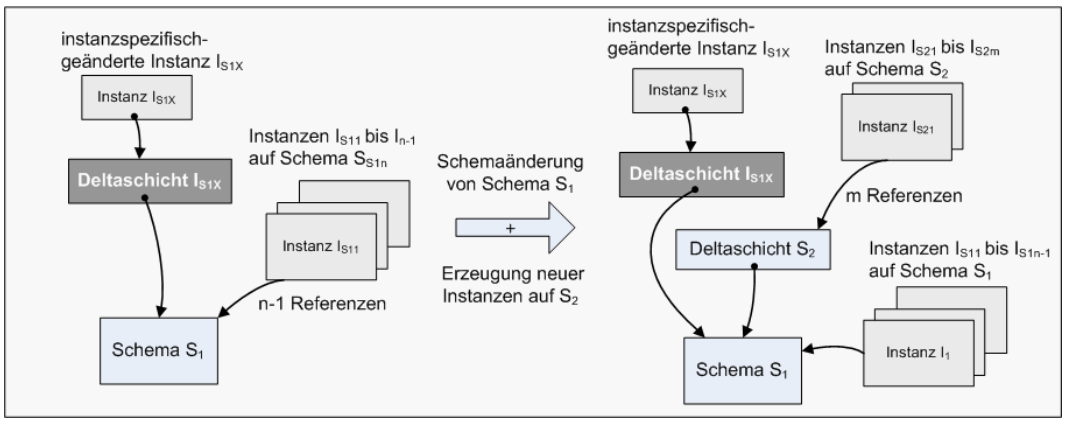


Abbildung 3.21 Situation bei Nicht-Materialisierung von Schemaänderungen

Die Instanzen I_{S11} bis I_{S1n} aus Abbildung 3.21 referenzieren bei ihrer Erzeugung die Schema-Version S_1 auf der sie beruhen. Wird eine beliebige Instanz I_{S1X} geändert, so wird eine Deltaschicht erzeugt und referenziert. Die Deltaschicht selbst referenziert daraufhin die ursprüngliche Schema-Version S_1. Wird in dieser Situation das Schema geändert, so muss bei Nicht-Materialisierung eine Deltaschicht S_2 auf das ursprüngliche Schema S_1 aufgesetzt werden, damit einerseits die bereits erzeugten Instanzen auf dem vorherigen Schema weiterlaufen und neue Instanzen auf der neuen Schema-Version S_2 erzeugt werden können. Diese Situation wird durch jede weitere Änderung noch komplizierter. Dies ist weder aus Übersichtlichkeits- noch aus Effizienzgründen vorteilhaft, weshalb zusätzlich zur Deltaschicht die Schemaänderungen auf jeden Fall materialisiert werden müssen.
Im Unterschied zur Deltaschicht bei Instanzänderungen, ist die Deltaschicht bei Schemaänderungen durch die zwingend erforderliche Materialisierung also lediglich als Meta-Information anzusehen mit der sich die Migration von Instanzen beschleunigen lässt.

3.6 Zusammenfassung

In diesem Kapitel wurden die konzeptionellen Grundlagen des Änderungsrahmenwerkes vorgestellt. Dabei wurden als Basis für die Änderungskonzepte die Modellierungskonstrukte und Laufzeitaspekte des *WSM-Net*-Prozessmodells erläutert. Um einen auf diesem Modell beruhen Prozess ändern zu können, wurde eine vollständige Menge an Änderungsoperationen entwickeln, mit Hilfe derer ein auf einem korrekten WA-Netz (*WSM-Net*) basierendes Prozessschema S bzw. instanzspezifisches Schema S_I geändert werden kann. Diese Änderungsoperationen lassen sich in verschiedene semantische Ebenen einteilen, wobei alle Operationen auf einer minimalen fest definierten Menge an Änderungsprimitiven basieren. Um die Korrektheit von S bzw. S_I bei der Anwendung einer solchen Operationen zu gewährleisten, wurden für jede Änderungsoperationen eindeutige Vor- und Nachbedingungen definiert. Diese verhindern zuverlässig die Anwendung einer Änderungsoperation, falls es dadurch zu einem bzgl. Ausführungszustand oder struktureller Korrektheit falschen S bzw. S_I kommen würde.
Aufbauend auf diesen Änderungsoperationen wurden die beiden Änderungsarten instanzspezifische Änderung und Schemaevolution getrennt betrachtet. Dabei hängt im Falle einer Instanzänderung die Zulässigkeit einer anzuwendenden Änderungsoperation von deren strukturellen und zustandsbasierten Vorbedingungen ab. Weiterhin muss nach der Anwendung einer zulässigen Änderungsoperation u. U.

die instanzspezifische Markierung M^I, durch Neubewerten der von der Änderungsoperation betroffenen Knoten, angepasst werden.
Zur Schemaänderung im Rahmen einer Schemaevolution werden ebenfalls die Änderungsoperationen eingesetzt. Erst die Migration laufender Instanzen auf ein geändertes Prozessschema erfordert zusätzliche Mechanismen. Um diese gezielt definieren zu können und die Komplexität zu reduzieren, wurden die zu migrierenden Instanzen in unveränderte (*unbiased*) und geänderte (*biased*) Instanzen unterteilt. Für die Instanzen der Klasse *unbiased* wurden Verträglichkeitskriterien beschrieben, mit Hilfe derer bestimmt werden kann, ob eine nicht geänderte Instanz (*unbiased*) *I* auf das geänderte Prozessschema *S'* migriert werden kann. Weiterhin sind nach einer Propagation der Schemaänderungen Δ_S auf eine verträgliche Instanz *I* Algorithmen notwendig um die schemaspezifische Instanzmarkierung anzupassen. Bei den Instanzen der Klasse *biased* unterscheidet man zwischen *disjoint* und *overlapping*, wobei letztere noch entlang des Überlappungsgrades der Schema- und Instanzänderungen in die Unterklassen *equivalent*, *subsumption equivalent* und *partially equivalent* unterteilt wird. Für die Klassen *subsumption equivalent* und *equivalent* wurden Verträglichkeitskriterien und Migrationsstrategien beschrieben, mit denen eine Instanz diesen Typs unter Wahrung der Korrektheit und ohne Eingriff des Benutzers auf ein geändertes Schema migriert werden kann. Bei Instanzen der Unterklasse *partially equivalent* ist es aufgrund der hohen Variabilität an instanzspezifischen Änderungen nicht möglich eine einheitliche und automatisch durchführbare Migrationsstrategie zu definieren. Eine Gruppierung der angewendeten Änderungen (*change projection*) ermöglicht es jedoch, dass für einige Instanzen der Klasse *partially equivalent* die automatisch durchzuführenden Migrationsstrategien der anderen Klassen anwendbar sind. Trotzdem bleibt eine Reihe von Instanzen, die nicht ohne Eingriff des Benutzers migriert werden können. Für diesen Teil der *partially equivalent* Instanzen lassen sich allerdings die Konflikte berechnen, wodurch der Benutzer gezielt bei der manuellen Migration unterstützt werden kann.
Neben geeigneten Konzepten zur Schema- und Instanzänderung ist für eine praktische Umsetzung auch die Art der internen Repräsentation durchgeführter Änderungen von entscheidender Bedeutung. Hierzu wurde das Deltaschicht-Konzept beschrieben, mit dem Instanzänderungen gekapselt werden, ohne dabei die Bindung an das ursprüngliche Schema *S* zu verlieren. Weiterhin wurde erläutert, dass neben der zwingend notwendigen Materialisierung von Schemaänderungen aus Effizienzgründen ebenfalls eine Deltaschicht erzeugt wird, die die Abweichungen vom ursprünglichen Schema enthält. Dadurch kann mit Hilfe des Deltaschicht-Konzeptes sowohl die Durchführung von Instanzänderungen als auch die Durchführung einer Schemaevolution bestmöglich unterstützt werden.

Welche konkreten Anforderungen sich aus den vorgestellten konzeptionellen Grundlagen ergeben und welche weiteren Anforderungen bei der Erstellung des Rahmenwerkes zu beachten sind wird im folgenden Kapitel beschrieben.

4 Anforderungen an das Änderungsrahmenwerk

Die Grobanforderungen an das Änderungsrahmenwerk sind weitestgehend mit den in Abschnitt 2.1 aufgestellten Anforderungen an geeignete konzeptionelle Ansätze identisch und entsprechen im Wesentlichen der von Anwenderseite geforderten Funktionalität eines PMS. Wie sich diese Grobanforderungen verfeinern und in Form funktionaler Anforderungen an das Änderungsrahmenwerk konkretisieren lassen wird in Abschnitt 4.1 beschrieben. Neben diesen rein funktionalen Anforderungen besteht von Benutzerseite der Wunsch, dass die Durchführung einer Änderung auch von weniger versierten Anwendern durchgeführt werden kann. Es sind bei der Entwicklung des Änderungsrahmenwerkes also auch „soft-facts“ wie Benutzerfreundlichkeit zu berücksichtigen. Diese und weitere softwaretypische Eigenschaften werden in Form nicht-funktionaler Anforderungen in Abschnitt 4.2 genauer erläutert.

4.1 Funktionale Anforderungen

Anhand der im vorherigen Kapitel beschriebenen Konzepte, lässt sich bereits erkennen, welche Funktionalität notwendig ist, um die Grobanforderungen zumindest auf konzeptioneller Ebene erfüllen zu können. Für das Änderungsrahmenwerk lassen sich daraus die folgenden funktionalen Anforderungen ableiten:

- Das Änderungsrahmenwerk stellt alle in Anhang A (Änderungsoperationen) definierten Änderungsoperationen bereit.
- Ausnahmslos alle Änderungsoperationen basieren auf Änderungsprimitiven.
- Mit den Änderungsoperationen können sowohl neue Prozessvorlagen (Schemata) erstellt, existierende Schemata geändert als auch laufende instanzspezifische Schemata modifiziert werden (Instanzänderung).
- Jede Änderungsoperation besitzt sowohl strukturelle als auch zustandsbasierte Vorbedingungen.
- Die Vorbedingungen gewährleisten, dass bei Anwendung einer Änderungsoperation die Korrektheit des erstellten/modifizierten Prozessgraphen erhalten bleibt.
- Zusätzlich muss das Rahmenwerk einen Mechanismus bereitstellen, der nach der Durchführung einer instanzspezifischen Änderung eine notwendige Zustandsneubewertung automatisch durchführt.
- Das Änderungsrahmenwerk ermöglicht die Migration von Instanzen auf ein geändertes Schema. Dies schließt insbesondere auch Instanzen ein, die im Zuge einer Instanzänderung modifiziert wurden.
- Es werden Mechanismen zur Verfügung gestellt, die es ermöglichen, Instanzen entlang des Überlappungsgrades zwischen Δ_I und Δ_S in Klassen einzuteilen.
- Für jede dieser Klassen gilt es, Verträglichkeitskriterien bereitzustellen, mit denen geprüft werden kann, ob sich eine Instanz dieser Klasse auf ein geändertes Schema migrieren lässt.
- Das Änderungsrahmenwerk ermöglicht die automatische Migration verträglicher Instanzen oder, falls dies nicht möglich ist, die Unterstützung des Anwenders bei der manuellen Migration.
- Bei der automatischen Migration ist die strukturelle und zustandsbasierte Korrektheit einer auf ein geändertes Schema migrierten Instanz zu gewährleisten. Hierzu gehört auch die Zustandsneubewertung nach erfolgter Propagation der Schemaänderungen.
- Der Benutzer ist bei der manuellen Migration durch Mechanismen zu unterstützen, die ihm die Konflikte aufzeigen, die dazu geführt haben, dass keine automatische Migration durchgeführt werden konnte.

- Die strukturelle und zustandsbasierte Korrektheit einer manuell angepassten Instanz soll so weit wie möglich vom Änderungsrahmenwerk gewährleistet werden.

4.2 Nicht-funktionale Anforderungen

Die nicht-funktionalen Anforderungen an das Änderungsrahmenwerk beschränken sich größtenteils auf die Unterstützung des Anwenders. Hierbei ist zu beachten, dass in diesem Fall unter dem Begriff „Anwender“ nicht nur der Endanwender verstanden wird. Vielmehr betrifft dies auch den Entwickler bzw. den Programmierer, der das Änderungsrahmenwerk durch Plug-Ins erweitert oder eine Komponente entwickelt, die auf die Funktionalität des Änderungsrahmenwerkes zurückgreift. Es handelt sich also um softwaretypische Anforderungen wie Erweiterbarkeit oder Modularität.

Auf eine Angabe von Anforderungen bzgl. der Effizienz von auszuführenden Algorithmen oder dem Speicherbedarf von Konstrukten wird hier verzichtet. Es wird eine möglichst effiziente und speichersparende Vorgehensweise als selbstverständlich vorausgesetzt. Sollte dies bei der Beschreibung einer Vorgehensweise einmal nicht der Fall sein, so wird dies dort gesondert begründet.

Im Folgenden werden die nicht-funktionalen Anforderungen getrennt nach Unterstützung für den Endanwender (Abschnitt 4.2.1) und Unterstützung des Anwendungsentwicklers bzw. -programmierers erläutert (Abschnitt 4.2.2).

4.2.1 Unterstützung des Endanwenders

Um die Benutzerfreundlichkeit zu steigern, stellt das Änderungsrahmenwerk eine *UNDO*-Funktion bereit. Mit Hilfe dieser Funktion kann der Anwender bei der Modellierung oder Änderung eines Schemas die Auswirkungen bereits durchgeführter Änderungsoperationen wieder rückgängig machen. Dies verbessert die Benutzbarkeit erheblich, da in der Praxis oftmals die optimale Änderung durch Ausprobieren erreicht wird.

Weiterhin ermöglicht das Änderungsrahmenwerk die Aufrufbarkeit einer Änderungsoperation im Vorfeld zu bestimmen. Dies kommt in der folgenden Situation zum Einsatz:
Der Anwender wählt in dem durch das GUI dargestellten Schema ein oder mehrere Elemente aus. Anhand dieser Auswahl erkennt das Änderungsrahmenwerk, welche Änderungsoperationen in dieser Situation möglich sind und schaltet in dem GUI entsprechende *Buttons/Icons* frei. Abbildung 4.1 zeigt beispielhaft, wie diese Funktionalität von einem GUI verwendet werden kann.

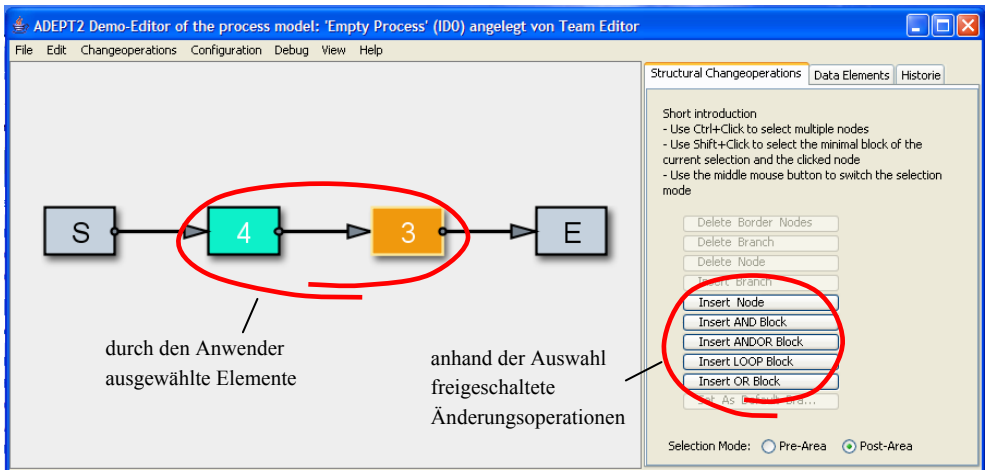


Abbildung 4.1 Situationsabhängiges Freischalten von Änderungsoperationen

Der Anwender kann durch diesen Mechanismus nur Änderungsoperationen ausführen, die in der Folge nicht zu einem inkorrekten Graphen führen. Dies ermöglicht es auch weniger versierten Benutzern, ein Schema zu modellieren bzw. zu ändern.

4.2.2 Unterstützung des Anwendungsentwicklers bzw. -programmierers

Eine bedeutende nicht-funktionale Anforderung ist die Möglichkeit, dass Änderungsrahmenwerk nachträglich erweitern zu können. Da dies im Regelfall die Aufgabe eines Anwendungsentwicklers bzw. -programmierers ist, bedeutet eine leichte Erweiterbarkeit des Rahmenwerkes auch eine Erleichterung für den jeweiligen Anwendungsentwickler bzw. -programmierer.

Zu diesem Zweck stellt das Änderungsrahmenwerk eine Plug-In Schnittstelle bereit, die es ermöglicht, auch nachträglich neue Änderungsoperationen in das System einzubringen. Dies erweist sich insbesondere dann als hilfreich, wenn in einer zukünftigen Implementierung der Einsatz eines Makrokonzeptes[7] erwünscht ist. Es könnten auf diese Weise die Makros mit Hilfe eines GUI modelliert und direkt zur weiteren Verwendung dynamisch geladen werden (vgl. [BGST04]).

Eng mit der vorhergehenden Anforderung verbunden ist die Forderung, dass das Änderungsrahmenwerk modular aufgebaut ist. Dies erleichtert insbesondere eine Anpassung an ein erweitertes Prozessmodell (z.B. um neue Knotentypen, Zeitaspekte oder Mitarbeiterzuordnungen). Speziell die Mechanismen zur Migration von Instanzen auf ein geändertes Schema profitieren von einem modularen Aufbau, da diese im Falle einer Erweiterung evtl. angepasst werden müssen.

Weiterhin gilt es, neben der bestmöglichen Unterstützung des ADEPT2-*Editors* auch Buttons/Icons und evtl. Hilfen in Form von *Tooltips* bereitzustellen, die ein GUI-Programmierer direkt in seine Oberfläche einbauen kann. In Abbildung 4.1 sind dies z.B. die *Buttons/Icons* zur Auswahl einer Änderungsoperation. Dadurch ist es möglich, die Funktionalität des Änderungsrahmenwerkes auch für andere Editoren zugänglich zu machen. Dies ist besonders im Hinblick auf einen Einsatz im unternehmerischen Umfeld notwendig, da dort oftmals einheitliche Arbeitsoberflächen zum Einsatz kommen.

Neben dieser graphischen Unterstützung ist die Funktionalität des Änderungsrahmenwerkes – wo sinnvoll möglich – auch von einem „reinen“ API bereitzustellen, damit eventuelle Änderungen auch ohne graphische Benutzeroberfläche ausgeführt werden können.

4.3 Zusammenfassung

Die Grobanforderungen an das Änderungsrahmenwerk ergeben sich direkt aus den Forderungen die von Anwenderseite an die Flexibilität eines PMS gestellt werden. Welche Anforderungen sich daraus im Detail ableiten lassen und welche Mechanismen das Änderungsrahmenwerk zur Steigerung der Benutzerfreundlichkeit bereitstellen muss wurde in diesem Kapitel beschrieben. Neben den Anforderungen von Anwenderseite existieren zusätzliche Anforderungen sowohl von Programmiererseite als auch aus softwaretechnischer Sicht. Dies betrifft Aspekte wie Erweiterbarkeit oder Modularität. Auch hierfür wurden konkrete Anforderungen formuliert.

[7] Unter Makrokonzept wird hier ein Mechanismus verstanden, mit dem man in einem GUI das Anwenden von Änderungsoperationen mitprotokollieren und daraus eine normal verwendbare Änderungsoperation erzeugen kann.

5 Architektur

Nachdem die Anforderungen an das Änderungsrahmenwerk feststehen wird in diesem Kapitel die Architektur des Rahmenwerkes vorgestellt und dieses entlang der Anforderungen in Komponenten unterteilt. Dabei ist es hilfreich das Änderungsrahmenwerk in einem ersten Schritt in die Gesamtarchitektur eines Prozess-Management-Systems einzuordnen und die Kommunikation mit umliegenden Komponenten zu beschreiben (Abschnitt 5.1). Hierbei ist insbesondere ein für ADEPT2 entwickeltes Datenmodell von Bedeutung, das als Schnittstelle zu den in der Prozess-Engine verwalteten Schema- und Instanzdaten fungiert und von allen Komponenten des PMS genutzt werden kann. In Abschnitt 5.2 wird die Architektur des Rahmenwerkes entlang der Anforderungen verfeinert und die Funktionalität der resultierenden Komponenten beschrieben.

5.1 Einbettung des Änderungsrahmenwerkes in die Architektur eines PMS

Die Einbettung des Änderungsrahmenwerkes in die Gesamtarchitektur eines PMS ist insofern von Bedeutung, da die Position darüber entscheidet, auf welche Daten und Funktionen das Rahmenwerk zurückgreifen kann. Weiterhin lässt sich durch die Betrachtung der Gesamtarchitektur vermeiden, dass im Änderungsrahmenwerk Funktionalität implementiert wird, die in den Aufgabenbereich anderer Komponenten fallen.

Um die Einbettung eines Änderungsrahmenwerkes zu verdeutlichen, wird die Architektur des ADEPT2-PMS verwendet. Es besteht allerdings auch die Möglichkeit das Änderungsrahmenwerk in die Architektur anderer Prozess-Management-Systeme zu integrieren. Die Verdeutlichung anhand des ADEPT2-PMS ist allerdings naheliegend, da das Änderungsrahmenwerk in erster Linie für dieses System entwickelt wird. Abbildung 5.1 zeigt die Position in der aktuellen Architektur:

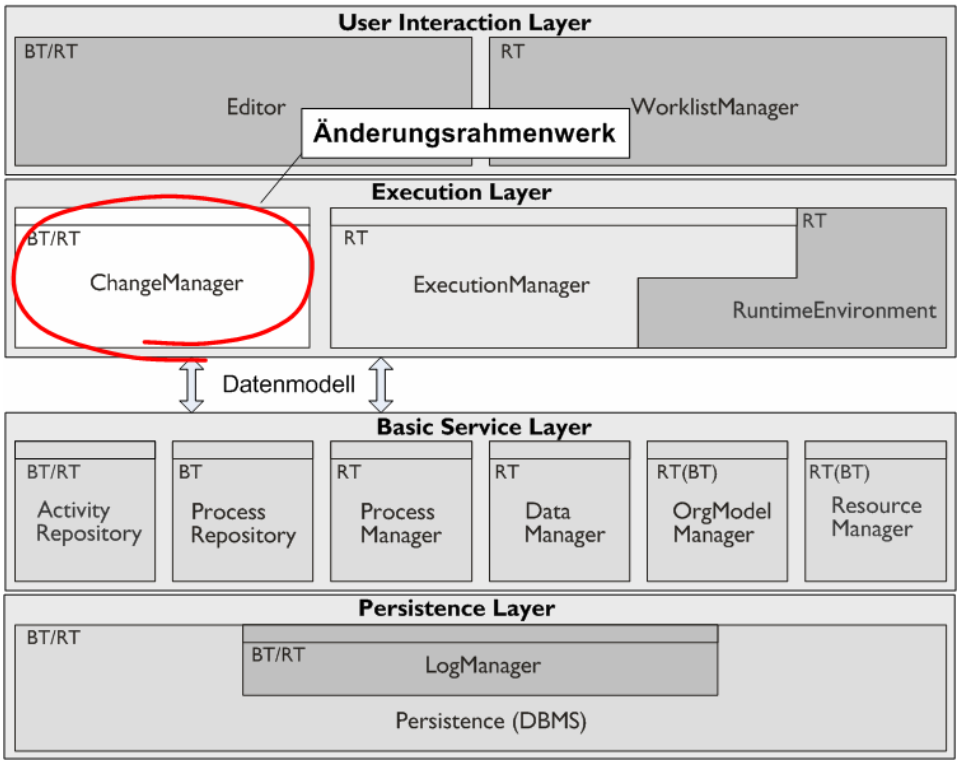


Abbildung 5.1 Einbettung des Änderungsrahmenwerkes in die Architektur von ADEPT2

Das Änderungsrahmenwerk (in der Abbildung als *ChangeManager* bezeichnet) befindet sich aufgrund der Art der bereitzustellenden Funktionalität in der *Execution Layer*. In dieser Schicht werden die so genannten „höheren“ Dienste und die Kern-Funktionalität des ADEPT2-PMS bereitgestellt. Bei der Migration von Instanzen auf ein geändertes Schema handelt es sich um einen solchen „höheren“ Dienst, während die Anwendung einer Änderungsoperation im Zuge einer Prozessmodellierung als Kern-Funktionalität eines PMS aufgefasst werden kann. Es empfiehlt sich also eine Einbettung an dieser Stelle.

Anhand dieser statischen Einbettung wird in den folgenden zwei Abschnitten die Kommunikation zwischen dem *ChangeManager* und den umliegenden Komponenten beschrieben. Dabei liegt das Hauptaugenmerk auf der Beschreibung derjenigen Funktionalität, auf die das Änderungsrahmenwerk zurückgreifen kann (Abschnitt 5.1.2). Bei der Beschreibung der angebotenen Funktionalität (Abschnitt 5.1.1) werden an dieser Stelle die „konsumierenden“ Komponenten und die Art der Kommunikation beschrieben.

5.1.1 Angebotene Schnittstellen

Wie Abbildung 5.1 zeigt, stellt das Rahmenwerk seine Funktionalität der *User Interaction Layer* zur Verfügung. Die Komponenten dieser Schicht bilden die Schnittstelle zum Benutzer. Sie besteht aus den beiden Komponenten *Editor* und *WorklistManager*. Dabei spricht der *Editor* die Funktionalität des Änderungsrahmenwerkes größtenteils über eine graphische Oberfläche (GUI) an, d.h. der *Editor* fungiert lediglich als Vermittler zwischen dem Änderungsrahmenwerk und dem Benutzer. Die Ablauflogik bei der Durchführung einer Änderung übernimmt vollständig das Änderungsrahmenwerk (vgl. Abschnitt 6.5). Der *WorklistManager* hingegen nutzt die Funktionalität ausschließlich über API-Aufrufe. Da aber im Regelfall das Erstellen und Ändern von Schemata, sowie die Migration laufender Instanzen vom *Editor* aus durchgeführt werden, handelt es sich bei dieser Komponente um den „Hauptkonsumenten“ der vom Änderungsrahmenwerk bereitgestellten Funktionalität. Dies gilt es bei der Umsetzung der Änderungskonzepte entsprechend zu berücksichtigen und zu unterstützen.

5.1.2 Verwendete Schnittstellen

Der *ChangeManager* selbst kann auf die Funktionalität der *Basic Service Layer* zugreifen. Diese Schicht beinhaltet die Mechanismen zur Bearbeitung der grundlegenden Aufgaben eines PMS. So verwaltet diese Schicht mit den Komponenten *ProcessRepository* und *ProcessManager* die erzeugten Schemata und die laufenden Instanzen, und mit den Komponenten *ActivityRepository* und *DataManager* die an Knoten gekoppelten Aktivitätenvorlagen/-instanzen bzw. die Datenelemente für den Datenfluss. Von besonderer Bedeutung für das Änderungsrahmenwerk sind hauptsächlich der *ProcessManager* und das *ProcessRepository*. Letztere Komponente übernimmt die Verwaltung von Prozessvorlagen (Schemata) und die darauf durchgeführten Änderungen. Der *ProcessManager* verwaltet aktuell laufende Instanzen, die entsprechenden Prozessvorlagen und instanzspezifische Änderungen. Die Schemaevolution erfolgt im Zusammenspiel der Daten aus *ProcessRepository* und *ProcessManager*.
Der *ChangeManager* erhält also von diesen beiden Komponenten die für dessen Funktionalität notwendigen Schema- bzw. Instanzdaten. Um hierbei eine größtmögliche Unabhängigkeit von der tatsächlichen Implementierung der Instanzen und Schemata zu gewährleisten, geschieht der Datenaustausch über ein für ADEPT2 entwickeltes Datenmodell (vgl. Abbildung 5.1). Dieses wird in Abschnitt 5.1.2.1 vorgestellt und in Abschnitt 5.1.2.2 so erweitert, dass die Mechanismen des Änderungsrahmenwerkes optimal unterstützt werden können.

5.1.2.1 Datenmodell

Das für ADEPT2 entwickelte Datenmodell dient dem Zweck von der eigentlichen Repräsentation und Verwaltung der im ADEPT2-PMS vorhandenen Daten zu abstrahieren. Neben der angebotenen Funktionalität bringt die Verwendung des Datenmodells für das Änderungsrahmenwerk aber auch den Vorteil, dass dieses von der darunterliegenden Prozess-Management-Engine unabhängig ist. Dadurch besteht die Möglichkeit, dass Rahmenwerk auch mit einer beliebigen anderen Prozess-Management-Engine zu betreiben.
Abbildung 5.2 und Abbildung 5.3 zeigen die für das Änderungsrahmenwerk wichtigsten Methoden des Datenmodells. Eine Darstellung des vollständigen Datenmodells findet sich in Anhang B (ADEPT2-Datenmodell).

Für die Modellierung oder Manipulation von Instanzen oder Schemata ist insbesondere die Schnittstelle *ProcessChangePrimitives* wichtig. Mit Hilfe der dort definierten Methoden können die Datenstrukturen der Schemata bzw. Instanzen direkt manipuliert werden. Es handelt sich dabei also um die Umsetzung der in den Änderungsoperationen verwendeten Änderungsprimitiven (vgl. Tabelle 0.1 Anhang A (Änderungsoperationen)). Weiterhin ist die Funktionalität des Interfaces *Node* bzw. dessen Subinterfaces von Bedeutung. Hierüber lassen sich die für eine Änderung notwendigen Informationen wie Knotentyp (*node.getType()*), zugehöriger Verzweigungsknoten (*node.getSplitNode()*), Vorgänger-/Nachfolgerbeziehung (*node1.isPredecessorOf(node2)* bzw. *node2.isPredecessorOf(node1)*)[8], Teilzweignummer (*getBranchID*) und Knotenzustand (*getNodeState()*) abrufen.

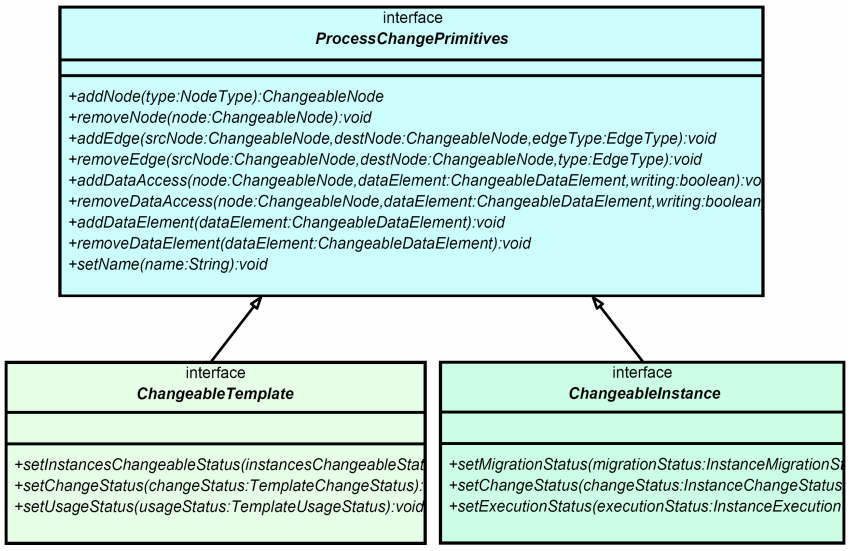


Abbildung 5.2 Relevante Schnittstellen des Datenmodells (1)

[8] Eine *isSuccessorOf*-Methode Für die Prüfung, ob ein Knoten *node1* Nachfolger eines Knotens *node2* ist, wird durch *node1.isPredecessorOf(node2)* simuliert.

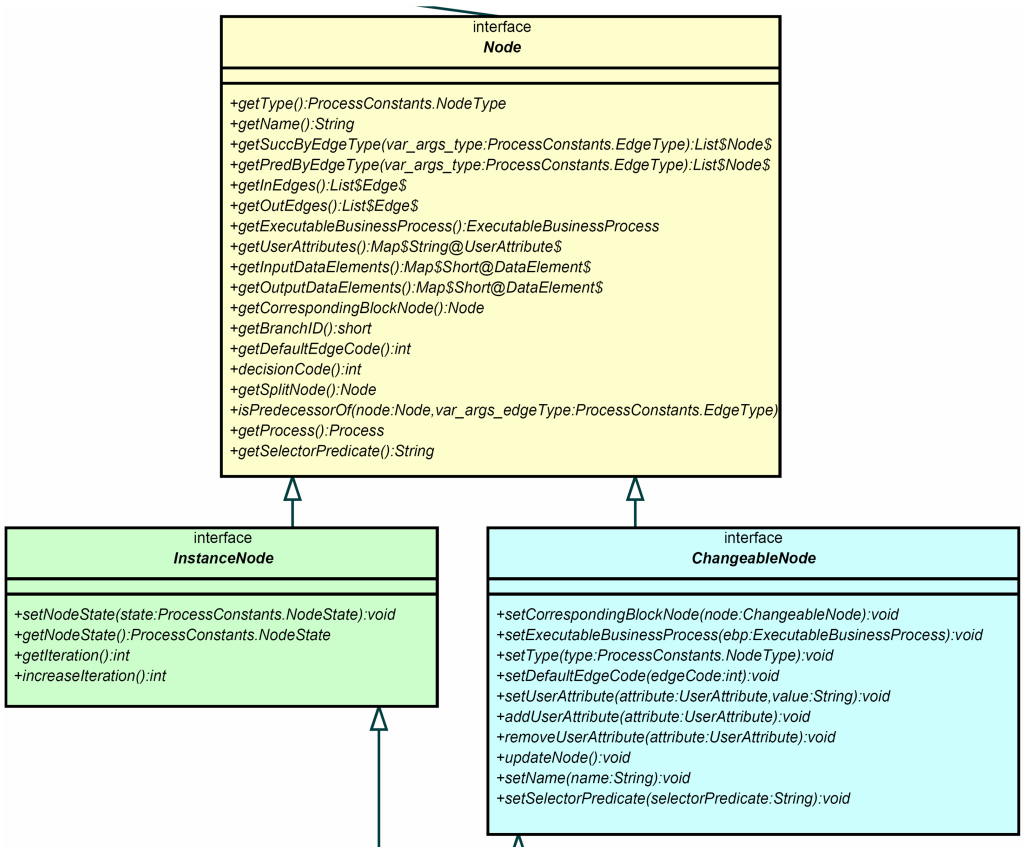


Abbildung 5.3 Relevante Schnittstellen des Datenmodells (2)

5.1.2.2 Erweiterung des Datenmodells

Für eine effiziente Unterstützung aller Änderungsarten sind die angebotenen Methoden allerdings noch nicht ausreichend. Dies hängt damit zusammen, dass bei der Entwicklung des Änderungsrahmenwerkes davon ausgegangen wird, dass die zugrunde liegende Prozess-Engine das Deltaschicht-Konzept implementiert (vgl. Abschnitt 3.5). Folglich benötigt das Änderungsrahmenwerk Methoden, mit Hilfe derer die Daten einer Deltaschicht manipuliert werden können. Das bisherige ADEPT2-Datenmodell enthält hierzu keine passende Schnittstelle. Das Datenmodell wird deshalb um die beiden Interfaces *DeltaLayer* und *ChangeableDeltaLayer* (vgl. Abbildung 5.4) erweitert. Dabei erbt die Schnittstelle *ChangeableDeltaLayer* die Operationen von *ProcessChangePrimitives*. Dies ist naheliegend, da eine Deltaschicht nach außen hin exakt die gleichen Schnittstellen wie ein Prozessschema bietet (vgl. Abschnitt 3.5)). Folglich können die von *ProcessChangePrimitives* geerbten Methoden analog zur Manipulation von Schemata und Instanzen auch zur Manipulation der Deltaschicht selbst herangezogen werden.
Ein schreibender Zugriff auf die Deltaschicht ist aber noch nicht ausreichend. Deshalb ermöglicht das Interface *DeltaLayer* den lesenden Zugriff auf die in der Deltaschicht gespeicherten Elemente und Datenstrukturen.

Diese Erweiterung ist für das Änderungsrahmenwerk von entscheidender Bedeutung, da die Funktionalität beider Schnittstellen Grundvoraussetzung für das Funktionieren der Mechanismen zur Durchführung einer Schemaevolution sind. Dies wird in Kapitel 7 (Umsetzung der Schemaevolution) genauer erläutert.

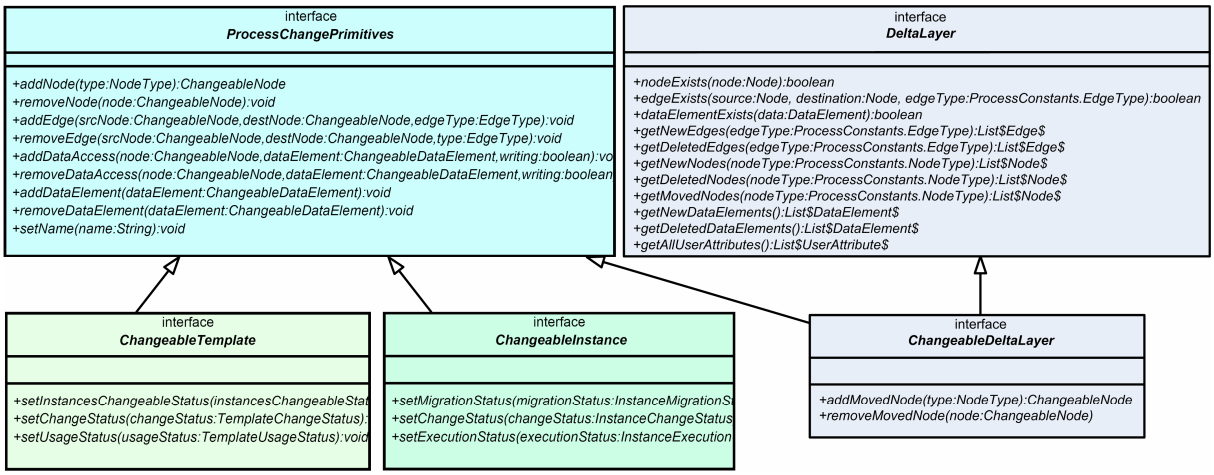


Abbildung 5.4 Erweiterung des Datenmodells

5.2 Verfeinerung des Änderungsrahmenwerkes

Neben der beschriebenen Kommunikation des Änderungsrahmenwerkes mit der „Außenwelt“ sind auch die Internas des Rahmenwerkes wichtig. Um diese genauer zu erläutern wird die *ChangeManager*-Komponente weiter verfeinert. Ziel dabei ist die Beschreibung derjenigen Komponenten, die bei einer Implementierung die Funktionen zur Erfüllung der in Kapitel 4 aufgestellten Anforderungen enthalten. Abbildung 5.5 zeigt die verfeinerte Architektur mit der logischen Einteilung in *Process Change*, *Change Operations*, *Migration*, *Compliance Test* und *Adaptation Layer* im Überblick.

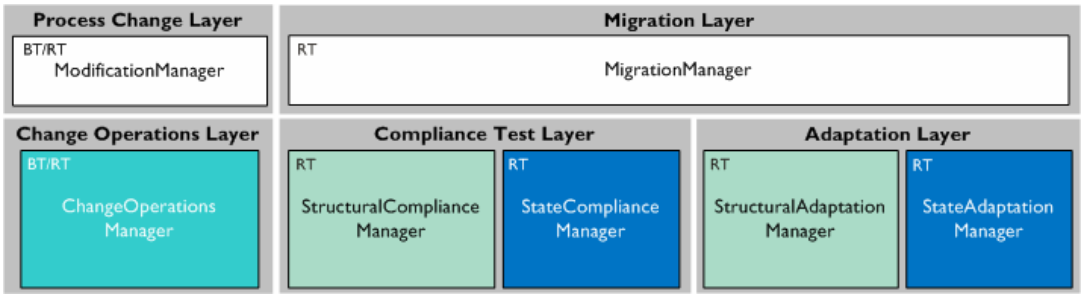


Abbildung 5.5 Verfeinerung der *ChangeManager*-Komponente

Auf Grund der Bezeichnungen erkennt man bereits, dass sich die vom Änderungsrahmenwerk bereitzustellende Funktionalität in die beiden großen Bereiche Prozessänderung und Migration unterteilen lässt.

Für das Modellieren oder Ändern eines Schemas und die Durchführung einer Instanzänderung sind die Komponenten *ModificationManager* und *ChangeOperationsManager* zuständig. Dabei verwaltet der *ModificationManager* das zu ändernde Schema bzw. die zu ändernde Instanz und koordiniert abhängig von der Art des Aufrufes (GUI oder API) die Ausführung einer Änderungsoperation. Weiterhin stellt er die Funktionen zur Unterstützung des Anwenders bereit (vgl. Abschnitt 4.2). Dies sind im Speziellen die *UNDO*-Funktion, die Möglichkeit, die Aufrufbarkeit einer Änderungsoperation im Vorfeld zu testen, und das Bereitstellen von Hilfen und Icons. Die dafür notwendigen Informationen bezieht der *ModificationManager* vom *ChangeOperationsManager*. Dieser verwaltet die im Änderungsrahmenwerk verfügbaren Änderungsoperationen inklusive der bei diesen definierten Vorbedingungen und Mechanismen zur Zustandsanpassung bei Instanzänderungen. Zusätzlich erfüllt der *ChangeOperationsManager* die Funktionalität des Plug-In-Konzeptes. Das nachträgliche Einfügen

von neuen Änderungsoperationen läuft folglich über diese Komponente. Wie die beschriebenen Komponenten ihre Funktionalität im Einzelnen bereitstellen, wird in Kapitel 6 detailliert beschrieben.

Die Komponenten der Schichten *Migration*, *Compliance Test* und *Adaptation* bieten die Funktionalität, mit Hilfe derer Instanzen auf das geänderte Schema migriert werden können. Dabei ist es Aufgabe des *MigrationManagers* die Instanzen in die unterschiedlichen Klassen (vgl. Abschnitt 3.4 und Abschnitt 7.2) einzuteilen und abhängig davon das weitere Vorgehen bei der Migration zu koordinieren. Dies gilt auch im Besonderen für den Fall, dass eine Instanz nicht automatisch migriert werden kann. Hierfür stellt der *MigrationManager* Mechanismen bereit, die den Anwender bei der manuellen Migration unterstützen.
Die für die Erfüllung seiner Aufgaben benötigte Funktionalität erhält der *MigrationManager* von den Komponenten der Schichten *Compliance Test* und *Adaptation*. So wird der Test der Verträglichkeit einer Instanz mit einem geänderten Schema von den Komponenten *Structural* und *StateComplianceManager* übernommen, während die in manchen Fällen notwendige Anpassung der Deltaschicht, in der Komponente *StructuralAdaptationManager* durchgeführt wird. Eine zum Abschluss einer Schemaevolution notwendige Zustandsneubewertung wird vom *StateAdaptationManager* übernommen. Detaillierte Beschreibungen zur Umsetzung der Funktionalität finden sich in Kapitel 7.

5.3 Zusammenfassung

In diesem Kapitel wurde das Änderungsrahmenwerk in die Gesamtarchitektur des ADEPT2-PMS eingegliedert, die Kommunikation mit den umliegenden Komponenten beschrieben und das Rahmenwerk entlang der Anforderungen aus Kapitel 4 in Komponenten unterteilt. Im Speziellen wurde dabei auf die Daten und Funktionen eingegangen auf die das Änderungsrahmenwerk zugreifen kann. In diesem Zusammenhang wurde das ADEPT2-Datenmodell als Schnittstelle zu den in der Prozess-Engine verwalteten Schema- und Instanzdaten vorgestellt. Damit das Datenmodell allerdings in der Lage ist, alle vom Änderungsrahmenwerk bereitzustellenden Änderungsarten optimal zu unterstützen, wurde dieses um weitere Schnittstellen und Methoden erweitert. Bei der Verfeinerung der Architektur wurden speziell die Komponenten des Rahmenwerkes beschrieben, die bei einer Implementierung die Funktionen zur Erfüllung der in Kapitel 4 aufgestellten Anforderungen enthalten.

Aufbauend auf den Komponenten der verfeinerten Architektur, den Anforderungen aus Kapitel 4 und den konzeptionellen Grundlage aus Kapitel 3, wird in den folgenden zwei Kapiteln umfassend beschrieben, mit welchen Konzepten und Algorithmen die Komponenten des Änderungsrahmenwerkes in der Lage sind, die geforderte Funktionalität zu erbringen.

6 Prozesserstellung und -änderung

Ein Teil der vom Änderungsrahmenwerk geforderten Funktionalität bezieht sich auf die Unterstützung der Schemaerstellung sowie der Schema- und Instanzänderung. Mit welchen Konzepten und Mechanismen das Rahmenwerk in der Lage ist diese Anforderungen zu erfüllen, wird in diesem Abschnitt detailliert beschrieben. Die konzeptionellen Grundlagen bilden dabei die Änderungsoperationen aus Abschnitt 3.2 und die Beschreibungen zur Instanzänderung aus Abschnitt 3.3. Bei deren Umsetzung in diesem Kapitel sind zusätzlich die nicht-funktionalen Anforderungen wie Benutzerfreundlichkeit und Erweiterbarkeit zu berücksichtigen, da sich diese maßgeblich auf die Implementierung der Konzepte auswirken. Im Einzelnen wird im Abschnitt 6.1 die Erstellung und Änderung eines Schemas und die notwendigen Schritte bei einer Instanzänderung beschrieben. Der Abschnitt 6.2 zeigt, wie die für eine Änderung notwendigen Änderungsoperationen im Rahmenwerk gespeichert und verwaltet werden (Plug-In-Schnittstelle). In Abschnitt 6.3 werden die Mechanismen zur Umsetzung einer *UNDO*-Funktion für durchgeführte Änderungen beschrieben. Abschnitt 6.4 stellt ein Konzept vor, bei dem Anwendern ausschließlich diejenigen Änderungsoperationen angeboten werden, die im momentanen Kontext zulässig sind. Um das Zusammenspiel der Konzepte zu verdeutlichen und die dabei auftretenden Probleme zu lösen, wird in Abschnitt 6.5 ein konkreter Anwendungsfall im Detail erläutert.

6.1 Funktionaler Ablauf

In diesem Abschnitt wird zunächst der Gesamtablauf bei einer Prozesserstellung bzw. -änderung mit allen durchzuführenden Schritten und den notwendigen Interaktionen mit anderen Komponenten des PMS erläutert. Dann wird detailliert beschrieben, wie im Änderungsrahmenwerk ein Prozess bzw. ein Schema mit Hilfe der Änderungsoperationen manipuliert wird und wie sich diese Änderungen auf die interne Repräsentation auswirken.

6.1.1 Gesamtablauf

Das Erstellen einer neuen Prozessvorlage (Schema) beginnt mit der Erzeugung eines sogenannten Null-Schemas. Dieses besteht lediglich aus einem Start- und einem Endknoten sowie einer leeren Deltaschicht. Ausgehend von diesem Schema, wird der Prozess durch Anwenden der vordefinierten Änderungsoperationen in die gewünschte Form gebracht. In diesem Punkt unterscheidet sich eine Schemaerstellung nicht von einer Schemaänderung. Der Unterschied besteht lediglich im Ausgangsschema. Während bei einer Schemaerstellung auf Basis eines Null-Schemas eine komplett neue Prozessvorlage entsteht, wird bei einer Schemaänderung lediglich eine bereits vorhandene Prozessvorlage aus dem *ProcessRepository* geholt und angepasst. Dabei wird nicht direkt die im *ProcessRepository* gespeicherte Prozessvorlage manipuliert, sondern eine Kopie erzeugt und die Änderungen darauf ausgeführt. Dies ist wichtig, da eine Schemaänderung immer als Änderungstransaktion angesehen wird. Es gilt demzufolge das Prinzip „Alles oder Nichts“. Sollte es nun im Rahmen einer Änderung zu einem nicht vorhersehbaren Fehler kommen, so bleibt die ursprüngliche Form und somit die Lauffähigkeit des gesamten Systems erhalten. Zusätzlich ermöglicht das Anlegen einer Kopie dem Anwender alle Auswirkungen der bis zu diesem Zeitpunkt angewendeten Änderungsoperationen rückgängig zu machen. Für Schemaerstellungen gilt dies natürlich implizit, da dass Null-Schema immer eine bekannte einheitliche Form besitzt.
Hat der Anwender die Prozessvorlage in die gewünschte Form gebracht so bestätigt er dies durch einen vom *ModificationManager* bereitgestellten *Commit*-Befehl. Daraufhin koordiniert dieser die Speicherung des neuen bzw. geänderten Schemas inkl. zugehöriger Deltaschicht im

ProcessRepository. Von dort können auf Grundlage dieser (geänderten) Prozessvorlage Prozessinstanzen erzeugt und ausgeführt werden.

Für das Ändern von Instanzen, die bisher noch auf dem Schema laufen auf dem sie ursprünglich erzeugt wurden (*unbiased*-Instanzen), wird analog zur Erstellung einer Prozessvorlage eine leere Deltaschicht erzeugt. Wird hingegen eine bereits geänderte Instanz erneut manipuliert, so wird deren Deltaschicht aus dem *ProcessManager* geholt und eine Kopie erzeugt. Diese erfüllt den gleichen Zweck wie bei einer Prozessänderung.
Bevor eine Instanz jedoch geändert werden kann, muss die zu ändernde Instanz „angehalten“ werden. Hierzu weißt das Änderungsrahmenwerk den *WorklistManager* an, alle Aktivitäten, die in den Arbeitslisten als ausführbar markiert sind, zu sperren oder zu entfernen. Dies ist notwendig damit ein Bearbeiter nicht während der instanzspezifischen Änderung eine Aktivität bearbeitet und somit die Instanz unter Umständen in einen nach Definition 2 inkorrekten Zustand versetzt. Danach können die gewünschten Änderungen durch Anwenden verschiedener Änderungsoperationen in die Instanz eingebracht werden.
Zur Speicherung der geänderten Instanz ruft der *ModificationManager* entsprechende Methoden im *ProcessManager* auf, um die neue Deltaschicht zu speichern bzw. die bereits vorhandene zu ersetzen. Danach wird der *WorklistManager* angewiesen die Instanz fortzusetzen, wodurch die im Zustand *ACTIVATED* befindlichen Aktivitäten in die entsprechenden Arbeitslisten eingefügt werden.

6.1.2 Anwenden der Änderungsoperationen

Wie beim Gesamtablauf beschrieben kann dass Schema bzw. die Instanz durch Anwendung der Änderungsoperationen in die gewünschte Form gebracht werden. Eine Unterscheidung zwischen Schemaerstellung und -änderung sowie zwischen erstmaliger und erneuter Instanzänderung ist im Folgenden nicht mehr notwendig.
Es ergibt sich allerdings ein Unterschied zwischen dem Anwenden einer Änderungsoperation auf Schema- und dem Anwenden auf Instanzebene. Um dies zu konkretisieren, wird im Folgenden anhand von *moveNodes*, stellvertretend für alle Änderungsoperationen, der Ablauf bei der Anwendung einer Änderungsoperation für beide „Änderungsarten“ im Detail erläutern. Als Ausgangsschema *S* bzw. instanzspezifisches Schema S_I dienen die Graphen aus Abbildung 6.1.

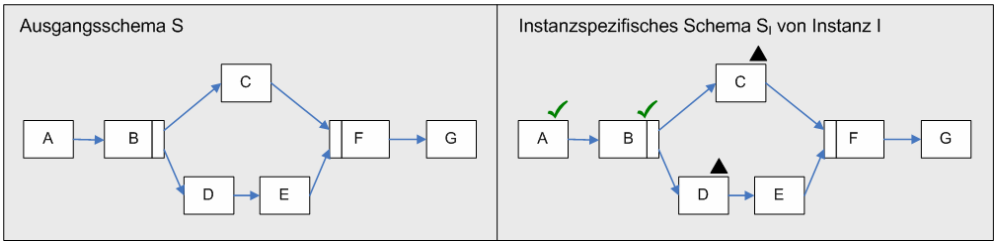


Abbildung 6.1 Ausgangssituation

Die für eine Anwendung von *moveNodes* notwendigen Informationen zeigt Tabelle 6.1.

moveNodes($S_{(I)}$, *first*, *last*, *pred*, *succ*)	
Vorbedingung (strukturell):	- *pred*, *succ, first und last* sind im Schema enthaltene Knoten - *pred* und *succ* sind durch eine Kontrollkante miteinander verbunden (dazu zählen auch Verzweigungs- und Vereinigungsknoten, die durch einen leeren Zweig verbunden sind) - *first* und *last* bilden einen gültigen Kontrollblock, d.h. sie befinden sich auf derselben Blockebene
Vorbedingung (zustandsbasiert):	- *first* und *succ* befinden sich in einem der Zustände *NOT_ACTIVATED, ACTIVATED*

Verwendete Änderungsprimitiven:	- *removeEdge*($S_{(I)}$, *pred*(*first*), *first*, *CONTROL*) - *removeEdge*($S_{(I)}$, *last*, *succ*(*last*), *CONTROL*) - *addEdge*($S_{(I)}$, *pred*(*first*), *succ*(*last*), *CONTROL*) - *removeEdge*($S_{(I)}$, *pred*, *succ*, *CONTROL*) - *addEdge*($S_{(I)}$, *pred*, *first*, *CONTROL*) - *addEdge*($S_{(I)}$, *last*, *succ*, *CONTROL*) - *addMoveNodes* für alle Knoten zwischen *first* und *last*: *addMovedNode*($S_{(I)}$, *first*), ..., *addMovedNode*(*last*)
Nachbedingungen:	- Es existieren keine Zyklen durch Sync-Kanten - Der Datenfluss bleibt korrekt, d.h. die von den verschobenen Knoten benötigten Daten werden alle noch vor Ausführung der Knoten geschrieben und alle verschobenen Knoten schreiben ihre Daten noch rechtzeitig.
Zustandsneubewertung:	- befindet sich *succ* im Zustand *ACTIVATED*, so muss dieser auf *NOT_ACTIVATED* und *first* auf *ACTIVATED* gesetzt werden. - befindet sich succ nicht im Zustand *ACTIVATED*, *first* allerdings schon, so muss *first* auf *NOT_ACTIVATED* gesetzt werden. - befindet sich der Vorgänger von first (an der Ursprungsstelle) im Zustand Completed so muss der Nachfolger von last (an der Ursprungsstelle) neu bewertet werden.

Tabelle 6.1 Die Änderungsoperation *moveNodes*

Eine vollständige Auflistung mit Informationen zu allen Änderungsoperationen findet sich im Anhang A (Änderungsoperationen).

Schemaänderung mittels moveNodes

Bei der Anwendung von *moveNodes*(*S*, *D*, *E*, (*F*, *G*)), wird in einem ersten Schritt wie bei jeder Änderungsoperation die Verträglichkeit mit dem zu ändernden Schema *S* geprüft. Es wird also getestet, ob die Anwendung der Änderungsoperation mit den angegebenen Parametern zu einem nach Definition 1 korrekten Graph führt. Dies ist genau dann der Fall, wenn die bei einer Änderungsoperation hinterlegten Vorbedingungen erfüllt sind. Wie man aus Tabelle 6.1 erkennen kann, sind dies Kriterien wie „sind die als Parameter übergebenen Knoten alle vorhanden“ oder „sind *first* und *last* auf der selben Blockebene[9]“. Um dies zu prüfen, hält das verwendete Datenmodell bereits entsprechende Methoden wie *nodeExists*, *edgeExists* oder *getBranchID* bereit (vgl. Abschnitt 5.1.2.1). Diese Methoden arbeiten alle lesend auf der internen Repräsentation eines Schemas. Somit erhält man mit ihrer Hilfe, alle notwendigen Informationen über ein Schema. Es lässt sich beispielsweise der Test, ob sich *D* und *E* auf derselben Blockebene und damit im selben Teilzweig befinden, durch Vergleich der Rückgabewerte von *getBranchID*(*D*) und *getBranchID*(*E*) durchführen.
Im betrachteten Beispiel aus Abbildung 6.1 sind alle strukturellen Vorbedingungen auf dem Schema *S* erfüllbar: *D*, *E*, *F* und *G* sind in *S* vorhanden, *F* und *G* sind über eine Kontrollkante verbunden und *D* und *E* befinden sich auf derselben Blockebene. Mit Erfüllung der strukturellen Vorbedingungen ist bei einer Schemaänderung bereits die Verträglichkeit gewährleistet. Zustandsbasierte Vorbedingungen müssen hier nicht geprüft werden, da ein Schema keine Laufzeitinformationen enthält.
Die durch *moveNodes* verursachten strukturellen Änderungen am Schema, werden durch Anwenden der in Tabelle 6.1 aufgeführten Änderungsprimitiven erreicht. Auch diese werden bereits vom Datenmodell bereitgestellt. Entscheidend ist jedoch die Datenstruktur auf der diese Änderungsprimitiven angewendet werden. Abbildung 6.2 verdeutlicht die beteiligten Strukturen vor und nach Durchführung der strukturellen Änderungen durch *moveNodes*(*S*, *D*, *E*, (*F*, *G*)).

[9] vgl. Abschnitt 2.1.1

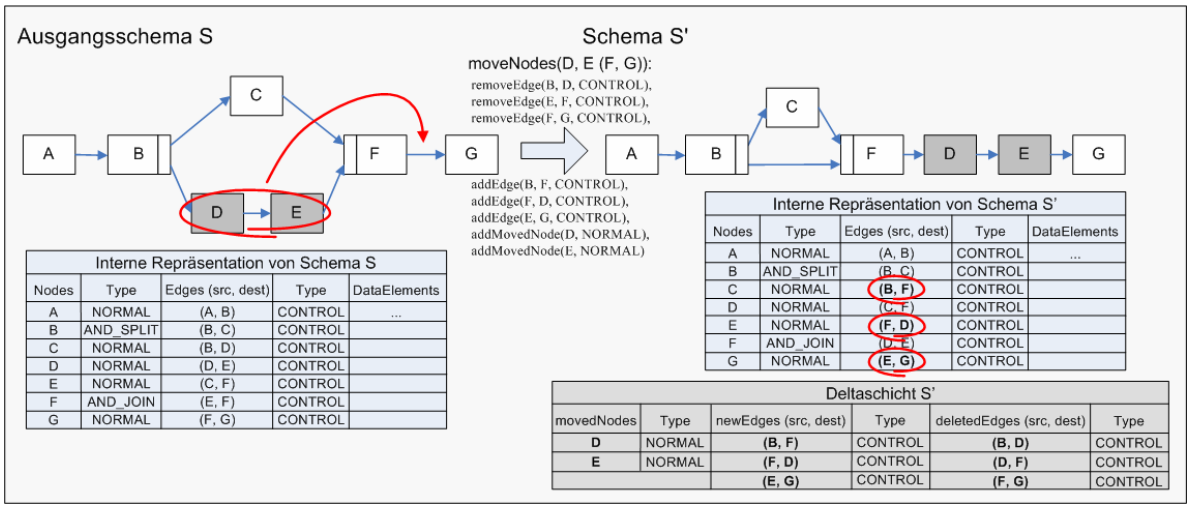


Abbildung 6.2 Schema S nach Anwendung der in *moveNodes* enthaltenen Änderungsprimitiven[10]

Wie man gut erkennen kann, werden die Änderungen direkt auf der internen Repräsentation des zu ändernden Schemas materialisiert. Die durch die Primitive *removeEdge* gelöschten Kanten (*B*, *D*), (*E*, *F*) und (*F*, *G*) werden tatsächlich aus *Edges* gelöscht und die durch *addEdges* neu eingefügten Kanten (*B*, *F*), (*F*, *D*) und (*E*, *G*) in *Edges* eingefügt. Die interne Repräsentation von *S'* repräsentiert also exakt das geänderte Schema. Zusätzlich werden die Abweichungen zum ursprünglichen Schema *S* aber auch als Meta-Information in der Deltaschicht gespeichert.
Die Nachbedingungen von *moveNodes* sind im betrachteten Beispiel automatisch erfüllt, da weder Sync-Kanten noch Datenelemente vorhanden sind.
Die Angaben zur Zustandsneubewertung bei *moveNodes* (vgl. Tabelle 6.1) sind bei einer Schemaänderung nicht zu beachten. Nach Einbringen der strukturellen Änderungen ist somit die Anwendung von *moveNodes*(*S*, *D*, *E*, (*F*, *G*)) im Rahmen einer Schemaänderung abgeschlossen.

Instanzspezifische Änderung mittels moveNodes

Bei der Anwendung von *moveNodes*(S_{I1}, *D*, *E*, (*F*, *G*)) im Rahmen einer Instanzänderung ergibt sich ein etwas anderes Bild. Während die strukturellen Vorbedingungen exakt gleich wie bei der Schemaänderung geprüft werden können, müssen bei einer Instanzänderung zusätzlich auch Laufzeitaspekte berücksichtigt werden. Um die Verträglichkeit von *moveNodes* mit dem instanzspezifischen Schema S_I bei einer Instanzänderung zu gewährleisten, ist nach Definition 3 zusätzlich sicherzustellen, dass die instanzspezifische Markierung M^I auch nach Anwendung der Operation korrekt bleibt. Dies ist genau dann der Fall, wenn die zustandsbasierten Vorbedingungen erfüllt sind (siehe Definition 4). Aus Tabelle 6.1 ergibt sich für das betrachtete Beispiel aus Abbildung 6.1, dass sich sowohl *D* als auch *G* in einem der Zustände *ACTIVATED* oder *NOT_ACTIVATED* befinden müssen. Um dies zu prüfen, benötigt man Zugriff auf die schemaspezifische Instanzmarkierung M^I. Hierfür bietet das Datenmodell in der Schnittstelle *Node* die Funktion *getNodeState*, mit dessen Hilfe der Zustand eines Knotens abgefragt werden kann. Im betrachteten Beispiel ergibt sich für *D ACTIVATED* und für *G NOT_ACTIVATED*. Die zustandsbasierte Verträglichkeit von S_I bei Anwendung von *moveNodes*(S_I, *D*, *E*, (*F*, *G*)) ist folglich gewährleistet.
Die strukturellen Änderungen an der Instanz bzw. an dessen instanzspezifischem Schema S_I werden durch Anwenden der in Tabelle 6.1 aufgeführten Änderungsprimitiven durchgeführt. Die Änderungen

[10] Es ist zu beachten, dass bei dem „reinen“ Deltaschichtkonzept aus Abschnitt 3.5, anders wie bei der hier dargestellten Deltaschicht, die verschobenen Knoten nicht explizit gespeichert werden. Wieso diese für das Änderungsrahmenwerk notwendig sind, wird in Abschnitt 7.2.2.1.3 genauer erläutert.

werden hier jedoch nicht materialisiert, sondern die Abweichungen zu S_I ausschließlich in der Deltaschicht gespeichert. Dabei ist zu beachten, dass im Gegensatz zu einer erneuten Schemaänderung, bei einer erneuten Instanzänderung nicht eine neue Deltaschicht erzeugt, sondern die Kopie der bereits bestehenden geändert wird. Um dies zu verdeutlichen, wird in Abbildung 6.3 davon ausgegangen, dass das instanzspezifische Schema S_I bereits im Zuge einer früheren instanzspezifischen Änderung manipuliert worden ist.

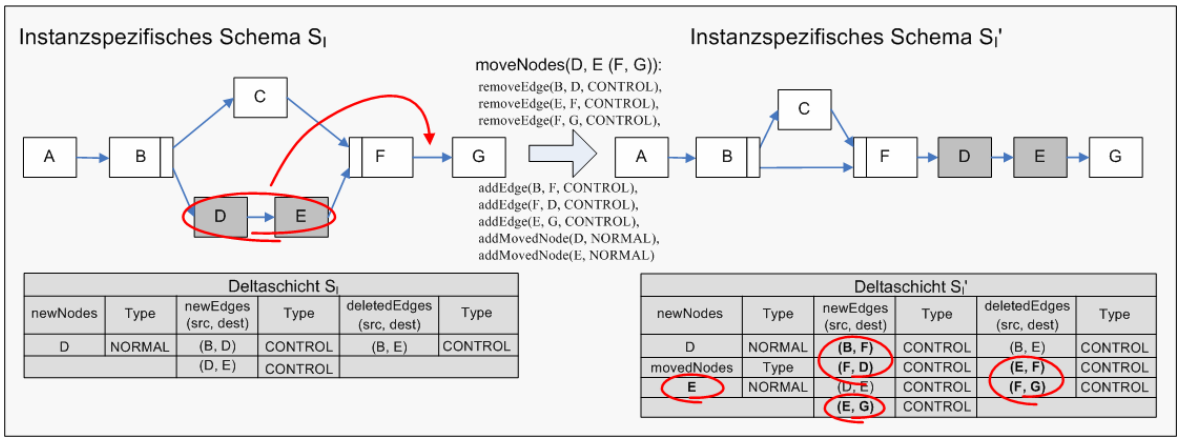
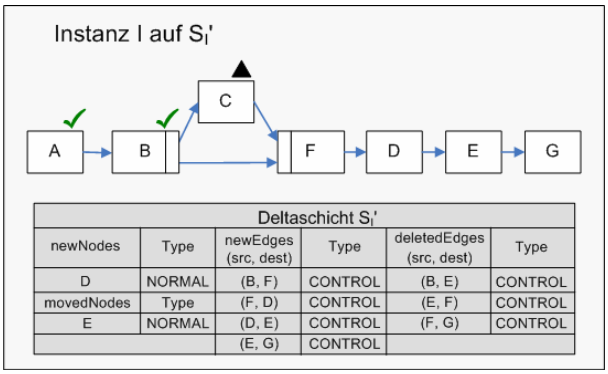


Abbildung 6.3 Anwenden von *moveNodes* auf eine bereits zuvor geänderte Instanz

Aus dieser bereits bestehenden Deltaschicht ergibt sich nach der Anwendung von *moveNodes* ein etwas anderes Bild für die resultierende Deltaschicht als es bei der Schemaänderung der Fall war (vgl. Abbildung 6.2). Dies kommt daher, dass die zu einer Instanz zugehörige Deltaschicht immer die Abweichungen von demjenigen Schema repräsentiert, das von der Instanz vor Ausführung der ersten Instanzänderung referenziert wurde. Im betrachteten Beispiel also die Abweichungen zu dem Schema ohne eingefügtem Knoten *D*. Daraus ergibt sich auch die Erklärung, wieso lediglich der Knoten *E* in *movedNodes* auftritt, obwohl bei der Anwendung von *moveNodes*(S_I, *D*, *E*, (*F*, *G*)) auch der Knoten *D* verschoben worden ist. Da nur die Auswirkungen der angewendeten Änderungen und nicht die tatsächlich angewendeten Änderungsoperationen von Interesse sind, repräsentiert die Deltaschicht S_I' einen Graphen, der gegenüber dem ursprünglichen Schema durch Verschieben des Knotens *E* und Einfügen des Knotens *D* resultiert. Die Deltaschicht enthält also exakt die Abweichungen von S_I' zu dem Schema auf dem *I* ursprünglich erzeugt worden ist. Dass es allerdings zu der gewünschten Darstellung kommt, liegt in der Verantwortung der Änderungsprimitiven. So ist es beispielsweise notwendig, dass in der Primitive *addMovedNode* geprüft wird, ob der verschobene Knoten nicht bereits bei einer früheren Änderung eingefügt worden ist. Sollte dies der Fall sein, wie hier bei Knoten *D*, so wird dieser nicht in *movedNodes* eingefügt, sondern verbleibt in *newNodes*. Ähnliche Anweisungen existieren für alle Änderungsprimitiven (vgl. Anhang C (Bereinigen der Deltaschicht)).
Konflikte durch Sync-Kanten oder fehlende Datenversorgung können im betrachteten Beispiel nicht auftreten, weshalb die Nachbedingungen von *moveNodes* automatisch erfüllt sind.
Das Anwenden von *moveNodes*(S_I, *D*, *E*, (*F*, *G*)) ist mit dem Prüfen der Nachbedingungen aber noch nicht abgeschlossen. Wie in Abschnitt 3.3 beschrieben, müssen unter Umständen im Anschluss an die strukturellen Änderungen die Knotenzustände angepasst werden. Welche Knoten davon betroffen sind, ergibt sich aus den bei jeder Änderungsoperation hinterlegten Kriterien zur Zustandsneubewertung. Wie sich aus Tabelle 6.1 entnehmen lässt, müssen im zugrunde liegenden Beispiel die Knoten *G*, *B, D* und *F* betrachtet werden. *G* befindet sich nicht im Zustand *ACTIVATED*, weshalb daraus keine Zustandsneubewertungen resultieren. *B* befindet sich im Zustand *Completed*, weshalb der Knoten *F* neu bewertet werden muss. Es ergibt sich allerdings keine Änderung, da sich *C*

nicht im Zustand *COMPLETED* befindet. *D* besitzt den Zustand *ACTIVATED*. Folglich gilt es noch den Zustand von Knoten *F* zu untersuchen. Da sich dieser im Zustand *NOT_ACTIVATED* befindet, darf dessen Nachfolger nicht den Zustand *ACTIVATED* besitzen. Dies ist aber der Fall. *D* muss also mit *NOT_ACTIVATED* markiert werden, um zu einer korrekten Instanzmarkierung M^I zu gelangen. Die dafür notwendige Methode *setNodeState* findet sich wiederum im Datenmodell.
Mit der Zustandsneubewertung ist die Anwendung von *moveNodes*(S_I, *D*, *E*, (*F*, *G*)) im Rahmen einer Instanzänderung beendet. Es resultiert die Instanz aus Abbildung 6.4.



Deltaschicht S_I'

newNodes	Type	newEdges (src, dest)	Type	deletedEdges (src, dest)	Type
D	NORMAL	(B, F)	CONTROL	(B, E)	CONTROL
movedNodes	Type	(F, D)	CONTROL	(E, F)	CONTROL
E	NORMAL	(D, E)	CONTROL	(F, G)	CONTROL
		(E, G)	CONTROL		

Abbildung 6.4 Resultierendes Schema S_I' mit Instanzmarkierung

6.2 Plug-In-Fähigkeit

Bisher wurde vereinfachend davon ausgegangen, dass die für das Änderungsrahmenwerk definierten Änderungsoperationen im System vorhanden sind. In welcher Form diese allerdings gespeichert werden, auf welche Weise sie ihre Funktionalität bereitstellen und wie das Änderungsrahmenwerk um neue Änderungsoperationen erweitert werden kann, wurde nicht beschrieben. Damit dies im Folgenden näher erläutert werden kann, wird die bisher verwendete abstrakte Sichtweise auf die Änderungsoperationen aufgegeben und stattdessen eine implementierungsnähere Beschreibung gewählt. Der Grund liegt in den unterschiedlichen Paradigmen der einzelnen Programmiersprachen, die in diesem Fall eine allgemein gültige Beschreibung verhindern. Für das Funktionieren der nachfolgend vorgestellten Konzepte wird eine objektorientierte Implementierung vorausgesetzt.

Wie die Beschreibungen im vorherigen Abschnitt zeigen, sind die notwendigen Mechanismen bei der Anwendung einer Änderungsoperation abhängig von der Art der durchzuführenden Änderung. So ist es im Rahmen einer Schemaänderung weder notwendig die zustandsbasierten Vorbedingungen zu testen noch eine Zustandsneubewertung durchzuführen. Andererseits werden sowohl bei der Schema- als auch bei der Instanzänderung die Mechanismen zur Durchführung der strukturellen Änderung benötigt. Für den Aufbau einer Änderungsoperation bedeutet dies, dass die Mechanismen zum Testen der strukturellen und zustandsbasierten Vorbedingungen, der Anwendung der verwendeten Änderungsprimitiven und der Zustandsneubewertung jeweils separat aufrufbar sein müssen. Unter Berücksichtigung der Forderung nach einer Plug-In-Schnittstelle für Änderungsoperationen, bietet sich hierfür an, eine Änderungsoperation als Klasse im objektorientierten Sinn zu implementieren und die notwendige Funktionalität in Form von Methoden bereitzustellen. Die Plug-In-Schnittstelle gibt dabei vor, welche Methoden von den Änderungsoperationsklassen implementiert werden müssen. Durch diese einheitliche Schnittstelle erreicht man die für eine Erweiterbarkeit notwendige Transparenz von der tatsächlichen Implementierung einer Änderungsoperation. Implementiert eine neu entwickelte Änderungsoperation die Methoden der Plug-In-Schnittstelle, so kann diese auch nachträglich in das Änderungsrahmenwerk eingebracht werden.

Mit der bisherigen Information ergeben sich für die Plug-In-Schnittstelle die folgenden von jeder Änderungsoperation zu implementierenden Methoden (vgl. hierzu auch Tabelle 6.1):

checkStructuralCompliance($S_{(I)}$, *inParams*[]):
Prüft, ob mit den in *inParams*[] enthaltenden Graphelementen, die bei einer Änderungsoperation definierten strukturellen Vorbedingungen erfüllt werden können.

checkStateCompliance(S_I, *inParams*[]):
Prüft, ob mit den in *inParams*[] enthaltenden Graphelementen, die bei einer Änderungsoperation definierten zustandsbasierten Vorbedingungen erfüllt werden können.

performChange($S_{(I)}$, *inParams*[]):
Führt die strukturelle Änderung mit Hilfe der Änderungsprimitiven durch.

reEvaluateNodeStates(S_I, *inParams*[]):
Passt die Knotenzustände an.

Die übergebenen Parameter S bzw. S_I bezeichnen hierbei das zu ändernde Schema bzw. instanzspezifische Schema. In *inParams*[] befinden sich die für eine Änderung notwendigen Graphelemente. Bei *insertNode* beispielsweise der neu einzufügende Knoten und die Knoten zwischen denen dieser eingefügt werden soll.

Insgesamt lässt sich die Plug-In-Funktionalität des Änderungsrahmenwerks schematisch folgendermaßen darstellen. Dabei ist zu beachten, dass der Name „Plug-In“ hier lediglich aus Verständnisgründen verwendet wird, in einer tatsächlichen Implementierung aber der Name *ChangeOperation* besser geeignet ist.

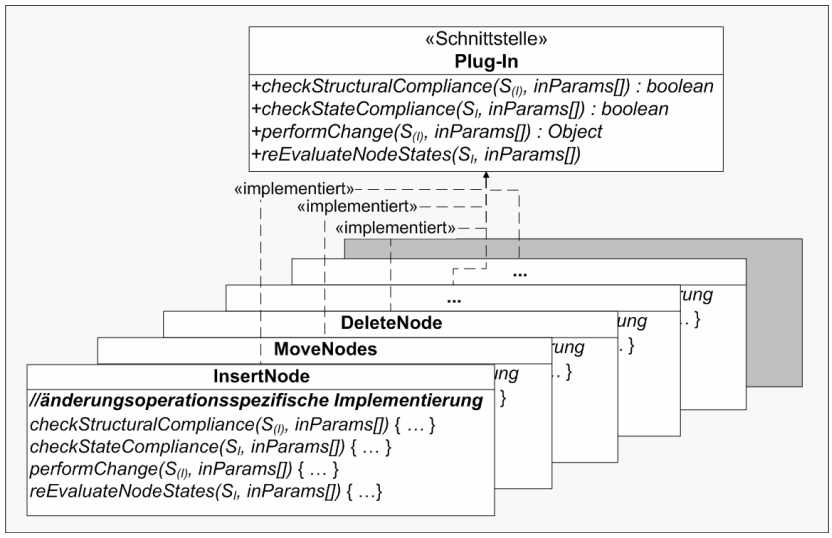


Abbildung 6.5 Plug-In-Schnittstelle des Änderungsrahmenwerkes

Die einheitliche Schnittstelle und die im Voraus nicht definierte Anzahl an Änderungsoperationen erfordern eine gesonderte Verwaltung der im Änderungsrahmenwerk verfügbaren Operationen. Dies übernimmt der in Abschnitt 5.2 vorgestellte *ChangeOperationsManager*. Er enthält eine geeignete Datenstruktur (z.B. Array oder Liste), in der von jeder im Änderungsrahmenwerk verfügbaren Änderungsoperationsklasse eine Objektinstanz gespeichert wird. Komponenten, die eine

Änderungsoperation verwenden wollen, greifen auf diese Datenstruktur zu, und erhalten durch die dort gespeicherten Objektinstanzen Zugriff auf die Methoden der gewünschten Änderungsoperation.

Die Änderungsoperationsklassen selbst, werden in einem festgelegten Verzeichnis gespeichert. Wird das Änderungsrahmenwerk gestartet, so liest der *ChangeOperationsManager* mit Hilfe einer *loadChangeOperations*-Methode die Klassen ein und erzeugt von jeder eine Objektinstanz. Diese Instanzen speichert er in der von ihm verwalteten Datenstruktur. Soll das Änderungsrahmenwerk nun zur Laufzeit um eine neue Änderungsoperation erweitert werden, so speichert man die zugehörige Änderungsoperationsklasse in dem festgelegten Verzeichnis und ruft die Methode *loadChangeOperations* erneut auf.

6.3 UNDO-Funktionalität

Die Implementierung der Änderungsoperationen in Form von Klassen ermöglicht nicht nur die einfache Erweiterbarkeit, sondern erleichtert auch die Implementierung der in Abschnitt 4.2.1 geforderten *UNDO*-Funktionalität. Dabei ist mit *UNDO* nicht das Rückgängigmachen der gesamten bis zu diesem Zeitpunkt durchgeführten Änderungen gemeint, sondern eine Funktion, mit der die Auswirkungen der jeweils zuletzt angewandten Änderungsoperation zurückgenommen werden können. Entscheidend für die Durchführung eines *UNDO* ist die Erzeugung und Verwaltung der Ausgangsdaten. Wie das Änderungsrahmenwerk diese Aufgabe erledigt, beschreibt Abschnitt 6.3.1. Die Durchführung des eigentlichen *UNDO* wird in Abschnitt 6.3.2 erläutert.

6.3.1 Erzeugung und Verwaltung der Ausgangsdaten

Die für *UNDO* notwendigen Daten ergeben sich direkt aus der Anwendung der Änderungsoperationen. Dabei ist für die *UNDO*-Funktion die Struktur des (instanzspezifischen) Schemas $S_{(I)}$ vor der Durchführung der strukturellen Änderungen relevant. Beim Aufruf der Methode *performChange* wird deshalb noch vor der Durchführung der strukturellen Änderungen eine Kopie des zu ändernden (instanzspezifischen) Schemas $S_{(I)}$ erzeugt. Diese Kopie repräsentiert folglich auch nach der Durchführung der strukturellen Änderungen die Struktur von $S_{(I)}$ vor der Anwendung der Änderungsoperation. Im Speziellen ist der Inhalt dieser Kopie abhängig von der Art der durchgeführten Änderung. Kommt die Änderungsoperation im Zuge einer Schemaänderung zum Einsatz, so enthält die Kopie sowohl das Schema selbst als auch die zugehörige Deltaschicht. Handelt es sich um eine instanzspezifische Änderung, so wird lediglich die Deltaschicht in der Kopie gespeichert.
Prinzipiell wäre es für eine *UNDO*-Funktion bereits ausreichend, die bei der Anwendung einer Änderungsoperation erzeugten Kopien in einer *Stack*-ähnlichen Weise zu speichern. Dies hat allerdings den Nachteil, dass allein anhand der in den Kopien enthaltenen Informationen nicht ersichtlich ist, welche Änderungsoperation in einem konkreten Fall angewendet worden ist. Um an diese Meta-Information zu gelangen, wird die Eigenschaft genutzt, dass die Änderungsoperationen in Form von Klassen implementiert werden. Jede Objektinstanz einer solchen Änderungsoperationsklasse enthält folglich die für die *UNDO*-Funktionalität zusätzlich gewünschten Meta-Informationen, wie beispielsweise den Namen einer Änderungsoperation. Die Änderungsoperationsklassen werden deshalb um ein Attribut erweitert, in dem $S_{(I)}$ gespeichert wird. Beim Aufruf der Methode *performChange*, wird nun zusätzlich zur Kopie des zu ändernden (instanzspezifischen) Schemas eine neue Objektinstanz der gerade ausgeführten Änderungsoperation erzeugt. Speichert man nun die Kopie in der erzeugten Änderungsobjektinstanz, so enthält diese alle für eine *UNDO*-Funktion notwendigen Informationen. Die Objektinstanzen selbst werden wiederum in

einer für die *UNDO*-Funktionalität geeigneten Struktur im *ModificationManager* gespeichert. Hierfür stellen viele Programmiersprachen, wie z.B. Java bereits vorgefertigte *UNDO*-Manager bereit, in denen die Objektinstanzen auf *Stack*-ähnliche Weise verwaltet werden. Zusammenfassend zeigt Abbildung 6.6 noch einmal in graphischer Form, wie die für ein *UNDO* notwendige Information erzeugt und verwaltet wird.

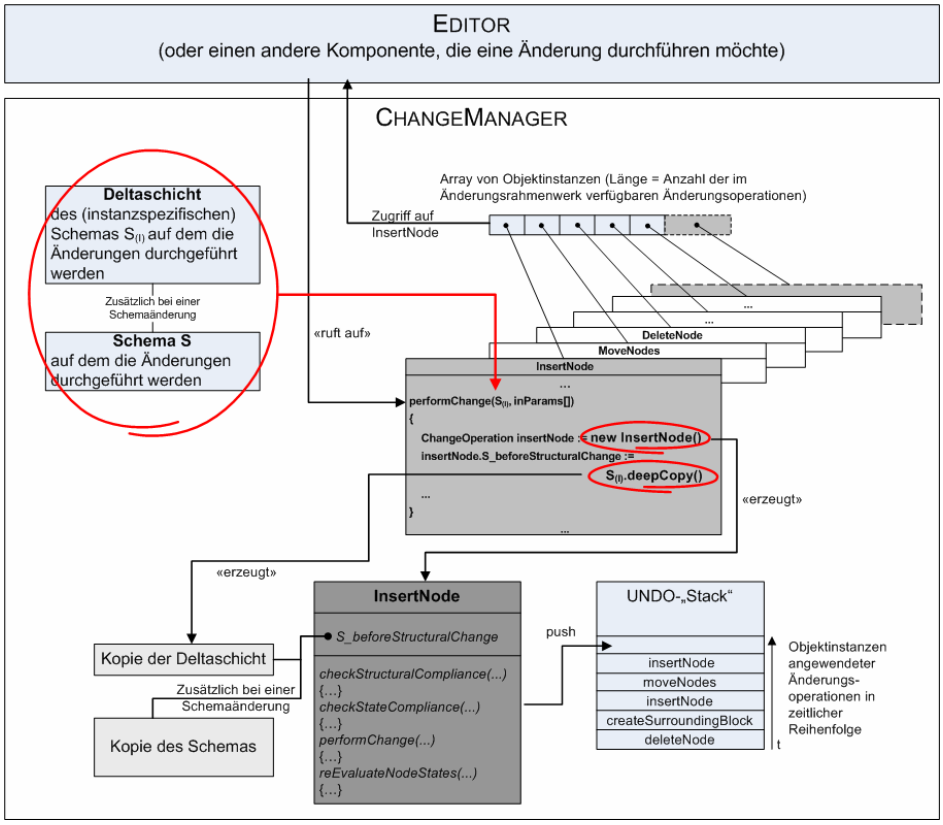


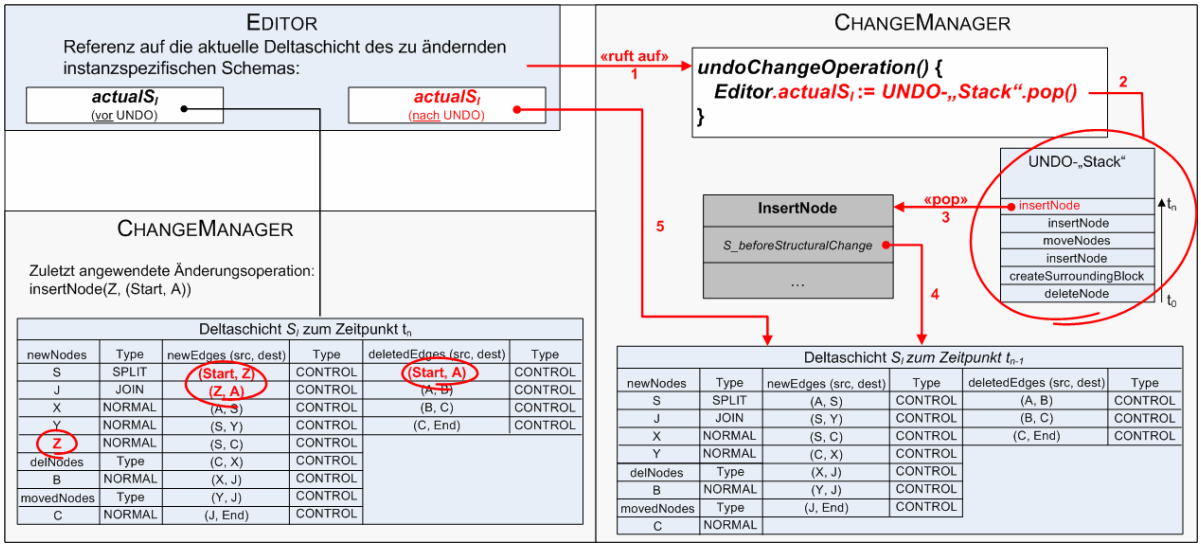
Abbildung 6.6 Erzeugen und Verwalten der für ein *UNDO* notwendigen Informationen

6.3.2 Ausführen der *UNDO*-Funktion

Für die Durchführung eines *UNDO* bietet der *ModificationManager* eine *UndoChangeOperation*-Methode an. Greift eine Komponente auf diese Methode zu, so wird die oberste Änderungsobjektinstanz vom *Stack* geholt. Diese Objektinstanz enthält das (instanzspezifische) Schema vor der Durchführung der von dieser Instanz repräsentierten Änderungsoperation. Um die Auswirkungen dieser Operation rückgängig zu machen, genügt es nun, dass zu bearbeitende (instanzspezifische) Schema durch das in der Objektinstanz gespeicherte (instanzspezifische) Schema zu ersetzen und die weiteren Änderungen darauf auszuführen.

Abbildung 6.7 zeigt anhand eines Beispiels den Ablauf bei der Durchführung eines *UNDO*:
In der linken Hälfte der Abbildung sieht man die Ausgangssituation. Es handelt sich in diesem Fall um eine instanzspezifische Änderung, bei der lediglich die Deltaschicht zu berücksichtigen ist. Wie Abbildung 6.7 zeigt, referenziert die Komponente, die die Funktionalität des Änderungsrahmenwerkes in Anspruch nimmt (hier der *Editor*), die Deltaschicht des instanzspezifischen Schemas, auf dem die Änderungen durchgeführt werden.
Im betrachteten Beispiel wurde zuletzt die Operation *insertNode*(*Z*, (*Start*, *A*)) angewendet. Folglich ist der oberste Eintrag im *Stack* eine Objektinstanz vom Typ *InsertNode* (siehe *UNDO*-Stack). Die

Auswirkungen der Änderungsoperation sind in der aktuellen Deltaschicht direkt ersichtlich (eingekreiste Einträge).
Um die Auswirkungen von *insertNode*(*Z*, (*Start*, *A*)) rückgängig zu machen, wird die Methode *undoChangeOperation* aufgerufen. Dadurch wird der oberste Eintrag vom *UNDO*-Stack geholt. Es handelt sich dabei genau um die Objektinstanz vom Typ *InsertNode*, die beim Anwenden von *insertNode*(*Z*, (*Start*, *A*)) erzeugt worden ist. Diese wiederum enthält die Deltaschicht in der Form, wie sie vor der Anwendung der Änderungsoperation bestand. Um das *UNDO* abzuschließen wird im *Editor* die Referenz in $actualS_l$ gelöscht und durch eine Referenz auf diese Deltaschicht ersetzt.
Der Ablauf der *UNDO*-Funktion im Zuge einer Schemaänderung gestaltet sich analog zum beschriebenen Beispiel. Der Unterschied besteht lediglich darin, dass bei einer Schemaänderung zusätzlich zur Deltaschicht auch das Schema selbst gespeichert wird. Folglich wird beim *UNDO* nicht nur die Referenz auf die Deltaschicht, sondern auch eine entsprechende Referenz auf das Schema ersetzt.



newNodes	Type	newEdges (src, dest)	Type	deletedEdges (src, dest)	Type
S	SPLIT	(Start, Z)	CONTROL	(Start, A)	CONTROL
J	JOIN	(Z, A)	CONTROL	(A, B)	CONTROL
X	NORMAL	(A, S)	CONTROL	(B, C)	CONTROL
Y	NORMAL	(S, Y)	CONTROL	(C, End)	CONTROL
Z	NORMAL	(S, C)	CONTROL		
delNodes	Type	(C, X)	CONTROL		
B	NORMAL	(X, J)	CONTROL		
movedNodes	Type	(Y, J)	CONTROL		
C	NORMAL	(J, End)	CONTROL		

newNodes	Type	newEdges (src, dest)	Type	deletedEdges (src, dest)	Type
S	SPLIT	(A, S)	CONTROL	(A, B)	CONTROL
J	JOIN	(S, Y)	CONTROL	(B, C)	CONTROL
X	NORMAL	(S, C)	CONTROL	(C, End)	CONTROL
Y	NORMAL	(C, X)	CONTROL		
delNodes	Type	(X, J)	CONTROL		
B	NORMAL	(Y, J)	CONTROL		
movedNodes	Type	(J, End)	CONTROL		
C	NORMAL				

Abbildung 6.7 Ausführen der *UNDO*-Funktion

6.4 Testen der Aufrufbarkeit einer Änderungsoperation im Vorfeld

Neben der Plug-In-Fähigkeit und der *UNDO*-Funktion wird in Abschnitt 4.2 gefordert, dass mit dem Änderungsrahmenwerk die Aufrufbarkeit einer Änderungsoperation im Vorfeld getestet werden kann. Der Grund für diese Forderung ergibt sich aus der Vorgehensweise die ein Anwender beim Modellieren eines Graphen verfolgt. Möchte dieser beispielsweise im ADEPT2-*Editor* eine Änderung durchführen, so markiert er durch Anklicken einzelner Graphelemente den zu ändernden Graphbereich (*Selection*). Ohne weitere Hilfsmittel würde er in einem nächsten Schritt die von ihm gewünschte Änderungsoperation aufrufen. Das Änderungsrahmenwerk prüft daraufhin die Verträglichkeit der ausgewählten Elemente und weist die angewendete Änderungsoperation bei Unverträglichkeit zurück. Es liegt bei dieser Vorgehensweise also in der Hand des Anwenders, in einer gegebenen Situation zu entscheiden, welche Änderungsoperation überhaupt erfolgversprechend aufgerufen werden kann. Die Anforderung lautet aber, den Anwender von dieser Aufgabe zu entlasten. Es ist also ein Konzept zu entwickeln, bei dem der Anwender nur diejenigen Änderungsoperationen auswählen kann, die im momentanen Kontext und mit den in der *Selection* enthaltenen Elementen sinnvoll aufrufbar sind. Abbildung 6.8 zeigt die geforderte Funktionalität:

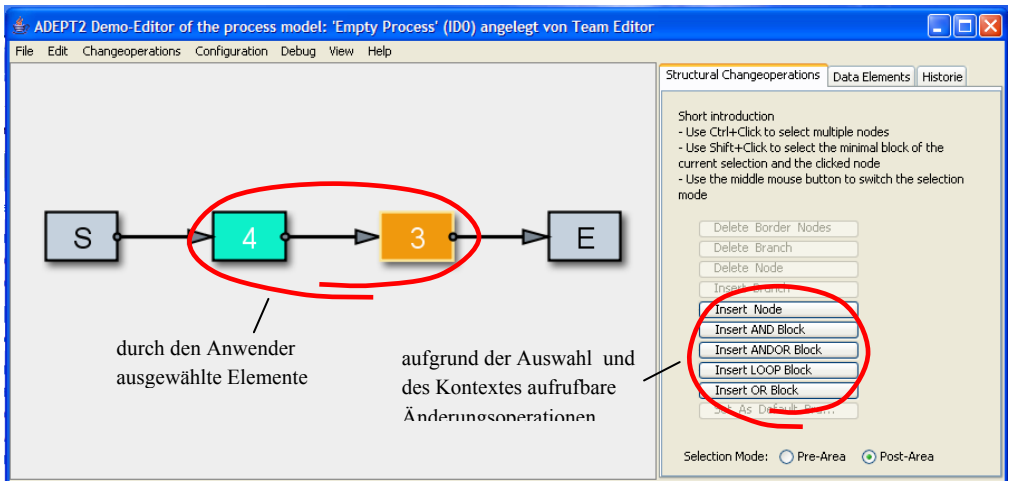


Abbildung 6.8 Kontextabhängiges Freischalten von Änderungsoperationen

Für eine Umsetzung im Änderungsrahmenwerk bedeutet diese Funktionalität, dass es möglich sein muss, für jede Änderungsoperation bereits im Vorfeld zu entscheiden, ob diese erfolgversprechend aufgerufen werden kann.
Wie dies im Einzelnen durchgeführt wird und welche Auswirkungen sich für die Änderungsschnittstelle ergeben, wird im Folgenden beschrieben. Grundlegend ist dabei die Unterscheidung zwischen Aufrufbarkeit und Ausführbarkeit. Dies wird in Abschnitt 6.4.1 genauer erläutert. Die Auswirkungen auf die Schnittstelle der Änderungsoperationen beschreibt Abschnitt 6.4.2 und die internen Abläufe bei der Prüfung der Aufrufbarkeit werden in Abschnitt 6.4.3 behandelt. Da ein Testen der Aufrufbarkeit im engen Zusammenhang mit der Benutzeroberfläche steht, wird bei den nachfolgend beschriebenen Mechanismen auch die Forderung nach Unterstützung des GUI-Programmierers berücksichtigt (*Buttons*/*Icons*, siehe rechte Seite der Abbildung).

6.4.1 Aufrufbarkeit vs. Ausführbarkeit

Die Unterscheidung zwischen Aufrufbarkeit und Ausführbarkeit ist deshalb so entscheidend, weil speziell bei Verschiebeoperationen die Aufrufbarkeit einer Änderungsoperation nicht mit der Ausführbarkeit übereinstimmt. So ist beispielsweise die Operation *moveNodes* aufrufbar, sobald vom Anwender diejenigen Knoten angewählt und vom Rahmenwerk geprüft worden sind, die den zu verschiebenden Block eingrenzen. Um die Ausführbarkeit bzw. Verträglichkeit der Operation *moveNodes* bestimmen zu können, werden aber auch die Knoten der Zielstelle benötigt. Diese können aber nicht gleichzeitig mit den Knoten des zu verschiebenden Blockes gewählt werden, da sonst nicht ersichtlich ist, welche Knoten den Block begrenzen und welche die Zielstelle markieren.
Wie das Beispiel *moveNodes* zeigt, ist es bei Verwendung eines Mechanismus zur Prüfung der Aufrufbarkeit nicht möglich, die Verträglichkeit und somit die Ausführbarkeit einer Änderungsoperation in einem Schritt zu prüfen. Es ist deshalb notwendig, die bei jeder Änderungsoperation hinterlegten Vorbedingungen in zwei Teile zu untergliedern. Im einen Teil wird getestet, ob eine Änderungsoperation im angegebenen Kontext zulässig ist. Im anderen Teil werden die Parameter geprüft, die bei der Ausführung der Änderungsoperation zusätzlich zu den bereits getesteten Elementen notwendig sind. Um zu überprüfen welche Änderungsoperationen für die vom Anwender markierten Graphelemente (*Selection*) zulässig sind und abhängig davon entsprechende *Buttons* freizuschalten, genügt es den ersten Teil zu betrachten. Erst wenn der Anwender eine der zulässigen Operationen über die freigeschalteten *Buttons* aufruft, ist es je nach Änderungsoperation notwendig, weitere Parameter nachzufordern und zu prüfen. Bei dem Test des ersten Teils, handelt es

sich also um den Aufrufbarkeitstest, während die Tests des ersten und des zweiten Teils zusammengenommen einem Verträglichkeitstest entsprechen.
Tatsächlich unterscheiden sich bei einigen Änderungsoperationen wie *deleteNode* oder *insertSyncEdge* die Aufrufbarkeitstest nicht von den Verträglichkeitstests. So genügt bei *deleteNode* bereits die Auswahl eines einzigen Knotens um die Verträglichkeit und somit die Ausführbarkeit der Operation zu bestimmen. Eine Unterteilung der Vorbedingungen ist in einem solchen zwar Fall nicht notwendig, wirkt sich aber auf die Schnittstelle der Änderungsoperationen aus, wie der nächste Abschnitt zeigt.

6.4.2 Auswirkungen auf die Änderungsschnittstelle

Wie im vorigen Abschnitt beschrieben, hat ein Mechanismus zur Berechnung der aufrufbaren Änderungsoperationen als Eingabe lediglich diejenigen Graphelemente zur Verfügung, die vom Graphmodellierer ausgewählt wurden (*Selection*). Da diese Elemente nicht in jedem Fall für das Testen der Verträglichkeit ausreichen, sind die in Abschnitt 6.2 vorgestellten Methoden *checkStructuralCompliance* und *checkStateCompliance* in ihrer jetzigen Form für einen Aufrufbarkeitstest ungeeignet. Beide Operationen benötigen immer alle für einen Verträglichkeitstest notwendigen Elemente. In Fällen wie *insertNode*(*node*, (*pred*, *succ*)) oder dem bereits angesprochenen *moveNodes(first*, *last*, (*pred*, *succ*)) ist es allerdings unmöglich, aus einer *Selection* alle notwendigen Elemente zu bekommen.

Um dieses Problem zu beheben, ist es nun die einfachste Lösung, diejenigen Änderungsoperationen, bei denen sich die Verträglichkeitstests von den Aufrufbarkeitstests unterscheiden, um eine Methode *isCallable* zu erweitern. In dieser Methode wird dann der Teil der Verträglichkeitstests durchgeführt, der für die Bestimmung der Aufrufbarkeit notwendig ist. Dafür wären die in einer *Selection* enthaltenen Graphelemente ausreichend.
Im Zusammenspiel mit dem Plug-In-Interface lässt sich eine solche Lösung aber nicht verwirklichen. Die für jede Änderungsoperation einheitliche Schnittstelle erfordert, dass auch diejenigen Änderungsoperationen, die ohne die Methode *isCallable* auskommen, diese trotzdem implementieren müssen. Als Folge hat die *isCallable*-Methode für viele Änderungsoperationen eine reine *Dummy*-Funktion. Dies ist nicht wünschenswert, da auf diese Weise das Programmieren neuer Änderungsoperationen unnötig erschwert wird. Folglich steht eine solche Vorgehensweise in direktem Konflikt mit der nicht-funktionalen Anforderung den Anwendungsentwickler/-programmierer zu unterstützen. Für die Umsetzung der Aufrufbarkeit ist eine Erweiterung der Änderungsoperationen um neue Methoden folglich nicht geeignet ist.

Da sich die Änderungsoperationen nicht um neue Methoden erweitern lassen, müssen die bestehenden Operationen angepasst werden. Dazu wird die Parameterliste der Änderungsoperationen *checkStructuralCompliance*($S_{(I)}$, *inParams*[][11]) und *checkStateCompliance*(S_I, *inParams*[]) um ein sogenanntes *Check-Flag* erweitert. Soll nun die Aufrufbarkeit einer Änderungsoperation getestet werden, so setzt die aufrufende Komponente dieses *Flag* auf „1“. Gilt es hingegen die Verträglichkeit zu prüfen, so ist das *Check-Flag* auf „0“ zu setzen. Änderungsoperationen, bei denen sich die Aufrufbarkeit von der Verträglichkeit unterscheidet, werten dieses *Flag* aus und führen abhängig davon die entsprechenden Tests durch. Bei den anderen Änderungsoperationen muss der Wert des *Flags* nicht berücksichtigt werden. Obwohl es sich bei letzterem Fall streng genommen ebenfalls um

[11] Der Parameter *inParams*[] entspricht hierbei der aus den im Editor ausgewählten Elementen bestehenden *Selection*.

einen *Dummy* handelt, ist diese Lösung der zuerst beschriebenen bzgl. Anwenderfreundlichkeit überlegen. So muss sich der Anwendungsprogrammierer bei Verwendung des *Check-Flags* nicht mit der Implementierung einer Funktion beschäftigen, die aus seiner Sicht unnötig ist.

6.4.3 Interner Ablauf

Mit den bis jetzt beschriebenen Mechanismen ergibt sich für das Freischalten der in einem bestimmten Kontext und unter Auswahl einer beliebigen Anzahl an Graphelementen (*Selection*) zulässigen Änderungsoperationen der folgende interne Ablauf:

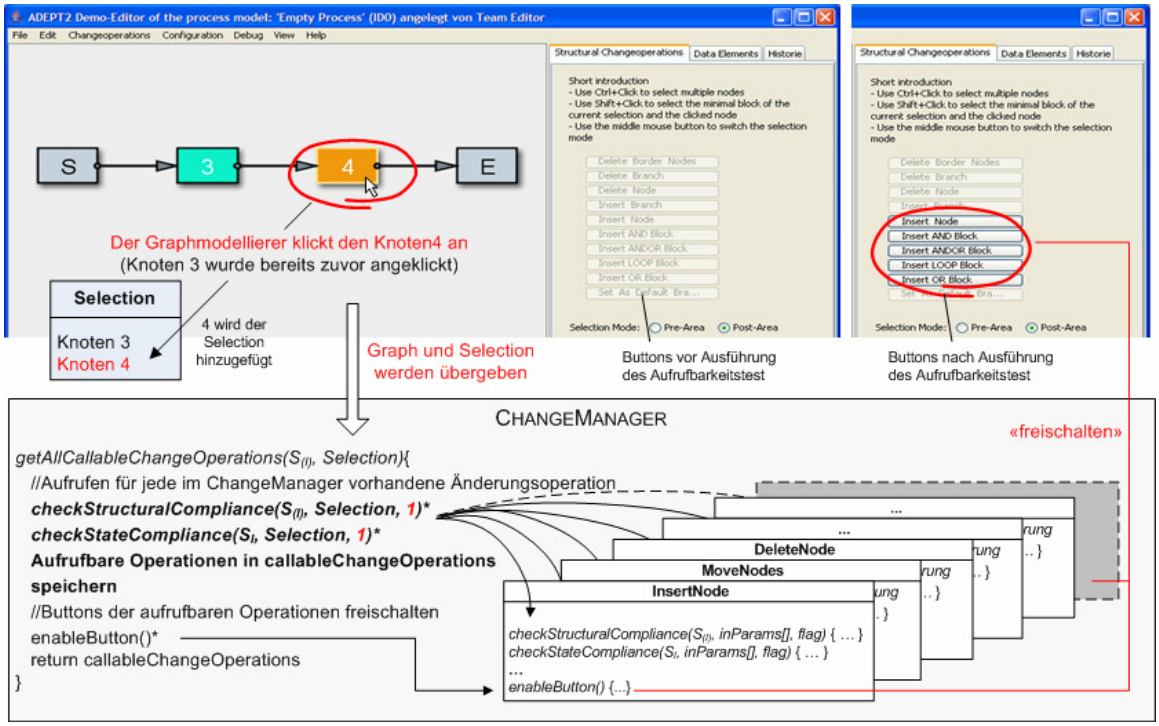


Abbildung 6.9 Interner Ablauf beim Freischalten von Änderungsoperationen

Wie Abbildung 6.9 zeigt, bewirkt das Anklicken eines Graphelements, dass dieses in die *Selection* aufgenommen wird. Gleichzeitig löst dieses Ereignis einen Aufruf an eine *getAllCallableChangeOperations*-Methode im Änderungsrahmenwerk aus. Dieser Methode werden als Parameter der aktuelle Graph (Schema S oder instanzspezifisches Schema S_I) und die Menge der aktuell ausgewählten Elemente übergeben. Die Methode selbst, ruft für jede im Änderungsrahmenwerk vorhandene Änderungsoperation die Methoden *checkStructuralCompliance* und *checkStateCompliance* auf. Letztere allerdings nur, wenn es sich beim aktuellen Graph um ein instanzspezifisches Schema S_I handelt.

Wie man anhand der Parameterliste erkennt, geschieht der Aufruf mit dem Wert „1“ im *Check-Flag*. Dies signalisiert den Methoden, dass es sich um einen Aufrufbarkeitstest handelt, worauf diese entsprechend reagieren und nur denjenigen Teil des Verträglichkeitstests durchführen, der für die Aufrufbarkeit notwendig ist. Absolviert eine Änderungsoperation die Aufrufbarkeitstest mit positivem Ergebnis, so wird sie in der Menge der aufrufbaren Operationen (*callableChangeOperations*) gespeichert.

Der nächste Schritt *enableButton()*, betrifft die nicht-funktionale Anforderung, dass das Änderungsrahmenwerk auch *Buttons*/*Icons* bereitstellen soll, die direkt von GUI-Programmierern verwendet werden können. Es bietet sich hier an, dass jede Änderungsoperation ihren eigenen *Button* inklusive damit verbundener Ereignisverarbeitung und entsprechender *Tooltips* zur Verfügung stellt.

Diese können dann wie in Abbildung 6.9 direkt im GUI eingebaut werden. Für den Ablauf ergibt sich, dass die *Buttons* genau dann freigeschaltet werden, wenn der Aufrufbarkeitstest erfolgreich ausgeführt wurde. Die Änderungsoperationen bieten hierfür die Methode *enableButton()* an.
Damit ist der Aufrufbarkeitstest beendet. Es können nun exakt diejenigen Änderungsoperationen ausgeführt werden, deren *Button* freigeschaltet wurde.
Für die Verwendung über ein GUI ist das *Button*-Konzept optimal geeignet, für „reine“ API-Aufrufe hingegen nicht. Deshalb werden zusätzlich alle aufrufbaren Änderungsoperationen in einer geeigneten Datenstruktur gespeichert und an die aufrufende Komponente zurückgegeben. Der Aufrufbarkeitstest lässt sich dadurch sowohl über ein GUI als auch über API-Aufrufe in Anspruch nehmen.

6.5 Zusammenspiel der vorgestellten Konzepte

In den Abschnitten 6.1 bis 6.4 wurden, analog zu den in Kapitel 4 definierten Anforderungen, einzelne Konzepte zur Schemaerstellung und Schema- bzw. Instanzänderung vorgestellt. Die Mechanismen wurden dabei größtenteils separat betrachtet. In diesem Abschnitt wird das Zusammenspiel der Konzepte untersucht und sich daraus ergebende Problemstellungen behandelt. Um den Gesamtablauf zu verdeutlichen, zeigt Abbildung 6.10 anhand eines Anwendungsfalls die Aufrufreihenfolge und die notwendigen Schritte.

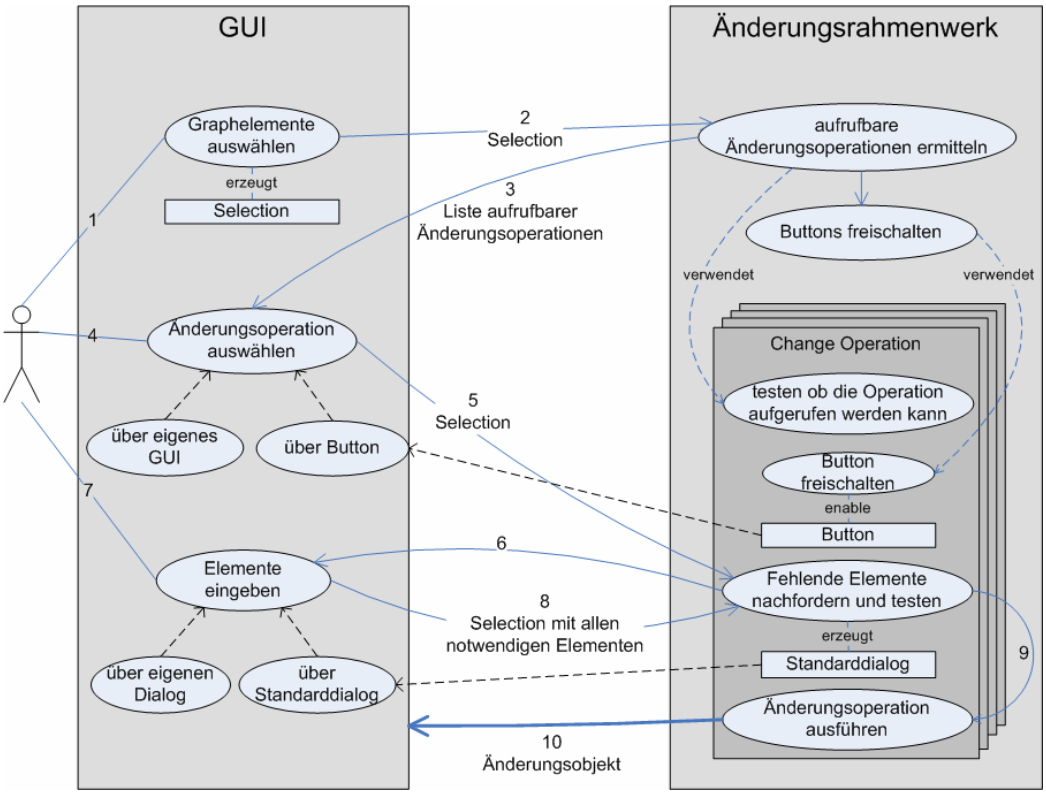


Abbildung 6.10 Notwendige Schritte bei der Änderung eines Graphen

Der Anwendungsfall beschreibt die Inanspruchnahme der Funktionalität des Änderungsrahmenwerkes über ein GUI. Bei der folgenden Ausführung wird deshalb vornehmlich auf diese Situation eingegangen. Wo sich allerdings Unterschiede zu der Inanspruchnahme der Funktionalität in Form von „reinen“ API-Aufrufen ergeben wird speziell darauf hingewiesen.
Im ersten Schritt wählt der Anwender eine Reihe von Graphelementen aus um den Änderungsbereich zu definieren. Dabei wird jedes Element beim Anklicken in einer *Selection* gespeichert. Mit dieser *Selection* wird geprüft, welche Änderungsoperationen aufrufbar sind (Schritt 2). Dann werden die

Buttons der aufrufbaren Änderungsoperationen freigeschaltet und zusätzlich eine Liste der aufrufbaren Operationen an die aufrufende Komponente (hier GUI) zurückgegeben (Schritt 3). Die entsprechenden Mechanismen und Methoden wurden in Abschnitt 6.4 vorgestellt. Im nächsten Schritt wählt der Anwender eine der freigeschalteten Änderungsoperationen durch Anwählen des entsprechenden *Buttons* aus. Dadurch wird direkt die in Abschnitt 6.2 vorgestellte Methode *performChange* aufgerufen. Dies ist natürlich auch ohne das *Button*-Konzept durchführbar, indem man eine Änderungsoperation aus der in Schritt 3 erhaltenen Liste auswählt und deren *performChange* Methode aufruft. In einigen Fällen können bzw. dürfen die in *performChange* durchzuführenden strukturellen Änderungen jedoch nicht sofort ausgeführt werden. Dies begründet sich damit, dass im bisherigen Ablauf in der *Selection* lediglich diejenigen Elemente vorhanden sind, die für die Aufrufbarkeit benötigt werden. Um nun Änderungsoperationen überhaupt durchführen zu können, bei denen sich die Anzahl der für die Aufrufbarkeit notwendigen Elemente von der Anzahl der für die Verträglichkeit notwendigen Elemente unterscheiden, müssen zuerst die noch fehlenden Elemente nachgefordert werden (Schritt 6). Bei *moveNodes* beispielsweise die Zielknoten *pred* und *succ*.
Hierfür bestehen mehrere Möglichkeiten:
Es kann direkt aus der Methode *performChange* ein Dialog aufgerufen werden, in dem der Benutzer die fehlenden Elemente auswählen kann. Hat er dies durchgeführt, so werden die Elemente sofort durch Aufruf von *checkStructural-* und ggf. *checkStateCompliance* auf ihre Verträglichkeit geprüft. Dabei gilt es allerdings redundante Tests zu vermeiden und ausschließlich die nachgeforderten Elemente zu prüfen. Um dies zu erreichen, wird für das *Check-Flag* neben „0“ und „1“ noch ein weiterer Wert „2“ eingeführt. Werden die Methoden mit „2“ aufgerufen, so signalisiert dies, dass lediglich die nachgeforderten Elemente der übergebenen *Selection* zu testen sind.
Alternativ zur Eingabe der fehlenden Elemente über einen bereitgestellten Dialog muss das Änderungsrahmenwerk aber auch die Möglichkeit bieten, dass diese Elemente auf andere Weise von dem GUI bereitgestellt werden. Bietet ein Editor – wie beispielsweise der ADEPT2-*Editor* – die Möglichkeit, Graphelemente per Mausklick auszuwählen, so ist es aus Anwendersicht am leichtesten nachvollziehbar, wenn die nachgeforderten Elemente ebenfalls auf diese Weise ausgewählt werden können. Damit dies trotz Verwendung des *Button*-Konzepts möglich ist, muss die Methode *performChange* bzw. die Änderungsoperationsschnittstelle entsprechend angepasst werden. Für *performChange* wird dazu ein weiteres *Flag* eingeführt. Dieses *Demand-Flag* kann die folgenden Werte annehmen:

„0“: Der Nachforderungsdialog soll verwendet werden.

„1“: Das GUI bietet einen eigenen Mechanismus um die nachgeforderten Elemente zu liefern.

Da aber auch bei Bereitstellung der noch fehlenden Elemente durch das GUI die *Buttons* verwendet werden sollen, müssen die bei den *Buttons* hinterlegten Ereignisbehandlungen angepasst werden, damit die unterschiedlichen *Demand-Flag*-Werte erzeugt werden können. Hierfür werden die Änderungsoperationen um die Methode *setUsage*(*boolean*) erweitert. Bei dem Wert „*true*“ wird die Methode *performChange* aus der Ereignisbehandlung eines *Buttons* mit dem *Demand-Flag*-Wert „0“ aufgerufen, bei „*false*“ mit „1“.
Für den Ablauf bei der Bereitstellung der noch fehlenden Elemente über das GUI ergibt sich der folgende Ablauf: Die Auswahl der Graphelemente, der Aufrufbarkeitstest und das Anwählen einer Änderungsoperation über einen *Button* bleiben unverändert. Die Methode *performChange* wird allerdings mit *Demand-Flag*-Wert „1“ aufgerufen. Dadurch wird nicht der Nachforderungsdialog aufgerufen, sondern die Methode abgebrochen und stattdessen ein Dialog angezeigt mit dem Hinweis, die fehlenden Elemente auszuwählen. Nach Anklicken der erforderlichen Elemente drückt der Anwender erneut den *Button* der gewünschten Änderungsoperation. Nun sind in der übergebenen *Selection* alle für die Ausführung der Änderungsoperation notwendigen Elemente vorhanden. Davon

müssen noch die neu hinzugekommenen Elemente getestet werden. Dies geschieht analog zur Durchführung mit Hilfe des Nachforderungsdialogs über den Aufruf der Methoden *checkStructural-* und ggf. *checkStateCompliance* mit dem *Check-Flag*-Wert „2“.
Die Durchführung der strukturellen Änderungen und die Rückgabe des Änderungsobjektes an das GUI (Schritte 9 und 10) zur Unterstützung der *UNDO*-Funktionalität, sind für beide Konzepte identisch. Wie die strukturellen Änderungen im Einzelnen ausgeführt werden, wurde bereits in Abschnitt 6.1 detailliert erläutert.

Der bisher vorgestellte Ablauf und die Konzepte beziehen sich alle auf die Verwendung der Änderungsoperationen mit einem GUI. Um die Funktionalität aber auch über ein „reines“ API, d.h. ohne direkte Interaktion mit dem Benutzer bereitstellen zu können, muss die Methode *performChange* erneut angepasst werden. Dies liegt daran, dass in diesem Fall gleich zu Beginn alle notwendigen Elemente vorliegen. Da diese logischerweise alle ungetestet sind, ist ein Aufruf der Methoden *checkStructural-* und ggf. *checkStateCompliance* mit dem *Check-Flag*-Wert „0“ notwendig. Damit die Änderungsoperation erkennt um welche Art von Änderung es sich handelt, wird der Parameterliste von *performChange* ein weiteres *Flag* hinzugefügt. Unter Berücksichtigung aller Aufrufmöglichkeiten ergeben sich die folgenden Werte für dieses *Call-Typ-Flag*:

„0“: Button-Konzept
„1“: Die Änderungsoperation wird in Form eines API-Aufrufs in Anspruch genommen
„2“: Die Änderungsoperation wird von einer komplexen Änderungsoperation verwendet.

Der Wert „2“ deckt dabei einen bisher noch nicht betrachteten Fall ab. Wie in Abschnitt 3.2 beschrieben wurde, können komplexe Änderungen auf einfachen Änderungen aufbauen. In einem solchen Fall werden allerdings nur die strukturellen Änderungen in Anspruch genommen. Die strukturellen und zustandsbasierten Verträglichkeitstests müssen hingegen nicht berücksichtigt werden. Die Methode *performChange* überspringt deshalb bei *Call-Typ-Flag*-Wert „2“ den Aufruf der Testmethoden und führt direkt die strukturellen Änderungen aus.

Tabelle 6.2 zeigt nochmals alle *Flags*, die möglichen Werte und deren Bedeutung im Überblick:

Wert / Flag	„0“	„1“	„2“
Check	Ausführbarkeit	Aufrufbarkeit	nachgeforderte Elemente
Demand	Nachforderungsdialog	Eigener Nachforderungsmechanismus	-
Call-Typ	Button-Konzept	API-Aufruf	Komplexe Änderungsoperation

Tabelle 6.2 *Flags* der Methode *performChange*

6.6 Zusammenfassung

In diesem Kapitel wurden die Konzepte zur Erfüllung der funktionalen und nicht-funktionalen Anforderungen im Bereich der Schemaerstellung und Schema- bzw. Instanzänderung entwickelt. Dabei wurde demonstriert wie sich die Anwendung einer Änderungsoperation auf die interne Repräsentation von Schemata und Instanzen auswirkt, wie mit Vor- und Nachbedingungen die Zulässigkeit geprüft wird und wie nach der Durchführung die Zustandsmarkierung bei Instanzen anzupassen ist. Für die nicht-funktionale Anforderung der Erweiterbarkeit wurde eine Plug-In-Konzept entwickelt mit dem neue Änderungsoperationen in das Änderungsrahmenwerk eingebracht werden können. Dazu war es erforderlich, die bis dahin verwendete abstrakte Beschreibungsebene zu verlassen und unter Verwendung eines objektorientierten Paradigmas eine konkrete Schnittstelle für

Änderungsoperationen zu definieren. Um auch die Anforderungen bezüglich Benutzerfreundlichkeit zu erfüllen, wurden eine *UNDO*-Funktion und ein Konzept zur kontextabhängigen Aufrufbarkeit einer Änderungsoperation beschrieben. Die *UNDO*-Funktion wurde dabei mit Hilfe eines *Stacks* realisiert. Auf diesem wird vor der Durchführung der strukturellen Änderung eine Kopie des ursprünglichen (instanzspezifischen) Schemas gespeichert. Um nach der Ausführung einer Änderungsoperation deren Auswirkungen rückgängig zu machen, wird das geänderte (instanzspezifische) Schema durch das auf dem Stack gespeicherte Schema ersetzt.
Mit dem Konzept zur Prüfung der kontextabhängigen Aufrufbarkeit ist es möglich, anhand einer beliebigen Menge an Graphelementen, dem Anwender diejenigen Änderungsoperationen zu berechnen, die im momentanen Kontext und mit der Anzahl an gewählten Elementen aufrufbar sind. Für die dadurch notwendige Unterscheidung zwischen Aufrufbarkeit und Ausführbarkeit wurden die Auswirkungen auf die Änderungsoperationsschnittstelle beschrieben und diese entsprechend angepasst. In diesem Zusammenhang wurde gleichzeitig durch Vorstellung des *Button*-Konzeptes der Forderung nach Unterstützung des Anwendungsentwicklers-/programmierers nachgegangen.
Um auch das Zusammenspiel der einzelnen Konzepte zu verdeutlichen wurde ein konkreter Anwendungsfall beschrieben und gezeigt, wie die vorgestellten Konzepte sowohl für eine Verwendung über ein GUI als auch für reine API-Aufrufe optimiert werden können.

Insgesamt erhält man bei einer Implementierung der vorgestellten Konzepte ein mächtiges Änderungsrahmenwerk in Bezug auf die Erstellung und Änderung von Prozessvorlagen, sowie der Manipulation von Instanzen. Dies allein ist aber noch nicht ausreichend um alle an das Änderungsrahmenwerk gestellten funktionalen Anforderungen zu erfüllen. Um das Rahmenwerk zu vervollständigen, wird im nächsten Kapitel detailliert beschrieben, wie die Konzepte zur Schemaevolution umgesetzt werden.

7 Umsetzung der Schemaevolution

Die Ausgangsbasis für die in diesem Kapitel beschriebenen Konzepte und Algorithmen zur Schemaevolution bilden die für jede Instanzklasse definierten Verträglichkeitskriterien und Migrationsstrategien aus den Abschnitten 3.4.3 und 3.4.4. Bei deren Umsetzung in diesem Kapitel werden sowohl implementierungsnahe Konzepte wie die Deltaschicht als auch Wechselwirkungen mit den im vorherigen Kapitel beschriebenen Konzepten zur Umsetzung der nicht-funktionalen Anforderungen berücksichtigt. Hierbei wird im Speziellen untersucht, ob die Auswirkungen der Erweiterbarkeit des Änderungsrahmenwerkes ein Umsetzen der auf konzeptioneller Ebene entwickelten Algorithmen überhaupt zulässt. In den Fällen wo dies nicht möglich ist, werden die Algorithmen komplett neu entwickelt und die zugrundeliegenden Konzepte erläutert.

Im Detail ist das Kapitel folgendermaßen aufgebaut:
Nach der Vorstellung des Gesamtablaufs einer Schemaevolution (Abschnitt 7.1), werden in 7.2 Algorithmen entwickelt mit denen die Klassenzugehörigkeit einer Instanz bestimmt werden kann. Aufbauend auf dieser Einteilung wird in den Abschnitten 7.3 - 7.7 beschrieben, wie die für die Migration der Instanzen notwendigen Konzepte umgesetzt werden. Die Gliederung erfolgt dabei nicht klassenweise, sondern nach den Mechanismen die insgesamt für die Durchführung notwendig sind. Dies liegt daran, dass die Verträglichkeitstests bzw. Migrationsstrategien unterschiedlicher Klassen in vielen Fällen Gemeinsamkeiten aufweisen und somit einzelne Algorithmen bei mehreren Klassen verwendet werden können. Im Detail befasst sich der Abschnitt 7.3 mit den Algorithmen zur strukturellen Verträglichkeit. Abschnitt 7.4 beschreibt die *Bias*-Berechnung bei Instanzen der Klasse *subsumption equivalent* ($\Delta_S < \Delta_I$). In Abschnitt 7.5 werden die Algorithmen zur Prüfung der zustandsbasierten Verträglichkeit und in Abschnitt 7.6 die Mechanismen zur korrekten Neubewertung von Knotenzuständen vorgestellt. Wie sich auch die nicht automatisch zu migrierenden Instanzen der Klasse *partially equivalent* korrekt migrieren lassen, wird in Abschnitt 7.7 erläutert.

7.1 Gesamtablauf

Für das Änderungsrahmenwerk ergibt sich bei einer Schemaevolution, im Zusammenwirken mit umliegenden Komponenten des PMS, der folgende Ablauf:
Eine Schemaevolution beginnt mit der strukturellen Änderung einer Prozessvorlage *S* (Schema *S*). Die dafür notwendigen Mechanismen wurden im vorherigen Kapitel behandelt. Als Ergebnis resultiert eine neue Version *S'* der Prozessvorlage (geändertes Schema *S'*). Auf diese neue Version kann der Anwender die auf dem ursprünglichen Schema *S* laufenden Instanzen migrieren. Hierfür bietet das Änderungsrahmenwerk im MigrationManager (vgl. Abschnitt 5.2) die Methode *migrateInstances*(*S*, *S'*) an. Diese koordiniert das Anhalten der auf dem Schema *S* laufenden Instanzen *I*. Dazu ist es – analog zu einer instanzspezifischen Änderung – notwendig, im WorklistManager Methoden aufzurufen, die verhindern, dass die Ausführungszustände der zu migrierenden Instanzen durch einen Anwender verändert werden. Sind die Instanzen angehalten wird nacheinander jede auf S laufende Instanz separat betrachtet: Es wird geprüft, ob eine Instanz *I* geändert wurde oder ob ihr instanzspezifisches Schema S_I noch mit der ursprünglichen Prozessvorlage *S* identisch ist (*unbiased*). Handelt es sich um eine geänderte Instanz, so wird in der Methode *getOverlapClass*(Δ_S, Δ_I), anhand des Überlappungsgrades zwischen der Schemaänderung Δ_S und der auf *I* durchgeführten instanzspezifischen Änderung Δ_I, die Klassenzugehörigkeit zu *biased disjoint*, (*subsumption*) *equivalent* oder *partially equivalent* bestimmt. Abhängig von der Klassenzugehörigkeit werden die

folgenden Schritte durch Aufruf entsprechender Methoden der *ComplianceTest*- und *Adaptation-Layer* durchgeführt (vgl. Tabelle 7.1):

- *unbiased*: Testen der zustandsbasierten Verträglichkeit und Anpassen der Knotenzustände nach der Migration.
- *disjoint*: Testen der strukturellen und zustandsbasierten Verträglichkeit und Anpassen der Knotenzustände nach der Migration.
- *subsumption equivalent* ($\Delta_I < \Delta_S$): Testen der zustandsbasierten Verträglichkeit für die Änderungen die auf Schema- nicht aber auf Instanzebene durchgeführt worden sind. Neubewertung der von diesen Änderungen betroffenen Knoten.
- *subsumption equivalent* ($\Delta_S < \Delta_I$): Berechnung der Abweichungen (*Bias*) zwischen der Instanz *I* und dem geänderten Schema *S'*.
- *partially equivalent*: Testen der zustandsbasierten Verträglichkeit und Gruppieren der in Δ_S und Δ_I angewendeten Änderungen nach ihrem Typ (*change projections*). Lässt sich eine Teilmengenbeziehung zwischen den *change projections* von Δ_S und Δ_I herstellen, so werden die Migrationsschritte der entsprechenden Klasse angewendet. Anderenfalls ist eine Interaktion mit dem Benutzer erforderlich.

Die verträglichen Instanzen der Klassen *unbiased*, *biased disjoint* und (*subsumption*) *equivalent* werden direkt nach der Ausführung der für die jeweiligen Klassen notwendigen Schritte auf das neue Schema umgehängt und der *Worklist-Manager* angewiesen diese fortzusetzen. Für die Instanzen der Klasse *partially equivalent*, die sich nicht automatisch migrieren lassen, werden hingegen im *MigrationManager* aussagekräftige Konfliktmeldungen erzeugt, mit deren Hilfe der Anwender eine manuelle Migration auf das geänderte Schema durchführen kann. Damit ist die Schemaevolution abgeschlossen. Die mit den Schemaänderungen verträglichen Instanzen referenzieren nun alle die neue Version *S'* der Prozessvorlage. Die nicht verträglichen Instanzen laufen weiterhin auf der ursprünglichen Schemaversion *S* weiter.

	Unbiased	Biased				
		Disjoint	equivalent	subsumption equivalent ($\Delta_I < \Delta_S$)	subsumption equivalent ($\Delta_S < \Delta_I$)	Partially equivalent
strukturelle Verträglichkeit	-	✓	-	-	-	Benutzereingriff erforderlich
zustandsbasierte Verträglichkeit	✓	✓	-	für $\Delta_S \setminus \Delta_I$	-	✓
Change projections	-	-	-	-	-	✓
Neuberechnung des *Bias* $\Delta_I(S')$	-	-	-	-	✓	Benutzereingriff erforderlich
Knotenzustände anpassen	✓	✓	-	für $\Delta_S \setminus \Delta_I$	-	Benutzereingriff erforderlich

Tabelle 7.1 Von der Klassenzugehörigkeit abhängige Schritte bei der Migration einzelner Instanzen

Nachdem in diesem Abschnitt der Gesamtablauf auf einer eher funktionalen Ebene beschrieben wurde, werden in den folgenden Abschnitten die im Änderungsrahmenwerk auszuführenden Schritte im Detail erläutert.

7.2 Einteilung zu migrierender Instanzen in Klassen

Um bei der Migration von Instanzen gezielt die notwendigen Algorithmen anwenden zu können, müssen diese entlang des Überlappungsgrades zwischen der Schemaänderung Δ_S und der Instanzänderung Δ_I in Klassen eingeteilt werden. Wie man unveränderte Instanzen erkennt, wird in Abschnitt 7.2.1 beschrieben. Mit den Algorithmen zur Einteilung instanzspezifisch-geänderter Instanzen befasst sich Abschnitt 7.2.2.

7.2.1 Unveränderte Instanzen

Instanzen ohne instanzspezifische Änderungen (*unbiased*) lassen sich auf Grund des Deltaschicht-Konzeptes sofort erkennen. Es genügt zu prüfen, ob für eine betrachtete Instanz eine Deltaschicht existiert. Sollte dies nicht der Fall sein, so besitzt die Instanz keine Abweichungen vom ursprünglichen Schema und gehört somit zur Klasse *unbiased*.

7.2.2 Instanzspezifisch-geänderte Instanzen

Instanzspezifisch-geänderte Instanzen besitzen alle eine Deltaschicht. Das Einteilen in die Klassen *unbiased*, *biased disjoint*, (*subsumption*) *equivalent* und *partially equivalent* erfordert deshalb Algorithmen, die den Grad der Überlappung zwischen den Änderungen von Δ_I und Δ_S zuverlässig bestimmen können. Wie die Klassenzugehörigkeit im Einzelnen berechnet werden kann, wird in den Abschnitten 7.2.2.2-7.2.2.5 beschrieben. Die notwendigen Grundkonzepte liefert Abschnitt 7.2.2.1.

7.2.2.1 Grundlagen

Als Grundlage für die Klasseneinteilung dient ein in [Rind04] vorgestellter hybrider Ansatz (*Hybrid Approach*). Dieser Ansatz, der die Vorteile eines strukturellen und eines operationalen Konzeptes vereint, liefert die für eine Klassenberechnung notwendigen Ausgangsmengen. Als Basis nutzt der hybride Ansatz die Eigenschaft, dass sich der Grad der Überlappung zwischen Δ_I und Δ_S auf folgende zwei Arten bestimmen lässt:

- Durch direkten Vergleich zwischen dem geänderten Schema *S'* bzw. instanzspezifischen Schema S_I und dem ursprünglichen Schema *S* (*structural approach*).
- Durch direkte Gegenüberstellung der in Δ_S und Δ_I vorhandenen Änderungen (*operational approach*), d.h. durch Vergleich der Änderungshistorien von *S'* und *I*.

Beide Ansätze haben ihre Vor- und Nachteile. Die Anwendung des strukturellen Ansatzes liefert präzise Informationen bzgl. Einfüge- und Löschoperationen (im Sinne von neu eingefügten bzw. gelöschten Elementen), eine genaue Beurteilung von Reihenfolgeänderungsoperationen ist jedoch nicht möglich. Der operationale Ansatz ermöglicht hingegen präzise Aussagen über Reihenfolgeänderungsoperationen, hat allerdings das Problem, dass der Vergleich zweier Änderungshistorien ohne zusätzlichen Aufwand zu falschen Ergebnissen führen kann. Die beiden Ansätze werden im Folgenden genauer beschrieben, bevor in Abschnitt 7.2.2.1.3 der hybride Ansatz und in 7.2.2.1.4 dessen Umsetzung im Änderungsrahmenwerk erläutert wird.

7.2.2.1.1 Struktureller Ansatz

Der strukturelle Ansatz nutzt die Eigenschaft des Prozess-Metamodells, Prozesse nicht nur graphisch, sondern auch mit Hilfe von Mengen beschreiben zu können. Der Unterschied zwischen einem geänderten Schemata *S'* bzw. einem geänderten instanzspezifischen Schema S_I und dessen ursprünglicher Form *S* ist somit relativ einfach zu bestimmen. Es werden die vom Typ identischen

Mengen verglichen, um an die Abweichungen zu gelangen. Diese in [Rind04] als *Difference Sets* bezeichneten Mengen definieren sich folgendermaßen:

Control Flow Difference Sets: $N_{\Delta}^{add} := N' \setminus N$ und $N_{\Delta}^{del} := N \setminus N'$

$CtrlE_{\Delta}^{add} := CtrlE' \setminus CtrlE$ und $CtrlE_{\Delta}^{del} := CtrlE \setminus CtrlE'$

$SyncE_{\Delta}^{add} := SyncE' \setminus SyncE$ und $SyncE_{\Delta}^{del} := SyncE \setminus SyncE'$

$LoopE_{\Delta}^{add} := LoopE' \setminus LoopE$ und $LoopE_{\Delta}^{del} := LoopE \setminus LoopE'$

Data Flow Difference Sets: $D_{\Delta}^{add} := D' \setminus D$ und $D_{\Delta}^{del} := D \setminus D'$

$DataE_{\Delta}^{add} := DataE' \setminus DataE$ und $DataE_{\Delta}^{del} := DataE' \setminus DataE$

Abbildung 7.1 zeigt anhand eines Beispiels die Berechnung der *Difference Sets* zwischen *S'* und *S*:

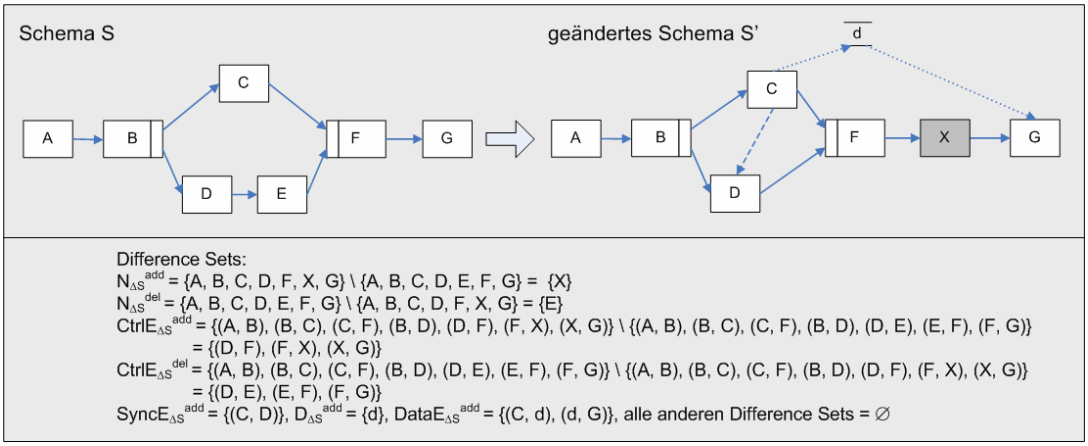


Abbildung 7.1 Difference Sets zwischen *S'* und *S*

Für ein umfassendes Änderungsrahmenwerk ist gegenüber [Rind04] noch eine weitere Menge zu berücksichtigen. Es handelt sich dabei um die Attribute, die gegenüber dem Originalschema geändert worden sind. Diese lassen sich analog zu den *Difference Sets* als:

NodeAttrChanged := *NodeAttributes'* \ *NodeAttributes* und

EdgeAttrChanged := *EdgeAttributes'* \ *EdgeAttributes* beschreiben.

Anhand der Konstruktion der *Difference Sets* können die Auswirkungen angewendeter Einfüge- und Lösch-Operationen direkt abgelesen werden. So signalisiert beispielsweise die Menge $N_{\Delta S}^{add} = \{X\}$, dass der Knoten *X* gegenüber dem ursprünglichen Schema eingefügt worden ist. Daraus resultiert, dass im Zuge der Klasseneinteilung ein Vergleich der *Difference Sets* von *S'* mit den *Difference Sets* von S_I konkrete Aussagen über Teilmengenbeziehungen zwischen neu eingefügten und gelöschten Elementen ermöglicht. Die bei einer Reihenfolgeänderungsoperation verschobenen Knoten lassen sich auf diese Weise jedoch nicht feststellen, da diese bereits im ursprünglichen Schema vorhanden sind und somit weder in der Menge N_{Δ}^{add} noch in der Menge N_{Δ}^{del} vorkommen (vgl. Abbildung 7.2). Eine Unterscheidung zwischen *subsumption* und *partially equivalent* ist allein mit den *Difference Sets* nicht möglich, wie das Beispiel aus Abbildung 7.2 zeigt:

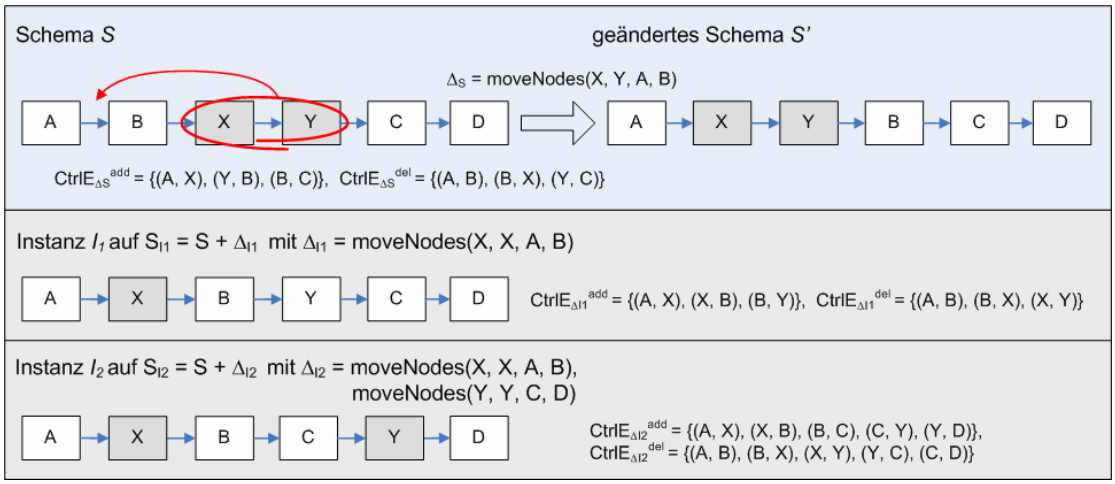


Abbildung 7.2 *Structural Approach* bei der Anwendung von Änderungsreihenfolgeoperationen

Für alle drei auf das Schema S angewendeten Änderungen Δ_S, Δ_{I1} und Δ_{I2} ergibt sich $N_{\Delta S}^{add} = N_{\Delta I1}^{add} = N_{\Delta I2}^{add} = \varnothing$ und $N_{\Delta S}^{del} = N_{\Delta I1}^{del} = N_{\Delta I2}^{del} = \varnothing$, da weder Knoten gelöscht noch eingefügt worden sind. Zusätzlich führt ein Vergleich der neu eingefügten und gelöschten Kontrollkanten zwischen Δ_S und Δ_{I1} bzw. zwischen Δ_S und Δ_{I2} zu den folgenden nicht-disjunkten Ergebnismengen: $CtrlE_{\Delta S}^{add} \cap CtrlE_{\Delta I1}^{add} = \{(A, X)\}$, $CtrlE_{\Delta S}^{del} \cap CtrlE_{\Delta I1}^{del} = \{(A, B), (B, X)\}$ bzw. $CtrlE_{\Delta S}^{add} \cap CtrlE_{\Delta I2}^{add} = \{(A, X), (B, C)\}$, $CtrlE_{\Delta S}^{del} \cap CtrlE_{\Delta I2}^{del} = \{(A, B), (B, X), (Y, C)\}$. Die einzige daraus mögliche Schlussfolgerung ist, dass sowohl $\Delta_S \cap \Delta_{I1} \neq \varnothing$ als auch $\Delta_S \cap \Delta_{I2} \neq \varnothing$ gilt. Das die Auswirkungen der Änderungen aus Δ_{I1} jedoch *subsumption equivalent*, die Auswirkungen der Änderungen von Δ_{I2} hingegen *partially equivalent* zu Δ_S sind, lässt sich daraus nicht feststellen. Weiterhin kann nicht bestimmt werden, ob S' durch Verschieben der Knoten X und Y an die Stelle zwischen A und B oder durch Verschieben des Knotens B zwischen Y und C entstanden ist. Eine präzise Beurteilung des Überlappungsgrades zwischen einer Schema- und einer Instanzänderung ist somit nicht möglich.

7.2.2.1.2 Operationaler Ansatz

Komplementär zum strukturellen Ansatz ermöglicht der operationale Ansatz Reihenfolgeänderungsoperationen präzise zu beurteilen. Anhand der in einer Änderungshistorie gespeicherten Änderungsoperationen lässt sich direkt bestimmen, welche Knoten verschoben worden sind. Der dadurch für eine Klasseneinteilung notwendige Vergleich zweier Änderungshistorien führt allerdings beim Auftreten von so genannter *noisy information* und bei kontextabhängigen Änderungsoperationen zu falschen Ergebnissen.

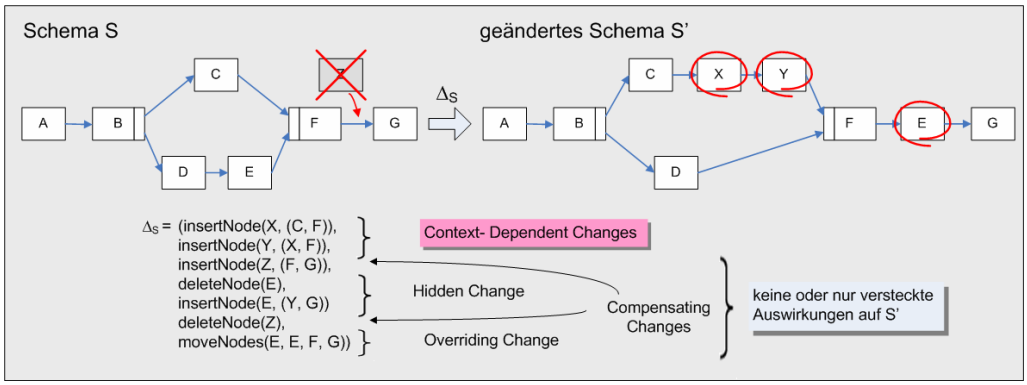


Abbildung 7.3 Beispiel einer Änderungshistorie mit *noisy information* [Rind04]

Wie Abbildung 7.3 zeigt, lässt sich die *noisy information* in die Kategorien versteckte (*hidden*), überschreibende (*overriding*) und kompensierende Änderungen (*compensating changes*) einteilen. Solche Änderungen kommen dadurch zustande, dass Anwender bei der Modifikation von Prozessen nicht immer zielorientiert vorgehen, sondern die optimale Lösung oftmals durch Ausprobieren erreichen. Unter solchen Umständen ist ein sinnvoller Vergleich zwischen Schema- und Instanzänderungen nicht möglich.
Ein weiteres Problem resultiert bei kontextabhängigen Änderungen. Dabei handelt es sich um Änderungen, die als Kontext einen ebenfalls im Zuge einer Änderung betroffenen Knoten besitzen. Aus der Anwendung der beiden Änderungsoperationen *insertNode*(X, (C, F)) und *insertNode*(Y, (X, F)) beispielsweise entsteht eine Sequenz $X \rightarrow Y$ zwischen den Knoten C und F. Das gleiche Ergebnis lässt sich aber auch durch Anwendung von *insertNode*(Y, (C, F)) und *insertNode*(X, (C, Y)) erreichen. Tritt die Situation auf, dass die eine Änderungskombination auf Instanzebene und die andere auf Schemaebene durchgeführt worden ist, so führt ein alleiniger Vergleich der Änderungshistorien zu dem Ergebnis, dass nicht die gleichen Änderungen zum Einsatz gekommen sind. Allerdings sind bei der Migration von Instanzen nicht die angewendeten Änderungsoperationen relevant sondern deren Auswirkungen. Diese sind in beiden Fällen gleich, weshalb eine solche Instanz zur Klasse *equivalent* zuzuordnen ist. Ein sinnvoller Vergleich zwischen Schema- und Instanzänderungen ist also auch mit dem operationalen Ansatz nicht möglich.

7.2.2.1.3 Hybrider Ansatz

Unter Ausnutzung der Vorteile beider Ansätze wurde in [Rind04] ein hybrider Ansatz (*Hybrid Approach*) entwickelt. Dieser verwendet die *Difference Sets* für die Informationen zu Einfüge- und Löschoperationen und die Änderungshistorie für verschobene Knoten. Um allerdings mit Hilfe des operationalen Ansatzes an die tatsächliche Menge der verschobenen Knoten (N_{Δ}^{move}) zu gelangen, müssen die Änderungshistorien zuerst von der *noisy information* befreit werden. Bei diesem als bereinigen (*purge*) bezeichneten Vorgehen wird eine Änderungshistorie $\Delta = (op_1, \ldots, op_n)$ in Rückwärtsrichtung (d.h. beginnend mit op_n) durchlaufen und für jede Änderungsoperation op_i ($i = 1, \ldots, n$) überprüft, ob diese Auswirkungen auf das Ausgangsschema S besitzt. Hat eine Operation keine Auswirkungen auf S, so wird diese abhängig von ihrem Typ gelöscht oder durch eine andere ersetzt [Laue04]. Dieses „Kürzen“ um *hidden*, *overriding* und *compensating changes* führt zu einer kanonischen Form der Änderungshistorie. In dieser Form befinden sich in der Änderungshistorie nur noch Reihenfolgeänderungsoperationen die tatsächlich Auswirkungen auf das Schema S besitzen. Die verschobenen Knoten können also direkt abgelesen werden.
Durch die Kombination von *Difference Sets* und bereinigter Änderungshistorie ermöglicht es der hybride Ansatz auf konzeptioneller Ebene alle für eine Klasseneinteilung notwendigen Ausgangsmengen zu bestimmen.

7.2.2.1.4 Umsetzung des hybriden Ansatzes

Ein direktes Umsetzen des hybriden Ansatzes im Änderungsrahmenwerk ist allerdings nicht möglich. Dies liegt daran, dass die für das Bereinigen der Änderungshistorie notwendige Information direkt bei den Änderungsoperationen hinterlegt werden muss. Zusätzlich hängt die Entscheidung, ob eine Änderungsoperation aus der Historie gelöscht oder durch eine neue ersetzt wird, von den anderen ebenfalls in der Änderungshistorie enthaltenen Operationen ab. Diese Abhängigkeiten sind bezüglich der Erweiterbarkeit des Änderungsrahmenwerkes sehr kritisch und somit ein Umsetzen des hybriden Ansatzes in der bestehenden Form nicht durchführbar. Um die leichte Erweiterbarkeit des Änderungsrahmenwerkes erhalten zu können muss eine Lösung entwickelt werden, mit der die Menge

der verschobenen Knoten N_{Δ}^{move} unabhängig von den tatsächlich angewendeten Änderungsoperationen und damit unabhängig von den dort gespeicherten Bereinigungsanweisungen ermittelt werden kann.

Hierzu werden noch einmal die *Difference Sets* betrachtet. Diese enthalten exakt die Abweichungen zwischen dem ursprünglichen Schema S und einem geänderten (instanzspezifischen) Schema S' (bzw. S_I). Daraus lässt sich direkt ableiten, dass die *Difference Sets* genau die gleiche Information enthalten wie eine Deltaschicht (vgl. Abschnitt 3.5). Als Folge davon können die *Difference Sets* direkt aus der Deltaschicht abgeleitet werden. Eine gesonderte Berechnung entfällt.
Hinzu kommt, dass die Deltaschicht analog zu den *Difference Sets* implizit bereinigt ist. So existieren keine *compensating changes*, da auf Instanzebene und somit auch in die Deltaschicht eingefügte Knoten nach deren Löschung wieder aus der Deltaschicht entfernt werden. Analog gibt es auch keine *hidden changes*, weil ein gelöschtes und danach wieder eingefügtes Element ebenfalls nicht in der Deltaschicht auftritt. Durch die Abstraktion der Deltaschicht von den eigentlich angewendeten Operationen treten auch keine *overriding changes* auf. Das bedeutet, dass das Phänomen bei dem die Auswirkungen früherer Änderungen durch Anwendung nachfolgender Änderungsoperationen überschrieben werden nicht beachtet werden muss. Die Deltaschicht bereinigt sich automatisch von solchen *overriding changes*.
Wie bei den *Difference Sets* sind die verschobenen Knoten aber auch aus der Deltaschicht nicht direkt ersichtlich. Um trotzdem die Vorteile der Deltaschicht nutzen zu können, wird diese für das Änderungsrahmenwerk um eine zusätzliche Knotenmenge *movedNodes* erweitert. Die folgende Abbildung zeigt beispielhaft eine um die Menge der verschobenen Knoten erweiterte Deltaschicht. Man beachte, dass hier nicht die *Difference Sets* sondern äquivalent die tatsächlichen Bezeichnungen der Deltaschicht verwendet werden.

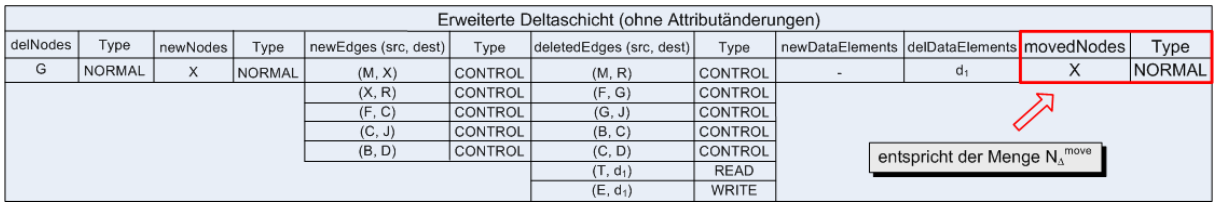


Abbildung 7.4 Beispiel einer erweiterten Deltaschicht

Die Deltaschicht bleibt trotz der Erweiterung implizit bereinigt wenn Folgendes beachtet wird:
Befindet sich ein zu löschender Knoten in der Menge der verschobenen Knoten, so wird dessen Eintrag aus *movedNodes* gelöscht. Ist ein verschobener Knoten X in einer vorherigen Änderungsoperation eingefügt worden ($X \in newNodes$), so wird X nicht in *movedNodes* eingefügt.

Oberflächlich scheinen sich die Bereinigungs-Anweisungen einer Deltaschicht nicht von den Bereinigungs-Mechanismen einer Änderungshistorie zu unterscheiden. Der Vorteil beim „erweiterten“ Deltaschicht-Konzept liegt jedoch im Speicherort der für das Bereinigen notwendigen Mechanismen. Diese werden nicht wie beim Bereinigen einer Änderungshistorie bei den Änderungsoperationen sondern direkt bei den Änderungsprimitiven hinterlegt. Dies ist möglich da alle (inkl. den in Zukunft in das Änderungsrahmenwerk eingebrachten) Änderungsoperationen bei der Manipulation eines (instanzspezifischen) Schemas auf diese fest definierte Anzahl an Änderungsprimitiven zurückgreifen (vgl. Abschnitt 3.2). Die Primitiven wiederum manipulieren direkt die Mengen der Knoten, Kanten, Datenelemente, etc. der Deltaschicht und somit letztendlich auch die benötigten *Difference Sets* inklusive N_{Δ}^{move}. Dadurch ist gewährleistet, dass die Deltaschicht einerseits immer bereinigt und somit die *Difference Sets* immer berechenbar sind und andererseits die Erweiterbarkeit des

Änderungsrahmenwerks gewährleistet bleibt. Weiterhin ergibt sich der Vorteil, dass die Deltaschicht direkt bei der Anwendung einer Änderungsoperation bereinigt wird, während dies bei einer Änderungshistorie erst nach der Durchführung der gesamten Änderung möglich ist. Die Deltaschicht ist somit zu jedem Zeitpunkt frei von *hidden*, *overriding* und *compensating changes*. Dies steigert die Effizienz bei der Schemaevolution maßgeblich, da unter Verwendung des erweiterten Deltaschicht-Konzeptes, die für die Klasseneinteilung notwendige Information bereits bereinigt vorliegt, während bei dem Konzept aus [Rind04] erst die Änderungshistorie bereinigt werden muss. Bei der in der Praxis üblichen Anzahl zu migrierender und somit einzuteilender Instanzen resultiert aus dem Konzept der erweiterten Deltaschicht ein entscheidender Vorteil.

Man erhält mit dieser erweiterten Deltaschicht einen Mechanismus, mit dem alle für die Klasseneinteilung notwendigen Ausgangsmengen unter Erhaltung der Erweiterbarkeit des Änderungsrahmenwerkes und mit maximaler Effizienz bestimmt werden können.

7.2.2.2 *Equivalent*-Instanzen

Die Erkennung von Instanzen der Klasse *equivalent*, also von Instanzen, bei denen die instanzspezifischen Änderungen Δ_I und die Schemaänderungen Δ_S exakt die gleichen Auswirkungen auf das ursprüngliche Schema S besitzen (vgl. Abschnitt 3.4.4.2.1), gestaltet sich relativ einfach. Es genügt, für die geänderte Instanz I und das geänderte Schema S' die *Difference Sets* aus den jeweiligen Deltaschichten abzuleiten und miteinander zu vergleichen. Sind die einzelnen *Difference Sets* von I mit den entsprechenden *Difference Sets* von S' identisch, so ist die betrachtete Instanz zur Klasse *equivalent* zuzuordnen. Dabei sind bei diesem speziellen Fall die Mengen N_{Δ}^{move} von I und S' nicht zu beachten, da lediglich die Auswirkungen der angewendeten Änderungen für den Vergleich relevant sind. So kann es sein, dass sich trotz Gleichheit aller anderen *Difference Sets*, die Mengen $N_{\Delta S}^{move}$ und $N_{\Delta I}^{move}$ unterscheiden. Dies ist immer genau dann der Fall, wenn die Anwendung unterschiedlicher Reihenfolgeänderungsoperationen zum selben Schema führen. Abbildung 7.5 zeigt einen solchen Fall:

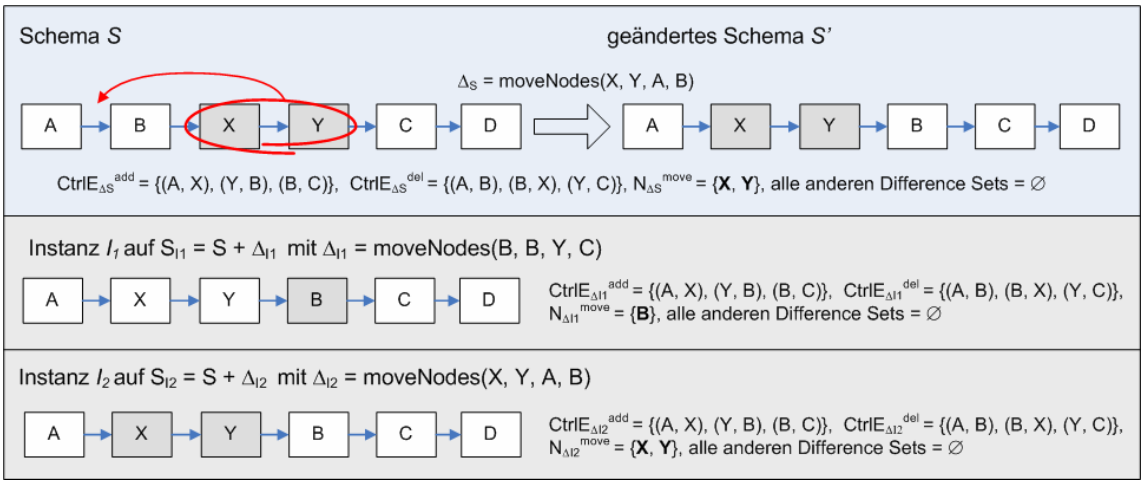


Abbildung 7.5 Instanzen der Klasse *equivalent*

Sowohl die Instanz I_1 als auch I_2 sind mit dem geänderten Schema S' identisch. Folglich gehören beide Instanzen zur Klasse *equivalent*. Dabei sind jedoch nicht dieselben Änderungsoperationen angewendet worden. Während bei Δ_{I2} und Δ_S die Knoten X und Y verschoben worden sind, signalisiert Δ_{I1}, dass dies bei Instanz I_1 für B gilt. Vergleicht man alle *Difference Sets* von S' mit denen von I_2, so ergibt

sich $CtrlE_{\Delta S}^{add} = CtrlE_{\Delta I2}^{add}$, $CtrlE_{\Delta S}^{del} = CtrlE_{\Delta I2}^{del}$ und $N_{\Delta S}^{move} = N_{\Delta I2}^{move}$ was dem gewünschten Ergebnis entspricht. Vergleicht man allerdings die *Difference Sets* von *S'* mit denen von I_1, so ergibt sich $N_{\Delta S}^{move} \neq N_{\Delta I1}^{move}$. Daraus zu folgern, dass die Änderungen von Δ_S und Δ_{I1} nicht identisch sind und die betrachtete Instanz somit nicht zur Klasse *equivalent* gehört, ist allerdings falsch. Für die Gleichheit entscheidend sind nicht die angewendeten Änderungsoperationen, sondern deren Auswirkungen, die sich in diesem Beispiel in den *Difference Sets* $CtrlE_{\Delta}^{add}$ und $CtrlE_{\Delta}^{del}$ niederschlagen. Diese sind sowohl für das Schema als auch für beide Instanzen identisch. Die Menge N_{Δ}^{move} darf somit bei der Bestimmung von *equivalent* Instanzen nicht zu einem Vergleich herangezogen werden, da es sich bei dieser lediglich um eine Hilfsmenge und nicht um eine die Abweichungen zum ursprünglichen Schema signalisierende Menge handelt.

Das für eine Instanz der Klasse *equivalent* zu erfüllende Kriterium lautet also folgendermaßen:

Kriterium 7 (Kriterium für Instanzen der Klasse *equivalent*)

Eine Instanz gehört zur Klasse *equivalent*, wenn die *Difference Sets* ohne $N_{\Delta S}^{move}$ von *S'* mit den entsprechenden *Difference Sets* ohne $N_{\Delta I}^{move}$ von *I* identisch sind.

7.2.2.3 *Disjoint*-Instanzen

Zur Klasse *disjoint* gehören diejenigen Instanzen, deren instanzspezifische Änderungen Δ_I nicht mit den Schemaänderungen Δ_S überlappen (vgl. Abschnitt 3.4.4.1). Um dies feststellen zu können werden analog zu *equivalent* Instanzen sowohl für die geänderte Instanz *I* als auch für das geänderte Schema *S'* die *Difference Sets* bestimmt und miteinander verglichen. Ergibt sich bei dem Vergleich der Difference Sets von *S'* und *I* in jedem Fall eine leere Menge, so ist das erste Kriterium für eine Instanz der Klasse *disjoint* erfüllt.
Dies ist aber noch nicht ausreichend, wie das Beispiel aus Abbildung 7.6 verdeutlicht.

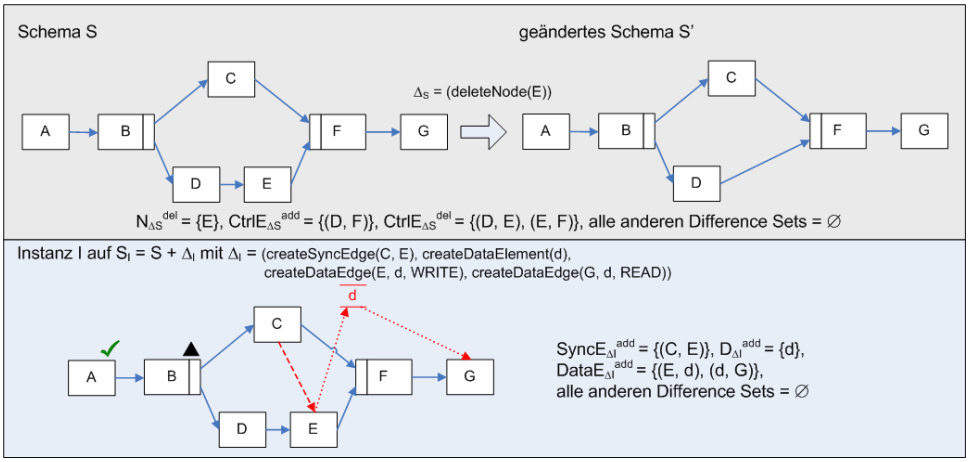


Abbildung 7.6 Probleme bei der Erkennung von *disjoint* Instanzen

Vergleicht man die *Difference Sets* von *S'* mit denen von *I*, so ergeben sich in jedem Fall disjunkte Mengen. Diejenigen *Difference Sets*, die bei *S'* nicht leer sind ($N_{\Delta S}^{del}$, $CtrlE_{\Delta S}^{add}$ und $CtrlE_{\Delta S}^{del}$), sind es bei *I*. Im umgekehrten Fall gilt dasselbe. Folglich sind alle *Difference Sets* disjunkt. Würde man es bei dieser Prüfung belassen, so wäre die Instanz *I* aus Abbildung 7.6 zur Klasse *disjoint* zuzuordnen. Dies ist allerdings falsch. Die Änderungen überlappen sich zwar nicht im Sinne von gleichen Auswirkungen auf das ursprüngliche Schema *S*, sie überlappen sich aber im jeweils verwendeten Zielkontext. So wird im Zuge der Schemaänderung der Knoten *E* gelöscht. Dieser dient aber den

Änderungen *createSyncEdge*(*C*, *E*) und *createDataEdge*(*E*, *d*, *WRITE*) von Δ_I als Kontext. Die Änderungen sind also nicht disjunkt sondern *partially equivalent*.
Dieses Problem der Kontext zerstörenden Änderungen (*context destroying changes*) wurde bereits in Abschnitt 3.4.4.2.3 (*Partially equivalent*) erläutert. Für die Einteilung in die Klasse *disjoint* müssen somit zusätzlich die folgenden Kriterien bzgl. kontextzerstörender Änderungen erfüllt sein; der entsprechende Algorithmus findet sich in Anhang D (Algorithmen zur Klasseneinteilung):

- Weder *src* noch *dest* einer Sync-Kante (*src*, *dest*) aus der Menge $SyncE_{\Delta S}^{add}$ bzw. $SyncE_{\Delta I}^{add}$ befinden sich in einer der Mengen $N_{\Delta I}^{move}$ oder $N_{\Delta I}^{del}$ bzw. $N_{\Delta S}^{move}$ oder $N_{\Delta S}^{del}$.
- Ein Datenelement *data* einer Kante der Menge $DataE_{\Delta S}^{add}$ (*node*, *data*) bzw. $DataE_{\Delta I}^{add}$ befindet sich nicht in einer der Mengen $D_{\Delta I}^{del}$ bzw. $D_{\Delta S}^{del}$ und *node* ist nicht in $N_{\Delta I}^{move}$ oder $N_{\Delta I}^{del}$ bzw. $N_{\Delta S}^{move}$ oder $N_{\Delta S}^{del}$ enthalten.
- Die Knoten der Knotenattribute aus der Menge $NodeAttrChanged_I$ bzw. $NodeAttrChanged_S$ sind nicht in $N_{\Delta S}^{del}$ bzw. $N_{\Delta I}^{del}$ enthalten.
- Die Kanten der Kantenattribute aus der Menge $EdgeAttrChanged_S$ bzw. $EdgeAttrChanged_I$ befinden sich nicht in der Menge $CtrlE_{\Delta S}^{del}$ bzw. $CtrlE_{\Delta I}^{del}$.

Die in Abschnitt 3.4.4.2.3 zusätzlich angegebene Situation, dass Δ_S bzw. Δ_I einen Knoten verschiebt oder löscht, der für eine Knoteneinfüge- oder -verschiebeoperation aus Δ_S bzw. Δ_I als Teil des Zielkontextes fungiert, muss bei *disjoint* nicht beachtet werden. Aufgrund der notwendigen Disjunktheit der *Difference Sets* ist eine solche Kombination von Änderungsoperationen nicht möglich.

Die für eine Instanz der Klasse *disjoint* zu erfüllenden Kriterien lauten also folgendermaßen:

Kriterium 8 (Kriterien für Instanzen der Klasse *disjoint*)

Eine Instanz gehört zur Klasse *disjoint*, wenn einerseits alle *Difference Sets* von *S'* zu den Difference Sets von *I* disjunkt sind und andererseits die Kriterien zum Ausschluss der für *disjoint* relevanten kontextzerstörenden Änderungen erfüllt sind.

7.2.2.4 *Subsumption equivalent*-Instanzen

Die Klasse *subsumption equivalent* teilt sich in die zwei Kategorien $\Delta_S < \Delta_I$ und $\Delta_I < \Delta_S$. Zur ersten Kategorie gehören die Instanzen, bei denen Δ_I die gleiche Auswirkung auf *S* hat wie Δ_S, Δ_I aber noch zusätzlich weitere Auswirkungen auf *S* besitzt. Bei Instanzen der zweiten Kategorie ist dies genau umgekehrt, Δ_S hat die gleichen Auswirkungen auf *S* wie Δ_I, Δ_S besitzt aber noch zusätzlich weitere Auswirkungen auf *S*. Bei den Mechanismen zur Bestimmung der Klassenzugehörigkeit kann zwischen den Schema- und Instanzänderungen ohne und denen mit kontextabhängigen Änderungen unterschieden werden.

7.2.2.4.1 Ohne kontextabhängige Änderungen

Enthalten weder Δ_I noch Δ_S kontextabhängige Änderungen, so lässt sich die Zugehörigkeit einer Instanz zur Klasse *subsumption equivalent* allein mit Hilfe der *Difference Sets* bestimmen. Dazu werden wiederum die *Difference Sets* von *S'* und *I* miteinander verglichen. Ergibt sich für jedes *Difference Set* eine Teilmengenbeziehung, so ist die betrachtete Instanz in jedem Fall zur Klasse *subsumption equivalent* zuzuordnen. Ob diese nun zur Klasse $\Delta_S < \Delta_I$ oder zur Klasse $\Delta_I < \Delta_S$ gehört, leitet sich direkt aus der Richtung der Teilmengenbeziehung ab. Sind alle *Difference Sets* von *S'* eine Teilmenge der entsprechenden *Difference Sets* von *I*, so gehört die Instanz zu ersten Kategorie, im umgekehrten Fall zur zweiten.

Kriterium 9 (Kriterium für Instanzen der Klasse *subsumption equivalent* ohne kontextabhängige Änderungen)

Enthalten weder Δ_I noch Δ_S kontextabhängige Änderungen, so gehört eine Instanz zur Klasse *subsumption equivalent* wenn jedes *Difference Set* von *S'* eine Teilmenge des entsprechenden *Difference Set* von *I* (*subsumption equivalent* $\Delta_S < \Delta_I$) oder jedes *Difference Set* von *I* eine Teilmenge des entsprechenden *Difference Set* von *S'* (*subsumption equivalent* $\Delta_I < \Delta_S$) ist.

7.2.2.4.2 Mit kontextabhängigen Änderungen

Enthält Δ_I und/oder Δ_S kontextabhängige Änderungen, so ist es allein mit den *Difference Sets* nicht möglich *subsumption equivalent* Instanzen zu erkennen wie das Beispiel aus Abbildung 7.7 zeigt.

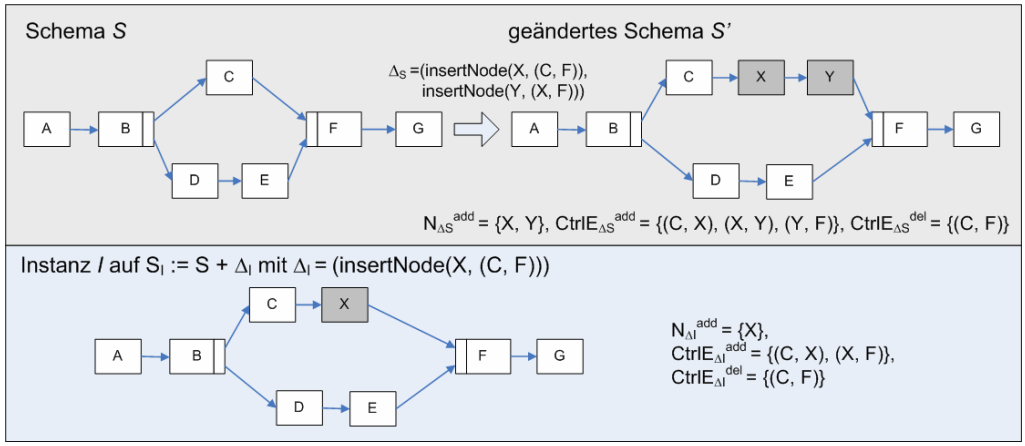


Abbildung 7.7 *subsumption equivalent* und kontextabhängige Änderungen

Vergleicht man die neu hinzugefügten Knoten und die gelöschten Kontrollkanten ergibt sich: $N_{\Delta I}^{add} \subseteq N_{\Delta S}^{add}$ und $CtrlE_{\Delta I}^{del} \subseteq CtrlE_{\Delta S}^{del}$. Es gilt also die Teilmengenbeziehung. Betrachtet man die neu eingefügten Kontrollkanten, so gilt die Beziehung allerdings nicht mehr. Sowohl $CtrlE_{\Delta S}^{add}$ als auch $CtrlE_{\Delta I}^{add}$ enthalten Kanten die im jeweils anderen *Difference Set* nicht vorhanden sind. Dies könnte man als Indiz für eine Instanz der Klasse *partially equivalent* deuten. Tatsächlich handelt es sich bei *I* aber um eine Instanz der Klasse *subsumption equivalent* mit $\Delta_I < \Delta_S$. Die Ursache, dass die Klassenzugehörigkeit nicht an Hand der *Difference Sets* erkannt werden kann, liegt an der Kontextabhängigkeit der in Δ_S angewendeten Änderungsoperationen. So verwendet die Operation *insertNode*(*Y*, (*X*, *F*)) den im vorherigen Schritt eingefügten Knoten *X* als Kontext. Als Folge davon besitzen *X* und *Y* bezogen auf das ursprüngliche Schema *S* dieselben Kontextknoten, nämlich *C* und *F*. Die Verbindung zum ursprünglichen Schema wird dabei durch die Kanten (*C*, *X*) und (*Y*, *F*) hergestellt. Da bei der Instanzänderung nur der Knoten *X* zwischen *C* und *F* eingefügt wurde und somit die Kanten (*C*, *X*) und (*X*, *F*) die Verbindung zu *S* übernehmen ist es zwangsläufig nicht möglich, eine Teilmengenbeziehung zwischen $CtrlE_{\Delta S}^{add}$ und $CtrlE_{\Delta I}^{add}$ zu erreichen.

Anhand des Beispiels lässt sich erkennen, dass bei kontextabhängigen Änderungen die Mengen $CtrlE_{\Delta}^{add}$ und $CtrlE_{\Delta}^{del}$ nicht beachtet werden dürfen, stattdessen aber der Kontext im ursprünglichen Schema *S* eine wichtige Rolle spielt. Dieser in [Rind04] als Anker (*Anchor*) bezeichnete Kontext lässt sich sowohl für neu eingefügte als auch für verschobene Knoten folgendermaßen berechnen (*AnchorIns*(Δ) bzw. *AnchorMove*(Δ)):

Suche für jeden Knoten X der Menge N_Δ^{add} bzw. N_Δ^{move} den letzten bereits im ursprünglichen Schema S vorhandenen Vorgänger (*left*) und den ersten bereits in S vorhandenen Nachfolger (*right*) und füge der Menge *AnchorIns*(Δ) bzw. *AnchorMove*(Δ) den Eintrag (*left*, *X*, *right*) hinzu.

Die angegebene Suche nach den bereits im ursprünglichen Schema vorhandenen Knoten ist mit Hilfe der vom Datenmodell bereitgestellten Funktionen *getPredByEdgeType*(*CONTROL*) und *getSuccByEdgeType*(*CONTROL*) des Interfaces *Node* leicht durchzuführen. Ein vollständiger Algorithmus findet sich in Anhang D (Algorithmen zur Klasseneinteilung).

Mit den Ankern allein, ist es bei kontextabhängigen Änderungen aber noch nicht möglich zuverlässig *subsumption equivalence* festzustellen, wie folgendes Beispiel zeigt:

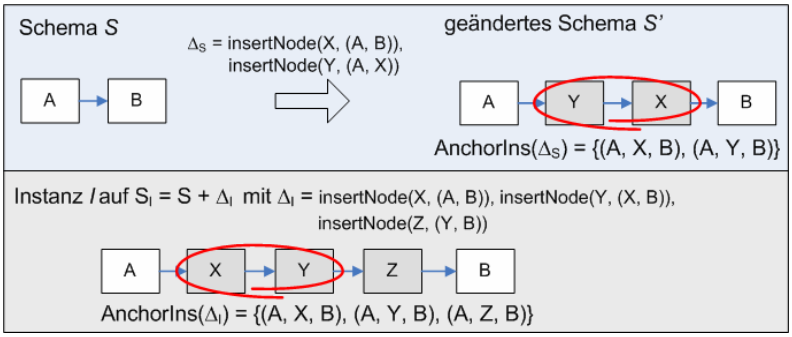


Abbildung 7.8 Teilmengenbeziehung zwischen Ankersätzen aber unterschiedliche Reihenfolge

Die Ankersätze von *S'* sind eine Teilmenge der Ankersätze von *I*. Es ergibt sich $AnchorIns(\Delta_S) \subseteq AnchorIns(\Delta_I)$. Die Instanz *I* gehört aber keinesfalls zur Klasse *subsumption equivalent*, da diejenigen Knoten die bei beiden neu eingefügt wurden in unterschiedlicher Reihenfolgebeziehung zueinander stehen. Die Instanz ist folglich vom Typ *partially equivalent*.

Zusätzlich zu den Ankern muss also auch die Reihenfolgebeziehung zwischen den Knoten mit denselben Ankern beachten werden. Dafür sind in einem ersten Schritt die Knoten aus N_Δ^{add} und N_Δ^{move} so zu gruppieren, dass jede Gruppe diejenigen Knoten enthält, die den gleichen Anker besitzen. Diese als *AnchorGroupsAgg* zusammengefassten Gruppen lassen sich folgendermaßen berechnen:

$$AnchorGroupsAgg(\Delta) = \{G = \{X_1, \ldots, X_n\} \mid \forall\, X_i, X_j \in G: \exists\, (left_i, X_i, right_i), (left_j, X_j, right_j) \in AnchorIns(\Delta) \cup AnchorMove(\Delta) \text{ mit } left_i = left_j \wedge right_i = right_j\ (i, j = 1, \ldots, n)\}$$

Eine Gruppe *G* wird also nur dann erstellt, wenn mindestens zwei Knoten denselben Anker besitzen, da nur in diesem Fall eine Reihenfolgebeziehung notwendig bzw. möglich ist.
Die Reihenfolgebeziehung selbst berechnet sich, indem man für jede Gruppe $G = \{X_1, \ldots, X_n\}$ aus *AnchorGroupsAgg*(Δ) jeweils zwei Aktivitäten X_i und X_j betrachtet und prüft, ob X_i (transitiver) Vorgänger oder (transitiver) Nachfolger von X_j ist. Im Falle einer Vorgängerbeziehung wird ein Eintrag (X_i, X_j) erzeugt, bei einer Nachfolgerbeziehung (X_j, X_i). Beide Einträge werden in der Menge *OrderAgg*(Δ) gespeichert (vgl. [Rind04], S. 169). Zur Feststellung der Vorgänger-/Nachfolgerbeziehung kann die Funktion *isPredecessorOf* im Interface *node* des Datenmodells verwendet werden (vgl. Abschnitt 5.1.2.1).

Angewendet auf das Beispiel in Abbildung 7.7 ergeben sich die Mengen $AnchorGoupsAgg(\Delta_S) = \{\{X, Y\}\}$ und $AnchorGoupsAgg(\Delta_I) = \{\{X, Y, Z\}\}$. Daraus berechnen sich die Reihenfolgebeziehungen $OrderAgg(\Delta_S) = \{(Y, X)\}$ und $OrderAgg(\Delta_I) = \{(X, Y), (Y, Z)\}$. Eine Teilmengenbeziehung ist also nicht vorhanden. Folglich lässt sich mit Hilfe dieser *OrderAgg*-Mengen feststellen, dass es sich nicht um eine Instanz der Klasse *subsumption equivalent*, sondern um eine Instanz der Klasse *partially equivalent* handelt.

Dass mit Hilfe der Reihenfolgebeziehungen aber auch das eigentliche Ziel – nämlich die Erkennung von *subsumption equivalent* Instanzen – erreicht wird, zeigt die folgende Berechnung für das Beispiel aus Abbildung 7.7. Für die bei *subsumption equivalent* zu prüfenden *Difference Sets* (entspricht den *Difference Sets* ohne $CtrlE_{\Delta}^{add}$ und $CtrlE_{\Delta}^{del}$) ergibt sich $N_{\Delta I}^{add} \subseteq N_{\Delta S}^{add}$. Auch die Ankergruppe $AnchorIns(\Delta_I) = \{(C, X, F)\}$ ist eine Teilmenge von $AnchorIns(\Delta_S) = \{(C, X, F), (C, Y, F)\}$. Es muss also noch geprüft werden, ob dies auch für die *OrderAgg*-Mengen der Fall ist. Dazu werden wiederum zuerst die *AnchorGroupsAgg*-Mengen berechnet. Es resultiert für $AnchorGroupsAgg(\Delta_S) = \{\{X, Y\}\}$, für $AnchorGroupsAgg(\Delta_I)$ hingegen die leere Menge, da zwischen denselben Ankern lediglich ein Knoten – nämlich X – eingefügt worden ist. Somit ist die Menge $OrderAgg(\Delta_I)$ zwangsläufig leer und damit in jedem Fall eine Teilmenge von $OrderAgg(\Delta_S) = \{(X, Y)\}$. Es ergibt sich für alle betrachteten Mengen eine Teilmengenbeziehung zwischen den Mengen aus *S'* und denen aus *I*. Folglich ist die betrachtete Instanz der Klasse *subsumption equivalent* ($\Delta_I < \Delta_S$) zuzuordnen.

Das für eine Instanz der Klasse *subsumption equivalent* zu erfüllende Kriterium im Falle kontextabhängiger Änderungen lautet also folgendermaßen:

Kriterium 10 (Kriterium für Instanzen der Klasse *subsumption equivalent* bei Auftreten kontextabhängiger Änderungen)

Enthält Δ_I und/oder Δ_S kontextabhängige Änderungen, so gehört eine Instanz genau dann zur Klasse

- *subsumption equivalent* ($\Delta_S < \Delta_I$), wenn die *Difference Sets* von *S'*, ausgenommen $CtrlE_{\Delta S}^{add}$ und $CtrlE_{\Delta S}^{del}$ eine Teilmenge der entsprechenden *Difference Sets* von *I* sind und folgende Beziehungen erfüllt werden:
  - $AnchorIns(\Delta_S) \subseteq AnchorIns(\Delta_I) \cap AnchorMove(\Delta_S) \subseteq AnchorMove(\Delta_I)$
  - $OrderAgg(\Delta_S) \subseteq OrderAgg(\Delta_I)$
- *subsumption equivalent* ($\Delta_I < \Delta_S$), wenn die *Difference Sets* von *I*, ausgenommen $CtrlE_{\Delta I}^{add}$ und $CtrlE_{\Delta I}^{del}$ eine Teilmenge der entsprechenden *Difference Sets* von *S'* sind und folgende Beziehungen erfüllt werden:
  - $AnchorIns(\Delta_I) \subseteq AnchorIns(\Delta_S) \cap AnchorMove(\Delta_I) \subseteq AnchorMove(\Delta_S)$
  - $OrderAgg(\Delta_I) \subseteq OrderAgg(\Delta_S)$

7.2.2.5 *Partially equivalent*-Instanzen

Für die Klasse *partially equivalent* gibt es keine gesonderten Mechanismen. Die Zugehörigkeit einer Instanz zu dieser Klasse resultiert automatisch aus den Prüfungen der anderen Klassen. Eine betrachtete Instanz ist also genau dann zur Klasse *partially equivalent* zuzuordnen wenn sie keine der Zugehörigkeitskriterien für *disjoint*, *equivalent* oder *subsumption equivalent* erfüllt.
Dies lässt sich folgendermaßen definieren:

Kriterium 11 (Kriterium für Instanzen der Klasse *partially equivalent*)

Die Auswirkungen der Änderungen aus Δ_I und Δ_S sind teilweise äquivalent und die Instanz I somit der Klasse *partially equivalent* zuzuordnen, wenn Δ_I weder *disjoint* noch *equivalent* noch *subsumption equivalent* zu Δ_S ist.

Das Kriterium gilt allerdings bei vielen als *partially equivalent* erkannten Instanzen nur, wenn man die instanzspezifischen Änderungen Δ_I und die Schemaänderungen Δ_S insgesamt betrachtet (vgl. Abschnitt 3.4.4.2.3). Gruppiert man die Änderungen von Δ_I und Δ_S jeweils nach ihrem Typ und vergleicht die resultierenden Gruppen (*change projections*) miteinander so erfüllen diese, für sich betrachtet, oftmals das Zugehörigkeitskriterium einer der anderen Klassen. Dies ist im Hinblick auf eine automatisch durchzuführende Migration von entscheidender Bedeutung, da die Instanzen der anderen Klassen automatisch migriert werden können, während dies bei Instanzen der Klasse *partially equivalent* nicht möglich ist. Es wird deshalb an dieser Stelle noch einmal genauer auf die Berechnung der *change projections* eingegangen.

In [Rind04] werden die im Zuge einer Schema- oder Instanzänderung angewendeten Änderungsoperationen über die folgende Funktion auf die verschiedenen Änderungsgruppen (*change projections*) verteilt:

$optype: Change \rightarrow OpType$

Change repräsentiert hierbei eine fest definierte Menge an Änderungsoperationen, *OpType* eine Gruppe, mit typgleichen Änderungsoperationen und *optype* die Funktion die jeder Änderungsoperation aus *Change* seine entsprechende Gruppe *OpType* zuordnet.

Um diese Funktion praktisch umzusetzen muss entweder die Gruppe bei jeder einzelnen Änderungsoperation hinterlegt sein oder Algorithmus existieren, der alle Änderungsoperationen kennt und entsprechend zuordnen kann. Beides ist im Hinblick auf das nachträgliche Einfügen von Änderungsoperationen über die Plug-In-Schnittstelle nicht optimal. Im ersten Fall muss der Plug-In-Programmierer wissen, welche Gruppen im System existieren und zu welcher Gruppe die von ihm programmierte Operation gehört, im zweiten Fall ist es sogar notwendig den Kern zu ändern, damit dieser die neue Änderungsoperation überhaupt erkennt.

Eine geeignete Umsetzung muss also von den angewendeten Änderungsoperationen abstrahieren und trotzdem eine analoge Gruppeneinteilung ermöglichen. Hierzu wird wiederum die Eigenschaft ausgenutzt, dass alle Änderungsoperationen auf einer fest definierten Anzahl von Änderungsprimitiven beruhen und diese direkt die Daten in der Deltaschicht und somit die *Difference Sets* manipulieren. Folglich muss es an Hand der *Difference Sets* möglich sein, die geforderte Gruppeneinteilung durchzuführen. Im direkten Vergleich mit dem Ansatz aus [Rind04] ergeben sich für das Beispiel aus Abbildung 7.9 die folgenden *change projections*:

$$\begin{aligned}
\Delta_S[ins_Node] &= \{N_{\Delta S}^{add} = \{X, Z, Y\},\\
&\quad CtrlE_{\Delta S}^{add} = \{(C, X), (X, Y), (Y, F), (A, Z),\\
&\quad\quad (Z, B)\},\\
&\quad CtrlE_{\Delta S}^{del} = \{(C, F), (A, B)\}\}\\
\Delta_S[del_Node] &= \{N_{\Delta S}^{del} = \{E\}, CtrlE_{\Delta S}^{add} = \{(D, F)\},\\
&\quad CtrlE_{\Delta S}^{del} = \{(D, E), (E, F)\}\}\\
\Delta_S[data] &= \{D_{\Delta S}^{add} = \{d\},\\
&\quad DataE_{\Delta S}^{add} = \{(X, d, write), (Y, d, read)\}
\end{aligned}$$

$$\begin{aligned}
\Delta_S[ins_Node] &= (insertNode(X, (C, F)),\\
&\quad insertNode(Z, (A, B)),\\
&\quad insertNode(Y, (X, F))\\
\Delta_S[del_Node] &= deleteNode(E)\\
\\
\Delta_S[data] &= (createDataElement(d),\\
&\quad createDataEdge(X, d, write),\\
&\quad createDataEdge(Y, d, read))
\end{aligned}$$

Es lässt sich erkennen, dass die in den jeweiligen Gruppen enthaltene Information äquivalent ist. Der Unterschied besteht darin, dass die hier erzeugten Gruppen die Auswirkungen der angewendeten Änderungen und nicht wie in [Rind04] die Änderungsoperationen selbst enthalten. Es handelt sich also lediglich um eine andere Darstellung der Information mit dem Vorteil, dass von den eigentlich angewendeten Änderungsoperationen abstrahiert wird.

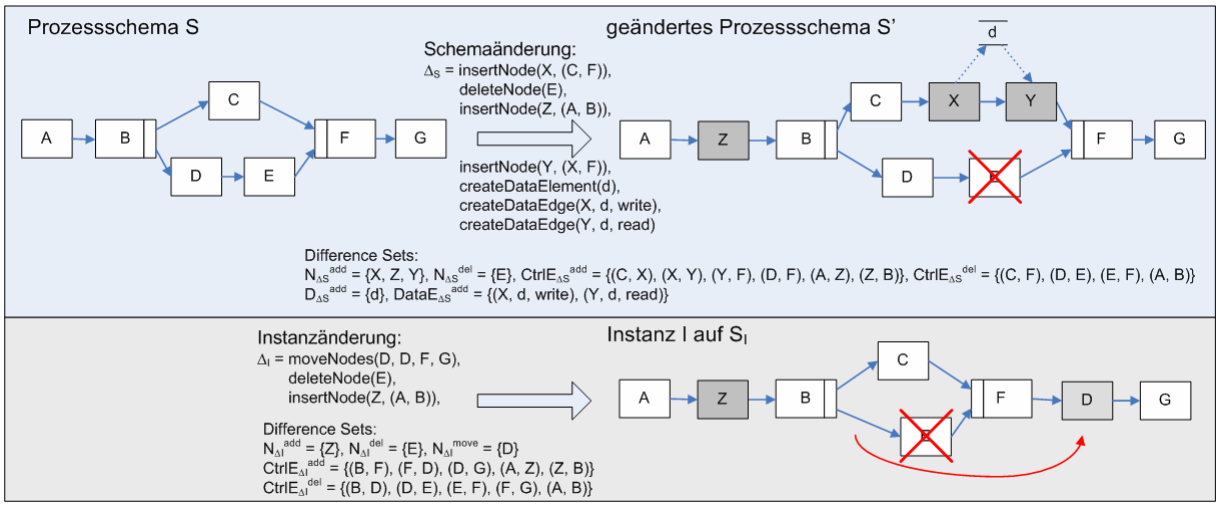


Abbildung 7.9 Δ_S und Δ_I inklusive resultierender *Difference Sets*

Um die Berechnung der *change projections* für das Beispiel aus Abbildung 7.9 zu vervollständigen werden noch die Gruppen für die Instanzänderung Δ_I bestimmt. In Gegenüberstellung zu den bereits berechneten Gruppen für die Schemaänderung Δ_S ergeben sich folgende Mengen:

$$\Delta_S[ins\_Node] = \{N_{\Delta S}^{add} = \{X, Z, Y\}, CtrlE_{\Delta S}^{add} = \{(C, X), (X, Y), (Y, F), (A, Z), (Z, B)\}, CtrlE_{\Delta S}^{del} = \{(C, F), (A, B)\}\}$$

$$\Delta_S[del\_Node] = \{N_{\Delta S}^{del} = \{E\}, CtrlE_{\Delta S}^{add} = \{(D, F)\}, CtrlE_{\Delta S}^{del} = \{(D, E), (E, F)\}\}$$

$$\Delta_S[data] = \{D_{\Delta S}^{add} = \{d\}, DataE_{\Delta S}^{add} = \{(X, d, write), (Y, d, read)\}$$

$$\Delta_I[ins\_Node] = \{N_{\Delta I}^{add} = \{Z\}, CtrlE_{\Delta I}^{add} = \{(A, Z), (Z, B)\}, CtrlE_{\Delta I}^{del} = \{(C, F), (A, B)\}\}$$

$$\Delta_I[del\_Node] = \{N_{\Delta I}^{del} = \{E\}, CtrlE_{\Delta I}^{add} = \{(D, F)\}, CtrlE_{\Delta I}^{del} = \{(D, E), (E, F)\}\}$$

$$\Delta_I[move\_Node] = \{N_{\Delta I}^{move} = \{D\}, CtrlE_{\Delta I}^{add} = \{(F, D), (D, G), (B, E)\}, CtrlE_{\Delta I}^{del} = \{(B, D), (D, E), (F, G)\}\}$$

Vergleicht man die einzelnen Gruppen von Δ_S und Δ_I miteinander, so resultieren die Beziehungen: $\Delta_I[ins\_Act] \subseteq \Delta_S[ins\_Act]$, $\Delta_I[del\_Act] = \Delta_S[del\_Act]$, $\Delta_S[move\_Act] \subseteq \Delta_I[move\_Act]$ und $\Delta_I[data] \subseteq \Delta_S[data]$.

Die Auswirkungen der Einfügeoperationen von Δ_I sind somit *subsumption equivalent* zu den Auswirkungen der Einfügeoperationen von Δ_S. Die Auswirkungen der Löschoperationen sind sogar bei beiden Änderungen identisch. Folglich ist Δ_I und Δ_S diesbezüglich *equivalent*. Weiterhin ist Δ_I bzgl. der Auswirkungen von Datenflussänderungen eine Teilmenge von Δ_S (*subsumption equivalent* $\Delta_I < \Delta_S$) und Δ_S eine Teilmenge von Δ_I bzgl. Verschiebeoperationen (*subsumption equivalent* $\Delta_S < \Delta_I$). Für die hier betrachtete *partially equivalent* Instanz lassen sich somit die automatischen Migrationsmechanismen der anderen Klassen verwenden.

Mit Hilfe der *change projections* kann also entschieden werden welche Instanzen der Klasse *partially equivalent* sich mit den Verträglichkeitstests und Migrationsstrategien der anderen Klassen automatisch verarbeiten lassen und welche nur mit Unterstützung des Anwenders migrierbar sind. Wie die *change projections* im Einzelnen berechnet werden, wird in Anhang F.1 (Berechnung der Änderungsprojektionen aus der Deltaschicht) erläutert.

7.2.3 Zusammenfassung

In diesem Abschnitt wurde beschrieben wie entschieden werden kann, zu welcher Klasse eine betrachtete Instanz gehört. Dabei wurde ausgehend von einem in [Rind04] vorgestellten hybriden Ansatz das erweiterte Deltaschicht-Konzept entwickelt. Dieses Konzept ermöglicht es, die für eine Klasseneinteilung notwendigen Ausgangsmengen ohne Betrachtung der Ausführungshistorie und mit maximaler Effizienz zu berechnen. Basierend auf diesen Mengen wurden für jede Klasse Algorithmen beschrieben und Kriterien aufgestellt mit denen die Klassenzugehörigkeit einer Instanz bestimmt werden kann. Dabei ist es im Einzelfall notwendig die aus der erweiterten Deltaschicht gewonnene Information durch weitere Berechnungen zu erweitern. Welche Mengen zur Bestimmung der Klassenzugehörigkeit benötigt werden, zeigt Tabelle 7.2 im Überblick:

<table>
<tr><td colspan="7">Notwendige Mengen zur Bestimmung der Klassenzugehörigkeit:</td></tr>
<tr><td></td><td>unbiased</td><td colspan="5">Biased</td></tr>
<tr><td></td><td></td><td>Disjoint</td><td>equivalent</td><td>Subsumption equivalent
($\Delta_S > \Delta_I$)</td><td>subsumption equivalent
($\Delta_S < \Delta_I$)</td><td>partially equivalent</td></tr>
<tr><td>Ausschließlich Deltaschicht</td><td>Delta existiert nicht</td><td colspan="4">Delta existiert</td><td></td></tr>
<tr><td>Difference Sets</td><td>-</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td rowspan="5">Die Zugehörigkeit zur Klasse partially equivalent ergibt sich automatisch, wenn eine Instanz keines der Zugehörigkeitskriterien der anderen Klassen erfüllt</td></tr>
<tr><td>Moved Nodes</td><td>-</td><td>-</td><td>-</td><td>✓</td><td>✓</td></tr>
<tr><td>AnchorIns und AnchorMove</td><td>-</td><td>-</td><td>-</td><td>✓</td><td>✓</td></tr>
<tr><td>AnchorGroupsAgg</td><td>-</td><td>-</td><td>-</td><td>✓</td><td>✓</td></tr>
<tr><td>OrderAgg</td><td>-</td><td>-</td><td>-</td><td>✓</td><td>✓</td></tr>
<tr><td>Besonderheiten:</td><td></td><td>Kontextzer-störende Änderungen</td><td></td><td>Anchor Sets, Anchor Groups, Order Sets nur notwendig, falls eine Erkennung mit den Difference Sets auf Grund kontextabhängiger Änderungen nicht möglich ist</td><td>Anchor Sets, Anchor Groups, Order Sets nur notwendig, falls eine Erkennung mit den Difference Sets auf Grund kontextabhängiger Änderungen nicht möglich ist</td><td>change projections, zur Erkennung von Teil-mengenbeziehungen auf Änderungstyp-ebene</td></tr>
</table>

Tabelle 7.2 Notwendige Mengen zur Berechnung einer Klassenzugehörigkeit

Die Mechanismen zur Klasseneinteilung bilden die Grundlage für das weitere Vorgehen bei der Migration von Instanzen. Die Klasseneinteilung ist dabei insofern von entscheidender Bedeutung, da – wie bereits in den Abschnitten 3.4.3 und 3.4.4 gezeigt – für jede Klasse unterschiedliche Verträglichkeitskriterien und Migrationsstrategien existieren. Mit der Klasseneinteilung ist das Änderungsrahmenwerk nun in der Lage, für jede beliebige Instanz, die für eine erfolgreiche Migration anzuwendenden Algorithmen zu bestimmen.

7.3 Strukturelle Konfliktbestimmung

Wird eine Instanz *I* im Rahmen der Klasseneinteilung als *biased disjoint* erkannt, so muss neben der zustandsbasierten auch die strukturelle Verträglichkeit von *I* geprüft werden. Dies liegt daran, dass es bei der Migration zustandsverträglicher Instanzen diesen Typs zu unerwünschten Wechselwirkungen

zwischen den Schema- und den instanzspezifischen Änderungen kommen kann. Die dabei auftretenden Konfliktsituationen wurden in Abschnitt 3.4.4.1 vorgestellt. [Rind04] beschreibt hierzu entsprechende Konflikttests die sich allerdings alle von der Semantik einiger fest definierter Änderungsoperationen ableiten. Eine Umsetzung im Änderungsrahmenwerk ist somit aufgrund des Plug-In-Konzeptes nicht möglich. Nachträglich eingefügte Änderungsoperationen könnten mit bereits vorhandenen Operationen Konflikte verursachen, was durch die auf bereits vorhandene Änderungsoperationen beschränkten Tests nicht erkannt werden kann. Es ist deshalb notwendig eine andere Lösung zu finden, welche die strukturelle Korrektheit auch im Falle neu hinzukommender Änderungsoperationen gewährleistet.

Als naive Lösung ist es möglich, die für eine neu einzufügende Änderungsoperation notwendigen Konflikttests bei der jeweiligen Operation selbst zu hinterlegen. Tritt eine solche Änderung bei der Migration einer *biased disjoint* Instanz auf, so kann deren Konflikttest aufgerufen und somit eine mögliche unerwünschte Wechselwirkung identifiziert werden. Aufgrund des einheitlichen Plug-In-Interfaces hat diese Vorgehensweise aber zur Folge, dass auch Änderungsoperationen die keine Konflikte verursachen, eine entsprechende *Dummy*-Schnittstelle bereitstellen müssen. Bezüglich Benutzerfreundlichkeit und Effizienz eine unbrauchbare Lösung.

Ein geeigneter Mechanismus muss folglich von den eigentlichen konfliktverursachenden Änderungsoperationen abstrahieren und Konflikte bereits auf einer tieferen Ebene identifizieren. Die geforderte Ebene findet sich in Form der Änderungsprimitiven. Diese bilden die Grundlage für alle Änderungsoperationen und manipulieren direkt die interne Repräsentation von Schemata und Instanzen. Dies bedeutet, dass sich die Auswirkungen aller angewendeten Änderungsoperationen (auch der nachträglich eingefügten) in den Deltaschichten des Schemas bzw. des instanzspezifischen Schemas widerspiegeln. Beim Entwurf änderungsoperationsunabhängiger Konflikttests ist also genau an diesen Deltaschichten anzusetzen.

In den folgenden Abschnitten werden die durch konkurrierende Anwendung von Schema- und Instanzänderungen auftretenden Konflikte beschrieben und entsprechende, auf der Deltaschicht basierende, Tests entwickelt.

7.3.1 *Deadlocks*

Die erste Situation, bei der ein Konflikt durch Wechselwirkungen zwischen den auf dem Schema ausgeführten Änderungen und den Auswirkungen einer instanzspezifischen Änderung auftreten kann ist die konkurrierende Anwendung von Sync-Kanten. Hierbei ist ein Migrieren der entsprechenden Instanz auf das geänderte Schema nicht möglich, wie Abbildung 7.10 zeigt.

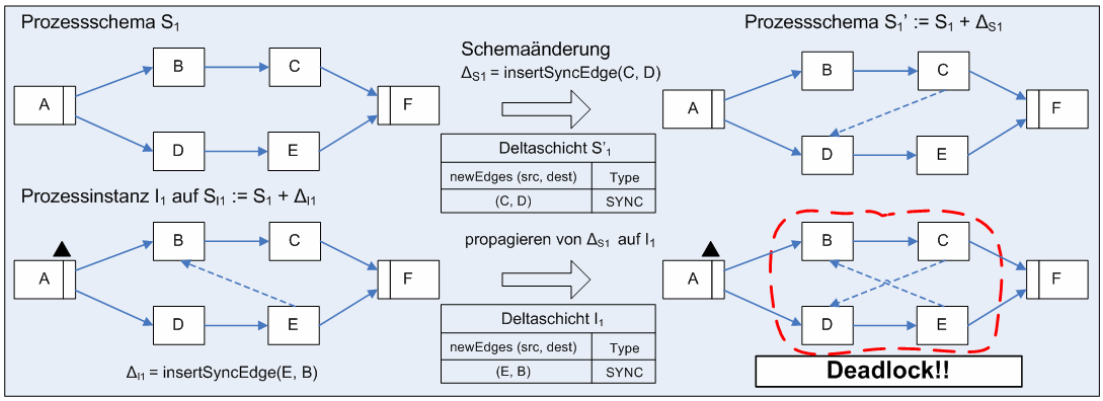


Abbildung 7.10 Konkurrierende Anwendung von Sync-Kanten

Durch das Anwenden der Änderungsoperationen *insertSyncEdge(C, D)* auf Schemaebene und *insertSyncEdge(E, B)* auf Instanzebene entsteht bei der Propagation ein *Deadlock* durch den Zyklus $C \rightarrow D \rightarrow E \rightarrow B \rightarrow C$. Dabei ist es für die jeweilige Deltaschicht unerheblich, ob eine Sync-Kante direkt durch die Änderungsoperation *insertSyncEdge* oder durch eine komplexere Änderungsoperation wie *insertNodeBetweenNodeSets* eingefügt worden ist. Für den hier vorgestellten Konflikttest ist die einzig wichtige Information, dass die Deltaschicht einen entsprechenden Eintrag für jede neu eingefügte Sync-Kante besitzt.
Damit lässt sich der folgende Konflikttest definieren:

Test 1 (Deadlocks)

Durch das Propagieren von Δ_S auf eine geänderte Instanz ($S + \Delta_I$) entsteht genau dann ein *Deadlock* verursachender Zyklus, wenn jeweils für ein Paar neu eingefügter Sync-Kanten aus der Deltaschicht von *S'* (Sync-Kante 1) und der Deltaschicht von *I* (Sync-Kante 2) die folgenden zwei Punkte erfüllt sind:

1. Das Ziel von Sync-Kante 1 ist Ursprung oder Vorgänger des Ursprungs von Sync-Kante 2 bzgl. Kontroll- und Sync-Kanten und
2. Das Ziel von Sync-Kante 2 ist Ursprung oder Vorgänger des Ursprungs von Sync-Kante 1 bzgl. Kontroll- und Sync-Kanten

In den meisten Fällen lassen sich die angegebenen Vorgängerbeziehungen ohne weiteres auf dem gemeinsamen Ursprungsschema *S* bestimmen. Enthalten allerdings die Deltaschichten neben den neu eingefügten Sync-Kanten zusätzlich neu eingefügte Knoten (und somit auch neu eingefügte Kontextkanten), so besteht die Möglichkeit, dass ein solcher Knoten Ursprung oder Ziel einer Sync-Kante ist. Dadurch kann die Vorgängerbeziehung ohne weitere Bemühungen nicht auf *S* bestimmt werden. Um dieses Problem zu umgehen, wurden in [Rind04] so genannte *Graph Reduction Rules* definiert. Im Zuge der Anwendung dieser Regeln wird jedes Sync-Kanten-Tupel[12], deren Ursprungs- oder Zielknoten neu eingefügt wurde, durch ein neu berechnetes ausschließlich aus bereits in *S* vorhandenen Knoten bestehendes Tupel ersetzt. Die Berechnung ist mit der Bestimmung der Ankerknoten (vgl. Abschnitt 7.2.2.4) vergleichbar. Ein Tupel berechnet sich folgendermaßen: Befindet sich ein Ursprungsknoten nicht in *S*, so suche mit Hilfe der in der Deltaschicht eingefügten Kontextkanten nach einem bereits in *S* vorhandenen Vorgänger. Befindet sich ein Zielknoten nicht in *S*, so suche analog einen in *S* bereits vorhandenen Nachfolger. Hat man die entsprechenden Knoten gefunden, werden die Knoten in den ursprünglichen Tupeln ersetzt. Danach ist ein Bestimmen der Vorgängerbeziehungen auf *S* problemlos möglich.
Aus welchem Grund durch die Neuberechnung solcher Tupel keine Verfälschung des Ergebnisses resultiert wird in [Rind04] genau erläutert. Auf eine weitere Erklärung wird an dieser Stelle verzichtet.

Ein detaillierter Algorithmus für die Durchführung des Konflikttests und die Anwendung der *Graph Reduction Rules* befindet sich in Anhang E (Algorithmen *für biased disjoint Instanzen)*. Hier soll lediglich die Funktionsweise anhand des Beispiels aus Abbildung 7.10 verdeutlicht werden:
Aus den Deltaschichten von S_1' und I_1 erhält man die Sync-Kanten Tupel (*C*, *D*) und (*E*, *B*). Da alle in den Tupeln enthaltenen Knoten im ursprünglichen Schema *S* enthalten sind, erübrigt sich in diesem Fall eine Anwendung der *Graph Reduction Rules*. Die Bestimmung der Vorgängerbeziehungen beginnt mit der Betrachtung des Zielknotens *D* der Sync-Kante aus S_1' und des Startknotens *E* der Sync-Kante aus I_1. Wie anhand des Schemas *S* berechenbar und aus Abbildung 7.10 leicht ersichtlich,

[12] Tupel bestehend aus dem Ursprung und dem Ziel einer Sync-Kante

ist *D* ein Vorgänger von *E*, womit Punkt 1 erfüllt ist. Gleiches gilt für den Ursprung der Kante (*E*, *B*). Auch hier ist der Zielknoten B ein Vorgänger des Ursprungs der Kante (*C*, *D*), womit auch Punkt 2 erfüllt wird. Weitere Sync-Kanten wurden nicht eingefügt, wodurch weitere Tests entfallen. Insgesamt erhält man also ein negatives Ergebnis. Folglich ist die Instanz I_1 nicht mit dem Schema S_1 verträglich und kann nicht migriert werden.

7.3.2 Fehlende Eingabedaten (*missing input parameter*) und Überschreiben noch nicht gelesener Daten (*lost update*)

Ein weiteres Problem kann sich durch die konkurrierende Manipulation von Datenkanten ergeben. Abbildung 7.11 zeigt einen solchen Fall. Bei der Schemaänderung werden die Schreibkante (*A*, *d*, '*write*') und die Lesekante (*B*, *d*, '*read*') gelöscht. Die Instanzänderung löscht die Schreibkante (*C*, *d*, '*write*'). Beide Änderungen sind aus der jeweiligen Deltaschicht unmittelbar ersichtlich und können getrennt betrachtet auf das Originalschema *S* korrekt angewendet werden. Erst die Kombination führt dazu, dass keine Aktivität das Datenelement *d* schreibt und somit keine Eingabedaten für *D* vorhanden sind. Eine Migration von Instanz I_2 ist folglich nicht möglich.

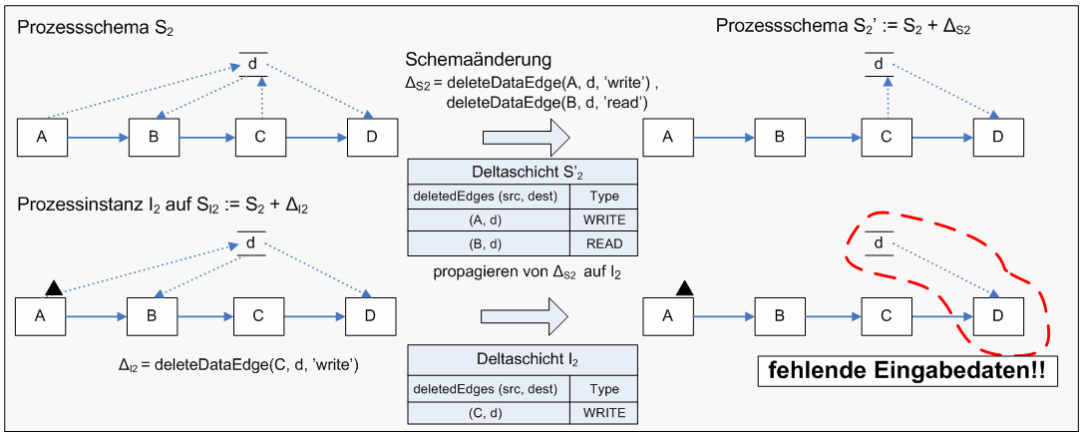


Abbildung 7.11 Fehlende Eingabedaten

Ein in [Rind04] für diesen Fall vorgestellter Test ist (neben der eingangs erwähnten mangelnden Abstraktion) hinsichtlich der geforderten Effizienz problematisch. Mit den dort beschriebenen Kriterien kann ein potentieller Konflikt zwar aufgedeckt werden kann, in Einzelfällen ist jedoch eine teure Materialisierung des Graphen erforderlich. Der hier vorgestellte Test kommt unter Ausnutzung der in den Deltaschichten und dem ursprünglichen Graphen gespeicherten Informationen ohne Materialisierung aus. Obgleich der Test effizienter ist, gestaltet er sich in der Beschreibung etwas aufwendiger, weshalb einige vorausgehende Mengendefinitionen notwendig sind.

Bestimmung von Datenelementen mit potentiellen Konflikten

Zuerst müssen alle Datenelemente bestimmt werden, bei denen es durch neu eingefügte oder gelöschte Datenkanten zu potentiellen Konflikten kommen kann. Abbildung 7.12 zeigt die konfliktverursachenden Kantenkombinationen. Die Pfeile zeigen dabei ausgehend von einer bestimmten Kante (bzw. Kantenmenge) diejenigen Kanten, mit denen es zu einem Konflikt kommen kann. Für eingefügte Schreibkanten des geänderten Schemas bedeutet dies beispielsweise, dass es sowohl mit eingefügten und gelöschten Schreibkanten als auch mit eingefügten Lesekanten der geänderten Instanz zu Konflikten kommen kann. Die Betrachtung ausgehend von anderen Kanten verhält sich analog.

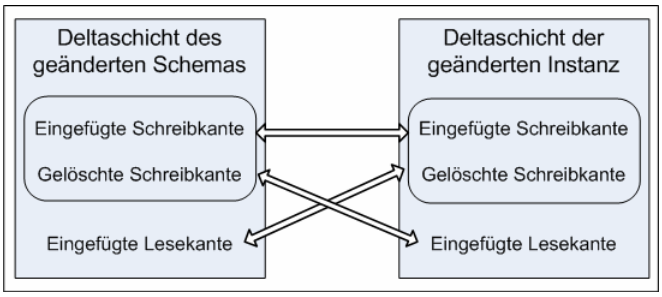


Abbildung 7.12 Konfliktverursachende Kantenkombinationen

Bestimmung der Mengen $Set_{Readers}$ und $Set_{Writers}$

In einem zweiten Schritt werden für jedes der identifizierten Datenelemente alle lesenden und schreibenden Datenkanten bestimmt. Dies schließt neben den entsprechenden Datenkanten aus beiden Deltaschichten auch diejenigen Datenkanten ein, die bereits im ursprünglichen Schema S vorhanden waren. Dabei bereinigen sich die resultierenden Mengen (Lesekantenmenge $Set_{Readers}$ und Schreibkantenmenge $Set_{Writers}$) automatisch, d.h. eine in S vorhandene Datenkante, die in einer Deltaschicht gelöscht wird, tritt nicht in der entsprechenden Lese- bzw. Schreibkantenmenge auf.
Mit diesen Kantenmengen lässt sich ein Konflikttest definieren, mit dem die beiden Konflikttypen *lost update* und *missing input data* erkannt werden können:

Test 2 (Fehlende Eingabedaten und Überschreiben nicht gelesener Daten)

Das Propagieren von Δ_S auf $S + \Delta_I$ führt weder zum Verlust von Eingabedaten noch zum Überschreiben noch nicht gelesener Daten (*lost updates*), falls die folgenden zwei Prüfungen mit den entsprechenden Kanten aus $Set_{Readers}$ und $Set_{Writers}$ erfolgreich durchgeführt werden können:

1. *lost update*: Prüfe für jede Schreibkante eines Datenelements, ob eine weitere Schreibkante auf dieses Element zugreift. Ist dies der Fall, so muss eine Lesekante zwischen den beiden schreibenden Knoten existieren, da dass zugrundeliegende Prozessmodell das Überschreiben eines nicht gelesenen Datenelements verbietet (siehe Abschnitt 3.1.1 Definition 1).
2. *missing input parameter*: Prüfe für jede Lesekante eines Datenelements, ob eine entsprechende Schreibkante existiert. Dabei ist es wichtig, dass sich der schreibende Knoten im Kontrollfluss vor dem lesenden Knoten befindet, da nur so ein korrekter Datenfluss zustande kommt.

Es besteht die Möglichkeit, dass betrachtete Datenkanten als Ursprung einen Knoten besitzen, der noch nicht im ursprünglichen Schema vorhanden war. Dadurch ist eine Prüfung der Reihenfolgebeziehung nicht möglich, weshalb auch hier ggf. die *Graph Reduction Rules* zum Einsatz kommen.

7.3.3 Überlappende Kontrollblöcke

Der Test zur Bestimmung eines Konfliktes durch überlappende Kontrollblöcke (vgl. Abbildung 7.13), in Folge des konkurrierenden Einfügens von Kontrollblöcken ist relativ einfach zu definieren. Der nachfolgende Konflikttest orientiert sich dabei an dem in [Rind04] vorgestellten Test zur Bestimmung überlappender Schleifenblöcke. Die dort definierte Bedingung geht allerdings auf Grund anderer Voraussetzungen nicht weit genug, weshalb eine komplette Neudefinition notwendig wird.

Test 3 (Überlappende Kontrollblöcke)

$(S + \Delta_I) + \Delta_S$ enthält keine überlappenden Kontrollblöcke, wenn die folgende Bedingung erfüllt ist:

- Befindet sich der Anfangsknoten eines in die Deltaschicht von *S'* bzw. *I* hinzugefügten Kontrollblocks zwischen dem Anfangsknoten und dem Endknoten eines in die Deltaschicht von *I* bzw. *S'* hinzugefügten Kontrollblocks, so muss sich der Endknoten des Blocks von *S'* bzw. *I* ebenfalls dazwischen befinden.

Auch hier werden wieder die *Graph Reduction Rules* notwendig, falls der Kontext der zu betrachtenden Knoten ebenfalls aus neu eingefügten Knoten besteht.

Ein Algorithmus zur Umsetzung des vorgestellten Tests befindet sich in Anhang E (Algorithmen *für biased disjoint Instanzen)*. Die Funktionsweise verdeutlicht das Beispiel aus Abbildung 7.13:

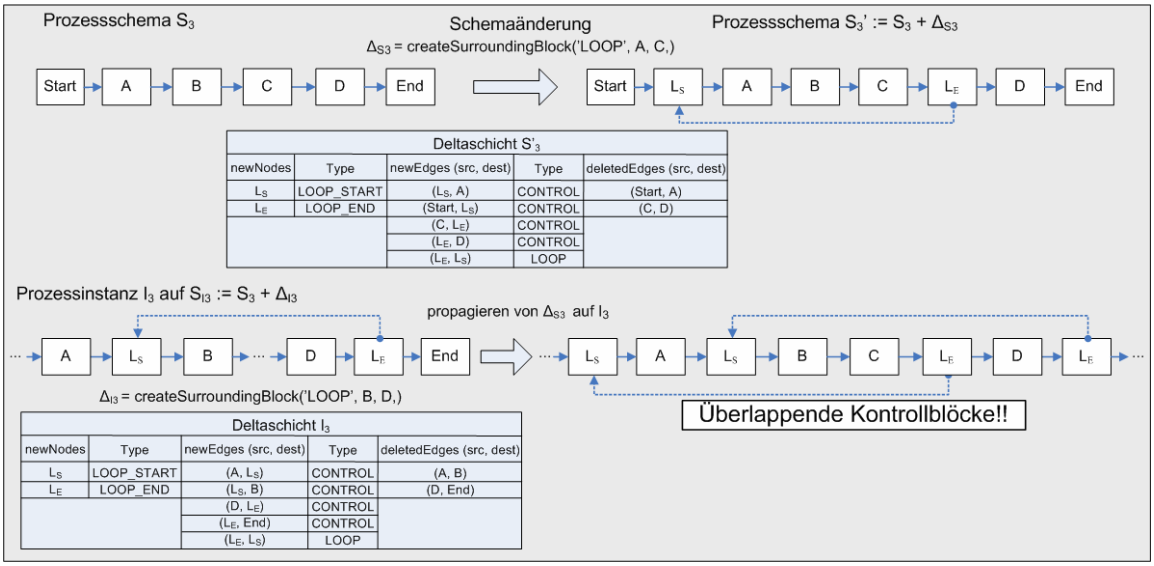


Abbildung 7.13 Überlappende Kontrollblöcke

Das konkurrierende Einfügen der beiden Schleifenblöcke führt zu überlappenden Kontrollblöcken. Dies verstößt gegen die in Abschnitt 3.1.1 definierte strukturelle Korrektheit. Folglich kann die Instanz I_3 nicht auf das geänderte Schema S_3' migriert werden. Gemäß dem im Anhang E (Algorithmen *für biased disjoint Instanzen)* beschriebenen Algorithmus kann dies folgendermaßen berechnet werden:
Zuerst werden separat aus beiden Deltaschichten anhand der neu eingefügten Kanten alle Knoten extrahiert und in Tupeln gespeichert, die Nachfolger eines Schleifenstart- bzw. Verzweigungsknotens oder Vorgänger eines Schleifenend- bzw. Vereinigungsknotens sind. Im Beispiel sind dies für *S'* die Knoten *A* und *C* und für *I* die Knoten *B* und *D*, woraus sich die Tupel (*A, C*) und (*B, D*) ergeben. Die Knoten sind alle bereits in *S* vorhanden, weshalb sich die Anwendung der *Graph Reduction Rules* erübrigt. Nun kann jeweils für ein Paar von Tupeln aus *S'* und *I* geprüft werden, ob sich beide Knoten eines Tupels innerhalb der Knoten des anderen Tupels befindet. Es befinden sich weder *A* und *C* zwischen (*B*, *D*) noch ist dies umgekehrt der Fall. Der Test erkennt folglich einen Konflikt, weswegen ein Migrieren der vorliegenden Instanz nicht möglich ist.

7.3.4 Sync-Kanten in oder aus Schleifenblöcken

Die letzte konfliktverursachende Konstellation entsteht durch das konkurrierende Einfügen von Sync-Kanten und Schleifenblöcken. Abbildung 7.14 zeigt einen solchen Fall. Auf die explizite Darstellung

der zugehörigen Deltaschichten wurde in diesem Fall verzichtet, da diese leicht aus den Deltaschichten der Beispiele zu Test 1 und 3 abgeleitet werden können.

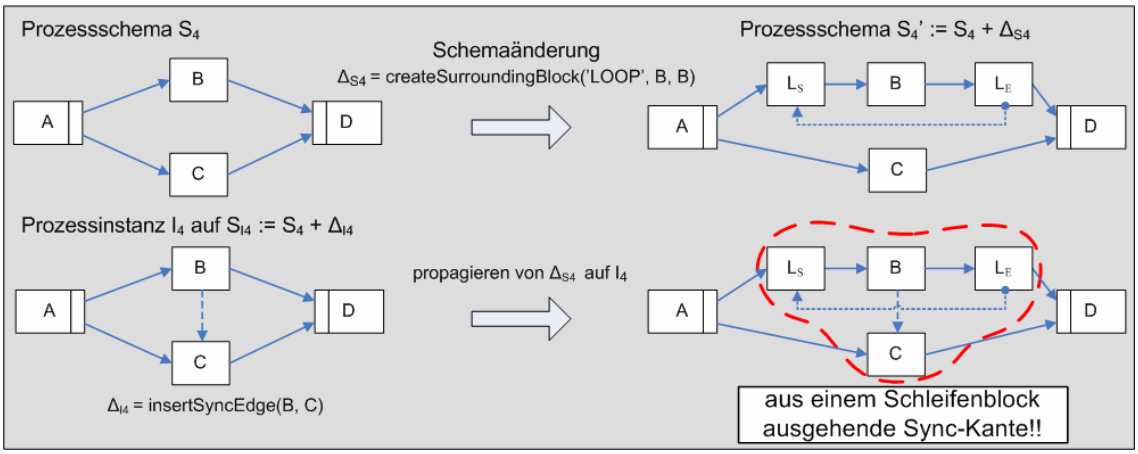


Abbildung 7.14 Sync-Kanten in oder aus Schleifenblöcken

Die im Zuge einer Instanzänderung eingefügte Sync-Kante (B, C) durchbricht die Grenze des in die Deltaschicht von S eingefügten Schleifenblocks. Diese führt nach Definition 1 zu einem inkorrekten Graphen. Deshalb ist eine Migration von Instanz I_4 auf das geänderte Schema S_4' nicht zulässig.
Ein allgemeiner Konflikttest definiert sich folgendermaßen:

Test 4 (Sync-Kanten in oder aus Schleifenblöcken)

$(S + \Delta_I) + \Delta_S$ enthält keine aus Schleifenblöcken ein- oder ausgehende Sync-Kanten wenn die folgende Bedingung erfüllt ist:

- Befindet sich der Startknoten/Zielknoten einer in die Deltaschicht von *S'* bzw. *I* eingefügten Sync-Kante innerhalb eines in die Deltaschicht von *I* bzw. *S'* eingefügten Schleifenblocks, so muss dies auch für dessen Zielknoten/Startknoten gelten.

Da auch bei diesem Test Reihenfolgebeziehungen auf dem ursprünglichen Schema zu bestimmen sind, kommen auch hier – wo erforderlich – die bekannten *Graph Reduction Rules* zum Einsatz.

Ein detaillierter Algorithmus findet sich wiederum im Anhang E (Algorithmen *für biased disjoint Instanzen)*. Angewendet auf das betrachtete Beispiel ergibt sich folgender Ablauf:
Zuerst werden analog zur Vorgehensweise bei Test 3 aus dem Nachfolgerknoten des Schleifenstartknotens und aus dem Vorgängerknoten des zugehörigen Schleifenendknotens Tupel erzeugt. Es ergibt sich lediglich das Tupel (*B*, *B*) für *S'*, weshalb im Folgenden die Betrachtung der Sync-Kanten aus der Deltaschicht von *I* genügt. Es wird nun für jede Sync-Kante aus dieser Deltaschicht geprüft, ob sich ihr Start- oder Endknoten im Kontrollfluss zwischen den beiden Knoten des Tupels befindet. Dies trifft im betrachteten Beispiel lediglich für den Startknoten *B* zu. Abschließend muss also nur noch getestet werden, ob sich der zugehörige Zielknoten der Sync-Kante (*B*, *C*) zwischen den Knoten des betrachteten Tupels befindet. Dies ist allerdings nicht der Fall, weshalb der Test anzeigt, dass eine Migration von *I* auf *S'* nicht möglich ist.

7.3.5 Zusammenfassung

In diesem Abschnitt wurde detailliert beschrieben, wie die strukturelle Korrektheit bei der Migration von *biased disjoint* Instanzen gewährleistet wird. Dazu wurden die Situationen erläutert, bei denen es

durch die konkurrierende Anwendung von Schema- und Instanzänderungen zu strukturellen Konflikten kommt und Tests entwickelt, mit denen diese erkannt werden. Dabei besitzen die Tests die Besonderheit, dass auch Änderungsoperationen berücksichtigt werden, die bisher noch nicht im Änderungsrahmenwerk vorhanden sind. Dies gelingt dadurch, dass die Konflikttests nur auf Daten zurückgreifen, die in der Deltaschicht enthalten sind. Da diese Schicht wiederum nur von einer festen Menge an Änderungsprimitiven manipuliert wird und alle (auch zukünftig in das Änderungsrahmenwerk eingebrachte) Änderungsoperationen auf diesen Primitiven beruhen erreicht man eine Abstraktion von den tatsächlich angewendeten Änderungsoperationen. Dies hat zur Folge, dass bei *biased disjoint* Instanzen auch die von nachträglich eingefügten Operationen verursachten Konflikte zuverlässig erkannt werden können.

7.4 Bias-Berechnung bei Instanzen der Klasse subsumption equivalent ($\Delta_S < \Delta_I$)

Im Gegensatz zu den Instanzen der Klasse *biased disjoint* müssen bei Instanzen der Klasse *subsumption equivalent*, bei denen die Schemaänderungen eine Teilmenge der Instanzänderungen sind ($\Delta_S < \Delta_I$), weder strukturelle noch zustandsbasierte Verträglichkeit geprüft werden. Weiterhin entfällt eine nachträgliche Zustandsanpassung, weil das für Laufzeitaspekte relevante instanzspezifische Schema I (bzw. S_I) bei der Migration auf das veränderte Schema S' nicht geändert wird. Da die Instanz I aber gegenüber S' noch weitere Änderungen enthält, weicht I auch nach der Migration noch von S' ab. Dieses *Bias* zwischen S' und I muss berechnet und daraus die neue Deltaschicht von I erzeugt werden. Erst dann lässt sich eine Instanz korrekt auf das geänderte Schema S' umhängen. Als Ergebnis erhält man eine Instanz, die das Schema S' referenziert und deren Deltaschicht exakt die Änderungen gegenüber S' widerspiegelt.
Ein in [Rind04] angegebener Algorithmus betrachtet zur Berechnung des *Bias* konkrete Operationen aus den Änderungshistorien von S' und I. Erweiterungen um neue Änderungsoperationen erfordern somit aufwendige Anpassungen des Algorithmus.
Um die Abhängigkeit von den angewendeten Änderungsoperationen zu umgehen, verwendet das in diesem Abschnitt entwickelte Konzept ausschließlich die Informationen aus S_I, S' und den Deltaschichten bzw. den daraus ableitbaren *Difference Sets* von I und S'. Dadurch ist garantiert, dass auch die Auswirkungen zukünftiger, in das System integrierter Änderungsoperationen ohne zusätzlichen Aufwand korrekt verarbeitet werden können.

Da die für eine *Bias*-Berechnung notwendigen Mechanismen davon abhängen, ob Δ_I und/oder Δ_S kontextabhängige Änderungen enthalten, wird im Folgenden zwischen der *Bias*-Berechnung ohne (Abschnitt 7.4.1) und der *Bias*-Berechnung bei Auftreten kontextabhängiger Änderungen unterschieden (Abschnitt 7.4.2). Wie mit den darin entwickelten Algorithmen das *Bias* für ein konkretes Beispiel berechnet werden kann, zeigt Abschnitt 7.4.3.

7.4.1 Ohne kontextabhängige Änderungen

Die *Bias*-Berechnung ohne kontextabhängige Änderungen wird angewendet, wenn weder die Schemaänderung Δ_S noch die Instanzänderung Δ_I kontextabhängige Änderungen enthalten. Dies ist immer genau dann der Fall, wenn bereits bei der Klasseneinteilung festgestellt wird, dass alle *Difference Sets* von Δ_S eine Teilmenge der *Difference Sets* von Δ_I sind (vgl. Abschnitt 7.2.2.4). Die Berechnung des *Bias* ist dadurch relativ einfach durchzuführen: Wie Abbildung 7.15 zeigt, ergibt sich das *Bias* durch Differenzmengenbildung zwischen den neu eingefügten und gelöschten Elementen von I und S'.

Formal:

Bias := Elemente der *Difference Sets* von *I* \ Elemente der *Difference Sets* von *S'*

Man prüft also für jedes Element aus den *Difference Sets* von *I*, ob es in den entsprechenden *Difference Sets* von *S'* vorhanden ist. Sollte dies für ein Element nicht der Fall sein, so wird es in dem zu bestimmenden *Bias* gespeichert.

Betrachtet man beispielsweise in Abbildung 7.15 die neu hinzugekommenen Kanten von *I* ((*F*, *X*), (*X*, *G*), (*D*, *F*)), so ergibt sich die Differenzmenge $CtrlE_{\Delta I \setminus \Delta S}^{add} = \{(F, X), (X, G)\}$, indem man die Menge $CtrlE_{\Delta I}^{add}$ um die Kante bereinigt, die auch in *S'* eingefügt wurde (*D*, *F*). Formal: $CtrlE_{\Delta I \setminus \Delta S}^{add} := CtrlE_{\Delta I}^{add} \setminus CtrlE_{\Delta S}^{add}$. Analog definieren sich auch die anderen Differenzmengen $N_{\Delta I \setminus \Delta S}^{add}$, $N_{\Delta I \setminus \Delta S}^{del}$, $N_{\Delta I \setminus \Delta S}^{move}$, $CtrlE_{\Delta I \setminus \Delta S}^{del}$ und die Mengen für Sync-Kanten, Daten-Kanten und -Elemente sowie für Attribute.

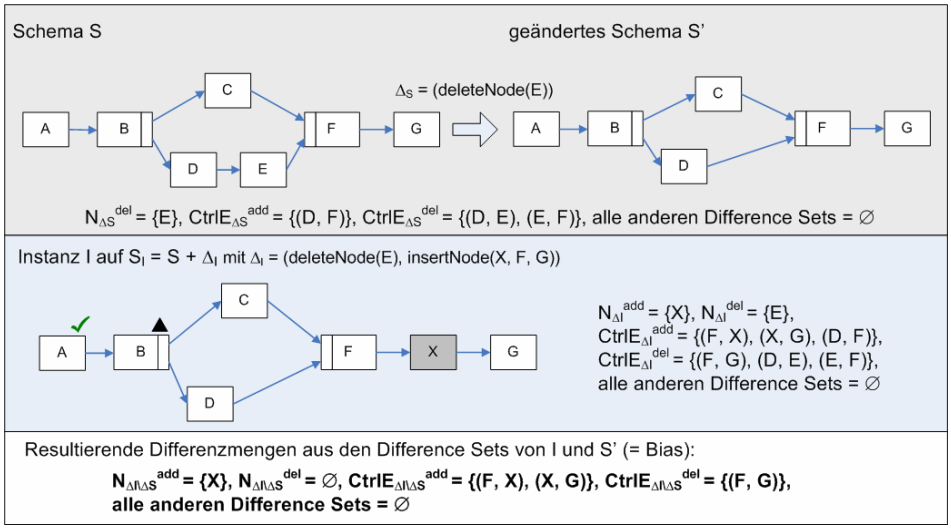


Abbildung 7.15 Bias bei Instanzen ohne kontextabhängige Änderungen

7.4.2 Bei kontextabhängigen Änderungen

Zur *Bias*-Berechnung bei Instanzen mit kontextabhängigen Änderungen werden in einem ersten Schritt ebenfalls die Differenzmengen zwischen allen neu hinzugefügten und gelöschten Elementen von *I* und *S'* gebildet. Da bei kontextabhängigen Änderungen aber keine Teilmengenbeziehung zwischen den eingefügten und/oder den gelöschten Kontrollkanten besteht, führt die Berechnung der Differenzmengen $CtrlE_{\Delta I \setminus \Delta S}^{add}$ und/oder $CtrlE_{\Delta I \setminus \Delta S}^{del}$ zu falschen Ergebnissen.

In Abbildung 7.16 ergibt sich diese Situation für $CtrlE_{\Delta I \setminus \Delta S}^{del}$. Aus der Differenzmengenberechnung $CtrlE_{\Delta I}^{del} \setminus CtrlE_{\Delta S}^{del}$ resultiert die leere Menge. Korrekt wäre allerdings (*X*, *F*), was bzgl. zu löschenden Kanten exakt die gewünschte Abweichung von *I* gegenüber *S'* widerspiegelt. Ein weiteres Beispiel, bei dem auch ein Berechnen der Menge $CtrlE_{\Delta I \setminus \Delta S}^{add}$ nicht über $CtrlE_{\Delta I}^{add} \setminus CtrlE_{\Delta S}^{add}$ möglich ist, zeigt Abbildung 7.20.

Eine korrekte Kontrollkantenmenge lässt sich bei kontextabhängigen Änderungen also nicht mit Hilfe der Differenzmengenbildung erzeugen. Wie die für ein korrektes *Bias* noch fehlenden Kontrollkantenmengen trotzdem berechnet werden können, wird in den folgenden Abschnitten detailliert beschrieben.

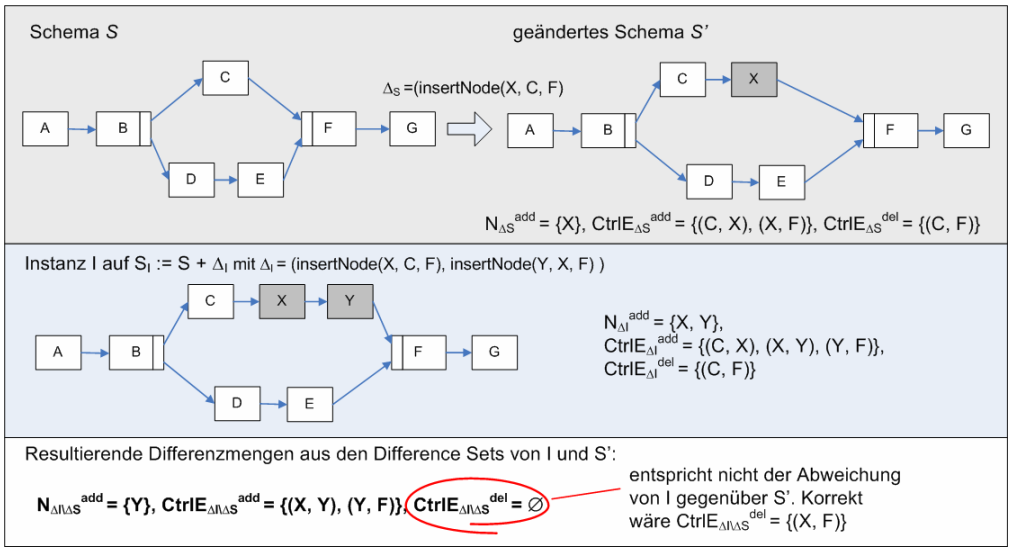


Abbildung 7.16 Differenzmengenbildung bei kontextabhängigen Änderungen

7.4.2.1 Berechnung der Kontrollkantenmengen

Als Grundlage für die Berechnung von $CtrlE_{\Delta I \setminus \Delta S}^{add}$ und $CtrlE_{\Delta I \setminus \Delta S}^{del}$ dienen die aus der Differenzmengenberechnung resultierenden Mengen $N_{\Delta I \setminus \Delta S}^{add}$, $N_{\Delta I \setminus \Delta S}^{del}$, $N_{\Delta I \setminus \Delta S}^{move}$ sowie I (bzw. S_I) und S'. Um das Verfahren übersichtlicher zu gestalten kommen zwei Hilfsmengen $TempCtrlE_{\Delta I \setminus \Delta S}^{add}$ und $TempCtrlE_{\Delta I \setminus \Delta S}^{del}$ zum Einsatz. Diese Mengen enthalten Kanten die während der Berechnung als potentiell in $CtrlE_{\Delta I \setminus \Delta S}^{add}$ bzw. in $CtrlE_{\Delta I \setminus \Delta S}^{del}$ einzufügende Tupel erkannt worden sind, eine abschließende Beurteilung zum Zeitpunkt der Berechnung aber noch nicht möglich war.
Bei der Berechnung der Mengen $CtrlE_{\Delta I \setminus \Delta S}^{add}$ und $CtrlE_{\Delta I \setminus \Delta S}^{del}$ werden im Folgenden die Ausgangsmengen $N_{\Delta I \setminus \Delta S}^{add}$, $N_{\Delta I \setminus \Delta S}^{del}$ und $N_{\Delta I \setminus \Delta S}^{move}$ getrennt betrachtet:

Betrachtung von $N_{\Delta I \setminus \Delta S}^{add}$:

1. Für alle Knoten N aus $N_{\Delta I \setminus \Delta S}^{add}$ suche Kanten (src, $dest$) in $CtrlE_{\Delta I}^{add}$ mit $src = N$ oder $dest = N$. Ist eine solche Kante gefunden, so kann sie direkt in die Menge $CtrlE_{\Delta I \setminus \Delta S}^{add}$ eingefügt werden, da eine solche Kante, bedingt durch die Konstruktion von $N_{\Delta I \setminus \Delta S}^{add}$, nicht in S' vorhanden sein kann.
2. Für jeden eingefügten Knoten N muss die an der Einfügestelle gelegene Kante (bzw. Kanten, falls N ein *Split*- oder *Join*-Knoten ist) gelöscht werden. Ob es sich bei einer solchen Kante allerdings um eine gegenüber S' tatsächlich zu löschende Kante handelt kann an dieser Stelle noch nicht abschließend beurteilt werden. Folglich wird die Kante in die Hilfsmenge $TempCtrlE_{\Delta I \setminus \Delta S}^{del}$ eingefügt. Die mangelnde Aussagesicherheit ergibt sich dadurch, dass es sich beim Vorgänger und oder Nachfolger von N ebenfalls um Knoten aus $N_{\Delta I \setminus \Delta S}^{add}$, $N_{\Delta I \setminus \Delta S}^{del}$ oder $N_{\Delta I \setminus \Delta S}^{move}$ handeln könnte. Wäre dies der Fall, so kann eine solche Kante nicht in S' vorkommen und darf folglich auch nicht in $CtrlE_{\Delta I \setminus \Delta S}^{del}$ eingefügt werden.
   Abbildung 7.17 verdeutlicht dies anhand der Graphausschnitte zweier Instanzen. Für die Instanz I_1 entspricht die potentiell zu löschende der tatsächlich zu löschenden Kante, da es sich lediglich um einen einzigen gegenüber S' eingefügten Knoten handelt. Bei Instanz I_2 hingegen, ergibt sich die oben beschriebene Vorgänger-/Nachfolgerbeziehung für X bzw. Y. Daraus ergeben sich die potentiell zu löschenden Kanten (A, Y) und (X, B). Keine dieser Kanten entspricht aber der gegenüber S' tatsächlich zu löschenden Kante (A, B). Dies ist aber

zum jetzigen Zeitpunkt noch nicht feststellbar weshalb (*A*, *Y*) und (*X*, *B*) in $TempCtrlE_{\Delta I \setminus \Delta S}^{del}$ gespeichert werden.

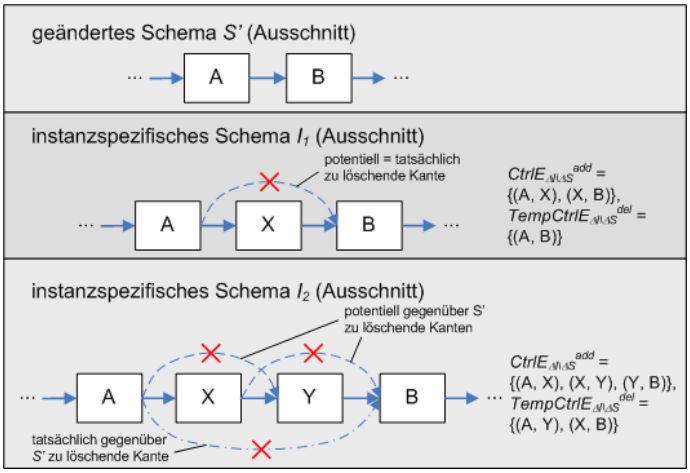


Abbildung 7.17 Beispiel einer potentiell gelöschten Kante

Betrachtung von $N_{\Delta I \setminus \Delta S}^{del}$:

1. Für alle Knoten *N* aus $N_{\Delta I \setminus \Delta S}^{del}$ suche Kanten (*src*, *dest*) in *S'* mit *src* = *N* oder *dest* = *N*. Ist eine gefundene Kante vom Typ *Control* so kann sie direkt in die Menge $CtrlE_{\Delta I \setminus \Delta S}^{del}$ eingefügt werden, da eine solche Kante in *I* nicht vorhanden sein kann. Dies hängt damit zusammen, dass der in *I* nicht vorhandene Knoten *N* in der betrachteten Kante als *src* oder *dest* fungiert.
2. Nach Löschen der Kontextkanten von *N*, durch Einfügen der entsprechenden Kanten in $CtrlE_{\Delta I \setminus \Delta S}^{del}$, muss nun die entstandene Lücke mit einer Kante (bzw. Kanten, falls *N* ein *Split*- oder *Join*-Knoten ist) geschlossen werden. Auch hier gilt wieder, dass es sich bei einer solchen Kante nicht zwangsläufig um eine gegenüber *S'* einzufügende Kante handeln muss. Die Kante wird analog zu Punkt 2 in die Hilfsmenge $TempCtrlE_{\Delta I \setminus \Delta S}^{add}$ eingefügt.

Betrachtung von $N_{\Delta I \setminus \Delta S}^{move}$:

Die Berechnung der Kanten für die Differenz der verschobenen Knoten $N_{\Delta I \setminus \Delta S}^{move}$ besteht aus der Kombination der Mechanismen für $N_{\Delta I \setminus \Delta S}^{add}$ und $N_{\Delta I \setminus \Delta S}^{del}$. Es ergibt sich der folgende Ablauf:

1. Für alle Knoten *N* aus $N_{\Delta I \setminus \Delta S}^{move}$ suche Kanten (*src*, *dest*) in $CtrlE_{\Delta I}^{add}$ mit *src* = *N* oder *dest* = *N*. Es ergibt sich hier quasi jeweils ein Einfügen des Knotens *N* aus $N_{\Delta I \setminus \Delta S}^{move}$ in den entsprechenden Kontext (*src, N*), (*N, dest*). Es lassen sich an dieser Stelle folglich die gleichen Mechanismen anwenden wie bei einem Knoten aus $N_{\Delta I \setminus \Delta S}^{add}$.
2. Nach dem Einfügen muss der Knotenkontext des Ursprung des verschobenen Knotens *N* aus $N_{\Delta I \setminus \Delta S}^{move}$ bereinigt werden. Dies entspricht der Vorgehensweise beim Löschen eines Knotens, weshalb der Knoten *N* aus $N_{\Delta I \setminus \Delta S}^{move}$ analog einem Knoten aus $N_{\Delta I \setminus \Delta S}^{del}$ behandelt werden kann.

Abbildung 7.18 verdeutlicht die Behandlung von *N* als Knoten aus $N_{\Delta I \setminus \Delta S}^{add}$ bzw. $N_{\Delta I \setminus \Delta S}^{del}$.

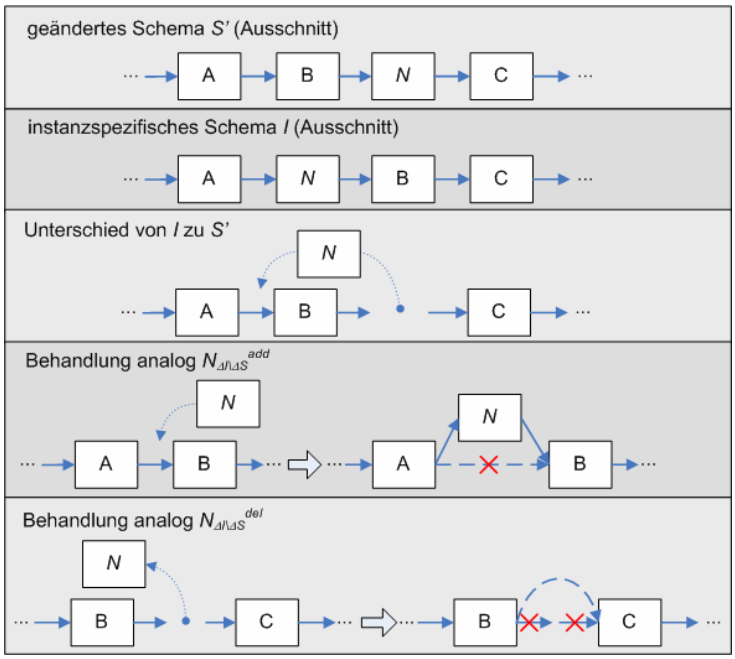


Abbildung 7.18 Behandlung von N^{move} als N^{add} bzw. N^{del}

7.4.2.2 Endgültige Menge gelöschter Kontrollkanten

Um die endgültige $CtrlE_{\Delta I \setminus \Delta S}^{del}$-Menge zu bestimmen werden die bei der Berechnung von $N_{\Delta I \setminus \Delta S}^{add}$ und $N_{\Delta I \setminus \Delta S}^{move}$ entstandenen, potentiell zu löschenden Kanten betrachtet. Dazu wird für jede Kante (*src*, *dest*) aus $TempCtrlE_{\Delta I \setminus \Delta S}^{del}$ geprüft, ob sich *src* in *S'* befindet[13]. Trifft dies zu, so müssen die folgenden Fälle unterschieden werden:

1. $src_{S'}$ ist vom Typ *Normal* oder *Loop_Start*/*Loop_End*: In *S'* kann nur eine Kante vom Typ *Control* existieren, die als Ursprung $src_{S'}$ besitzt. Deshalb füge Kante ($src_{S'}$, $succ(src_{S'})$) von *S'* in $CtrlE_{\Delta I \setminus \Delta S'}^{del}$ ein. Hierbei ist zu beachten, dass $succ(src_{S'})$ nicht mit *dest* von der betrachteten Kante aus $TempCtrlE_{\Delta I \setminus \Delta S}^{del}$ übereinstimmen muss. Ein solcher Fall ergibt sich bei der Betrachtung der potentiell zu löschenden Kante (*A*, *Y*) der Instanz I_2 aus Abbildung 7.17. Bei Anwendung der beschriebenen Vorgehensweise ergibt sich die tatsächlich zu löschende Kante (*A*, *B*), wobei $succ(src_{S'})$ (= *B*) nicht mit *dest* aus (*A*, *Y*) übereinstimmt. (*A*, *B*) entspricht aber der tatsächlich gegenüber *S'* zu löschenden Kante, weshalb die Vorgehensweise das gewünschte Ergebnis für $CtrlE_{\Delta I \setminus \Delta S}^{del}$ liefert.
2. $src_{S'}$ ist vom Typ *Split*: Handelt es sich bei dem betrachteten Knoten um einen *Split*-Knoten, so prüfe für jedes Kantentupel ($src_{S'}$, $dest_{S'}$) vom Typ *Control*, ob dieses Tupel noch in *I* vorhanden ist (d.h. die Kante ($src_{S'}$, $dest_{S'}$) befindet sich nicht in $CtrlE_{\Delta I}^{del}$). Befindet sich ($src_{S'}$, $dest_{S'}$) in $CtrlE_{\Delta I}^{del}$, so handelt es sich um eine gegenüber *S'* gelöschte Kante, die in die Menge $CtrlE_{\Delta I \setminus \Delta S}^{del}$ eingefügt werden muss.

Dass bei der Betrachtung eines $src_{S'}$ Knotens zwischen den Typen *Normal* und *Split* unterschieden werden muss begründet sich folgendermaßen:
Handelt es sich bei $src_{S'}$ um einen Split-Knoten, so existieren in *S'* mehrere Kanten, die diesen Knoten als Ursprung besitzen. Es lässt sich aber, falls der zugehörige *dest* Knoten der Kante aus $TempCtrlE_{\Delta I \setminus \Delta S}^{del}$ ein lediglich bei *I* neu eingefügter Knoten ist, nicht feststellen, in welchem Teilzweig sich die potentiell zu löschende Kante befindet. Da dies nicht entschieden werden kann,

[13] Ein mit *src* bzw. *dest* identischer Knoten in *S'* wird im Folgenden mit $src_{S'}$ bzw. $dest_{S'}$ bezeichnet

werden – wie bei 2 beschrieben – alle Kanten aus *S'* geprüft, die $src_{S'}$ als Ursprung besitzen. Abbildung 7.19 verdeutlicht diesen Umstand.

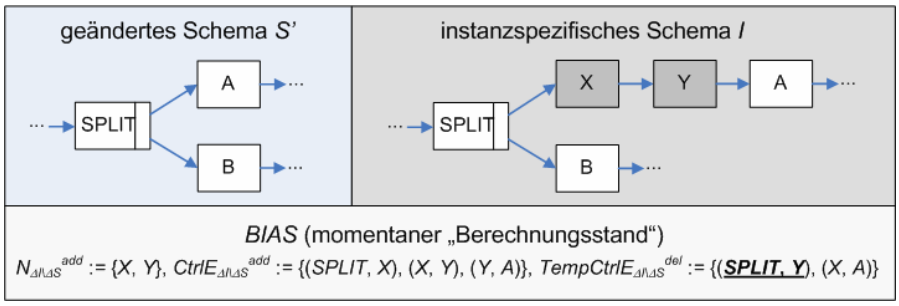


Abbildung 7.19 Prüfen aller aus einem Splitknoten ausgehenden Kanten

Bei der Betrachtung des *src*-Knotens der potentiell zu löschenden Kante (*SPLIT*, *Y*), finden sich in *S'* die Kanten (*SPLIT*, *A*) und (*SPLIT*, *B*). Da *Y* aber nicht in *S'* vorhanden ist, lässt sich nicht feststellen, dass die potentiell zu löschende Kante den oberen Teilzweig betrifft. Folglich werden einfach in *S'* alle von Split ausgehenden Kanten geprüft. Es ergibt sich als Ergebnis die in $CtrlE_{\Delta I \setminus \Delta S}^{del}$ einzufügende Kante (*SPLIT*, *A*). Die zweite Kante (*SPLIT*, *B*) ist hingegen auch in *I* vorhanden, weshalb sie nicht der Menge $CtrlE_{\Delta I \setminus \Delta S}^{del}$ hinzugefügt wird. Dies entspricht genau dem gewünschten Verhalten.

Befindet sich *src* nicht in *S'*, so muss geprüft werden, ob dies evtl. für *dest* gilt. Trifft dies zu, so müssen auch hier die folgenden Fälle unterschieden werden:

1. $dest_{S'}$ ist vom Typ *Normal* oder *Loop_Start*/*Loop_End*: Es kann in *S'* nur eine Kante vom Typ *Control* mit Ziel $dest_{S'}$ existieren. Deshalb füge Kante ($pred(dest_{S'})$, $dest_{S'}$) von *S'* in $CtrlE_{\Delta I \setminus \Delta S}^{del}$ ein. Auch hier ist wieder zu beachten, dass $pred(dest_{S'})$ nicht mit *src* der Kante aus $TempCtrlE_{\Delta I \setminus \Delta S}^{del}$ übereinstimmen muss. Ein solcher Fall ergibt sich für die potentielle Kante (*X*, *B*) der Instanz I_2 aus Abbildung 7.17, deren Betrachtung die tatsächlich einzufügende Kante (*A, B*) ergibt.
2. $dest_{S'}$ ist vom Typ *Join*: Handelt es sich bei dem betrachteten Knoten um einen *Join*-Knoten, so muss für jedes Kantentupel ($src_{S'}$, $dest_{S'}$) vom Typ *Control* geprüft werden, ob es noch in *I* vorhanden ist. Ist dies für eine Kante nicht der Fall, so füge diese in die Menge $CtrlE_{\Delta I \setminus \Delta S}^{del}$ ein.

Die durchgeführte Unterscheidung zwischen *Normal* und *Join* lässt sich analog zu der Unterscheidung zwischen *Normal* und *Split* begründen. Eine solche Situation entsteht beispielsweise, wenn man gegenüber *S'* zwei Knoten (*X* und *Y*) seriell vor einem *Join*-Knoten einfügt. Die potentiell zu löschende Kante (*X*, *JOIN*) erfordert dabei exakt die in Punkt 2 beschriebene Vorgehensweise, um zu einem korrekten Ergebnis zu führen.

Enthält *S'* weder *src* noch *dest*, so muss die Kante nicht weiter betrachtet werden, da es sich bei *src* und *dest* um Knoten handelt, die nur bei *I* eingefügt worden sind.

7.4.2.3 Endgültige Menge neu eingefügter Kontrollkanten

Die Berechnung der endgültigen $CtrlE_{\Delta I \setminus \Delta S}^{add}$ Menge ist mit der Berechnung im vorigen Abschnitt vergleichbar. Dabei ist zu beachten, dass sich die Suche eines *src* bzw. *dest* Knotens und dessen Kanten auf den Bereich $CtrlE_{\Delta I}^{add}$ beschränkt. Es ergibt sich der folgende Ablauf:

Für jede Kante (*src*, *dest*) aus $TempCtrlE_{\Delta I \setminus \Delta S}^{add}$ wird geprüft, ob sich *src* in Δ_I befindet. Trifft dies zu, so sind die folgenden beiden Fälle zu unterscheiden:

1. *src* ist vom Typ *Normal* oder *Loop_Start*/*Loop_End*: In $CtrlE_{\Delta I}^{add}$ kann nur eine Kante existieren, die als Ursprung *src* besitzt. Deshalb füge Kante (*src*, *succ*(*src*)) von *I* in $CtrlE_{\Delta I \setminus \Delta S}^{add}$ ein. Auch hier muss *succ*(*src*) nicht mit *dest* der betrachteten Kante aus $TempCtrlE_{\Delta I \setminus \Delta S}^{add}$ übereinstimmen.
2. *src* ist vom Typ *Split*: Handelt es sich bei dem betrachteten Knoten um einen *Split*-Knoten, so prüfe für jedes Kantentupel (*src*, *dest*), ob dieses Tupel bereits in *S'* vorhanden war. Ist dies für eine Kante nicht der Fall, so füge diese in die Menge $CtrlE_{\Delta I \setminus \Delta S}^{add}$ ein.

Ist *src* nicht in Δ_I vorhanden, so prüfe ob dies zumindest für *dest* gilt. Befindet sich *dest* in Δ_I, so wird wiederum eine Fallunterscheidung notwendig:

1. *dest* ist vom Typ *Normal* oder *Loop_Start*/*Loop_End*: Es kann in Δ_I nur eine Kante vom Typ *Control* mit Ziel *dest* existieren. Folglich ist die Kante (*pred*(*dest*), *dest*) von Δ_I in $CtrlE_{\Delta I \setminus \Delta S}^{add}$ einzufügen. Der Knoten *pred*(*dest*) muss auch hier wiederum nicht mit *src* von der betrachteten Kante aus $TempCtrlE_{\Delta I \setminus \Delta S}^{add}$ identisch sein.
2. *dest* ist vom Typ *Join*: In diesem Fall muss für jedes Kantentupel (*src*, *dest*) vom Typ *Control* geprüft werden, ob es in *S'* vorhanden ist. Trifft dies für eine Kante nicht zu, so füge diese in die Menge $CtrlE_{\Delta I \setminus \Delta S}^{add}$ ein.

Enthält $CtrlE_{\Delta I}^{add}$ weder *src* noch *dest*, so muss die Kante nicht weiter betrachtet werden, da es sich bei *src* und *dest* um Knoten handelt, die nur bei *I* gelöscht worden sind.

7.4.2.4 Sonderfall

Um die Berechnung der Mengen $CtrlE_{\Delta I \setminus \Delta S}^{add}$ und $CtrlE_{\Delta I \setminus \Delta S}^{del}$ abzuschließen ist noch ein Spezialfall zu berücksichtigen, der nicht mit Hilfe der Mengen $N_{\Delta I \setminus \Delta S}^{add}$, $N_{\Delta I \setminus \Delta S}^{del}$, $N_{\Delta I \setminus \Delta S}^{move}$ bearbeitet werden kann. Es handelt sich hierbei um die Auswirkungen der Änderungsoperationen *insertEmptyBranch* und *deleteEmptyBranch*. Diese Operationen fügen keiner der Mengen $N_{\Delta I \setminus \Delta S}^{add}$, $N_{\Delta I \setminus \Delta S}^{del}$ und $N_{\Delta I \setminus \Delta S}^{move}$ Knoten hinzu, weshalb die bisher vorgestellten Algorithmen keine Wirkung zeigen. Um trotzdem die Auswirkungen dieser Änderungsoperationen für die Mengen $CtrlE_{\Delta I \setminus \Delta S}^{add}$ und $CtrlE_{\Delta I \setminus \Delta S}^{del}$ bestimmen zu können, werden die folgenden Berechnungen notwendig:

$CtrlE_{\Delta I \setminus \Delta S}^{add}$: Es muss für jede Kante (*src*, *dest*) aus $CtrlE_{\Delta I}^{add}$ geprüft werden, ob *src* ein *Split*- und *dest* ein *Join*-Knoten ist. Trifft dies zu, so handelt es sich um einen hinzugefügten leeren Teilzweig. Es ist zu prüfen, ob dieser Teilzweig auch in *S'* hinzugefügt worden ist. Dazu sucht man die Kante (*src*, *dest*) in $CtrlE_{\Delta S}^{add}$. Ist diese nicht vorhanden, so wird die Kante (*src*, *dest*) in die Menge $CtrlE_{\Delta I \setminus \Delta S}^{add}$ eingefügt.
Auf Grund der Semantik von Einfügeoperationen muss ein weiterer Fall beachtet werden. Dieser entsteht wenn man beispielsweise die Operation *insertNode(node, pred, succ)* auf Schema- und Instanzebene anwendet und *pred* vom Typ *Split* und gleichzeitig *succ* vom Typ *Join* ist, d.h. *node* wird in einen leeren Teilzweig von *S* eingefügt. Dies erfordert, dass die ursprüngliche Kante (*pred*, *succ*) gelöscht und somit $CtrlE_{\Delta S}^{del}$ bzw. $CtrlE_{\Delta I}^{del}$ hinzugefügt wird. Fügt man nun auf Instanzebene durch die Änderungsmethode *insertEmptyBranch(pred*, *succ)* wieder ein leerer Teilzweig ein, so wird der Eintrag (*pred*, *succ*) auf Grund der impliziten Bereinigungs-Funktion der

Deltaschicht (vgl. Abschnitt 7.2.2.1.3) aus $CtrlE_{\Delta I}^{del}$ entfernt. Die Deltaschicht von *I* enthält also keinen Eintrag der diese Kante erwähnt. Bei einer Migration der Instanz *I* auf das geänderte Schema *S'* ist diese Kante aber relevant, da sie auf Schemaebene im Zuge der Einfügung von *node* gelöscht worden ist. Das berechnete *Bias* muss folglich einen Eintrag besitzen, der diese Kante als explizit eingefügt beschreibt, d.h. der Menge $CtrlE_{\Delta I \setminus \Delta S}^{add}$ muss die Kante (*pred*, *succ*) hinzugefügt werden. Berechnen lässt sich dies folgendermaßen:

Prüfe für jede Kante (*src*, *dest*) aus $CtrlE_{\Delta S}^{del}$, ob *src* ein *Split*- und *dest* ein *Join*-Knoten ist. Trifft dies zu, so muss dieser Teilzweig auch in $CtrlE_{\Delta I}^{del}$ enthalten ist. Anderenfalls wird die Kante (*src*, *dest*) in die Menge $CtrlE_{\Delta I \setminus \Delta S}^{add}$ eingefügt.

$CtrlE_{\Delta I \setminus \Delta S}^{del}$: Für die Bestimmung der gelöschten Teilzweige ergibt sich ein ähnlicher Ablauf. Hier wird analog für jede Kante (*src*, *dest*) aus der Menge $CtrlE_{\Delta I}^{del}$ geprüft, ob *src* ein *Split*- und *dest* ein *Join*-Knoten ist. Ist dies der Fall, so sucht man die Kante (*src*, *dest*) in der Menge $CtrlE_{\Delta S}^{del}$. Ist diese vorhanden, so handelt es sich um eine Änderung, die sowohl auf Instanz als auch auf Schemaebene durchgeführt wurde. Folglich muss die Kante nicht in $CtrlE_{\Delta I \setminus \Delta S}^{del}$ eingefügt werden. Befindet sich das Kantentupel allerdings nicht in $CtrlE_{\Delta S}^{del}$, so wird es der Menge $CtrlE_{\Delta I \setminus \Delta S}^{del}$ hinzugefügt.

Mit den vorgestellten Mechanismen ist es möglich, auch die letzten noch fehlenden Mengen $CtrlE_{\Delta I \setminus \Delta S}^{add}$ und $CtrlE_{\Delta I \setminus \Delta S}^{del}$ korrekt zu berechnen und somit die nach der Migration von *I* auf *S'* resultierende Deltaschicht von *I* zu erzeugen.

7.4.3 Anwendungsbeispiel

Um zu zeigen, wie die vorgestellten Algorithmen in der Praxis angewendet werden, wird in diesem Abschnitt die *Bias*-Berechnung anhand eines konkreten Beispiels detailliert erläutert. Das Hauptaugenmerk liegt dabei auf der Berechnung von $CtrlE_{\Delta I \setminus \Delta S}^{add}$ und $CtrlE_{\Delta I \setminus \Delta S}^{del}$.

Wie aus Abbildung 7.20 ersichtlich ist, handelt es sich bei *I* um eine Instanz der Klasse *subsumption equivalent* mit $\Delta_S < \Delta_I$. Die Abweichungen zwischen *I* und *S'* lassen sich nach der in den Abschnitten 7.4.2.1-7.4.2.4 beschriebenen Vorgehensweise folgendermaßen berechnen: Zuerst werden die Differenzmengen bestimmt. Es ergeben sich die Mengen: $N_{\Delta I \setminus \Delta S}^{add} = \{Y, Z\}$, $N_{\Delta I \setminus \Delta S}^{del} = \{A, B\}$, $N_{\Delta I \setminus \Delta S}^{move} = \{E\}$ und $SyncE_{\Delta I \setminus \Delta S}^{add} = \{(C, Y)\}$. Diese können direkt in das *Bias* übernommen werden.

Für die Berechnung der Kontrollkantenmengen $CtrlE_{\Delta I \setminus \Delta S}^{add}$ und $CtrlE_{\Delta I \setminus \Delta S}^{del}$ werden in einem ersten Schritt aus $N_{\Delta I \setminus \Delta S}^{add}$ die Kanten bestimmt, die garantiert in $CtrlE_{\Delta I \setminus \Delta S}^{add}$ eingefügt werden müssen. Sucht man, wie in Abschnitt 7.4.2.1 beschrieben, in Δ_I nach Kanten, die als *src* oder *dest Y* bzw. *Z* besitzen, so erhält man (*SPLIT*, *Y*) und (*Y*, *X*), sowie (*Start*, *Z*) und (*Z*, *SPLIT*). Diese können direkt in $CtrlE_{\Delta I \setminus \Delta S}^{add}$ eingefügt werden. Dann werden aus diesen Tupeln die Kanten bestimmt die potentiell in $CtrlE_{\Delta I \setminus \Delta S}^{del}$ eingefügt werden müssen. Es ergeben sich die folgenden in $TempCtrlE_{\Delta I \setminus \Delta S}^{del}$ einzufügenden Kanten: (*SPLIT*, *X*) und (*Start*, *SPLIT*).

Im zweiten Schritt werden die Knoten aus $N_{\Delta I \setminus \Delta S}^{del}$ betrachtet. Sucht man in *S'* (nicht in der Deltaschicht von *S*') die entsprechenden Kanten, so ergeben sich die Tupel (*Start*, *A*) und (*A*, *B*), sowie (*A*, *B*) und (*B*, *SPLIT*). Diese werden sofort der Menge $CtrlE_{\Delta I \setminus \Delta S}^{del}$ hinzugefügt, wobei ein doppeltes Einfügen von (*A*, *B*) auf Grund der Mengenoperation automatisch vermieden wird. Um die Lücken zu schließen werden in $TempCtrlE_{\Delta I \setminus \Delta S}^{add}$ die Kanten (*Start*, *B*) und (*A*, *SPLIT*) eingefügt.

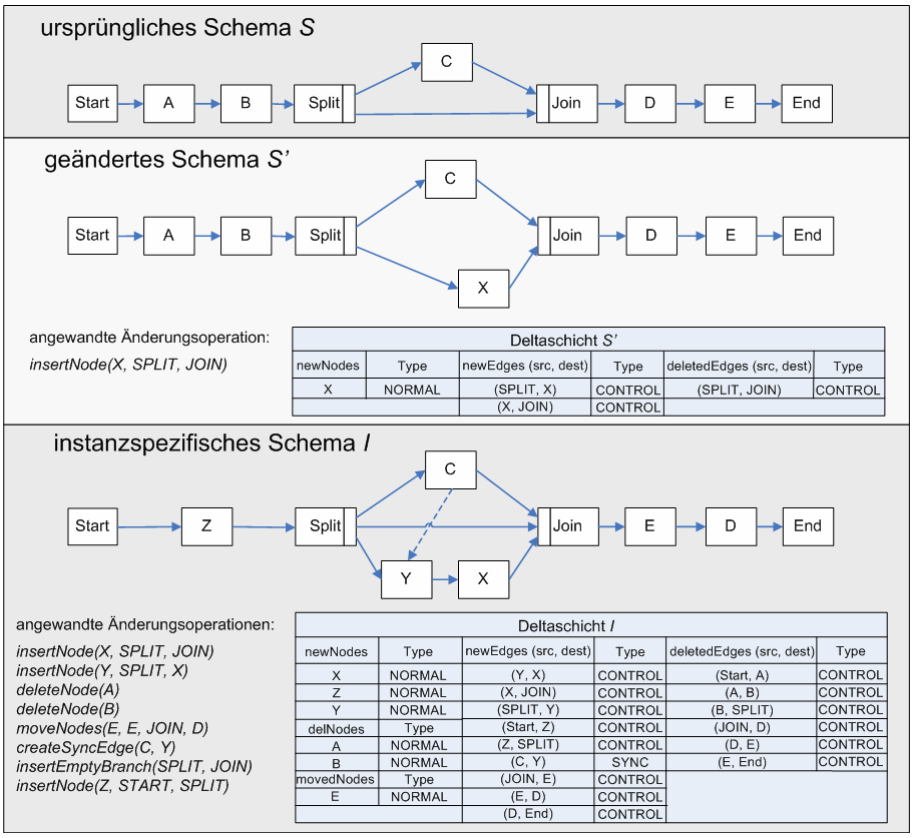


Abbildung 7.20 Ausgangssituation zur *Bias*-Berechnung

Für den Knoten E aus $N_{\Delta I \setminus \Delta S}^{move}$ werden zuerst die entsprechenden eingefügten Kanten aus Δ_I betrachtet. Man erhält (*JOIN*, *E*) und (*E*, *D*). Diese können wiederum sofort in $CtrlE_{\Delta I \setminus \Delta S}^{add}$ eingefügt werden. Für $TempCtrlE_{\Delta I \setminus \Delta S}^{del}$ ergibt sich (*JOIN*, *D*).

Im nächsten Schritt müssen die garantiert zu löschenden Kanten bestimmt werden. Es ergeben sich aus *S'* (*D*, *E*) und (*E*, *End*). Daraus resultiert die in $TempCtrlE_{\Delta I \setminus \Delta S}^{add}$ einzufügende potentielle Kante (*D*, *End*).

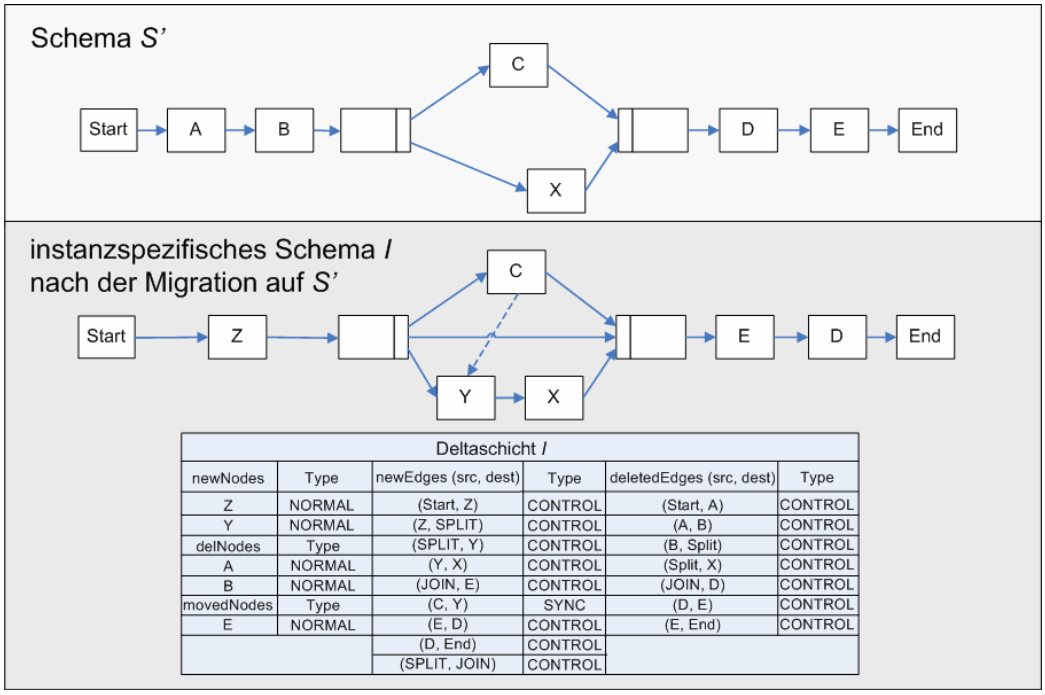
Damit sind alle garantiert und potentiell in $CtrlE_{\Delta I \setminus \Delta S}^{add}$ bzw. $CtrlE_{\Delta I \setminus \Delta S}^{del}$ einzufügenden Kanten bestimmt. Jetzt sind die potentiellen Kanten aus $TempCtrlE_{\Delta I \setminus \Delta S}^{add}$ und $TempCtrlE_{\Delta I \setminus \Delta S'}^{del}$ genauer zu untersuchen und die endgültigen $CtrlE_{\Delta I \setminus \Delta S}^{add}$ und $CtrlE_{\Delta I \setminus \Delta S}^{del}$ zu berechnen.

Zuerst werden die Kanten aus der Menge $TempCtrlE_{\Delta I \setminus \Delta S}^{del}$ betrachtet. Sucht man den *src*-Knoten der Kante (*SPLIT, X*) in *S'* so fällt auf, dass es sich hierbei um einen Knoten vom Typ *Split* handelt. Deshalb wird für alle Kanten aus *S'* mit $src_{S'} = src$ geprüft, ob diese noch in *I* vorhanden sind. Dies trifft für die Kante (*SPLIT*, *X*) nicht zu. Folglich wird dieses Tupel direkt in $CtrlE_{\Delta I \setminus \Delta S}^{del}$ eingefügt. Für den *src*-Knoten der zweiten Kante aus $TempCtrlE_{\Delta I \setminus \Delta S}^{del}$ (*Start*) findet sich bei der Suche in *S'* (*Start*, *A*). Dieses Tupel müsste eigentlich in $CtrlE_{\Delta I \setminus \Delta S}^{del}$ eingefügt werden. Da dieses aber bereits vorhanden ist, wird es nicht erneut eingefügt. Sucht man abschließend für die letzte Kante (= (*JOIN*, *D*)) aus $TempCtrlE_{\Delta I \setminus \Delta S}^{del}$ einen entsprechenden *src*-Knoten in *S'*, so findet man die Kante (*JOIN*, *D*). Diese ist in $CtrlE_{\Delta I \setminus \Delta S}^{del}$ noch nicht vorhanden und wird folglich dort eingefügt. Hiermit ist die endgültige Berechnung von $CtrlE_{\Delta I \setminus \Delta S}^{del}$ abgeschlossen. Es resultiert die Menge $CtrlE_{\Delta I \setminus \Delta S}^{del}$:= {(*Start*, *A*), (*A*, *B*), (*B*, *SPLIT*), (*SPLIT*, *X*), (*JOIN*, *D*), (*D*, *E*), (*E*, *End*)}.

Nun sind die Kanten aus der Menge $TempCtrlE_{\Delta I \setminus \Delta S}^{add}$ zu untersuchen. Für den *src*-Knoten des Tupels (*Start*, *B*), findet sich in $CtrlE_{\Delta I}^{add}$ von Δ_I die Kante (*Start*, *Z*). Diese ist allerdings bereits in $CtrlE_{\Delta I \setminus \Delta S}^{add}$ enthalten weshalb sie nicht eingefügt wird. Sucht man den *src*-Knoten der zweiten Kante

(*A*, *SPLIT*) aus $TempCtrlE_{\Delta I \setminus \Delta S}^{add}$ in $CtrlE_{\Delta I}^{add}$, so findet man keinen entsprechenden Eintrag mit gleichem *src*. Deshalb muss nach einem entsprechenden Eintrag mit gleichem *dest* gesucht werden. Es findet sich der Eintrag (*Z*, *SPLIT*), der ebenfalls bereits in $CtrlE_{\Delta I \setminus \Delta S}^{add}$ vorhanden ist uns somit nicht eingefügt wird. Für die letzte Kante (*D*, *End*) aus $TempCtrlE_{\Delta I \setminus \Delta S}^{add}$ existiert eine Kante mit gleichem *src* in der Deltaschicht von *I*. Dass diese sogar exakt identisch ist spielt hierbei keine Rolle. Sie wird in $CtrlE_{\Delta I \setminus \Delta S}^{add}$ eingefügt. Damit ist (vorerst) auch die Berechnung der $CtrlE_{\Delta I \setminus \Delta S}^{add}$-Menge abgeschlossen. Es ergibt sich $CtrlE_{\Delta I \setminus \Delta S}^{add} := \{(Start, Z), (Z, SPLIT), (SPLIT, Y), (Y, X), (JOIN, E), (E, D), (D, End)\}$. Abschließend muss noch der Spezialfall durch neu hinzugefügte bzw. gelöschte leere Teilzweige behandelt werden. Sucht man in $CtrlE_{\Delta I}^{add}$ nach Kanten vom Typ *Control*, deren *src*-Knoten vom Typ *Split*- und deren *dest*-Knoten vom Typ *Join* sind, so findet sich kein Eintrag. Eine Suche in $CtrlE_{\Delta S}^{del}$ liefert hingegen die Kante (*SPLIT*, *JOIN*). Diese ist nicht in $CtrlE_{\Delta I}^{del}$ enthalten und wird deshalb in $CtrlE_{\Delta I \setminus \Delta S}^{add}$ eingefügt. Der zweite Test des Spezialfalls liefert bei der Suche in $CtrlE_{\Delta I}^{del}$ keine Ergebnisse, d.h. die Menge $CtrlE_{\Delta I \setminus \Delta S}^{del}$ bleibt unverändert.

Abbildung 7.21 zeigt die Instanz *I* inklusive der aus den Differenzmengen der *Bias*-Berechnung erzeugten Deltaschicht. Für *I* bedeutet dies, dass sie nach der erfolgreichen Migration *S'* referenziert und die Deltaschicht die Unterschiede zwischen *S'* und *I* kapselt. Damit ist die Schemaevolution für die Instanz *I* der Klasse *subsumption equivalent* mit $\Delta_S < \Delta_I$ abgeschlossen.



Deltaschicht *I*

newNodes	Type	newEdges (src, dest)	Type	deletedEdges (src, dest)	Type
Z	NORMAL	(Start, Z)	CONTROL	(Start, A)	CONTROL
Y	NORMAL	(Z, SPLIT)	CONTROL	(A, B)	CONTROL
delNodes	Type	(SPLIT, Y)	CONTROL	(B, Split)	CONTROL
A	NORMAL	(Y, X)	CONTROL	(Split, X)	CONTROL
B	NORMAL	(JOIN, E)	CONTROL	(JOIN, D)	CONTROL
movedNodes	Type	(C, Y)	SYNC	(D, E)	CONTROL
E	NORMAL	(E, D)	CONTROL	(E, End)	CONTROL
		(D, End)	CONTROL		
		(SPLIT, JOIN)	CONTROL		

Abbildung 7.21 Resultierendes *Bias*

7.4.4 Zusammenfassung

Bei der Migration einer Instanz *I* der Klasse *subsumption equivalent* ($\Delta_S < \Delta_I$) weicht *I* auch nach dem Umhängen auf das geänderte Schema *S'* noch von diesem ab. Damit die Korrektheit von *I* auch nach der Migration erhalten bleibt muss dieses *Bias* bestimmt werden. Mit den in diesem Abschnitt beschriebenen Algorithmen ist das Änderungsrahmenwerk in der Lage, diese Abweichung ($Bias_{neu}$) zuverlässig zu berechnen. Dabei lässt sich das $Bias_{neu}$ für den Fall, dass weder Δ_I noch Δ_S kontextabhängige Änderungen enthalten, durch einfache Differenzmengenbildung zwischen den Element-Mengen der Deltaschichten von *I* und den Element-Mengen von *S'* berechnen. Im Falle

kontextabhängiger Änderungen erfordert insbesondere die Berechnung der Abweichung zwischen den neu eingefügten bzw. gelöschten Kontrollkanten von *I* und denen von *S'* aufwendigere Berechnungen. Hierzu wurden effiziente Algorithmen entwickelt, die als Eingabe ausschließlich die Informationen aus den Deltaschichten bzw. den Element-Mengen des ursprünglichen Schemas benötigen. Daraus resultiert der Vorteil, dass gegenüber dem konzeptionellen Ansatz aus [Rind04] weder teure Zugriffe auf eine Änderungshistorie notwendig sind noch eine Beschränkung auf eine fest definierte Menge an Änderungsoperationen besteht. Mit den beschriebenen Algorithmen kann also auch noch nach Einbringen bisher nicht bekannter Änderungsoperationen ein korrektes $Bias_{neu}$ berechnet werden.

7.5 Zustandsbasierte Verträglichkeit

Die bisher vorgestellten Mechanismen zur Konfliktbestimmung und zur *Bias*-Berechnung werden jeweils nur von Instanzen einer einzelnen Klasse benötigt. Eine Überprüfung, ob eine Instanz bezüglich ihres Ausführungszustandes korrekt auf ein geändertes Schema migriert werden kann, muss hingegen bei Instanzen mehrerer Klassen ausgeführt werden. So erfordern sowohl die Instanzen der Klassen *unbiased* und *biased disjoint* als auch die Instanzen der Klasse *subsumption equivalent* ($\Delta_S > \Delta_I$) Mechanismen zur Überprüfung der zustandsbasierten Verträglichkeit. Im Speziellen ist also ein Algorithmus zu entwickeln, mit dem für Instanzen aller angegebenen Klassen geprüft werden kann, ob die im Rahmen der Schemaänderungen durchgeführte Manipulation am ursprünglichen Schema *S* mit der aktuellen instanzspezifischen Markierung von *I* verträglich ist.

Die in Abschnitt 3.4.3 auf konzeptioneller Ebene vorgestellten Mechanismen zur Verträglichkeitsprüfung sind für eine praktische Umsetzung allerdings nicht geeignet. Neben der Abhängigkeit von konkreten Änderungsoperationen muss die Verträglichkeit anhand der zustandsbasierten Verträglichkeitskriterien jeder einzelnen im Zuge einer Schemaänderung angewendeten Änderungsoperation geprüft werden. Aus Effizienzgründen ist es jedoch sinnvoller, die resultierenden Effekte aller angewendeten Änderungsoperationen zur Definition eines Verträglichkeitstests heranzuziehen. Dadurch lassen sich die Änderungen nicht nur separat, sondern auch im Kontext der anderen Änderungen betrachten. In einer Vielzahl der Fälle kann so die Zahl der zu prüfenden Knoten drastisch gesenkt werden. Abbildung 7.22 verdeutlicht dies durch Gegenüberstellung der beiden Vorgehensweisen.

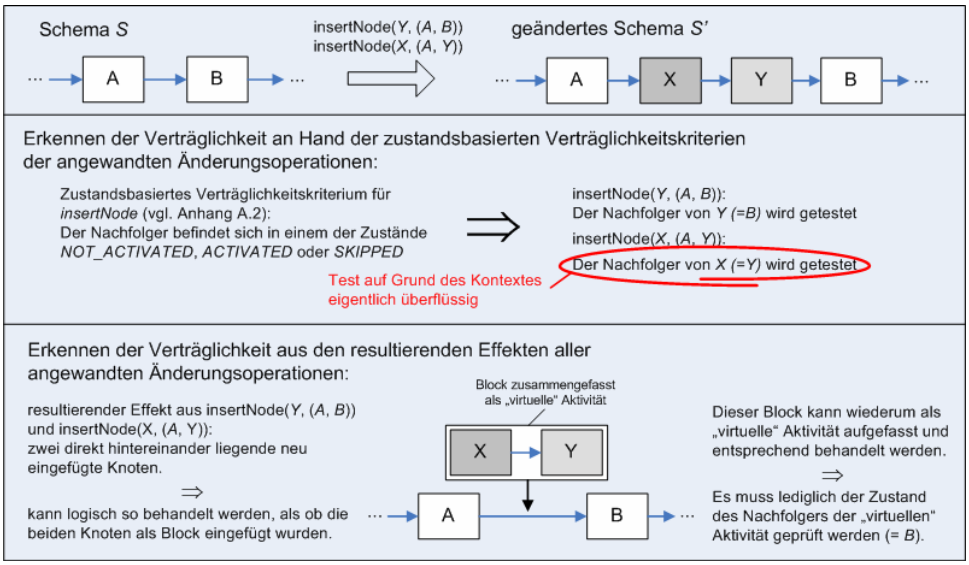


Abbildung 7.22 Vergleich der Konzepte zur zustandsbasierten Verträglichkeit

Die für den effizienten Verträglichkeitstest notwendigen Effekte aller angewandten Änderungsoperationen lassen sich direkt aus der Deltaschicht von *S'* ableiten. Der in diesem Abschnitt beschriebene Verträglichkeitstest verwendet deshalb als Ausgangsbasis die Informationen aus der Deltaschicht von *S*'.
Die Zielvorgabe des Tests ist es dabei, ausnahmslos alle verträglichen Instanzen zu erkennen, auch wenn dadurch in Einzelfällen teure Zugriffe auf die im Sekundärspeicher gehaltene Ausführungshistorie notwendig werden. Die Vorgabe zu Gunsten einer maximal erkennbaren Menge verträglicher Instanzen und gegen einen evtl. effizienteren Verträglichkeitstest begründet sich damit, dass es im Falle einer gesetzlichen Änderung u. U. notwendig ist, ein Schema anzupassen und alle diejenigen Instanzen zu migrieren, die in ihrem Ausführungszustand den von der Gesetzesänderung betroffenen Graphbereich noch nicht erreicht haben (vgl. Abschnitt 1.2). Werden in diesem Fall Instanzen die verträglich sind, aus Effizienzgründen als unverträglich erkannt, so müssen diese Instanzen zwingend durch Benutzereingriff an das geänderte Schema angepasst werden. Da in der Praxis mehrere 1000 Instanzen eines Schemas Normalität sind, steht die mögliche Zeiteinsparung durch einen effizienteren Verträglichkeitstest in keinem Verhältnis zu dem Zeitaufwand, den ein Bearbeiter in jede fälschlicherweise als unverträglich erkannte Instanz investieren muss.

Der im Folgenden vorgestellte Verträglichkeitstest wird in die Bereiche eingefügte und verschobene Knoten (Abschnitt 7.5.1), gelöschte Knoten (Abschnitt 7.5.2) und die Manipulation an Sync- und Datenkanten, Datenelementen sowie Knoten- und Kantenattributen (Abschnitt 7.5.3) unterteilt. Um zu zeigen, wie die einzelnen Algorithmen im Zusammenspiel agieren wird in Abschnitt 7.5.4 die zustandsbasierte Verträglichkeit für ein konkretes Beispiel berechnet.

7.5.1 Neu eingefügte und verschobene Knoten

In diesem Abschnitt werden die zustandsbasierten Verträglichkeitstests für diejenigen Knoten beschrieben, die an Einfügungen oder Verschiebungen von Knoten, Knotenmengen oder Teilzweigen beteiligt sind. Dabei wird eine direkte Betrachtung der Mengen *newNodes* ($= N_{\Delta S}^{add}$) und *movedNodes* ($= N_{\Delta S}^{move}$) vermieden, um die Anzahl der betrachteten Knoten so gering wie möglich zu halten. Dass eine Reduzierung der betrachteten Menge nicht mit einer Verfälschung des Ergebnisses einhergeht, zeigt Abbildung 7.23. Dabei sei darauf hingewiesen, dass die abgebildete Materialisierung der Änderungen auf I_1 und I_2 lediglich das Verständnis des Sachverhaltes erleichtern soll und keinesfalls für die Durchführung des Verträglichkeitstests notwendig ist.

Wie aus der Abbildung ersichtlich, ist es im Fall mehrerer hintereinander eingefügter Knoten unnötig, alle neu eingefügten Knoten zu betrachten. Für die zustandsbasierte Verträglichkeit ist nur der Zustand des Nachfolgers des letzten Knotens – also *B* – relevant. Die Verträglichkeit der unmittelbar davor eingefügten Knoten (*X* und *Y*) ergibt sich hingegen implizit aus der Verträglichkeit des letzten eingefügten Knotens (*Z*).
Beim Verschieben eines Teilzweiges oder einer Knotenmenge ergibt sich eine ähnliche Situation. Hier sind lediglich der erste und der letzte Knoten für die zustandsbasierte Verträglichkeit von Bedeutung. Die dazwischen liegenden Knoten beeinflussen das Ergebnis hingegen nicht.

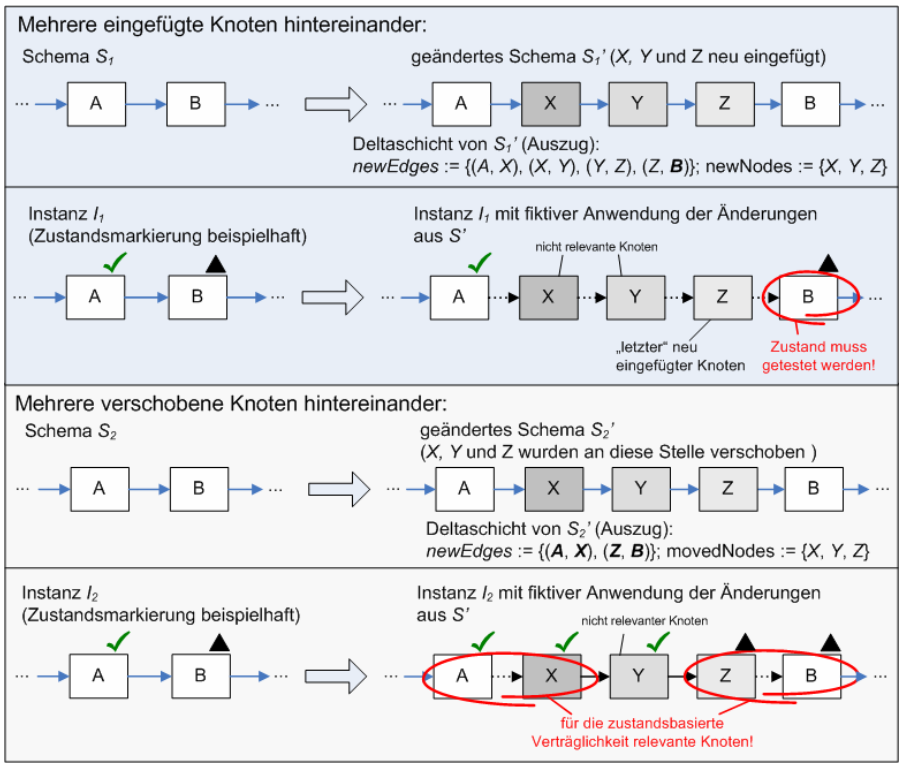


Abbildung 7.23 Reduzierung der zu betrachtenden Knotenmenge

7.5.1.1 Bestimmung der relevanten Knoten

Entscheidend für die Effizienz des Verträglichkeitstests ist es also, die relevanten Knoten möglichst schnell und einfach berechnen zu können. Dies ist in vielen Fällen bereits mit Hilfe von I bzw. der instanzspezifischen Markierung M^I möglich. Bei neu eingefügten Knoten ist beispielsweise der Zustand eines Nachfolgerknotens nur dann relevant, wenn dieser bereits in I und somit in M^I vorhanden ist. Für verschobene Knotenmengen gilt prinzipiell das Gleiche, allerdings nur für den Nachfolger des letzten Knotens einer verschobenen Menge. Die ebenfalls relevanten ersten und letzten Knoten einer verschobenen Menge lassen sich so nicht feststellen, da diese immer in M^I vorhanden sind. Allerdings können die für einen Verträglichkeitstest irrelevanten Knoten, die sich zwischen dem ersten und dem letzten Knoten einer verschobenen Menge befinden, direkt aus der Menge *newEdges* ($= CtrlE_{\Delta S}^{add}$) abgeleitet und somit von einer weiteren Betrachtung ausgeschlossen werden. Dies kommt daher, dass *newEdges* nur Kanten enthält, bei denen der erste Knoten einer verschobenen Knotenmenge als Kantenziel oder der letzte Knoten als Kantenursprung auftritt. Die dazwischen liegenden Knoten treten hingegen weder als Ursprung noch als Ziel einer Kante auf (vgl. S_2', Abbildung 7.23).

Im Detail ergibt sich für die Berechnung der für eine zustandsbasierte Verträglichkeit relevanten Knoten der folgende Ablauf:

- Für jedes Kantenziel (*dest*) einer Kante aus $CtrlE_{\Delta S}^{add}$ wird geprüft, ob sich dieses sowohl in M^I als auch in $N_{\Delta S}^{move}$ befindet. Trifft dies zu, so wird dieser in einer Hilfsmenge *movedNodesFirst* gespeichert. Ist *dest* lediglich in M^I enthalten, so ist dieser Knoten in die Menge *nodesToCheck* einzufügen. Eine Sonderbehandlung ergibt sich bei Instanzen der Klasse *subsumption equivalent* ($\Delta_I < \Delta_S$). Hier wird ein Knoten aus $N_{\Delta S}^{move}$ nur dann in die Menge *movedNodesFirst* eingefügt, wenn sich dieser nicht in der Menge $N_{\Delta I}^{move}$ befindet. Analog wird ein Knoten nur

dann der Menge *nodesToCheck* hinzugefügt, wenn die zugehörige Kante aus $CtrlE_{\Delta S}^{add}$ nicht in der Menge $CtrlE_{\Delta I}^{add}$ vorhanden ist. Hierdurch werden nur diejenigen Knoten beachtet, die sich auch dann ergeben, wenn man als Ausgangsbasis $\Delta_S \setminus \Delta_I$ verwendet.

Speichert man M^I und $N_{\Delta S}^{move}$ als *HashMenge*, so ist der Algorithmus mit einer Komplexität von O(n) durchführbar. Als Ergebnis erhält man in der Menge *movedNodesFirst* alle verschobenen Knoten bzw. bei einer verschobenen Knotenmenge oder einem verschobenen Teilzweig jeweils den am weitesten vorne liegenden Knoten. In *nodesToCheck* befinden sich hingegen Knoten, die den Nachfolgerkontext verschiedener Änderungsoperationen darstellen. Abbildung 7.24 zeigt die möglichen Situationen bei denen es zu einer Einfügung eines Knotens in die Menge *nodesToCheck* kommt.

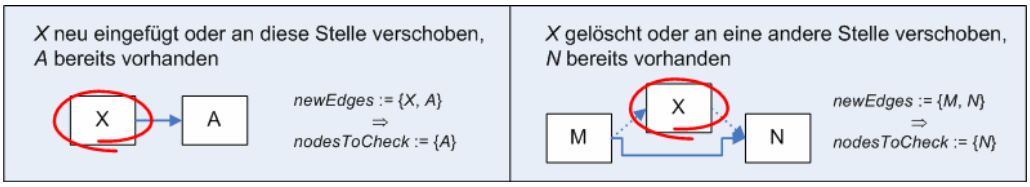


Abbildung 7.24 Situationen für das Einfügen eines Knotens in *nodesToCheck*

7.5.1.2 Prüfung der relevanten Knoten

Abhängig vom jeweiligen Zustand eines in *nodesToCheck* bzw. *movedNodesFirst* eingefügten Knotens bzw. dem Zustand dessen Vorgängers, werden unterschiedliche Verfahrensweisen bei der Verträglichkeitsprüfung notwendig. Diese werden im Folgenden detailliert beschrieben.

***nodesToCheck*:**
Für die Knoten der Menge *nodesToCheck*, die in M^I einen der Zustände *ACTIVATED*, *NOT_ACTIVATED* oder *SKIPPED* besitzen, ist die Verträglichkeit automatisch gewährleistet. Für die Knoten mit einem anderen Zustand müssen die folgenden zwei Schritte ausgeführt werden:

1. Bestimme den Vorgänger (bzw. die Vorgänger im Falle eines *Join*-Knotens) des betrachteten Knotens mit Hilfe von $CtrlE_{\Delta S}^{del}$. Dadurch lässt sich feststellen, ob der betrachtete Knoten in *I* der Nachfolger eines verschobenen Knotens, Teilzweiges oder einer Knotenmenge mit dem Zustand *COMPLETED* ist. In einem solchen Fall ist ein *RUNNING*- oder *COMPLETED*-Zustand für einen Knoten der Menge *nodesToCheck* zulässig.
2. Überprüfe den Typ der Vorgänger aus $CtrlE_{\Delta S}^{add}$. Es ergeben sich die folgenden drei Fälle:
   - Typ *Normal*: Bei Knoten vom Typ *Normal* kann es nur einen Vorgänger (*pred*) in $CtrlE_{\Delta S}^{add}$ geben. Dieser muss sich in M^I im Zustand *COMPLETED* befinden, anderenfalls ist die Instanz unverträglich. Befindet sich *pred* im Zustand *COMPLETED* und ist der Knoten nicht in der Menge $N_{\Delta S}^{move}$ enthalten, so handelt es sich bei *pred* um einen Knoten, der bereits vor der Schemaänderung im Kontrollfluss vor dem betrachteten Knoten lag. Dadurch ist dessen Zustand implizit richtig und weitere Betrachtungen sind überflüssig. Ist *pred* jedoch in $N_{\Delta S}^{move}$, so muss in der Ausführungshistorie von *I* (vgl. Abschnitt 3.1.2) dessen *End*-Eintrag vor dem *Start*-Eintrag des betrachteten Knotens aus *nodesToCheck* stehen.
   - Typ *OR_JOIN*: Bei *OR_JOIN* finden sich in *newEdges* mehrere Vorgänger. Davon muss sich genau einer im Zustand *COMPLETED* befinden. Handelt es sich bei diesem Knoten um ein Element aus $N_{\Delta S}^{move}$, so muss analog zur Vorgehensweise beim Typ *Normal* die Reihenfolgebeziehung in der Ausführungshistorie geprüft werden.

- Typ *AND_JOIN*: Handelt es sich bei dem Knoten aus *nodesToCheck* um einen *AND_JOIN*-Knoten, so existieren in *newEdges* wiederum mehrere Vorgänger. Diese müssen allerdings anders als bei der Behandlung von *OR_JOIN*-Knoten alle den Zustand *COMPLETED* besitzen. Sind weiterhin ein oder mehrere der Vorgänger in $N_{\Delta S}^{move}$ enthalten, so muss für jeden einzelnen die *End-Start*-Reihenfolgebeziehung in der Ausführungshistorie überprüft werden.

movedNodesFirst:
Für jeden Knoten aus der Menge *movedNodesFirst* wird zuerst der Zustand in M^I geprüft. Befindet sich ein solcher Knoten in einem der Zustände *NOT_ACTIVATED*, *ACTIVATED* oder *SKIPPED*, so bedarf der betrachtete Knoten keiner weiteren Überprüfung. Besitzt ein Knoten jedoch einen der Zustände *RUNNING* oder *COMPLETED*, so wird dessen Vorgänger (*pred*) mit Hilfe von $CtrlE_{\Delta S}^{add}$ bestimmt. Ist *pred* nicht in *I* und somit in M^I vorhanden, handelt es sich um einen Knoten der nur in *S'* neu eingefügt wurde. Dieser kann damit nicht den für den Knoten aus *movedNodeFirst* notwendigen Zustand *COMPLETED* besitzen und somit die Instanz nicht mit *S'* verträglich sein. Befindet sich *pred* in M^I, so muss in der Ausführungshistorie von *I* der *End*-Eintrag von *pred* vor dem *Start*-Eintrag des betrachteten Knotens aus *movedNodesFirst* stehen.

Mit dem angegeben Algorithmus kann die zustandsbasierte Verträglichkeit für neu eingefügte und verschobene Knoten zuverlässig und durch die Reduzierung auf relevante Knoten mit maximaler Effizienz bestimmt werden.

7.5.2 Gelöschte Knoten

Auch für gelöschte Knoten, ist eine Reduzierung der zu betrachtenden Knotenmenge möglich, wie Abbildung 7.25 zeigt:

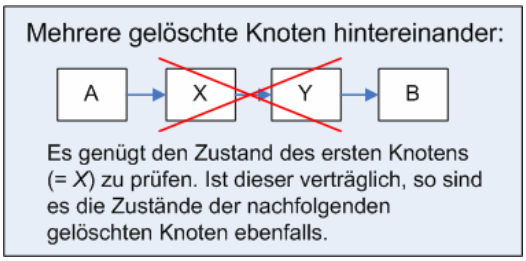


Abbildung 7.25 Reduzierung der zu prüfenden Knotenmenge

Für einen Verträglichkeitstest genügt es allerdings, lediglich den Zustand der gelöschten Knoten zu prüfen. Aus Effizienzgründen wird daher auf eine Reduzierung verzichtet.
Die Verträglichkeit für gelöschte Knoten lässt sich mit dem folgenden Algorithmus berechnen:

- Es muss für alle Knoten der Menge $N_{\Delta S}^{del}$ geprüft werden, ob sich diese in M^I in einem der Zustände *ACTIVATED*, *NOT_ACTIVATED* oder *SKIPPED* befinden. Besitzt einer der Knoten einen anderen Zustand, so ist die ganze Instanz nicht mit den Schemaänderungen von *S'* verträglich.

Bei Instanzen der Klasse *subsumption equivalent* ($\Delta_I < \Delta_S$) ist es möglich, dass ein betrachteter Knoten aus $N_{\Delta S}^{del}$ gar nicht in M_I enthalten ist. Dies tritt immer dann auf, wenn dieser Knoten auch im Zuge der Instanzänderung gelöscht worden ist. Ist dies der Fall, so muss dieser Knoten nicht weiter beachtet werden. Die Instanz bleibt verträglich.

7.5.3 Manipulationen an Sync- und Datenkanten, Datenelementen sowie Knoten- und Kantenattributen

Die Verträglichkeitstests für den Fall, dass auf Schemaebene Sync-Kanten, Datenkanten, Datenelemente, Knoten- oder Kantenattribute manipuliert worden sind, werden nachfolgend beschrieben. Dabei ist wiederum zu beachten, dass bei Instanzen der Klasse *subsumption equivalent* ($\Delta_I < \Delta_S$) die Tests nur dann ausgeführt werden wenn eine betrachtete Änderung aus Δ_S nicht auch im Zuge der instanzspezifischen Änderung durchgeführt worden ist.

Wird ein Knoten- oder Kantenattribut geändert, so ist dies nur zulässig, wenn der Knoten, der dieses Attribut besitzt bzw. der Zielknoten einer Kante deren Attribut geändert wird, sich in einem der Zustände *NOT_ACTIVATED*, *ACTIVATED* oder *SKIPPED* befindet. Dies lässt sich testen, indem man den Zustand des jeweiligen Knotens in M^I prüft. Tritt der Fall ein, dass M^I den Knoten gar nicht enthält, da dieser nur in *S'* eingefügt worden ist, so ist die Attributänderung immer zulässig.
Ein spezieller Fall von Knotenattributänderung ergibt sich, wenn der Typ eines *Split*-Knotens geändert wird. Eine solche Änderung wird z.B. durch Anwendung der Methode *convertBlock* erreicht. Hier ist es zulässig, dass der betrachtete Knoten sogar den Zustand *COMPLETED* besitzt. Allerdings nur dann, wenn sich die nachfolgenden Knoten in einem der Zustände *NOT_ACTIVATED, ACTIVATED* oder *SKIPPED* befinden.
Bei den Datenelementen werden für jedes Element aus $D_{\Delta S}^{del}$ (= Menge der gegenüber *S* gelöschten Datenelemente) Knoten in *I* gesucht, die dieses Element lesen oder schreiben. Diese Knoten müssen in M^I die Zustände *RUNNING* oder *COMPLETED* besitzen, da bereits geschriebene bzw. gelesene Datenelemente nicht gelöscht werden dürfen.
Die Prüfung von neu eingefügten oder gelöschten Lesekanten erfordert die Zustandsprüfung der lesenden Knoten. Diese lassen sich aus der Menge $DataE_{\Delta S}^{add}$ bzw. $DataE_{\Delta S}^{del}$ erkennen. Jeder der lesenden Knoten muss sich in einem der Zustände *NOT_ACTIVATED*, *ACTIVATED* oder *SKIPPED* befinden.
Neu eingefügte oder gelöschte Schreibkanten lassen sich ebenfalls aus den Mengen $DataE_{\Delta S}^{add}$ und $DataE_{\Delta S}^{del}$ erkennen. Hier darf jedoch der schreibende Knoten nicht den Zustand *COMPLETED* besitzen. Alle anderen Zustände sind zulässig.
Um die zustandsbasierte Verträglichkeit neu eingefügter Sync-Kanten zu testen werden zuerst alle Zielknoten der Kanten aus $SyncE_{\Delta S}^{add}$ auf ihren Zustand in M^I geprüft. Besitzen sie einen der Zustände *NOT_ACTIVATED*, *ACTIVATED* oder *SKIPPED*, so ist das Einfügen der Sync-Kante verträglich. Sollte sich ein Zielknoten in einem der Zustände *RUNNING* oder *COMPLETED* befinden, müssen zwei Spezialfälle unterschieden werden:

1. Befindet sich der Quellknoten der Sync-Kante in M^I im Zustand *COMPLETED*, so muss dessen *End*-Eintrag in der Ausführungshistorie von *I* vor dem *Start*-Eintrag des Zielknotens sein. Dies ist der Fall, wenn der Quellknoten vor der Aktivierung des Zielknotens beendet wurde.
2. Besitzt der Quellknoten in M^I den Zustand *SKIPPED*, so ist das Einfügen der Sync-Kante genau dann verträglich, wenn der *End*-Eintrag des ersten nicht *SKIPPED* Vorgängers des Quellknotens vor dem Start-Eintrag des Sync-Kanten-Zielknotens steht. Da bei jedem Knoten eines Teilzweiges sein zugehöriger *Split*-Knoten gespeichert wird, lässt sich der erste Vorgängerknoten, der sich nicht im Zustand *SKIPPED* befindet, effizient über die vom Datenmodell bereitgestellte Methode *getSplitNode* finden (ggf. rekursiv). Die Knoten zwischen dem *Split*-Knoten und dem betrachteten Knoten müssen dabei nicht beachtet werden, da immer alle Knoten eines Teilzweiges mit *SKIPPED* markiert sind.

Durchläuft eine Instanz auch diese Verträglichkeitstests erfolgreich, so ist sie mit den Schemaänderungen von *S'* verträglich, d.h. die bei der Schemaänderung von *S* auf *S'* durchgeführten Änderungen können auf die Instanz *I* propagiert werden ohne die Korrektheit der Instanzmarkierung nach Definition 2 zu verletzen.

7.5.4 Anwendungsbeispiel

Um zu zeigen, wie sich unter Anwendung der in den einzelnen Abschnitten beschriebenen Algorithmen, die zustandsbasierte Verträglichkeit berechnen lässt, wird anhand des Beispiels aus Abbildung 7.26 noch einmal ausführlich die Vorgehensweise bei der Überprüfung der zustandsbasierten Verträglichkeit erläutert.

Die Prüfung beginnt mit der Bestimmung der relevanten Knoten aus Einfüge- und Verschiebeoperationen. Analog zur beschriebenen Vorgehensweise werden in einem ersten Schritt diejenigen Zielknoten (*dest*) aus *newEdges* (genauer aus $CtrlE_{\Delta S}^{add}$) extrahiert, die sich auch in der schemaspezifischen Instanzmarkierung M^I befinden. Im betrachteten Beispiel ergeben sich die Knoten *R*, *C*, *J* und *D*. Bei *C* handelt es sich um einen verschobenen Knoten, weshalb dieser in die Menge *movedNodesFirst* eingefügt wird. Die anderen Knoten werden der Menge *nodesToCheck* hinzugefügt.
Im zweiten Schritt werden nacheinander mit Hilfe von M^I die Zustände der Knoten aus der Menge *nodesToCheck* geprüft. *R* befindet sich im Zustand *ACTIVATED*, *J* im Zustand *NOT_ACTIVATED*. Folglich sind deren Zustände mit den Schemaänderungen verträglich. *D* befindet sich hingegen im Zustand *COMPLETED*, wodurch für diesen Knoten weitere Tests erforderlich sind. Zuerst wird der Vorgänger mit Hilfe von *delEdges* ($CtrlE_{\Delta S}^{del}$) bestimmt. Dadurch lässt sich feststellen, ob *D* der Nachfolger eines verschobenen Knotens, Teilzweiges oder einer Knotenmenge mit dem Zustand *COMPLETED* ist. Dies ist hier der Fall, da *D* vor der Schemaänderung Nachfolger des verschobenen Knotens *C* war. Dieses Kriterium ist also erfüllt. In einem weiteren Schritt muss noch der Vorgänger von *D* in *newEdges* ($CtrlE_{\Delta S}^{add}$) bestimmt werden. Für unser Beispiel ergibt sich *B*. Dieser Knoten ist vom Typ *Normal* und befindet sich nicht in der Menge der verschobenen Knoten (vgl. Abschnitt 7.5.1.2, *nodesToCheck* Punkt 2). Daraus folgt, dass er vor der Schemaänderung im Kontrollfluss vor dem betrachteten Knoten gelegen haben muss und somit dessen Zustand implizit richtig ist. Dies bestätigt sich. *B* lag bereits vor der Änderung im Kontrollfluss vor dem *COMPLETED* Knoten *D* und hat somit zwangsläufig den für die Verträglichkeit notwendigen Zustand *COMPLETED*. Damit sind alle Knoten der Menge *nodesToCheck* geprüft.
Im nächsten Schritt werden die Knoten der Menge *movedNodesFirst* betrachtet. Hier ist dies lediglich der Knoten *C*. Dieser befindet sich im Zustand *COMPLETED*, d.h. die betrachtete Instanz *I* ist nur dann verträglich, wenn die folgenden zwei Kriterien erfüllt sind: Zum einen muss der Vorgänger von *C* den Zustand *COMPLETED* besitzen und zum anderen muss der *END*-Eintrag des Vorgängers in der Ausführungshistorie von *I* vor dem *START*-Eintrag von *C* stehen. Für unser Beispiel ergibt sich als Vorgänger von *C* der Knoten *F*. Dieser befindet sich im gewünschten Zustand *COMPLETED*. Weiterhin erkennt man bei der Betrachtung der Ausführungshistorie in Abbildung 7.26, dass der *END*-Eintrag von *F* vor dem *START*-Eintrag von *C* steht. Die Kriterien sind somit erfüllt, und die Verträglichkeit ist weiterhin gewährleistet. Damit ist die zustandsbasierte Verträglichkeitsprüfung für neu eingefügte und verschobene Knoten beendet.

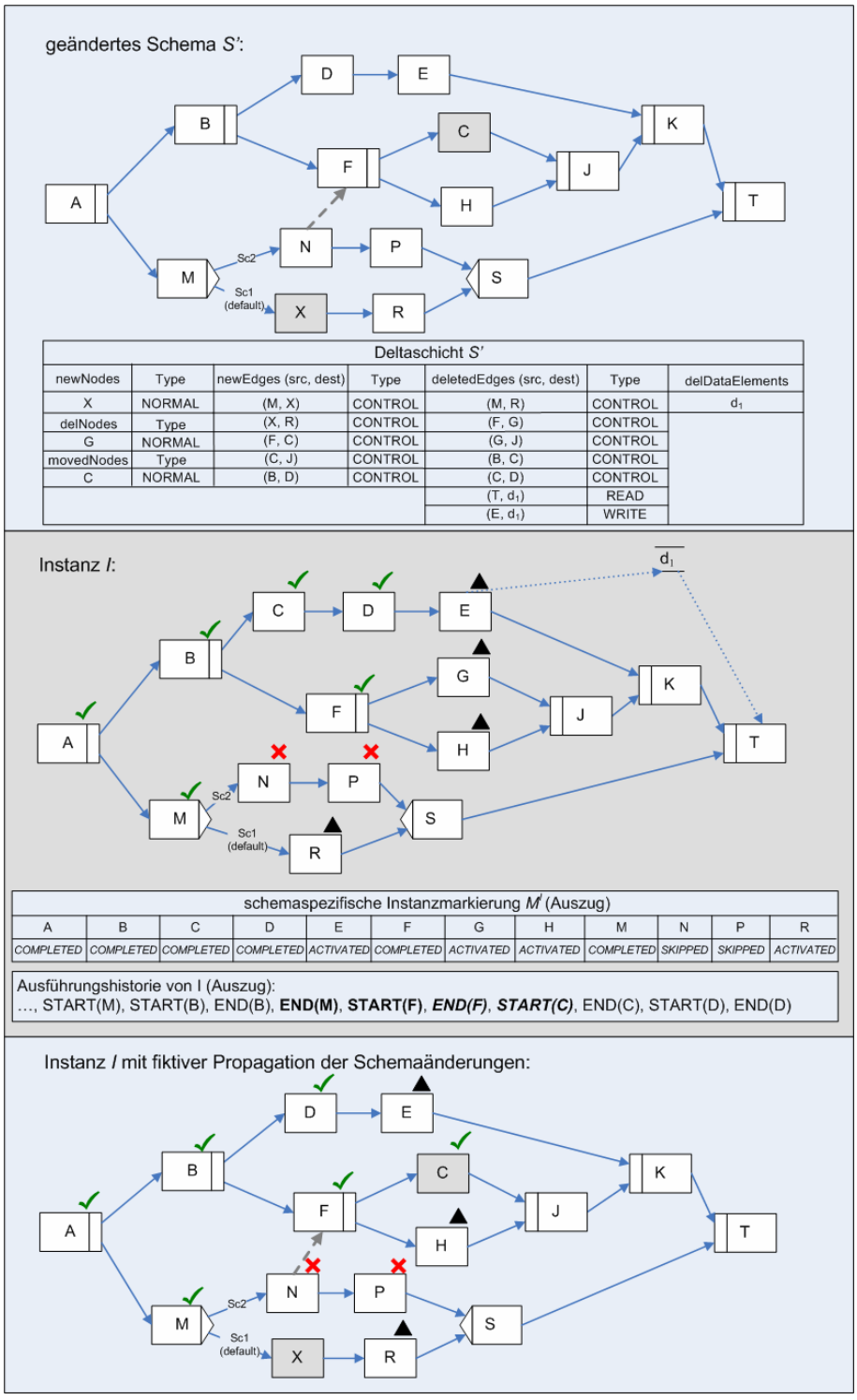


Abbildung 7.26 Beispiel zur zustandsbasierten Verträglichkeit

Nun geht es daran die gelöschten Knoten zu prüfen. Es ergibt sich lediglich der Knoten *G*. Dieser hat in M^I den für die Verträglichkeit zulässigen Zustand *ACTIVATED*, womit auch nach dieser Prüfung die Instanz *I* verträglich bleibt.

In einem letzten Schritt muss noch die zustandsbasierte Verträglichkeit für Manipulationen an Sync- und Datenkanten, Datenelementen sowie Knoten- und Kantenattributen überprüft werden. Für Datenkanten und -elemente ergibt sich bei der Betrachtung der Deltaschicht von *S'* die gelöschte Lesekante (T, d_1), die Schreibkante (E, d_1)) und das gelöschte Datenelement d_1. Wie in Abschnitt 7.5.3 beschrieben, erfordert die Manipulation an Lese- und Schreibkanten sowie das Löschen von

Datenelementen eine Zustandsüberprüfung der lesenden bzw. schreibenden Knoten. Es ist also notwendig die Zustände der Knoten *E* und *T* zu überprüfen. Die Prüfung in M^I ergibt die Zustände *ACTIVATED* für *E* und *NOT_ACTIVATED* für *T*. Beide Zustände beeinträchtigen die Verträglichkeit nicht. Bezüglich Sync-Kanten findet man in *newEdges* die neu eingefügte Sync-Kante (*N*, *F*). Deren Zielknoten *F* befindet sich im Zustand *COMPLETED*. Dies ist nur dann zulässig, wenn der Startknoten – also *N* – vor dem Zielknoten *F* beendet wurde. *N* befindet sich allerdings im Zustand *SKIPPED*, weshalb der direkte Vergleich nicht möglich ist. Es muss also zuerst derjenige Vorgänger von *F* bestimmt werden, der sich im Zustand *COMPLETED* befindet (vgl. Spezialfälle bei Sync-Kanten). Mit Hilfe von *getSplitNode* erhält man den Knoten *M*. Im betrachteten Beispiel befindet sich der *END*-Eintrag dieses Knotens in der Ausführungshistorie vor dem *START*-Eintrag des Sync-Kanten Zielknotens *F*. Folglich ist die Instanz weiterhin mit den Änderungen von *S'* verträglich. Manipulationen an Knoten- oder Kantenattributen wurden im betrachteten Beispiel nicht durchgeführt. Ein entsprechender Test ist somit nicht notwendig, womit die Prüfung der zustandsbasierten Verträglichkeit abgeschlossen ist. Als Ergebnis folgt, dass die Schemaänderungen von *S'* mit dem Zustand der Instanz *I* verträglich sind. Oder genauer, dass – analog zu Kriterium 1 – die Ausführungshistorie der Instanz auch auf dem geänderten Schema erzeugbar ist.

7.5.5 Zusammenfassung

Für eine Instanz der Klasse *unbiased*, *biased disjoint*, oder *subsumption equivalent* ($\Delta_I < \Delta_S$) muss im Zuge der Migration geprüft werden, ob diese bezüglich ihres Ausführungszustandes mit dem geänderten Schema verträglich ist. Mit den in diesem Abschnitt vorgestellten Verträglichkeitstests lässt sich diese Prüfung für ausnahmslos jede Instanz der genannten Klassen korrekt durchführen. Dabei wird die zu prüfende Knoten-Menge nicht anhand der einzelnen im Zuge einer Schemaänderung angewendeten Änderungsoperationen bestimmt, sondern anhand der Effekte aller angewendeten Operationen. Dadurch lässt sich die Zahl der zu betrachtenden Knoten entscheidend reduzieren und somit die Effizienz steigern. Die Effekte der angewendeten Änderungsoperationen können zudem direkt aus der Deltaschicht bestimmt werden, wodurch die Algorithmen auch bei einer Erweiterung des Änderungsrahmenwerkes korrekte Ergebnisse liefern. Ein weiterer Vorteil resultiert aus der Unabhängigkeit der einzelnen Tests. Dadurch ist die Reihenfolge der Vorgehensweise bei der Bestimmung der zustandsbasierten Verträglichkeit keinesfalls verbindlich. Somit liegt es in der Hand des Implementierers, abhängig vom jeweiligen Anwendungsgebiet und somit von der zu erwartenden Anzahl an zu löschenden, neu einzufügenden und zu verschiebenden Knoten abzuwägen, in welcher Reihenfolge die Tests in der Mehrheit der Fälle am schnellsten eine unverträgliche Instanz erkennen. Werden beispielsweise in einem bestimmten Anwendungsgebiet bei Instanzänderungen in der Regel hauptsächlich Knoten gelöscht, so empfiehlt es sich, den zustandsbasierten Verträglichkeitstest mit den gelöschten Knoten zu beginnen, da dort zwangsläufig am häufigsten mit Konflikten zu rechnen ist.

7.6 Neuberechnung von Knotenzuständen nach der Propagation von Schemaänderungen auf Instanzen

Wie die im vorherigen Abschnitt beschriebene zustandsbasierte Verträglichkeit kommt auch die Neubewertung von Knotenzuständen bei Instanzen mehrerer Klassen zum Einsatz. So ist es um eine Schemaevolution erfolgreich abzuschließen, für Instanzen der Klassen *unbiased*, *biased disjoint* und *subsumption equivalent* ($\Delta_I < \Delta_S$) notwendig, die instanzspezifische Markierung M^I neu zu berechnen. Dies muss eigentlich auch bei Instanzen vom Typ *partially equivalent* durchgeführt werden. Bei

diesen lassen sich die Zustände allerdings nicht automatisch berechnen, weshalb solche Instanzen hier nicht weiter berücksichtigt werden.
Die Neuberechnung von M^I wird notwendig, da durch die Propagation der Schemaänderungen von *S'* auf das instanzspezifische Schema von *I* (= S_I) neue Knoten hinzugefügt, verschoben oder entfernt wurden. Da das zugrundeliegende Laufzeitmodell, Manipulationen unmittelbar vor bzw. an bereits als *ACTIVATED* markierten Knoten zulässt, besteht nach dieser Propagation die Möglichkeit, dass Knotenzustände der ursprünglichen instanzspezifischen Markierung zurückgenommen und andere Knoten neu bewertet werden müssen. Dies wurde bereits in Abschnitt 3.4.3 (Neubewertung von Knotenzuständen) erläutert. Ein entsprechender Algorithmus der diese Aufgabe übernimmt wird in [Rind04] vorgestellt. Dieser verwendet allerdings als Ausgangsbasis die bei jeder Änderungsoperation zu hinterlegende Menge an neu zu bewertenden Knoten (vgl. Tabelle 0.6 Anhang A (Änderungsoperationen)). Um die Erweiterbarkeit des Änderungsrahmenwerkes zu gewährleisten, muss aber auch an dieser Stelle von konkreten Änderungsoperationen abstrahiert werden, weshalb bei der Entwicklung entsprechender Algorithmen wiederum nur auf die Deltaschicht zurückgegriffen werden darf.

Um die Menge der neu zu bewertenden Knoten so gering wie möglich zu halten wird nach der Initialisierung der neuen instanzspezifischen Markierung (Abschnitt 7.6.1), in den Abschnitten 7.6.2 und 7.6.3 beschrieben, welche Knoten für eine Neubewertung relevant sind und wie diese berechnet werden können. Die Algorithmen zur Neubewertung der relevanten Knoten werden in Abschnitt 7.6.4 vorgestellt und die dabei auftretenden Sonderfälle in Abschnitt 7.6.5 behandelt. Wie mit den entwickelten Algorithmen eine Zustandsneubewertung durchgeführt wird, zeigt Abschnitt 7.6.6 anhand eines konkreten Beispiels.

7.6.1 Initialisierung der instanzspezifischen Markierung

In einem ersten Schritt wird eine instanzspezifische Markierung erzeugt (M^I_{neu}), deren Struktur an die neuen Gegebenheiten angepasst ist. Das bedeutet für Instanzen der Klassen *unbiased* und *subsumption equivalent*, dass M^I_{neu} exakt die Knoten beinhaltet, die in *S'* vorhanden sind. Um eine korrekte Struktur bei Instanzen der Klasse *biased disjoint* zu erhalten werden hingegen zuerst alle Knoten aus *S'* in M^I_{neu} übernommen und danach diejenigen entfernt, die in der Deltaschicht von *I* als gelöscht markiert (*delNodes*) und diejenigen eingefügt, die in der Deltaschicht als neu eingefügt (*newNodes*) deklariert sind.
Nachdem die statische Struktur von M^I_{neu} feststeht, werden die in M^I_{neu} enthaltenen Knoten folgendermaßen initialisiert: Knoten die auch in *I* enthalten sind, erhalten genau den gleichen Zustand wie der Knoten in M^I. Knoten aus M^I_{neu} die nicht in M^I enthalten sind, werden dagegen mit *NOT_ACTIVATED* initialisiert. Abbildung 7.27 verdeutlicht die bisher beschriebenen Schritte anhand eines Beispiels.

Beziehen sich die Schemaänderungen auf einen Bereich, der bei der Ausführung von *I* noch nicht erreicht wurde, so ist die hier erzeugte instanzspezifische Markierung M^I_{neu} bereits korrekt. Wurden allerdings Manipulationen unmittelbar vor bzw. an bereits mit *ACTIVATED* markierten Knoten durchgeführt (vgl. Abbildung 7.27), ist eine Neubewertung von Knoten zwingend erforderlich.

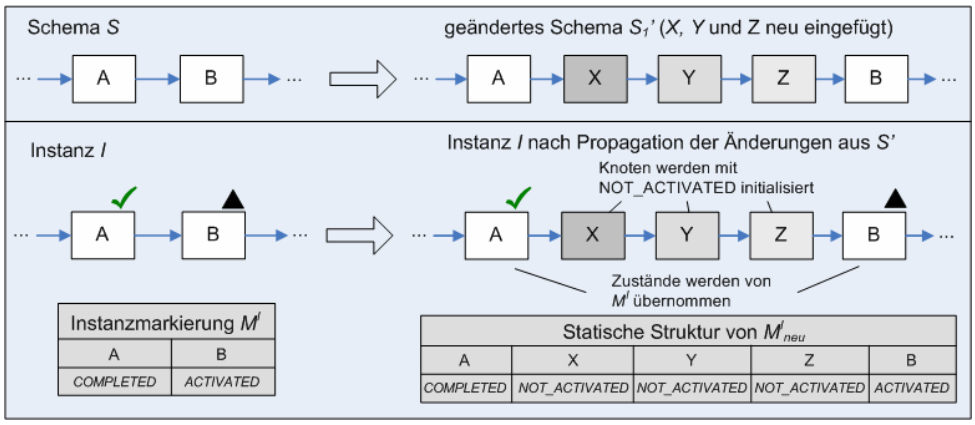


Abbildung 7.27 Statische Struktur von M^I_{neu} mit Initialisierung

7.6.2 Reduzieren der potentiell zu bewertenden Knotenmenge

Um eine nach Definition 2 korrekte Instanzmarkierung zu erhalten, ist es nicht notwendig, alle in M^I_{neu} vorhandenen Knoten neu zu bewerten (vgl. Abschnitt 3.4.3). So ist es im Normalfall[14] überflüssig, Knoten zu bewerten, deren Vorgänger ebenfalls neu eingefügt oder an die betrachtete Stelle verschoben worden sind. Abbildung 7.28 zeigt eine solche Situation. Hier muss der neu eingefügte Knoten *Y* nicht bewertet werden, da es sich bei dessen Vorgänger *X* ebenfalls um einen neu eingefügten Knoten handelt.

Eine ähnliche Reduzierung der zu bewertenden Knoten ergibt sich bei gelöschten oder von einer Stelle verschobenen Knoten. Wie Abbildung 7.28 zeigt, sind die Nachfolger solcher Knoten nicht neu zu bewerten, falls diese ebenfalls gelöscht oder von dieser Stelle verschoben worden sind.

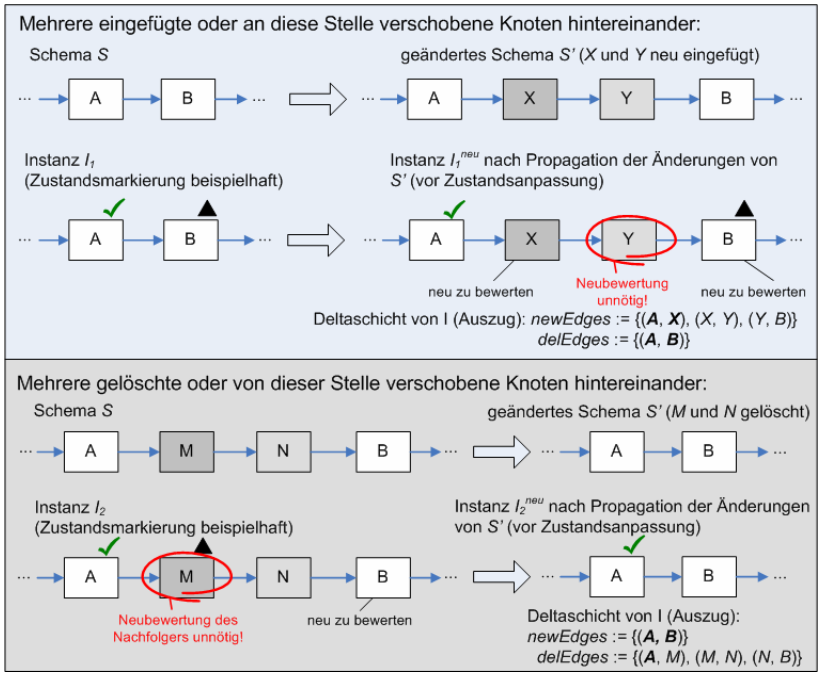


Abbildung 7.28 Reduzierung der neu zu bewertenden Knotenmenge

Bei genauer Betrachtung der Instanzen aus Abbildung 7.28 erkennt man noch ein weiteres Kriterium mit dem die neu zu bewertende Menge weiter eingeschränkt werden kann. Wie die beiden

[14] Die Ausnahmen werden in Abschnitt 7.6.5 beschrieben

abgebildeten Instanzen zeigen, ist eine Neubewertung im Normalfall nur dann notwendig, wenn der vordere Kontextknoten – im Beispiel *A* – den Zustand *COMPLETED* oder *SKIPPED* besitzt. Nur in diesem Fall kann der in I^{neu} auf den vorderen Kontextknoten nachfolgende Knoten neu bewertet werden kann. Bei Instanz I_1^{neu} ist dies der Knoten *X*, bei I_2^{neu} der Knoten *B*. Weiterhin ist es auch nur in diesem Fall möglich, dass Knotenmarkierungen zurückgenommen werden müssen (z.B. *B* in I_1). Allgemein betrachtet gilt dies auch für den Zustand von *M* in Instanz I_2. Hier erübrigt sich allerdings die Zustandsneubewertung, da gelöschte Knoten logischerweise keinen Einfluss auf die resultierende instanzspezifische Markierung M^I_{neu} haben.

7.6.3 Berechnung der relevanten Knotenmenge

Wie die für eine Zustandsneubewertung relevanten Knoten im Einzelnen berechnet werden, beschreibt der nachfolgende Algorithmus. Die notwendige Information kommt dabei aus der Deltaschicht von *S'* bzw. den daraus ableitbaren *Difference Sets* und der instanzspezifischen Markierung M^I.

I. In $CtrlE_{\Delta S}^{del}$ werden alle Quellknoten (*src*) gesucht, die in der instanzspezifischen Markierung M^I einen der Zustände *COMPLETED* oder *SKIPPED* besitzen. Ist ein solcher *src*-Knoten gefunden, so wird der zugehörige Zielknoten (*dest*) in die Menge der neu zu bewertenden Knoten (*nodesToReEvaluate*) eingefügt. Dies allerdings nur dann, wenn sich dieser nicht in der Menge der gelöschten Knoten $N_{\Delta S}^{del}$ befindet. Dadurch wird eine unnötige Neubewertung von gelöschten Knoten ausgeschlossen. Als Resultat enthält die Menge *nodesToReEvaluate* die neu zu bewertenden Knoten aus den folgenden Situationen:
   - Ein Knoten, eine Knotenmenge oder einer Teilzweig ist verschoben worden (vgl. Abbildung 7.29). Dies hat zur Folge, dass die Kante zwischen dem ersten verschobenen Knoten (*first*) und dessen Vorgänger (*pred*) in der Menge $CtrlE_{\Delta S}^{del}$ enthalten ist. Somit ist *first* in *nodesToReEvaluate* enthalten, falls sich *pred* in einem der Zustände *SKIPPED* oder *COMPLETED* befindet. *first* hat hierbei einen der Zustände *COMPLETED*, *ACTIVATED* oder *SKIPPED*.

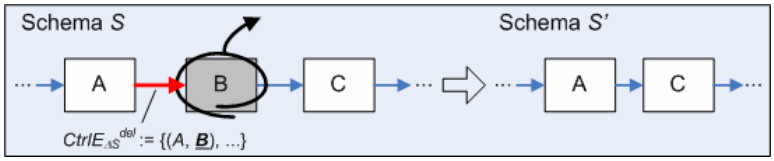


Abbildung 7.29 Ein Knoten, eine Knotenmenge oder ein Teilzweig wurde(n) verschoben

   - Es ist/sind ein oder mehrere Knoten zwischen zwei Knoten (*X*, *Y*) aus *S* eingefügt oder an diese Stelle verschoben worden (vgl. Abbildung 7.30). Dafür muss die Kante zwischen (*X*, *Y*) gelöscht werden. Es existiert also ein entsprechender Eintrag in $CtrlE_{\Delta S}^{del}$. Folglich ist *Y* in *nodesToReEvaluate*, falls *X* einen der Zustände *SKIPPED* oder *COMPLETED* besitzt. *Y* hat hierbei einen der Zustände *NOT_ACTIVATED* (möglich, wenn *dest* vom Typ *Join*), *ACTIVATED* oder *SKIPPED*. Zusätzlich ist bei verschobenen Knoten der Zustand *COMPLETED* möglich.

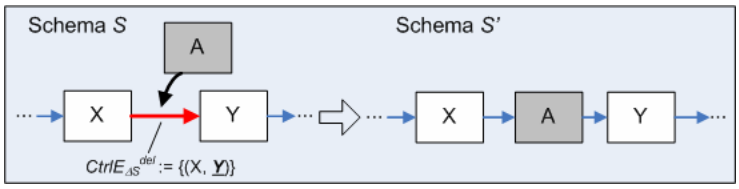


Abbildung 7.30 Ein oder mehrere Knoten wurden eingefügt

- Es ist ein Knoten gelöscht oder ein Knoten, eine Knotenmenge oder ein Teilzweig von einer bestimmten Stelle aus verschoben worden. $CtrlE_{\Delta S}^{del}$ enthält folglich eine Kante zwischen dem gelöschten (*del*) bzw. dem letzten verschobenen Knoten (*last*) und dessen Nachfolger (*succ*) in *S*. Somit ist *succ* in *nodesToReEvaluate* enthalten, falls sich *del* bzw. *last* in einem der Zustände *SKIPPED* oder *COMPLETED* befinden. *succ* hat hierbei einen der Zustände *NOT_ACTIVATED*, *ACTIVATED* oder *SKIPPED*. Zusätzlich ist bei verschobenen Knoten der Zustand *COMPLETED* möglich.

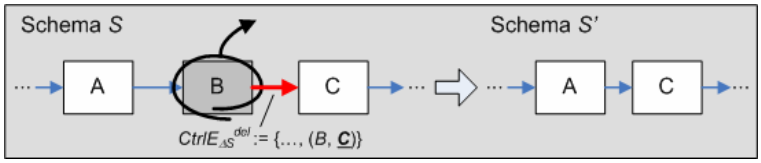


Abbildung 7.31 Ein Knoten wurde gelöscht oder ein Knoten eine Knotenmenge oder ein Teilzweig wurde(n) verschoben

Durch Anwendung der Vorgehensbeschreibung aus Punkt I lassen sich bereits die meisten neu zu bewertenden Knoten bestimmen. Was allerdings nicht aus $CtrlE_{\Delta S}^{del}$ ersichtlich ist, sind diejenigen Knoten, die im Zuge der Schemaänderung neu eingefügt worden sind. Zur Bestimmung dieser Knoten muss die Menge $CtrlE_{\Delta S}^{add}$ herangezogen werden. Es ergibt sich folgender Ablauf:

II. Für jeden bereits in Punkt I aus der Menge $CtrlE_{\Delta S}^{del}$ bestimmten Quellknoten (*src*), werden die Kanten aus $CtrlE_{\Delta S}^{add}$ ermittelt, die als Quellknoten exakt diesen *src* Knoten besitzen. Hat man eine solche gefunden, so wird der Zielknoten (*dest*) in die Menge *nodesToReEvaluate* eingefügt.
Durch diese Vorgehensweise wird die Anzahl der Zugriffe auf $CtrlE_{\Delta S}^{add}$ minimal gehalten. Dies wird dadurch möglich, weil einerseits die in Punkt I betrachteten *src* Knoten einen der Zustände *SKIPPED* oder *COMPLETED* besitzen und andererseits nur der direkt hinter einem solchen Knoten neu eingefügte Knoten bewertet werden muss (vgl. Abbildung 7.28). Alle anderen *src*-Knoten aus $CtrlE_{\Delta S}^{add}$ sind für die Neubewertung nicht von Bedeutung. Die folgende Abbildung verdeutlicht dies nochmals:

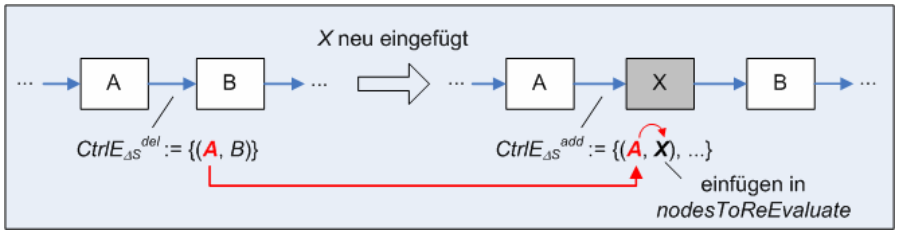


Abbildung 7.32 Bestimmung der neu zu bewertenden Knoten in $CtrlE_{\Delta S}^{add}$

Nach Abarbeitung der Punkte I und II enthält die Menge *nodesToReEvaluate* alle aus der Anwendung von Einfüge-, Lösch- und Verschiebeoperationen resultierenden neu zu bewertenden Knoten. Dies ist allerdings noch nicht ausreichend, da auch neu eingefügte oder gelöschte Sync-Kanten ein Neubewerten von Knoten erfordern können.
Bei neu eingefügten Sync-Kanten ist dies immer genau dann der Fall, wenn sich der Zielknoten im Zustand *ACTIVATED* befindet. In dieser Konstellation besteht die Möglichkeit, dass diese Zustandsmarkierung zurückgenommen werden muss, da sich der Quellknoten der neu eingefügten Sync-Kante noch nicht im erforderlichen Zustand *COMPLETED* befindet.
Bei gelöschten Sync-Kanten ist die Situation prinzipiell genau umgekehrt. Hier ist es möglich, dass der ehemalige Zielknoten aktiviert werden kann, da außer dem Quellknoten der Sync-Kante alle anderen Vorgängerknoten den Zustand *COMPLETED* besitzen.
Bestimmen lassen sich diese Knoten, indem man für die Kanten aus den Mengen $SyncE_{\Delta S}^{add}$ und $SyncE_{\Delta S}^{del}$ die gerade beschriebenen Zustandstests durchführt. Befindet sich ein Zielknoten in einem der geforderten Zustände, so wird dieser in die Menge *nodesToReEvaluate* eingefügt.

7.6.4 Neubewertung

Bei der Neubewertung eines Knotens aus der Menge *nodesToReEvaluate* wird in einem ersten Schritt geprüft, ob sich dieser in M^{I}_{neu} in einem der Zustände *NOT_ACTIVATED* oder *ACTIVATED* befindet. Diese Zustandsabfrage ist notwendig, da sich einerseits verschobene Knoten auch in einem der Zustände *SKIPPED*, *RUNNING* oder *COMPLETED* befinden können und andererseits auch das Einfügen neuer Knoten in *SKIPPED*-Zweige erlaubt ist. Die *SKIPPED* Knoten bedürfen einer gesonderten Betrachtung (siehe Sonderfall 3 und 4, Abschnitt 7.6.5), während eine Neubewertung von *RUNNING*- und *COMPLETED*-Knoten überhaupt nicht ausgeführt werden darf, da sich sonst falsche Zustände ergeben.
Die Bewertung selbst erfolgt analog zu den in Abschnitt 3.1.2 vorgestellten Markierungs- und Ausführungsregeln. Um dazu an die für eine Bewertung notwendigen Zustände der Vorgängerknoten zu gelangen, können direkt die Methoden *getSuccByEdgeType* und *getNodeState* des Datenmodells verwendet werden (vgl. Abschnitt 5.1.2.1).

7.6.5 Behandlung von Sonderfällen

Neben den im vorherigen Abschnitt beschriebenen Ablauf sind noch einige Sonderfälle zu beachten, von denen die ersten beiden bereits in [Rind04] behandelt werden:

1. Ist der *Selection Code* einer Kante geändert worden und besitzt der *src*-Knoten dieser Kante den Zustand *COMPLETED*, so besteht die Möglichkeit, dass ein vormals gewählter Teilzweig einer alternativen Verzweigung das Entscheidungskriterium nicht mehr erfüllt und somit nicht gewählt werden darf. Als Folge davon, müssen alle Nachfolger des *Split*-Knotens neu bewertet werden. Ändert sich der Zustand eines Knotens von *SKIPPED* auf *ACTIVATED* oder umgekehrt, so müssen nach dessen Neubewertung wiederum die Nachfolger dieses Knotens auf gleiche Weise neu bewertet werden. Dies wird so lange wiederholt, bis sich der Zustand eines Knotens nach dessen Neubewertung nicht mehr verändert. Dies ist genau dann der Fall, wenn der zum *Split*-Knoten zugehörige *Join*-Knoten erreicht ist. Das bei diesem Vorgang unter Umständen die Neubewertung von in Punkt I und II (siehe Abschnitt 7.6.3) betrachteten Knoten überschrieben wird, beeinflusst nicht die Korrektheit der resultierenden instanzspezifischen Markierung.
2. Ist durch die Methode *convertBlock* der Typ eines Verzweigungsknotens von *OR*- auf *AND*-*Split* oder umgekehrt geändert worden und befindet sich dieser Knoten im Zustand

COMPLETED, so hat dies Auswirkungen auf dessen Nachfolgerknoten. Eine Änderung eines *OR-Split*-Knotens in einen *AND-Split*-Knoten erfordert ein Zurücknehmen der *SKIPPED*-Zustände der abgewählten Zweige. Stattdessen müssen jeweils die vordersten Knoten eines solchen Zweiges auf *ACTIVATED* gesetzt werden. Bei der Änderung eines *AND-SPLIT*-Knotens in eine *OR-SPLIT*-Knoten lässt sich hingegen exakt der gleiche Mechanismus verwenden, der auch bei Sonderfall 1 zum Einsatz kommt, da hier *Selection Codes* neu eingefügt und damit geändert werden.

3. Ist die Bearbeiterzuordnung eines als *ACTIVATED* markierten Knotens geändert worden, so wirkt sich dies zwar nicht auf die Neubewertung von Knoten aus, es muss aber ein entsprechender Bearbeitungsauftrag – entsprechend der geänderten Bearbeiterzuordnung – aus der Arbeitsliste eines Bearbeiters gelöscht und in die Arbeitsliste eines anderen Bearbeiters eingefügt werden. Dieser Vorgang ist allerdings nicht Aufgabe des Änderungsrahmenwerkes, weshalb in einem solchen Fall lediglich entsprechende Methoden des *WorklistManagers* aufgerufen werden.
4. Ein weiterer speziell zu behandelnder Fall tritt immer dann auf, wenn ein Knoten, eine Knotenmenge oder ein Teilzweig verschoben wird, deren Knoten sich im Zustand *SKIPPED* befindet/befinden. Da diese Knoten nicht in der Ausführungshistorie auftreten, können sie auch noch verschoben werden, nachdem im Kontrollfluss nachfolgende Knoten bereits beendet worden sind. Als Ziel der Verschiebung sind alle Stellen zulässig, an denen der Nachfolgeknoten einen der Zustände *NOT_ACTIVATED*, *ACTIVATED* oder *SKIPPED* besitzt. Das Neubewerten des ersten Knotens (*first*) einer verschobenen Knotenmenge reicht hier nicht aus, wenn die Neubewertung von *first* den Zustand *NOT_ACTIVATED* oder *ACTIVATED* ergibt. In diesem Fall müssen analog zum Sonderfall 1 die Zustände der folgenden verschobenen Knoten von *SKIPPED* auf *NOT_ACTIVATED* gesetzt werden.
5. Sind Knoten in eine *SKIPPED*-Zweige neu eingefügt worden, so ist ein Bewerten des nachfolgenden Knotens unnötig, da sich dessen Zustand durch den neu eingefügten Knoten nicht verändern kann.

7.6.6 Anwendungsbeispiel

Um zu zeigen, wie mit den in den Abschnitten 7.6.1-7.6.5 beschriebenen Algorithmen die Zustände von einer Migration betroffener Knoten korrekt bewertet werden können, wird die Erzeugung der neuen instanzspezifischen Markierung anhand des Beispiels aus Abbildung 7.33 im Einzelnen erläutert.

Bei der Instanz *I*, im betrachteten Beispiel, handelt es sich um eine Instanz vom Typ *unbiased.* Um die Struktur der neuen instanzspezifischen Markierung M^I_{neu} zu erzeugen, genügt es also, alle Knoten aus *S'* in M^I_{neu} zu übernehmen. Danach werden die Knoten initialisiert. Dafür prüft man für jeden Knoten aus M^I_{neu}, ob dieser auch in M^I vorhanden ist und übernimmt gegebenenfalls dessen Zustandsmarkierung. Die restlichen Knoten werden mit *NOT_ACTIVATED* initialisiert. Man erhält M^I_{neu} := {({*A*, *B*, *C*, *D*, *F*, *M*}, *COMPLETED*), ({*E*, *H*}, *ACTIVATED*), ({*N*, *M*}, *SKIPPED*), ({*X*, *J*, *K*, *S*, *T*}, *NOT_ACTIVATED*) (vgl. Abbildung 7.34).

Dann müssen die neu zu bewertenden Knoten bestimmt werden. Nach der in Punkt I beschriebenen Vorgehensweise sucht man zuerst in $CtrlE_{\Delta S}^{del}$ (bzw. in *delEdges* vom Typ *Control*) nach Quellknoten die sich in einem der Zustände *SKIPPED* oder *COMPLETED* befinden. Im hier betrachteten Beispiel erfüllen die Knoten *M*, *F*, *B* und *C* dieses Kriterium. Befinden sich die zu diesen Quellknoten zugehörigen Zielknoten nicht in der Menge der gelöschten Knoten, so werden diese in die Menge der

neu zu bewertenden Knoten (*nodesToReEvaluate*) eingefügt. Es ergibt sich bis hierhin für *nodesToReEvaluate* := {*R*, *C*, *D*}.

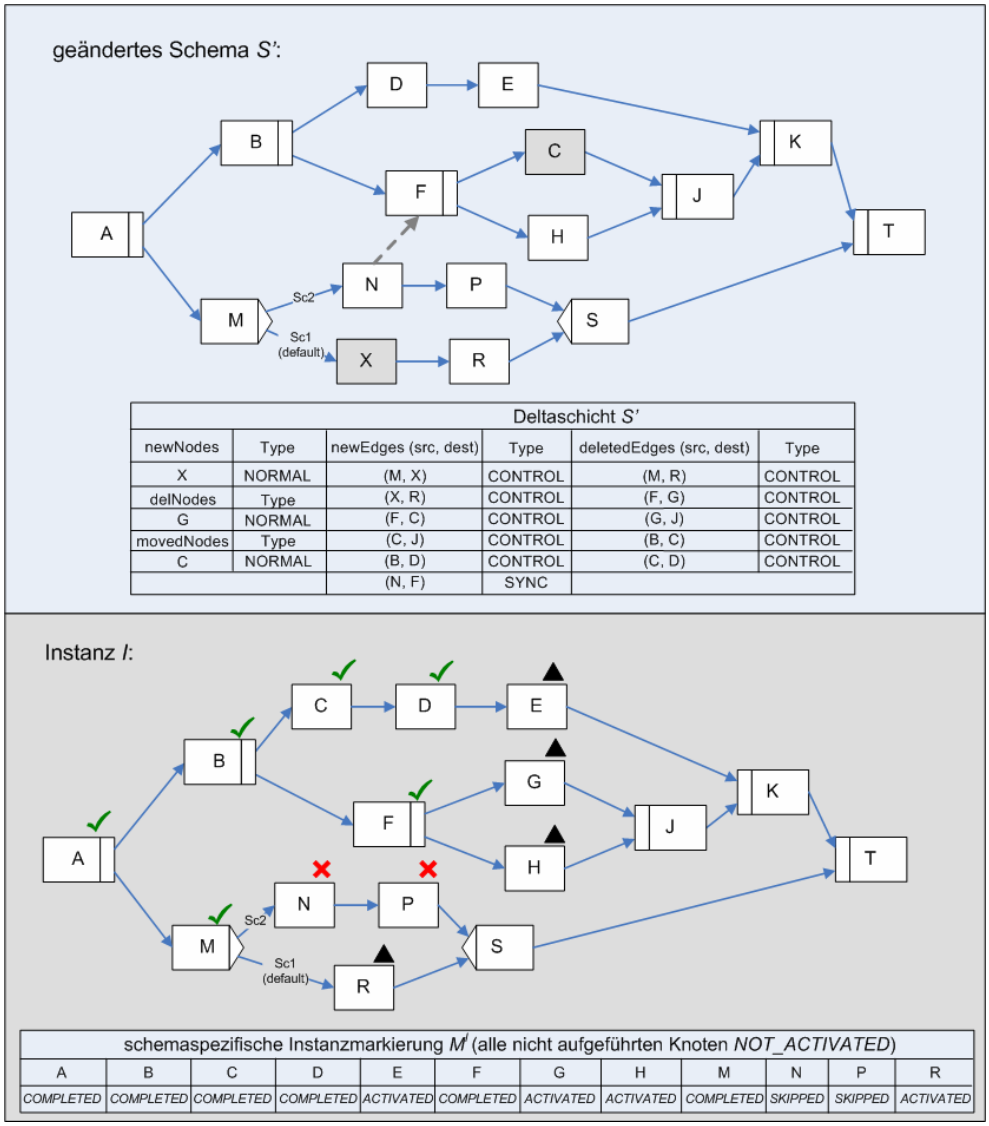
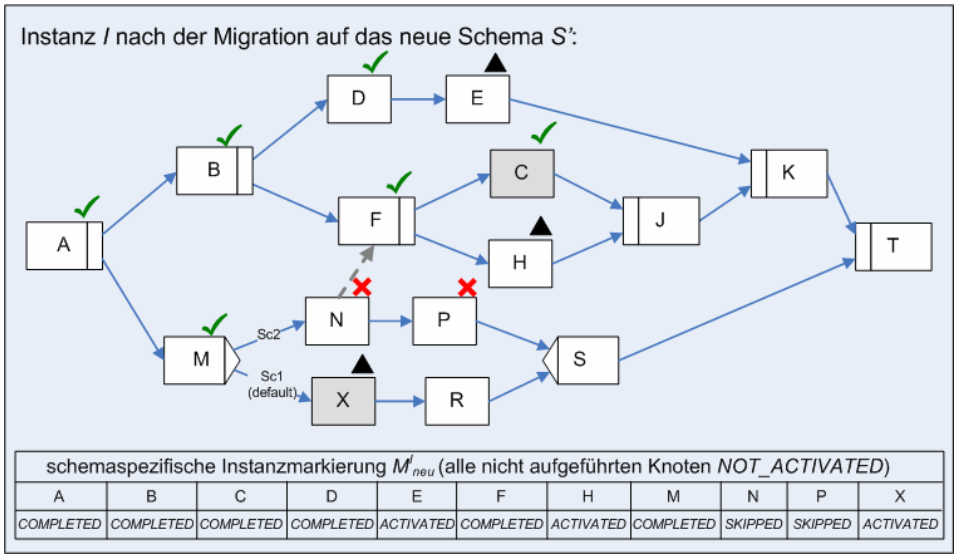


Abbildung 7.33 Beispiel zur Zustandsanpassung

Anschließend wird analog zu Punkt II für jeden der gefundenen Quellknoten *M*, *F*, *B* und *C* in $CtrlE_{\Delta S}^{add}$ (bzw. in *newEdges* vom Typ *Control*) nach Einträgen gesucht die ebenfalls einen dieser Knoten als Quellknoten besitzen. Man findet die Kanten (*M*, *X*), (*F*, *C*), (*B*, *D*) und (*C*, *J*). Hiervon sind jeweils die Zielknoten neu zu bewerten und somit der Menge *nodesToReEvaluate* hinzuzufügen. Weiterhin ist im Zuge der Schemaänderung eine Sync-Kante zwischen *N* und *F* neu eingefügt worden. Da sich der Zielknoten *F* allerdings nicht im Zustand *ACTIVATED* befindet muss dieser nicht neu bewertet werden. Somit ist die Menge der neu zu bewertenden Knoten vollständig. Man erhält für *nodesToReEvaluate* := {*R*, *C*, *D*, *X*, *J*}.

Im nächsten Schritt werden die gefundenen Knoten neu bewertet. Dies darf jedoch nur für diejenigen Knoten durchgeführt werden die sich nicht in einem der Zustände *RUNNING* oder *COMPLETED* befinden. Die Knoten *C* und *D* scheiden somit von vornherein aus, da sie sich implizit im richtigen Zustand befinden. Die Zustände der Knoten *R*, *X* und *J* sind hingegen anzupassen. Dabei werden – wie beschrieben – die Zustände der Vorgänger verwendet. Für den Knoten *R* ergibt sich der neue Zustand *NOT_ACTIVATED*, da nach der Migration der Instanz *I* sein einziger Vorgänger *X* den Zustand

NOT_ACTIVATED besitzt. *X* erhält den Zustand *ACTIVATED*, da sich dessen direkter Vorgänger in dem für die Aktivierung von *X* notwendigen Zustand *COMPLETED* befindet. Dabei spielt es keine Rolle, ob zuerst *X* und dann dessen Nachfolger *R* bewertet wird. Denn selbst wenn *X* vor der Neubewertung von *R* bereits mit *ACTIVATED* markiert worden ist, ändert sich an dem resultierenden Zustand für *R* nichts. Bei *J* handelt es sich um einen *AND_JOIN*-Knoten. Dieser wird genau dann mit *ACTIVATED* markiert, wenn sich alle seine Vorgänger im Zustand *COMPLETED* befinden. In dem Beispiel aus Abbildung 7.34 ist das nicht der Fall. Der Vorgänger *C* befindet sich zwar im Zustand *COMPLETED*, der zweite Vorgänger *H* besitzt aber den Zustand *ACTIVATED*. Folglich wird *J* nicht aktiviert sondern erhält den Zustand *NOT_ACTIVATED*. Damit sind alle Knoten der Menge *nodesToReEvaluate* bewertet. Die Sonderfälle treten im betrachteten Beispiel nicht auf, weshalb hiermit die Berechnung der neuen instanzspezifischen Markierung abgeschlossen ist. Es resultiert die in der Abbildung 7.34 gezeigte Markierung M^{I}_{neu}.



schemaspezifische Instanzmarkierung M^{I}_{neu} (alle nicht aufgeführten Knoten *NOT_ACTIVATED*)

A	B	C	D	E	F	H	M	N	P	X
COMPLETED	*COMPLETED*	*COMPLETED*	*COMPLETED*	*ACTIVATED*	*COMPLETED*	*ACTIVATED*	*COMPLETED*	*SKIPPED*	*SKIPPED*	*ACTIVATED*

Abbildung 7.34 Resultierende Markierung nach Migration auf *S'*

7.6.7 Zusammenfassung

Um die Migration von Instanzen der Klassen *unbiased*, *biased disjoint* und *subsumption equivalent* ($\Delta_I < \Delta_S$) erfolgreich abzuschließen, müssen die Zustände der von den Schemaänderungen betroffenen Knoten angepasst werden. Die dazu notwendigen Algorithmen wurden in diesem Abschnitt beschrieben. Dabei wurde unter Ausnutzung des Kontextes und der instanzspezifischen Markierung von *I*, die Menge der für eine Neubewertung relevanten Knoten minimal gehalten, was die Neuberechnung einer Instanzmarkierung beschleunigt. Weiterhin werden von allen Algorithmen ausschließlich Informationen aus der Deltaschicht und aus der instanzspezifischen Markierung von *I* verwendet. Dadurch ist die Abstraktion von konkreten Änderungsoperationen gewährleistet, wodurch die Erweiterbarkeit des Änderungsrahmenwerks erhalten bleibt.

7.7 Migration von partially equivalent Instanzen

Die in den Abschnitten 7.3 - 7.6 vorgestellten Algorithmen ermöglichen es die Instanzen der Klassen *disjoint*, *equivalent* und *subsumption equivalent* abschließend auf ein geändertes Schema zu migrieren. Ein Benutzereingriff ist dabei nicht notwendig. Das Änderungsrahmenwerk übernimmt, abhängig von der Klassenzugehörigkeit einer Instanz, automatisch die Durchführung der notwendigen Schritte.
Instanzen der Klasse *partially equivalent* können hingegen nur dann ohne Eingriff des Benutzers migriert werden, wenn sich bei einem Vergleich der jeweiligen Änderungsprojektionen (*change*

projections) von Δ_I und Δ_S ergibt, dass diese analog zu den Änderungen der Klassen *disjoint*, *equivalent* oder *subsumption equivalent* behandelt werden können (vgl. Abschnitt 7.2.2.5). Ist allerdings eine Änderungsprojektion von Δ_I wiederum *partially equivalent* zu der entsprechenden Projektion von Δ_S, so ist eine vollständig automatische Migration nicht möglich.
Damit das Änderungsrahmenwerk aber auch in diesem Fall sinnvoll eingesetzt werden kann, ist ein Konzept zu entwickeln, bei dem die Migration dieser *partially equivalent* Instanzen so weit wie möglich automatisiert und der Anwender bei manuell durchzuführenden Migrationsschritten bestmöglich unterstützt wird.

Dazu werden in Abschnitt 7.7.1 diejenigen Konflikte zwischen den Änderungen aus Δ_I und den Änderungen aus Δ_S beschrieben, die dazu führen, dass eine Instanz nicht automatisch migriert werden kann. Anhand dieser Konflikte wird in Abschnitt 7.7.2 ein Verträglichkeitstest entwickelt mit dem die vom Ausführungszustand verträglichen Instanzen zuverlässig erkannt werden können. Wie diese auf das geänderte Schema migriert werden und welche Konzepte dabei das Änderungsrahmenwerk zur Unterstützung des Benutzers bereitstellt, wird in Abschnitt 7.7.3 erläutert.

7.7.1 Konflikte

Nach dem bei der Klasseneinteilung in Abschnitt 7.2.2.5 angegebenen Kriterium ist eine Instanz zur Klasse *partially equivalent* zuzuordnen, wenn sie keines der Kriterien der anderen Klassen erfüllt. Aus dieser Komplementbildung lässt sich nicht erkennen welche Änderungen aus Δ_I und Δ_S Konflikte verursachen und somit verantwortlich sind, dass eine Instanz nicht ohne Benutzereingriff migriert werden kann. Für einen automatisch durchführbaren Verträglichkeitstest und bei der Unterstützung der manuellen Migration ist es aber zwingend notwendig die aufgetretenen Konflikte genau zu kennen. Deshalb werden im Folgenden die möglichen Konfliktsituationen im Einzelnen beschrieben (vgl. [Rind04]):

- Unterschiedliche Reihenfolge (*Different Order*): Sowohl auf Schema- als auch auf Instanzebene sind die gleichen Knoten (oder eine Untermenge) in den gleichen Zielkontext eingefügt oder verschoben worden. Allerdings unterscheidet sich die resultierende Reihenfolge der Knoten auf Instanzebene von derjenigen auf Schemaebene (vgl. Abbildung 7.35).

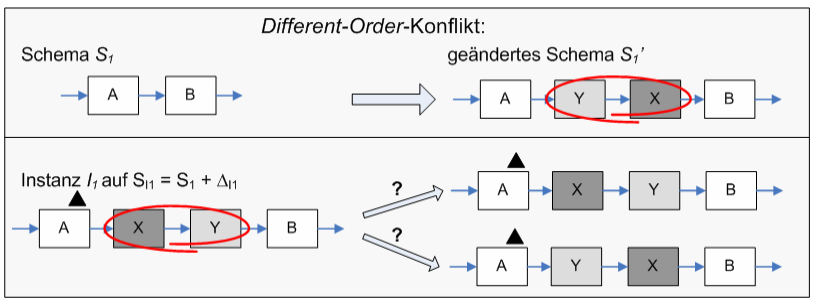


Abbildung 7.35 *Different-Order*-Konflikt

- Unterschiedliche Anker (*Different Anchors*): Es sind die gleichen Knoten in unterschiedliche Zielkontexte eingefügt oder verschoben worden (vgl. Abbildung 7.36).

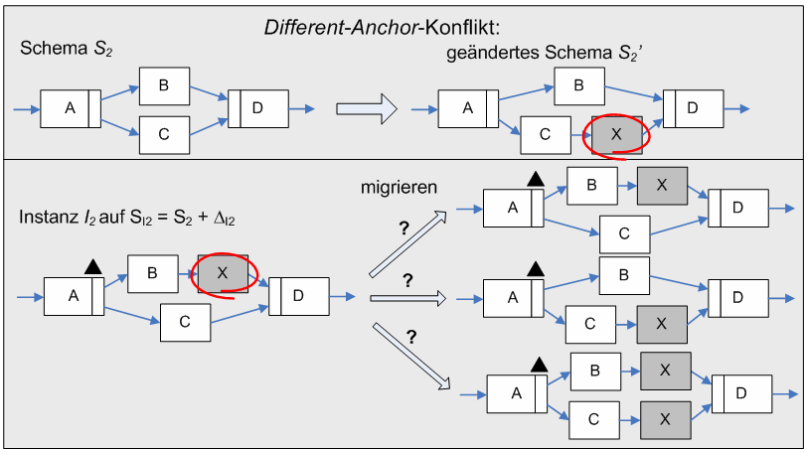


Abbildung 7.36 *Different-Anchor*-Konflikt

- Konkurrierender Zielkontext (*Conflicting Target Context*)[15]: Auf Schema- und Instanzebene ist ein Knoten (bzw. mehrere Knoten) in den gleichen Zielkontext eingefügt oder verschoben worden. Es handelt sich jedoch nicht um den/die gleichen Knoten (vgl. Abbildung 7.37).

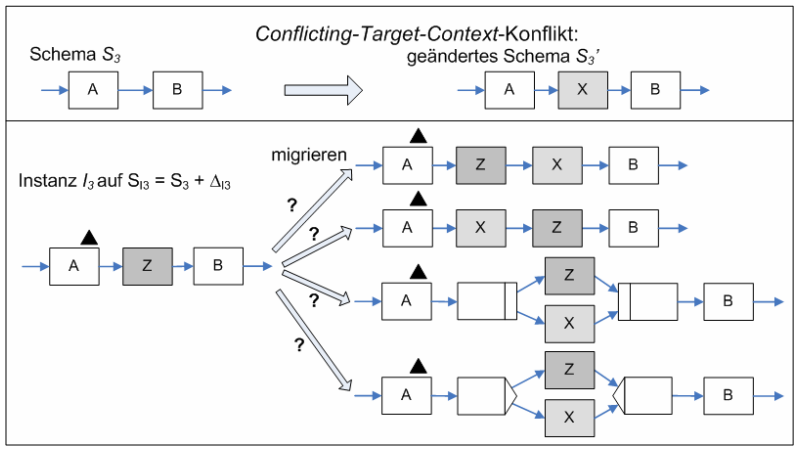


Abbildung 7.37 *Conflicting-Target-Context*-Konflikt

- Gleiche Elemente (*Multiple Operations*): Änderungen auf Schema- und Instanzebene betreffen das gleiche Element (z.B. doppeltes Einfügen oder Löschen des gleichen Knotens, vgl. Abbildung 7.38).

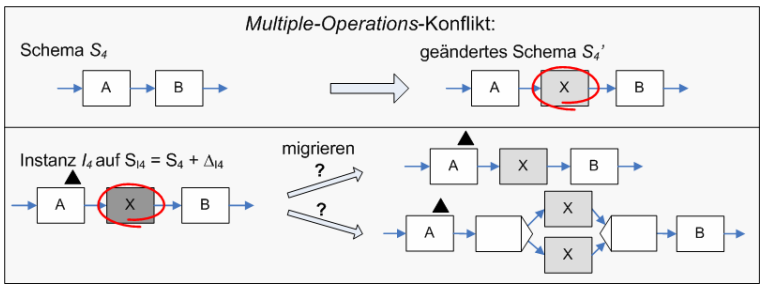


Abbildung 7.38 *Multiple-Operations*-Konflikt

[15] Siehe hierzu die Erläuterungen in Abschnitt 3.4.4.2.3

- Kontextzerstörende Änderungen (*Context Destroying Operations*)[15]: Eine Änderung auf Schemaebene zerstört den Zielkontext einer Änderung auf Instanzebene oder umgekehrt (vgl. Abbildung 7.39).

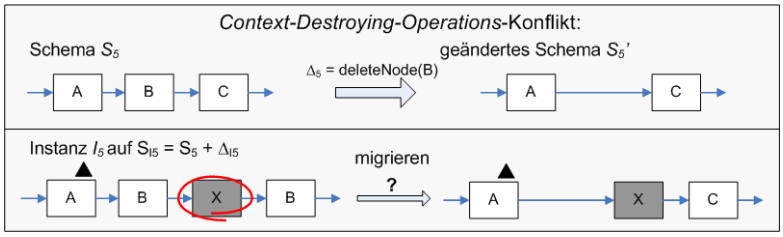


Abbildung 7.39 *Context-Destroying-Operations*-Konflikt

Mit welchen Algorithmen diese Konflikte anhand der Deltaschichten von *I* und *S'* berechnet werden können, wird in Anhang F (Algorithmen und Tests zu *partially equivalent)* beschrieben.

7.7.2 Zustandsbasierte Verträglichkeit

Um von vornherein die Anzahl manuell zu migrierender Instanzen minimal zu halten, ist es mit Hilfe der Konflikte möglich, die zustandsbasierte Verträglichkeit der *partially equivalent*-Instanzen zu bestimmen. Dabei ist die Kenntnis über die zwischen Δ_I und Δ_S aufgetretenen Konflikte von entscheidender Bedeutung, da der in Abschnitt 7.5 für die Instanzen der Klassen *unbiased*, *biased disjoint* und *subsumption equivalent* ($\Delta_S > \Delta_I$) vorgestellte Verträglichkeitstest, angewendet auf *partially equivalent*-Instanzen, bei Auftreten eines solchen Konflikts zu einem falschen Ergebnis führt. Das Beispiel aus Abbildung 7.40 zeigt einen solchen Fall:

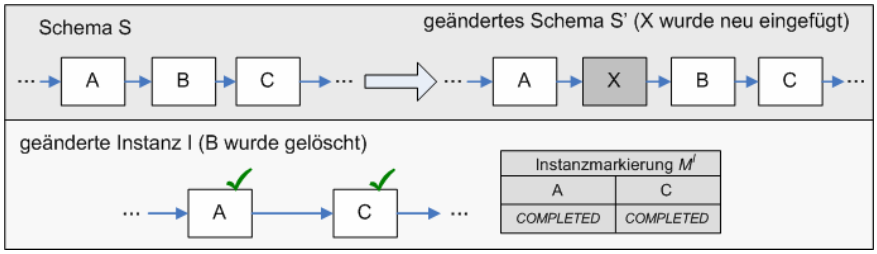


Abbildung 7.40 Zustandsbasierte Verträglichkeit wird falsch erkannt

Der auf Instanzebene gelöschte Knoten *B* fungiert bei der Einfügung von *X* auf Schemaebene als hinterer Kontextknoten. Da sich der Knoten *C* zum Zeitpunkt der Migration von *I* auf *S'* im Zustand *COMPLETED* befindet, führt ein Propagieren der Schemaänderungen in jedem Fall zu einer inkonsistenten Zustandsmarkierung. Der Test aus Abschnitt 7.5 erkennt hier allerdings keinen Zustandskonflikt. Dies liegt daran, dass im Zuge der Reduzierung der für einen Verträglichkeitstest relevanten Knoten, nur derjenige direkte Nachfolger eines neu eingefügten Knotens betrachtet wird, der auch in M^I vorhanden ist. Da es sich bei *C* aber nicht um eine direkten Nachfolger von *X* handelt und *B* nicht in M^I enthalten ist erkennt der Test die Instanz als verträglich.
Um dies zu verhindern muss für *partially equivalent* Instanzen ein zusätzlicher Verträglichkeitstest hinter den Test aus Abschnitt 7.5 geschaltet werden. Dieser kommt immer genau dann zum Einsatz, wenn ein oder mehrere der in Abschnitt 7.7.1 beschriebenen Konflikte entdeckt worden sind. Um hierbei gezielt vorgehen zu können, muss für jeden der Konflikte ein entsprechender Verträglichkeitstest definiert werden. Für den *Context-Destroying-Operations*-Konflikt – wie in Abbildung 7.40 – muss beispielsweise der folgende Test ausgeführt werden:

Prüfe ob der erste ebenfalls in I vorhandene Nachfolger des auf Schemaebene eingefügten Knotens in M^I maximal den Zustand *ACTIVATED* besitzt.

Wie diese Tests auch für die anderen *partially equivalent*-Konflikte aussehen, wird in Anhang F (Algorithmen und Tests zu *partially equivalent)* ausführlich beschrieben.

7.7.3 Unterstützung des Anwenders bei der manuellen Migration

Wie der Benutzer bei der manuellen Migration der als verträglich erkannten *partially equivalent* Instanzen vom Änderungsrahmenwerk unterstützt werden kann, wird in den folgenden zwei Abschnitten beschrieben.

7.7.3.1 Konflikt-Objekte

Um dem Anwender die manuelle Migration zu erleichtern, kann für jeden entdeckten Konflikt zwischen Δ_I und Δ_S ein sogenanntes Konflikt-Objekt erzeugt werden. Dieses enthält neben den Elementen die in Konflikt zueinander stehen, auch Informationen zu deren Kontext (z.B. Anker betroffener Knoten). Aus diesen Objekten lässt sich im *Editor* eine Liste erzeugen anhand derer dem Anwender die Konflikte angezeigt werden. Mit Hilfe der präzisen Information kann dieser dann die betrachtete Instanz durch geeignete Anpassungen auf das geänderte Schema migrieren.

7.7.3.2 Hybrid-Graph

Die Konflikt-Objekte allein reichen aber nicht aus um eine manuelle Migration umfassend zu unterstützen. Ohne weitere Maßnahmen ist es die Aufgabe des Anwenders die Instanz so zu ändern, dass sie auf das geänderte Schema korrekt migriert werden kann. Es ist aus Benutzersicht aber nicht nachvollziehbar, dass einerseits bei der Durchführung von Schema- und Instanzänderungen die Korrektheit eines Schemas bzw. einer Instanz vom System garantiert wird, andererseits aber bei der manuellen Migration die Verantwortung in der Hand des Anwenders liegt. Es muss also ein Konzept entwickelt werden, bei dem die Änderungen an der zu migrierenden Instanz mit Hilfe der ohnehin angebotenen Änderungsoperationen durchgeführt werden können und gleichzeitig die Möglichkeit einer korrekten Migration auf das geänderte Schema erhalten bleibt.

Entscheidend hierbei ist das Ausgangsschema auf dem die Änderungsoperationen angewendet werden. Es gibt grundsätzlich zwei Möglichkeiten:

1. Man geht von dem geänderten Schema aus und manipuliert die Instanz entsprechend.
2. Man geht von der geänderten Instanz aus und führt auf dieser die Schemaänderungen durch.

Beide Möglichkeiten haben Vor- und Nachteile. So ist es im ersten Fall zwar möglich, die Instanzen auf das geänderte Schema zu migrieren, d.h. die Instanz auf das geänderte Schema umzuhängen, andererseits enthält das Schema aber keine Zustände. Dadurch kann man nicht automatisch verhindern, dass der Anwender Änderungen an Graphteilen durchführt, die bereits auf Instanzebene durchlaufen wurden und somit einen inkorrekten Graph erzeugt. Geht man hingegen von der geänderten Instanz aus, so enthält der Graph die notwendigen Markierungen. Damit dieser aber auf das geänderte Schema umgehängt werden kann, muss der Anwender exakt alle Änderungen durchführen die zum geänderten Schema geführt haben. Anderenfalls muss die Instanz auf dem alten Schema weiterlaufen.

Keine der beschriebenen Ausgangschemata ist also wirklich praktikabel. Es ist daher notwendig für die manuelle Migration einen Ausgangsgraphen zu erzeugen, der die Stärken beider Konzepte vereint. Man muss folglich einen „hybriden“ Graph erzeugen. Dieser setzt sich folgendermaßen zusammen:
Ausgangsbasis ist das geänderte Schema *S'*. Auf diesem werden automatisch diejenigen Instanzänderungen eingebracht, die in dem auf Instanzebene bereits durchlaufen Graphabschnitt liegen. Dies ist immer möglich, da nur die Instanzen verträglich sind, bei denen die zu propagierenden Schemaänderungen in einem Graphabschnitt liegen, der auf Instanzebene noch nicht durchlaufen wurde.
Dann wird die instanzspezifische Markierung derjenigen Knoten von *I* übernommen, die sich nicht im Zustand *NOT_ACTIVATED* befinden.
Auf dem so erzeugten Graphen können die Instanzänderungen des Graphteils, der noch nicht durchlaufen wurde, mit den Standard-Änderungsoperationen durchgeführt werden. Dadurch sind einerseits die Korrektheit der Instanz und andererseits die Möglichkeit, die Instanz auf das geänderte Schema umzuhängen, implizit gewährleistet.

Wie ein solcher *Hybrid-Graph* zusammen mit der Liste der aufgetretenen Konflikte aus Abschnitt 7.7.3.1 bei der manuellen Migration eingesetzt werden kann, lässt sich aus Abbildung 7.41 erkennen.

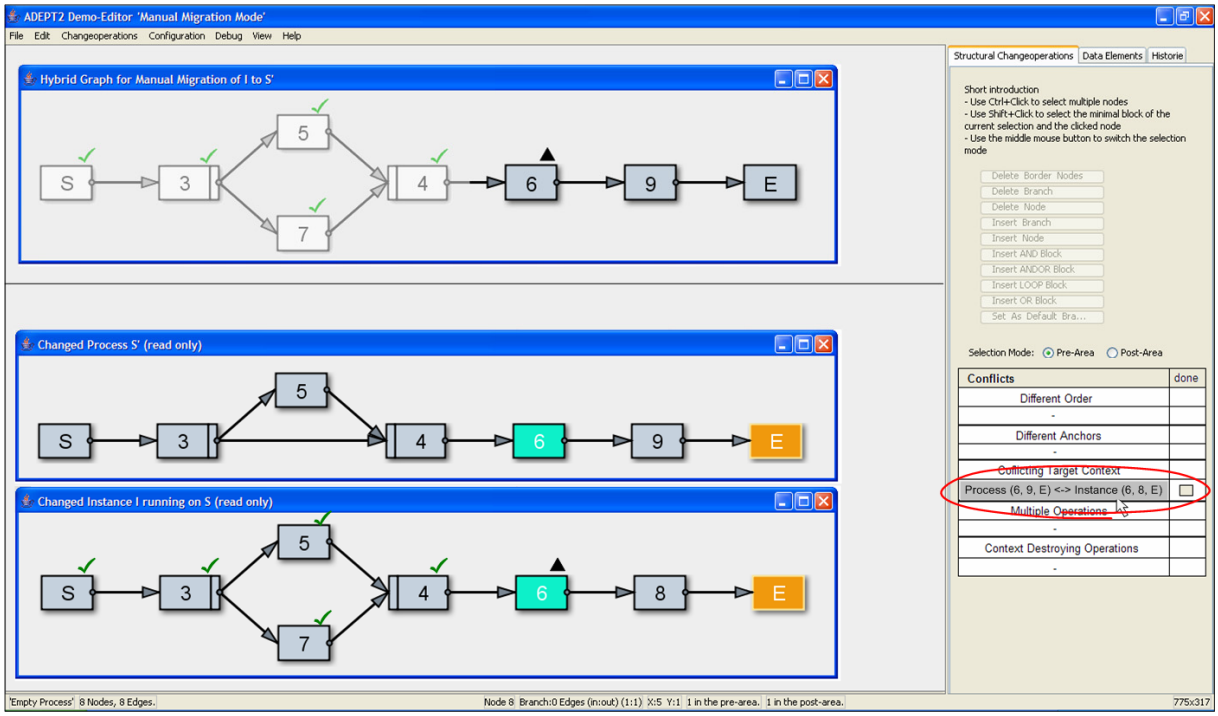


Abbildung 7.41 Beispiel für eine manuelle Migration mit Hilfe des Hybrid-Graphen

Die Abbildung zeigt den hybriden Graphen zusammen mit dem geänderten Schema und der geänderten Instanz. Dabei ist das geänderte Schema durch Einfügen des Knotens *9* zwischen den Knoten *6* und *E* entstanden. Bei der Instanz sind gegenüber dem Originalschema *S* der Knoten *7* zwischen *2* und *4* und der Knoten *8* zwischen *6* und *E* hinzugekommen. Die Änderungen auf Schemaebene sind folglich mit denen auf Instanzebene *partially equivalent*. Bei Berechnung der Konflikte wird deshalb ein *Conflicting-Target-Context*-Objekt mit dem Inhalt „*Process* (*6*, *9*, *E*)“ und „*Instance* (*6*, *8*, *E*)“ erzeugt. Dieses wird dem Benutzer in der Konfliktliste angezeigt. Der *Hybrid-Graph* ergibt sich für das betrachtete Beispiel, indem man ausgehend von dem geänderten Schema den Knoten *7* automatisch zwischen *3* und *4* einfügt und die Instanzmarkierung bis zum Knoten *6*

übernimmt. Dabei wird der bereits auf Instanzebene durchlaufene Graph ausgegraut dargestellt, um auch optisch zu verdeutlichen, dass in diesem Teil keine Änderungen mehr durchgeführt werden können.

Bei der manuellen Änderung ergibt sich daraus der folgende Ablauf:
Der Anwender klickt in der Konfliktliste auf einen Konflikt. Im Beispiel aus Abbildung 7.41 auf den Eintrag „*Process* (*6*, *9*, *E*) <-> *Instance* (*6*, *8*, *E*)“. Dadurch werden die betroffenen Knoten sowohl im Instanzgraph als auch im Schema optisch markiert. Nun kann der Anwender entscheiden, ob er im *Hybrid-Graph*, mit Hilfe der Änderungsoperation Insert Node, den Knoten *8* vor oder hinter dem Knoten *9* einfügen will. Er kann aber auch den hybriden Graphen – falls ihm das sinnvoll erscheint – durch Anwenden anderer Änderungsoperationen beliebig ändern. Dadurch bleibt zwar dem Anwender überlassen, welche Änderungsoperationen er anwendet. Die Korrektheit und die Möglichkeit die resultierende Instanz auf das geänderte Schema umzuhängen ist dabei aber in jedem Fall garantiert.

7.7.4 Zusammenfassung

Instanzen der Klasse *partially equivalent* können nicht in jedem Fall automatisch migriert werden. Dies ist immer dann nicht möglich, wenn die Kombination aus Schema- und Instanzänderungen ein Konflikt erzeugt, der ohne Benutzereingriff nicht gelöst werden kann. Für diesen Fall muss das Änderungsrahmenwerk Mechanismen bereitstellen, die den Benutzer bei der manuellen Migration bestmöglich unterstützen. Dazu wurden als Ausgangsbasis die möglichen Konflikte beschrieben und deren Berechnung erläutert. Anhand dieser Konflikte wurde ein Verträglichkeitstest entwickelt, mit dem die vom Zustand unverträglichen Instanzen zuverlässig erkannt und somit von vornherein von der manuellen Migration ausgeschlossen werden können. Zur korrekten Migration verträglicher Instanzen wurde das Konzept des Hybrid-Graphen vorgestellt. Dieses aus Teilen der zu migrierenden Instanz und Teilen des geänderten Schemas bestehende Graph repräsentiert die zu migrierende Instanz und kann vom Benutzer durch Anwenden der Standard-Änderungsoperationen in die gewünschte Form gebracht werden. Damit diese Änderungen zielgerichtet ausgeführt werden können, wird dem Anwender graphisch verdeutlicht, an welchen Stellen Konflikte aufgetreten sind und der Graph somit zu ändern ist. Dabei besitzt das Konzept des Hybrid-Graphen den entscheidenden Vorteil, dass bei jeder durchgeführten Änderung die Korrektheit der Instanz und die Möglichkeit, die aus den Änderungen resultierende Instanz auf das geänderte Schema umzuhängen, erhalten bleibt.

7.8 Zusammenfassung der Konzepte zur Schemaevolution

In diesem Kapitel wurden für das Änderungsrahmenwerk die Konzepte zur Durchführung einer Schemaevolution entwickelt. Hierfür wurden für jede der bereits auf konzeptioneller Ebene identifizierten Instanzklassen effiziente Algorithmen entwickelt, mit denen zuverlässig die Klassenzugehörigkeit einer Instanz bestimmt werden kann. Um dabei ohne teure Zugriffe auf Historiendaten auszukommen wurde das Deltaschicht-Konzept um eine zusätzliche Datenstruktur für verschobene Knoten erweitert. Dadurch lässt sich die Zugehörigkeit einer beliebigen Instanz ausschließlich mit Informationen aus der Deltaschicht berechnen. Aufbauend auf der Klasseneinteilung wurden die Verträglichkeitskriterien und Migrationsstrategien der einzelnen Klassen auf Gemeinsamkeiten untersucht und daraus die insgesamt bzgl. Migration umzusetzenden Funktionen bestimmt. Im Einzelnen muss das Änderungsrahmenwerk zur erfolgreichen Migration aller Arten von Instanzen die folgende Funktionalität bereitstellen:

- Prüfung der strukturellen Verträglich bei *disjoint*-Instanzen.
- *Bias*-Berechnung bei Instanzen der Klasse *subsumption equivalent* ($\Delta_S < \Delta_I$).

- Prüfung der zustandsbasierten Verträglichkeit bei Instanzen der Klassen *unbiased*, *disjoint*, *subsumption equivalent* ($\Delta_I < \Delta_S$) und *partially equivalent*.
- Korrekte Neubewertung einer Instanzmarkierung bei *unbiased*-, *disjoint*- und *subsumption equivalent*-Instanzen mit ($\Delta_I < \Delta_S$).
- Unterstützung des Anwenders bei der manuellen Migration von *partially equivalent*- Instanzen

Bei der Umsetzung der strukturellen Verträglichkeitsprüfung wurden die auf konzeptioneller Ebene auf bestimmte Änderungsoperationen fixierten Konflikttests verallgemeinert, indem nicht die Konflikte konkreter Änderungsoperationen selbst, sondern die Konflikte durch deren Auswirkungen betrachtet wurden. Zur zuverlässigen Prüfung dieser Konflikte wurden entsprechende Algorithmen entwickelt.
Zur effizienten *Bias*-Berechnung wurde zwischen dem Fall ohne Auftreten und dem Fall bei Auftreten kontextabhängiger Änderungen unterschieden. Im ersten Fall ist die *Bias*-Berechnung durch einfache Differenzmengenbildung zwischen den Mengen aus der Deltaschicht der Instanz und der Deltaschicht des geänderten Schemas berechenbar. Im zweiten Fall sind insbesondere für die Berechnung der Kontrollkantenmengen aufwendigere Algorithmen notwendig. Diese wurden detailliert beschrieben.
Mit den entwickelten zustandsbasierten Verträglichkeitstest für Instanzen der Klassen *unbiased*, *disjoint*, *subsumption equivalent* ($\Delta_I < \Delta_S$) und *partially equivalent* ist es möglich ausnahmslos für jede Instanz der genannten Klassen die Verträglichkeit korrekt zu bestimmen. Dabei wird die zu prüfende Knotenmenge nicht anhand der einzelnen im Zuge einer Schemaänderung angewendeten Änderungsoperationen bestimmt, sondern anhand der Effekte aller angewendeten Operationen. Dadurch lässt sich die Zahl der zu betrachtenden Knoten entscheidend reduzieren und somit die Effizienz der Algorithmen steigern.
Auch für die Mechanismen zur Neubewertung einer Instanzmarkierung konnte unter Ausnutzung des Kontextes und der instanzspezifischen Markierung der Instanz *I* die Menge der für eine Neubewertung relevanten Knoten minimal gehalten werden. Die Neuberechnung einer Instanzmarkierung wird dadurch entscheidend beschleunigt.
Zur korrekten Migration von *partially equivalent* Instanzen wurde das Konzept des Hybrid-Graphen entwickelt. Ein solcher Hybrid-Graph besteht aus Teilen der zu migrierenden Instanz und Teilen des geänderten Schemas. Dieser repräsentiert im Folgenden die zu migrierende Instanz. Um diese Instanz in die gewünschte Form zu bringen und somit migrieren zu können, werden die Standard-Änderungsoperationen auf dem Hybrid-Graphen angewendet. Entscheidend dabei ist, dass bei jeder durchgeführten Änderung die Korrektheit der Instanz und die Möglichkeit die aus den Änderungen resultierende Instanz auf das geänderte Schema umzuhängen aufgrund der Beschaffenheit des Hybrid-Graphen automatisch erhalten bleibt.

Insgesamt besitzen die beschriebenen Konzepte bzw. Algorithmen den Vorteil, dass sie ausschließlich auf die in der Deltaschicht enthaltenen Informationen zurückgreifen. Konkrete Änderungsoperationen und deren Semantik werden nicht verwendet. Dies macht teure Zugriffe auf Historiendaten überflüssig und gewährleistet die Erweiterbarkeit des Änderungsrahmenwerkes. So funktionieren die Algorithmen sogar mit bisher noch nicht im Rahmenwerk enthaltenen Änderungsoperationen, da diese alle auf einer fest definierten Menge an Änderungsprimitiven beruhen und sich somit die Auswirkungen aller (auch zukünftig in das Rahmenwerk eingebrachter) Änderungsoperationen in der Deltaschicht widerspiegeln.

Mit den in diesem Kapitel entwickelten Konzepten und Algorithmen ist das Änderungsrahmenwerk in der Lage, die Verträglichkeit unveränderter und geänderter Instanzen zuverlässig zu bestimmen und

diese ggf. korrekt auf ein geändertes Schema zu migrieren. Zusammen mit den Konzepten zur Prozesserstellung und Änderung erfüllt das Rahmenwerk nun alle der in Kapitel 4 aufgestellten Anforderungen.

8 Related Work

Im Folgenden wird beschrieben welche Funktionalität existierende Forschungsprototypen und die Änderungskomponenten kommerzieller Systeme bereitstellen. Dabei basieren einige der vorgestellten Prototypen auf den in Kapitel 2 beschriebenen konzeptionellen Ansätzen und besitzen somit auch deren Schwächen. An dieser Stelle werden deshalb hauptsächlich die Art der Umsetzung und evtl. vorhandene Unterschiede zwischen der Implementierung und der konzeptionellen Grundlage erläutert.

8.1 Prototypen

Ad-hoc-Instanzänderungen

Breeze [Sadi00] und *WASA*$_2$ [Wesk00] bieten einen einfachen Editor über den strukturelle Änderungen an bereits laufenden Instanzen durchgeführt werden können. Änderungsoperationen bei deren Anwendung die Korrektheit des Graphen erhalten bleibt, stehen allerdings weder bei *Breeze* noch bei *WASA*$_2$ zur Verfügung. Die in diesen Editoren angebotenen Änderungsmöglichkeiten befinden sich also auf einer semantisch sehr niedrigen Stufe, weshalb diese nur für erfahrene Benutzer geeignet sind.

Vorgeplante Instanzänderungen

Im Gegensatz zu Ad-hoc-Instanzänderungen, wie sie auch von dem in dieser Arbeit entwickelten Änderungsrahmenwerk angeboten werden, behandeln die Prototypen *AgentWork* [Müll02], *DYNAMITE* [HJKW96] und *EPOS* [LiCo93] Abweichungen vom Prozessablauf bereits zur Modellierzeit. Diese Art der vorgeplanten Änderung ist allerdings nur in den Einsatzgebieten praktikabel, in denen es nur selten zu unvorhersehbaren Abweichungen vom normalen Prozessablauf kommt. Der Vorteil an vorgeplanten Änderungen besteht darin, dass diese zur Laufzeit automatisch durchgeführt werden können. Ein fehleranfälliges Ändern laufender Prozesse ist somit nicht notwendig. Damit diese Systeme aber in der Lage sind auf Ausnahmen im Prozessablauf reagieren zu können, müssen die Fehler erkannt, die notwendigen Änderungen bestimmt und schließlich korrekt ausgeführt werden. Die Prototypen verwenden hierzu unterschiedliche Konzepte. In *AgentWork* werden beispielsweise ECA-Regeln (Event/Condition/Action) verwendet um logische Fehler im Prozessablauf zu erkennen und die notwendigen Änderungen zu bestimmen. *EPOS* verwendet ein ziel-basiertes Konzept, bei dem das Prozessziel formalisiert (z.B. in Form des gewünschten Prozessoutputs) wird. Notwendige Instanzänderungen, wie beispielsweise das Ersetzen einer fehlerverursachenden Aktivität, werden automatisch durchgeführt, sobald eine Aktivität einen Fehler verursacht, der zu einer Verletzung des Prozessziels führt.
Insgesamt sind die meisten Prototypen zur Unterstützung vorgeplanter Änderungen zu restriktiv (keine Beachtung von Schleifen) und die zu definierenden Regeln zu aufwendig. So kann eine Änderung bzw. die Planung einer Änderung ähnlich wie bei *Breeze* und *WASA*$_2$ nur von erfahrenen Anwendern durchgeführt werden. Die Korrektheit wird also auch hier nicht vom System gewährleistet.

Schemaänderungen und Migration laufender Instanzen auf ein geändertes Schema

Chautauqua [ElMa97] erlaubt dynamische Änderungen ausschließlich auf Schemaebene, da alle Instanzen auf demselben Petri-Netz (*Flow-Net*) laufen. Durchgeführt werden diese Änderungen über einen graphischen Editor und unter Verwendung einer Menge an vordefinierten Änderungsoperationen. Die strukturelle Korrektheit wird dabei von den Änderungsoperationen gewährleistet. Die Migration laufender Instanzen geschieht implizit durch die Änderung auf Schema-Ebene. Ein Überprüfen der Verträglichkeit und das Erzeugen eines korrekten Ausführungszustandes werden jedoch nicht vom System durchgeführt. Es werden lediglich diejenigen Tokens angezeigt, die

im geänderten Schema keine Stelle mehr besitzen („fall out“ Tokens). Der Benutzer muss die betroffenen Instanzen durch Setzen entsprechender Tokens in einen korrekten Ausführungszustand bringen. Wie nach einer Änderung Datenfluss und Arbeitslisten anzupassen sind, wird in [ElMa97] nicht beschrieben.
In *WASA*$_2$ können neben den bereits beschriebenen Instanzänderungen auch Schemaänderungen durchgeführt und Instanzen auf das geänderte Schema migriert werden. Dies ist allerdings nur für unveränderte Instanzen möglich. Bereits auf Instanzebene geänderte Prozesse werden hingegen von einer Migration ausgeschlossen.

8.2 Kommerzielle Systeme

Dynamische Änderungen
Die meisten auf dem Markt verfügbaren Prozess-Management-Systeme bieten zwar umfangreiche Prozessunterstützungsfunktionen, sind in den meisten Fällen aber sehr inflexibel. So bieten beispielsweise *WebSphere MQ Workflow* [IBMW04] und *Staffware* [Staf04] keine Möglichkeit instanzspezifische Änderungen durchzuführen. Die Systeme *TIBCO InConcert*, *SER Workflow* und *FileNet Ensemble* erlauben zwar dynamische Änderungen an laufenden Prozessen (Schemata), ermöglichen dies aber, indem sie eine Kopie des Schemas erzeugen und die Änderungen darauf ausführen. Ein Propagieren von Schemaänderungen auf diese Instanzen wird somit von vornherein ausgeschlossen. Hinzu kommt, dass die Systeme den Anwender bei der Durchführung dynamischer Änderungen nicht unterstützen. So bietet keines der Systeme Änderungsoperationen bei deren Anwendung die Korrektheit des (instanzspezifischen) Schemas erhalten bleibt. Der Anwender ist also selbst dafür verantwortlich, dass es in der Folge einer Änderung nicht zu Systemabstürzen oder fehlenden Eingabedaten kommt. In einem flexiblen Umfeld, in dem es häufig zu Abweichungen vom ursprünglichen Prozess kommt, ist der sinnvolle Einsatz eines solchen Prozess-Management-Systems folglich nicht möglich.
Um die Problematik bei der Durchführung einer Instanzänderung von vornherein zu umgehen, können bei Fall-basierten Systemen wie *FLOWer* (*Pallas Athena*) [AaBe01] Daten manipuliert und hinzugefügt werden bevor die Aktivitäten, die eigentlich diese Daten erzeugen, gestartet worden sind. Die Entscheidung, welche Aktivitäten als nächstes ausgeführt werden, hängt also nicht von zuvor ausgeführten Aktivitäten ab, sondern von der Verfügbarkeit der für eine Ausführung notwendigen Daten. Somit ergibt sich mit diesem daten-getriebenen Prozessmodell von vornherein eine Vielzahl an möglichen Ausführungen für einen einzigen modellierten Prozess. Zusätzlich erlaubt *FLOWer* aber auch Abhängigkeiten zwischen Aktivitäten zu modellieren. Ob dieser Mix aus prozess- und datengetriebener Prozessausführung tatsächlich ausreicht, um strukturelle Änderungen an Prozessen komplett überflüssig zu machen, bleibt abzuwarten [RReD04].

Schemaänderungen und Migration laufender Instanzen auf ein geändertes Schema
In der Regel erlauben es kommerzielle Systeme nicht, Schemaänderungen auf laufende Instanzen zu propagieren. Stattdessen werden einfache Versionierungskonzepte verwendet die garantieren, dass laufende Instanzen auf der ursprünglichen Schemaversion beendet werden können. Lediglich Staffware ermöglicht es Schemaänderungen auf laufende Instanzen zu propagieren. Dies ist allerdings nur deshalb möglich, weil alle Instanzen eines Typs dieselbe Prozessvorlage referenzieren. Eine Prüfung, ob die Instanzen mit der Schemaänderung verträglich sind, wird nicht durchgeführt. So lassen sich beispielsweise Aktivitäten löschen die sich in Ausführung befinden oder Aktivitäten, deren Daten im weiteren Verlauf benötigt werden. Ein korrektes Propagieren der Änderungen kann also auch hier nicht durchgeführt werden.

8.3 Allgemeine Aspekte und Fazit

Weder die Prototypen noch die kommerziellen Systeme erfüllen die vom Anwender geforderte Unterstützung bzgl. Flexibilität. Die existierenden Prototypen beschränken sich meist auf Modellierungs- und Laufzeitsimulationen. Anhand dieser Simulationen wird verdeutlicht, dass eine bestimmte Funktionalität im Prinzip realisiert werden kann. Es zeigt aber nicht, wie sich diese performant in der Praxis umsetzen lässt. Hinzu kommt, dass sowohl Prototypen als auch kommerzielle Systeme meist nur eine Änderungsart unterstützen, wobei sich diese Unterstützung auch nur auf die reine Ausführung struktureller Änderungen beschränkt. Die Verantwortung für eine korrekte Durchführung trägt der Anwender. Dies ist in der Praxis weder aus Sicht der Benutzerfreundlichkeit noch aus Gründen der uneingeschränkten Einsetzbarkeit akzeptabel. Macht man sich beispielsweise bewusst, wie sich in Staffware eine fehlerhaft durchgeführte Änderung auf u.U. mehrere tausend Instanzen auswirkt, so zeigt sich, dass ein flexibles PMS in jedem Fall die Korrektheit bei der Durchführung einer Änderung gewährleisten muss. Dabei ist es irrelevant, ob es sich nun um eine Schema- bzw. Instanzänderung oder um die Propagation von Schemaänderungen auf laufende Instanzen handelt. Keines der bisher existierenden Systeme ist jedoch in der Lage alle Änderungsarten zu unterstützen und dabei in jedem Fall die Korrektheit zu gewährleisten.

Insgesamt liefern die Prototypen zwar interessante Ergebnisse in speziellen Bereichen der Prozess- oder Instanzänderung, für einen Einsatz in der Praxis sind sie jedoch nur sehr bedingt geeignet. Bei den kommerziellen Systemen sind die Möglichkeiten der Prozessänderung meist so eingeschränkt und aufwendig durchzuführen, dass ein Einsatz in einem Umfeld mit häufig ändernden Prozessabläufen nicht möglich ist.

9 Weitergehende Überlegungen

An dieser Stelle werden Aspekte beschrieben, die im Anschluss an diese Arbeit weiterverfolgt werden können. Darunter finden sich sowohl Anweisungen für die weitere Vorgehensweise, als auch Ideen zur Optimierung vorgestellter Konzepte.

9.1 Gleichheit zweier Aktivitäten

Bei den Mechanismen zur Schemaevolution wurde die Gleichheit zweier Aktivitäten (Knoten) über den Namen bestimmt. Nur so war es möglich, die entscheidenden Abläufe klar hervorzuheben. Für eine konkrete Implementierung in Form einer Änderungskomponente für ein PMS ist diese Gleichheitsdefinition allerdings noch zu überarbeiten, wie folgendes Szenario aus der Praxis verdeutlicht:

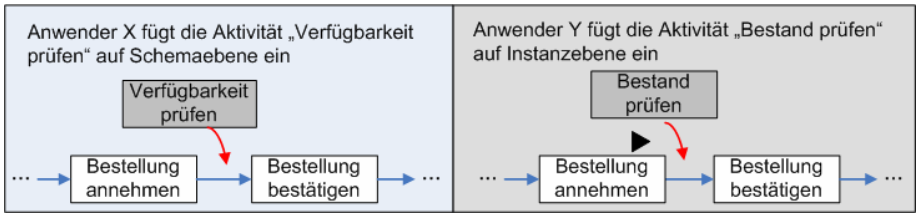


Abbildung 9.1 Gleichheit zweier Aktivitäten

Die Ausgangssituation bildet ein modellierter Geschäftsprozess (Prozessschema) der die beiden Aktivitäten „Bestellung annehmen“ und „Bestellung bestätigen“ enthält. Zusätzlich existiert eine laufende Ausprägung dieses Geschäftsprozesses (Instanz). Nun wird festgestellt, dass aus Performanzgründen der Geschäftsprozess durch einen weiteren Schritt „Verfügbarkeit prüfen“ erweitert werden muss. Anwender X ändert deshalb das Prozessschema und hinterlegt die neu eingefügte Aktivität mit einem Methodenaufruf *checkStock()*. Ein anderer Anwender Y, hat auf Kundenwunsch die (noch auf dem ursprünglichen Schema) laufende Instanz geändert, indem er die Aktivität „Bestand prüfen“ eingefügt hat. Auch hier wurde die Aktivität mit dem Methodenaufruf *checkStock()* hinterlegt.
Nun will Anwender X die Schemaänderung auf die laufenden Instanzen übertragen. Vergleicht man bei den Mechanismen zur Schemaevolution nur die Namen, so würde dies zu dem Ergebnis führen, dass die Instanz zur Klasse *partially equivalent* zuzuordnen ist. Tatsächlich ist die Instanz aber so zu behandeln als ob sie zur Klasse *equivalent* gehört, da die Auswirkungen der Änderungen exakt gleich sind. In beiden Fällen wird zur Laufzeit dasselbe ausgeführt.
Vergleicht man also lediglich die Namen, so werden in der Praxis viele Instanzen als unverträglich erkannt, obwohl sie es eigentlich nicht sind. Soll ein PMS umfassend einsetzbar sein, so kann man nicht verlangen, dass Anwender über Abteilungs- oder sogar Unternehmensgrenzen hinweg eindeutige Namen für bestimmte Aktivitäten verwenden.

Es muss also noch genau untersucht werden, auf welche Weise sich die Gleichheit zweier Aktivitäten festlegen lässt, damit die Mechanismen der Schemaevolution nicht ins „Leere laufen“. Hier sind insbesondere auch die einer Aktivität zuzuordnenden Attribute wie Bearbeiter, Zeit oder Aktivitätenvorlage zu berücksichtigen.

Zusätzlich besteht dieses Gleichheitsproblem nicht nur bei Aktivitäten, sondern auch bei Datenelementen. Hier gestalten sich der Versuch klare Gleichheitskriterien zu finden noch

schwieriger, da ein Datenelement weit weniger Attribute besitzt die für eine Gleichheitsprüfung herangezogen werden können. Dies ist in Zukunft noch genauer zu untersuchen.

9.2 Null-Knoten

Eng verbunden mit der in Abschnitt 9.1 beschriebenen Problematik ist die Behandlung von Null-Knoten. Zu diesen gehören alle Arten von *Split*- und *Join*-Knoten, sowie Schleifenanfangs- und Endknoten. Das besondere an den Knoten ist der Umstand, dass bei diesen weder Name, Aktivitätenvorlage noch Bearbeiter hinterlegt werden. Sie liefern also auch keinen direkten Beitrag zur Erfüllung der Aufgabe eines Geschäftsprozesses. Trotzdem sind sie durch ihre *Routing*-Funktion für die erfolgreiche Durchführung eines Prozesses unerlässlich.
Für die Mechanismen einer Schemaevolution sind die Null-Knoten insofern problematisch, da sie kaum bzw. gar keine Informationen enthalten, die sie von anderen Null-Knoten desselben Typs unterscheiden. Dieses Problem tritt vor allem dann auf, wenn im Zuge einer Änderung mehr wie zwei Verzweigungsblöcke des gleichen Typs eingefügt, gelöscht oder verschoben werden. Erfolgt im Zuge einer Schemaevolution ein Vergleich, so ist allein anhand eines Null-Knotens aus dem geänderten Schema nicht feststellbar, ob dieser mit einem Null-Knoten einer Instanz übereinstimmt.
Es muss also untersucht werden, wie solche Null-Knoten eindeutig bestimmt werden können. Dabei erweist sich auch hier ein einfaches Benennen der Null-Knoten analog zu Abschnitt 9.1 als ungeeignet. Es gilt demzufolge andere Lösungen zu finden. Vielversprechend scheinen in diesem Zusammenhang die Kontextknoten (*Anchor*s) eines Null-Knotens zu sein.

9.3 Erzeugung einer Änderungshistorie aus der Deltaschicht

Für die in dieser Arbeit vorgestellten Mechanismen ist das Erzeugen bzw. Vorhandensein einer Änderungshistorie nicht zwangsläufig erforderlich. Dies begründet sich hauptsächlich damit, dass eine solche Historie konkrete Änderungsoperationen enthält. Von genau diesen muss aber abstrahiert werden, um die Erweiterbarkeit des Änderungsrahmenwerkes zu gewährleisten. Trotzdem besteht die Möglichkeit, dass evtl. andere Komponenten eines PMS oder darauf aufbauende Anwendungsprogramme die Informationen einer Änderungshistorie benötigen. Dabei ist es nicht ausreichend jede angewendete Änderungsoperation einfach mitzuprotokollieren. Eine so erzeugte Historie enthält in der Regel *noisy information*, was im Normalfall nicht erwünscht ist.
Der in [Rind04] zur Bereinigung von *noisy information* beschriebene Algorithmus kann im Rahmenwerk aber nicht eingesetzt werden, da sich die für ein Bereinigen notwendigen Schritte auf konkrete Änderungsoperationen beziehen. Zusätzlich bestehen bei dem Algorithmus Abhängigkeiten zwischen einzelnen Operationen, die bei einem Einsatz des Algorithmus im Rahmenwerk dazu führen, dass bei einer Erweiterung um neue Operationen, die Wechselwirkungen zwischen allen Operationen überarbeitet werden müssen.
Da aber die in einer bereinigten Historie enthaltenen Informationen den Auswirkungen aller angewendeten Änderungsoperationen entsprechen, müssen die für eine bereinigte Änderungshistorie notwendigen Informationen auch aus der Deltaschicht erzeugbar sein. Ein geeigneter Algorithmus muss also aus den Informationen der Deltaschicht eine Änderungshistorie generieren.

In [Rind04] findet sich ein Algorithmus (Algorithmus 10) mit dem für Instanzen der Klasse *subsumption equivalent* ($\Delta_S < \Delta_I$) das aus der Migration auf ein geändertes Schema resultierende *Bias* bestimmt werden kann. Als Input verlangt dieser Algorithmus die aus der Deltaschicht ableitbaren *Difference Sets*. Da der Ansatz in [Rind04] nicht auf der Deltaschicht beruht, sondern immer Änderungsoperationen betrachtet, besteht auch das bei Algorithmus 10 erzeugte *Bias* aus

Änderungsoperationen. Betrachtet man dieses *Bias* als Änderungshistorie, so erhält man einen Algorithmus, der aus der Deltaschicht eine Änderungshistorie erzeugt.
Der Algorithmus ist allerdings noch so anzupassen, dass dieser nicht wie bei der *Bias*-Berechnung, nur die Elemente der Abweichungen zwischen Δ_I und Δ_S berücksichtigt, sondern für alle Elemente einer gegebenen Deltaschicht (bzw. den daraus ableitbaren *Difference Sets*) entsprechende Änderungshistorieneinträge erzeugt. Um den Algorithmus tatsächlich im Änderungsrahmenwerk einsetzen zu können, ist zusätzlich zu prüfen, ob der aus den in [Rind04] verwendeten Änderungsoperationen bestehende Output, an die ADEPT2-Änderungsoperationen angepasst werden kann.

9.4 Verfeinerung der Architektur

In Kapitel 5 wird die Architektur der *ChangeManager*-Komponente auf einer eher konzeptionellen Ebene beschrieben. Ebenso werden die in Kapitel 6 im Zuge der Beschreibung implementierungsnaher Konzepte entwickelten Schnittstellen und Methoden in abstrakter Form vorgestellt. In den Abschnitten zur Schemaevolution wird größtenteils sogar komplett auf die Definition von Methoden verzichtet. Vor der Implementierung des Änderungsrahmenwerks müssen also in einem nächsten Schritt, für die in dieser Arbeit vorgestellten implementierungsnahen Konzepte, konkrete Klassendiagramme und Schnittstellen beschrieben werden. Dies sollte problemlos durchführbar sein da, anders als bei einer direkten Umsetzung der konzeptionellen Arbeiten, bei den hier beschriebenen Mechanismen sowohl Internas eines PMS, als auch softwarespezifische Anforderungen wie Benutzerfreundlichkeit und Erweiterbarkeit bereits berücksichtigt werden.

9.5 Optimierung der Schemaänderung

Die Materialisierung der Schemaänderungen erfolgt bisher direkt bei der Ausführung einer Änderungsoperation bzw. der darin enthaltenen Änderungsprimitiven. Es wird aber gleichzeitig auch eine Deltaschicht erzeugt. Durch diese Deltaschicht genügt es eigentlich, die Änderungen erst dann zu materialisieren, wenn der Benutzer den *Commit*-Befehl des *ChangeOperationsManagers* aufruft. Die Materialisierung erfolgt daraufhin anhand der Informationen die in der Deltaschicht gespeichert wurden. Dies ist bei Schemaänderungen immer möglich, da – anders als bei Instanzänderungen – immer von einer anfangs leeren Deltaschicht ausgegangen wird.
Die Materialisierung läuft folgendermaßen ab:

1. Füge jeden Knoten *X* aus *newNodes* und jedes Datenelement *D* aus *newDataElements* in *Nodes* bzw. *DataElements* des zu ändernden Schemas *S'* ein.
2. Lösche jeden Knoten *Y* von *delNodes* und jedes Datenelement *D* von *delDataElements* aus der Menge *Nodes* bzw. *DataElements* von *S'*
3. Füge jede Kante E^{add} von *newEdges* in *Edges* von *S'* ein
4. Lösche jede Kante E^{del} von *deletedEdges* aus der Menge *Edges* von *S'*.
5. Analog für alle neu eingefügten und gelöschten Attribute

Daraus entsteht der Vorteil, dass die Schema- und Instanzänderungen bis zum *Commit*-Befehl exakt gleich behandelt werden können. Weiterhin wird der Ressourcenbedarf der *UNDO*-Funktion bei einer Schemaänderung entscheidend reduziert, da das Speichern des Schemas entfällt.

Ob diese Optimierung jedoch durchführbar ist muss noch geprüft werden. Probleme können dadurch entstehen, dass die von den Änderungsoperationen aufgerufenen Methoden des Datenmodells bei einem Schema nicht auf der Deltaschicht arbeiten, sondern direkt auf den Schemadaten. Werden im

Zuge einer Schemaänderung mehrere Änderungen durchgeführt und diese nicht sofort materialisiert, so kann dies zu strukturell inkorrekten Schemata führen. Es gilt also zu untersuchen, ob sich das Datenmodell bzw. die interne Repräsentation von Schemata so anpassen lässt, dass diese zumindest bis zum *Commit*-Befehl wie Instanzen behandelt werden können.

9.6 Optimierung der UNDO-Funktionalität

Zur Durchführung eines *UNDO* wird in Abschnitt 6.3 bei einer Schemaänderung der komplette Schemagraph inkl. Deltaschicht und bei einer Instanzänderung die Deltaschicht gespeichert. Dies ist bezüglich Speicherausnutzung noch nicht optimal. Wesentlich besser ist es nur die Änderungen zu speichern die aus der Anwendung einer Änderungsoperation resultieren. Bei Durchführung eines *UNDO* werden mit Hilfe der gespeicherten Änderungen die Auswirkungen der Änderungsoperation wieder rückgängig gemacht. Welche Form dabei die gespeicherten Änderungen besitzen müssen und wie diese Form erzeugt werden kann muss noch geprüft werden.

9.7 Erzeugung des Hybrid-Graphen

Im Zusammenhang mit der manuellen Migration von *partially equivalent* Instanzen wurde ein Hybrid-Graph eingeführt. Hier ist noch genauer zu untersuchen wie sich der hybride Graph aus Teilen der geänderten Instanz und Teilen des geänderten Schemas zusammenbauen lässt. Insbesondere für die Schnittstelle zwischen dem Instanz- und dem Schemateil muss ein Konzept entwickelt werden, das für alle angewendeten Änderungsoperationen eine klare Linie zwischen den Teilen ziehen kann.

10 Zusammenfassung

Der durch die Globalisierung gestiegene Wettbewerbsdruck zwingt die Unternehmen zu einer effizienten Nutzung der vorhandenen Kapazitäten, um sich am Markt behaupten zu können. Hierbei spielen insbesondere die Geschäftsprozesse eines Unternehmens eine entscheidende Rolle. Um diese unterstützen zu können, bietet es sich an, Prozess-Management-Systeme (PMS) einzusetzen. Da sich Geschäftsprozesse aber häufig ändern, muss ein solches PMS in der Lage sein, die von ihm verwalteten Geschäftsprozesse (Prozessschemata) und die darauf basierenden Instanzen bei Bedarf rasch anpassen zu können. Dies betrifft neben den einfachen Änderungen von Prozessschemata und -instanzen auch die Fähigkeit, Schemaänderungen auf laufende Instanzen zu übertragen (Schemaevolution). Weiterhin muss ein flexibles PMS auch das Zusammenspiel zwischen Schema- und Instanzänderungen unterstützen.

Die heutigen auf dem Markt verfügbaren PMS bieten aber entweder gar keine Möglichkeit bestehende Prozesse und Instanzen anzupassen oder die angebotenen Mechanismen sind derart unzureichend implementiert, dass es in der Folge einer Änderung zu Inkonsistenzen oder Systemabstürzen kommen kann. Hinzu kommt, dass bei den Systemen die Änderungen ermöglichen, diese nur von erfahrenen Prozessmodellierern durchgeführt werden können; semantisch höhere Operationen, die automatisch die Korrektheit des geänderten Schemas garantieren, sucht man vergeblich.
Auch Ansätze aus der Forschung behandeln die Problematik in vielerlei Hinsicht nur unzureichend. So existiert kein Ansatz, der bei vollständigem Metamodell (d.h. keine Einschränkungen bzgl. Modellierungskonstrukten, wie z.B. Schleifen) alle Änderungsarten unterstützt und dabei zusätzlich die Korrektheit gewährleistet. Weiterhin ist anhand der konzeptionellen Beschreibung in diesen Ansätzen nicht ersichtlich, ob sich die entwickelten Mechanismen, unter Berücksichtigung der an eine Softwarekomponente gestellten Anforderungen, wie Benutzerfreundlichkeit, Effizienz und Erweiterbarkeit, überhaupt implementieren und in die Gesamtstruktur eines PMS integrieren lassen.

Ausgehend von dieser Problematik war es das Hauptziel dieser Arbeit ein implementierungsnahes Konzept für ein Änderungsrahmenwerk zu entwickeln, das alle geforderten Änderungsarten unterstützt, die Korrektheit automatisch gewährleistet, den Benutzer bei der Durchführung einer Änderung unterstützt und direkt in Form einer Änderungskomponente in ein Prozess-Management-System (ADEPT2) integriert werden kann.

Die Ausgangsbasis für die Erstellung eines umfassenden Änderungsrahmenwerkes bilden konzeptionelle Ansätze zum Thema Flexibilität in Prozess-Management-Systemen. Dazu wurde untersucht, in wie weit in existierenden Ansätzen enthaltene Konzepte als formale Grundlage für das Änderungsrahmenwerk geeignet sind. Dabei hat sich gezeigt, dass ein Graph-/Aktivitäten-basiertes Prozess-Metamodell als Grundlage zur Unterstützung aller Änderungsarten besser geeignet ist als ein Petri-Netz-basiertes Metamodell. Im Speziellen hat sich besonders der *WSM-Nets*-Ansatz als gute Grundlage zur Prozessmodellierung herausgestellt. Zusammen mit den in [Reic00] definierten Korrektheitskriterien und den dort ebenfalls beschriebenen Konzepten zur Instanzänderung, bietet dieser Ansatz sowohl ein vollständiges Metamodell als auch die Möglichkeit, Änderungen auf Instanzebene korrekt durchzuführen. Zusätzlich spricht für diesen Ansatz, dass in [Rind04] beschriebene Konzepte zur Schemaevolution auf *WSM-Nets* basieren.

Um bzgl. Funktionalität eine vollständige konzeptionelle Grundlage für das Änderungsrahmenwerk zu erhalten, wurden die Konzepte aus [Reic00] und [Rind04] zu einem Ansatz vereint. Die Schwierigkeit bestand darin, dass dabei bereits der zukünftige Einsatzzweck des Änderungsrahmenwerkes berücksichtigt werden musste. So wurden die Konzepte an speziell für ADEPT2 entwickelte Änderungsoperationen angepasst.

Neben den Konzepten zur Durchführung einer Änderung ist für eine praktische Umsetzung auch die Art der internen Repräsentation von Schema- und Instanzänderungen von entscheidender Bedeutung. Hierzu wurden existierende Konzepte untersucht und das Deltaschicht-Konzept vorgestellt. Mit diesem können Instanzänderungen gekapselt werden, ohne dabei die Bindung an das ursprüngliche Schema *S* zu verlieren. Um nicht nur Instanzänderungen optimal unterstützen zu können, sondern auch die Durchführung einer Schemaevolution zu beschleunigen, wurde das Konzept erweitert. So wird nun neben der zwingend notwendigen Materialisierung von Schemaänderungen aus Effizienzgründen ebenfalls eine Deltaschicht erzeugt, die die Abweichungen vom ursprünglichen Schema enthält. Dadurch können die bei einer Schemaevolution notwendigen Vergleiche zwischen Schema- und Instanzänderungen direkt über die Deltaschichten durchgeführt werden.

Aus der konzeptionellen Grundlage ergeben sich konkrete Anforderungen an das Änderungsrahmenwerk. Diese wurden in Form von funktionalen Anforderungen definiert. Da es sich bei dem Änderungsrahmenwerk um ein implementierungsnahes Konzept handelt, das praktisch umgesetzt wird, müssen auch Anforderungen von Anwenderseite und softwaretypische Eigenschaften berücksichtigt werden. Diese wurden in Form von nicht-funktionalen Anforderungen konkretisiert.

Aufbauend auf den Anforderungen wurde eine konkrete Architektur für das Änderungsrahmenwerk entwickelt. Dabei wurden auch der Gesamtkontext und die Interaktionen mit den umliegenden Komponenten eines PMS berücksichtigt. Dies ist von entscheidender Bedeutung, da sich nur aus dem Gesamtkontext ergibt, auf welche Daten und Funktionen das Änderungsrahmenwerk zurückgreifen kann, um die eigene Funktionalität bereitstellen zu können. In diesem Zusammenhang wurde das ADEPT2-Datenmodell als Schnittstelle zu den in der Prozess-Engine verwalteten Schema- und Instanzdaten vorgestellt. Damit dieses allerdings in der Lage ist, alle vom Änderungsrahmenwerk bereitzustellenden Änderungsarten optimal zu unterstützen, wurde das Datenmodell um eine Schnittstelle für den direkten Zugriff auf die Daten der Deltaschicht erweitert.

Basierend auf den konzeptionellen Grundlagen, den funktionalen Anforderungen und der Architektur wurden umfassende Konzepte und Algorithmen zur Prozesserstellung/-änderung und zur Durchführung einer Schemaevolution entwickelt. Weiterhin wurden unter Verwendung eines objektorientierten Paradigmas konkrete Mechanismen und Methoden zur Erfüllung der nicht-funktionalen Anforderungen beschrieben.

Im Einzelnen wurden zur Prozesserstellung und -änderung die auf konzeptioneller Ebene vorhandenen ADEPT2-Änderungsoperationen um konkrete strukturelle und zustandsbasierte Vor-/Nachbedingungen und Anweisungen zur Zustandsneubewertung erweitert. Diese garantieren, dass es in der Folge der Anwendung einer solchen Operation nicht zu einem strukturell inkorrekten Schema bzw. einer bzgl. Struktur und Ausführungszustand inkorrekten Instanz kommt. Zusätzlich besitzen diese Operationen die Eigenschaft, dass sie alle auf einer minimalen fest definierten Menge an Änderungsprimitiven beruhen die wiederum direkt die interne Repräsentation (Knoten-, Kantenmengen, etc.) eines Prozesses manipulieren (vgl. Abbildung 10.1).

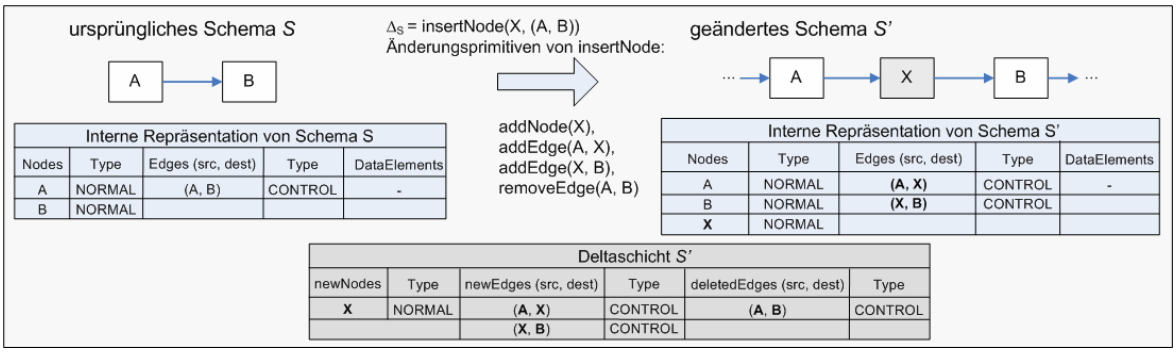


Abbildung 10.1 Änderungsprimitiven und deren Auswirkungen auf Schemaebene

Für die nicht-funktionale Anforderung der Erweiterbarkeit wurde ein Plug-In-Konzept entwickelt, mit dem das Rahmenwerk um neue Änderungsoperationen erweitert werden kann. Dazu wurde eine konkrete Schnittstelle für Änderungsoperationen definiert und ein Mechanismus entwickelt, mit dem neue Operationen dynamisch in das Änderungsrahmenwerk geladen werden können. Um auch die Anforderungen bezüglich Benutzerfreundlichkeit zu erfüllen wurde gezeigt, wie eine *UNDO*-Funktion und ein Konzept zur kontextabhängigen Aufrufbarkeit einer Änderungsoperation implementiert werden kann. Dabei lassen sich mit Hilfe der *UNDO*-Funktion die Auswirkungen einer angewendeten Änderungsoperation vollständig rückgängig machen. Mit dem Konzept zur Prüfung der kontextabhängigen Aufrufbarkeit ist es möglich, anhand einer beliebigen Menge an Graphelementen, dem Anwender diejenigen Änderungsoperationen zu berechnen und anzubieten, die im aktuellen Kontext und mit der Anzahl an gewählten Elementen aufrufbar sind.
Insgesamt wurden die Konzepte und Mechanismen so entwickelt, dass sie auch im Zusammenspiel optimal funktionieren und sowohl für eine Verwendung über ein GUI als auch für reine API-Aufrufe (ohne direkte Interaktion mit dem Endanwender) gleichgut geeignet sind.

Bei der Umsetzung der Konzepte zur Durchführung einer Schemaevolution wurde gleich zu Beginn deutlich, dass die in [Rind04] auf konzeptioneller Ebene beschriebenen Algorithmen für eine direkte Umsetzung im Änderungsrahmenwerk nicht geeignet sind. Dies hängt damit zusammen, dass die für eine erfolgreiche Schemaevolution notwendigen Verträglichkeitstests und Migrationstrategien die Semantik einzelner fest definierter Änderungsoperation verwenden. Aufgrund der Erweiterbarkeit des Änderungsrahmenwerkes sind Anzahl und Typ der Änderungsoperationen aber nicht im Voraus bekannt. In der Folge wurden Konzepte und Algorithmen entwickelt, die vollständig unabhängig von den tatsächlich angewendeten Änderungsoperationen die exakt gleiche Funktionalität erbringen, wie die auf konzeptioneller Ebene beschriebenen Vorgehensweisen.

Dazu wurde das Deltaschicht-Konzept so erweitert, dass auch verschobene Knoten von der Deltaschicht verwaltet werden können. Dadurch lassen sich alle Informationen für die bei einer Schemaevolution anzuwendenden Algorithmen aus der Deltaschicht ableiten, wodurch eine vollständige Unabhängigkeit von konkreten Änderungsoperationen erreicht wird. So funktionieren die in dieser Arbeit entwickelten Algorithmen sogar mit bisher noch nicht im Rahmenwerk enthaltenen Änderungsoperationen, da diese alle auf einer fest definierten Menge an Änderungsprimitiven beruhen und sich somit die Auswirkungen aller (auch zukünftig in das Rahmenwerk eingebrachter) Änderungsoperationen in der von den Algorithmen verwendeten Deltaschicht widerspiegeln.

Zusätzlich wurde durch konsequentes Ausnutzen der in den Deltaschichten gespeicherten Informationen, die Effizienz gegenüber den auf konzeptioneller Ebene beschriebenen Algorithmen gesteigert. So ist es gelungen, die für die Anwendung gezielter Verträglichkeitstests und Migrationstrategien notwendige Berechnung des Überlappungsgrades zwischen Schema- und Instanzänderungen dahingehend zu optimieren, dass keine teuren Zugriffe auf Historiendaten notwendig sind. Weiterhin wurde die Prüfung, ob Schemaänderungen mit einer Instanzmarkierung verträglich sind (zustandsbasierte Verträglichkeit) deutlich verbessert, indem die zu prüfende Knotenmenge nicht anhand der einzelnen, im Zuge einer Schemaänderung angewendeten Änderungsoperationen bestimmt wird, sondern anhand der Effekte aller angewendeten Operationen. Dadurch wird die Zahl der zu betrachtenden Knoten und somit der benötigte Zeitaufwand eines zustandsbasierten Verträglichkeitstests entscheidend reduziert. Auch für die Mechanismen zur Neubewertung einer Instanzmarkierung wurde durch geschicktes Ausnutzen der Deltaschichtinformationen, die Menge der für eine Neubewertung relevanten Knoten minimal gehalten.

Um auch für die auf konzeptioneller Ebene als nicht automatisch migrierbar erkannten Instanzen (*partially equivalent*-Instanzen) eine adäquate Unterstützung zu bieten, wurde das Konzept des Hybrid-Graphen entwickelt. Ein solcher Hybrid-Graph besteht aus Teilen der zu migrierenden Instanz und Teilen des geänderten Schemas. Dieser repräsentiert im Folgenden die zu migrierende Instanz. Um diese Instanz in die gewünschte Form zu bringen und somit migrieren zu können, werden die Standard-Änderungsoperationen auf dem Hybrid-Graphen angewendet. Entscheidend dabei ist, dass bei jeder durchgeführten Änderung die Korrektheit der Instanz und die Möglichkeit die aus den Änderungen resultierende Instanz auf das geänderte Schema umzuhängen aufgrund der Beschaffenheit des Hybrid-Graphen automatisch erhalten bleibt.

Insgesamt erhält man bei einer Implementierung der vorgestellten Konzepte ein mächtiges Änderungsrahmenwerk, das eingesetzt in einem PMS, bezüglich angebotener Funktionalität, Korrektheit der durchführbaren Änderungen, Erweiterbarkeit und Benutzerfreundlichkeit alle geforderten Kriterien an die Flexibilität eines Prozess-Management-Systems erfüllt.

Literaturverzeichnis

[AaBa02] W.M.P.v.d. Aalst, T. Basten: *Inheritance of workflows: an approach to tackling problems related to change*. Theoret Comp. Sci. 270 (1-2) (2002) 125-203.

[AgDe00a] A. Agostini, G. DeMichelis: *Improving flexibility of workflow management systems*. BPM ’00, LNCS, vol. 1806, 2000, S. 218-234.

[AgDe00b] A. Agostini, G. DeMichelis: A light workflow management system using simple process models, Int. J. Collab, Comp. [16] 335-363, 2000.

[ADEPT] *ADEPT – Next Generation Workflow Technology*. http://www.informatik.uni-ulm.de/dbis/01/forschung/projects/adept/adept.htm (Stand: 06.03.2006).

[AaBe01] W.M.P.v.d. Aalst, P. Berens: *Beyond workflow management: product-driven case handling*. Proceedings on the Conference on Supp. Group Work, New York, 2001, S. 42-51.

[AWWi03] W.M.P.v.d. Aalst, M. Weske, G: Wirtz: *Advanced topics in workflow management: Issues, requirements and solutions*. Int. J. Integrat. Design Process Sci. 7 (3) (2003).

[BGST04] M. Berroth, K. Göser, A. Stieger , J. Tcheho, T. Weitmann: *Implementierung eines Prozesseditors für ADEPT2*. Praktikumsdokumentation, Universität Ulm, Abteilung DBIS, SS04.

[CCPP98] F. Casati, S. Ceri, B. Pernici, G. Pozzi: *Workflow evolution*. Data and Knowledge Engineering 24 (3) (1998) 211-238.

[DaRe04] P. Dadam, M. Reichert: *ADEPT-Prozess-Management-Technologie der nächsten Generation*. In: D. Spath, K. Haasis (Eds.): Aktuelle Trends in der Softwareforschung – Tagungsband zum doIT Softwareforschungstag 2003, IRB-Verlag Stuttgart 2004, S. 27-43.

[EKRo95] C. A. Ellis, K. Keddara, G. Rozenberg: *Dynamic change within workflow systems*. Proceedings of International ACM Conference COOCS ’95, Milpitas, CA, August1995, S.10-21.

[ElKe00] C. Ellis, K. Keddara: *A workflow change is a workflow*. BPM ’00, LNCS, vol. 1806, 2000, S. 516-534.

[ElMa97] C. A. Ellis, C. Maltzahn: *The Chautauqua workflow system*: Proceedings of the International Conference on System Science, Maui, HI, 1997.

[HaCh94] M. Hammer, J. Champy: *Business Reengineering: Die Radikalkur für das Unternehmen*. Campus Verlag Frankfurt, New York 1994.

[HJKW96] P. Heimann, G. Joeris, C. Krapp, B. Westfechtel: *DYNAMITE: dynamic task nets for software process management*. Proceedings of the 18th International Conference Software Engineering (ICSE), Berlin, März 1996, S. 331–341.

[IBMW04] IBM: *IBM WebSphere MQ Workflow V3.5 Release - Advanced production workflow with MQ and portal integration*. 2004 http://www-306.ibm.com/common/ssi/rep_ca/4/897/ENUS204-044/ENUS204-044.PDF.

[JBSc99] S. Jablonski, M. Böhm, W. Schulze: *Workflow Management Entwicklung von Anwendungen und Systemen*. Dpunkt, 1999.

[JMSW04] M. Jurisch, T. Mihalca, P. Sauter, M. Waimer: *Verwaltung von Prozessinstanzen in Workflow-Management-Systemen*. Praktikumsdokumentation, Universität Ulm, Abteilung DBIS, WS03/04.

[KrAc04] U. Kreher, H. Acker: *Spezifikation einer Änderungsschnittstelle für blockorientierte Prozessgraphen*. Internes Arbeitspapier, Universität Ulm, Fakultät für Informatik, Abteilung DBIS, 2004.

[KrGe99] M. Kradolfer, A. Geppert: *Dynamic workflow schema evolution based on workflow type versioning and workflow migration*. CoopIS '99, Edinburgh, 1999, S. 104-114.

[Laue04] M. Lauer: *Effiziente Realisierung von Prozess-Schemaevolution in Hochleistungs-Prozess-Management-Systemen*. Diplomarbeit, Universität Ulm, Fakultät für Informatik, Abteilung DBIS, 2004.

[LeRo99] F. Leymann, D. Roller: *Production Workflow: Concepts and Techniques*. Prantice Hall PTR, September 1999.

[LiCo93] C. Liu, R. Conradi: *Automatic replanning of task networks for process model evolution*. Proceedings of European Software Engineers Conference, Garmisch-Partenkirchen, Germany, 1993, S. 434–450.

[LRRe04] M. Lauer, S. Rinderle, M. Reichert: *Repräsentation von Schema und Instanzobjekten in adaptiven Prozess-Management-Systemen*. Proc Informatik '04, Bd. 2, Ulm (2004), S. 555-560.

[Müll02] R. Müller: *Event-oriented dynamic adaptation of workflows*. Dissertation, Universität Leipzig, 2002.

[Ober96] A. Oberwies: *Modellierung und Ausführung von Workflows mit Petri-Netzen*. Teubner, 1996.

[Öste95] H. Österle: *Business Engineering. Prozeß- und Systementwicklung I. Entwurfstechniken*, Berlin et al.: Springer 1995 S.13ff.

[Reic00] M. Reichert: *Dynamische Ablaufänderungen in Workflow-Management-Systemen*. Dissertation, Universität Ulm, Fakultät für Informatik, Abteilung DBIS, 2000.

[Rind04] S. Rinderle: *Schema Evolution in Process Management Systems*. Dissertation, Universität Ulm, Fakultät für Informatik, Abteilung DBIS, 2004.

[RReD02] S. Rinderle, M. Reichert, P. Dadam: *Effiziente Verträglichkeitsprüfung und automatische Migration von Workflow-Instanzen bei der Evolution von Workflow-Schemata.* Informatik – Forschung und Entwicklung, Band 17: S. 177-197, 2002.

[RReD04] S. Rinderle, M. Reichert, P. Dadam: *Correctness criteria for dynamic changes in workflow systems – a survey*. Data and knowledge engineering, Vol. 50, No. 1, 2004 Special Issue on Advances in Business Process Management, S. 9-34

[Sadi00] S. Sadiq: *Handling dynamic schema changes in workflow processes*. Proceedings of the 11th Australian Database Conference 2000.

[SMOr00] S. Sadiq, O. Marjanovic, M. Orlowska: Managing change and time in dynamic workflow processes. IJCIS 9 (1-2) (2000) 93-116.

[Staf04] Staffware. Staffware Process Suite Brochure, 2004. http://www.staffware.com/downloads/ (Stand: 06.03.2006).

[Wesk00] M. Weske: *Workflow management systems: Formal foundation, Conceptual design, implementation aspects.* Dissertation, Universität Münster, 2000.

[Wesk01] M. Weske: *Formal foundation and conceptual design of dynamic adaptations in a workflow management system*: HICCS-34, 2001.

[WiGl06] http://de.wikipedia.org/wiki/Globalisierung (Stand 06.03.2006).

Abkürzungsverzeichnis

API:	application programming interface
EAI:	enterprise application integration
ERP:	enterprise resource planning
GUI:	graphical user interface
i.W.:	im Wesentlichen
M^I:	instanzspezifische Markierung
PMS:	Prozess-Management-System
u.a.:	unter anderem
vs.:	versus

Glossar

EAI:	Integrationsplattform zur prozessorientierten Integration von Anwendungssystemen in heterogenen IT-Anwendungsarchitekturen.
Änderungsprimitive:	kleinste Einheit einer Änderungsoperation. Manipuliert direkt die Datenstrukturen durch die ein Schema oder eine Instanz repräsentiert wird. Wird vom Datenmodell bereitgestellt.
Änderungshistorie:	Enthält die auf ein Schema oder eine Instanz angewendeten Änderungsoperationen in zeitlicher Reihenfolge.
Ausführungshistorie:	Speichert bei der Ausführung einer Instanz die Start- und End-Ereignisse von Aktivitäten und die von diesen geschriebenen und gelesenen Daten.
Aktivität:	Buildtime: Knoten mit zugehöriger Aktivitätenvorlage; Runtime: kleinste Arbeitseinheit im Prozess, die einem Bearbeiter vom PMS zugewiesen werden kann.
Aktivitätenvorlage:	Anwendungscode, der bei einem Prozessknoten hinterlegt und bei dessen Ausführung aufgerufen wird.
API:	Schnittstelle für Anwendungsprogramme
ERP-System:	Integriert alle Funktionen, Prozesse und Daten eines Unternehmens in einem System.
Geschäftsprozess:	Funktionsübergreifende Verkettung wertschöpfender Aktivitäten, mit dem Ziel dem Unternehmenszweck zu dienen.
GUI:	graphische Benutzerschnittstelle
Instanz:	Eine konkrete Ausprägung eines Schemas. Auf logischer Ebene: das Schema mit Lautzeitinformationen
Instanzspezifisches Schema:	Das Schema das von einer Instanz referenziert wird inklusive des aktuellen Ausführungszustands einer Instanz. Bei geänderten Instanzen, die Kombination aus Deltaschicht und ursprünglichen Schema.
Migration:	Beinhaltet alle notwendigen Schritte um eine Instanz an ein geändertes Schema anzupassen.
Noisy Information:	Als *noisy information* bezeichnet man in Änderungshistorien Informationen über Änderungsoperationen, die keine oder nur versteckte Effekte auf das zugrunde liegende Schema besitzen.
PMS:	Dient der aktiven Steuerung arbeitsteiliger Prozesse; mit Hilfe von Prozess-Management-Systemen lassen sich Prozess- und Anwendungslogik voneinander trennen
Propagation:	Bezeichnet den Vorgang des Anwendens von Schemaänderungen auf eine Instanz.
Prozess:	Ein im System gespeichertes Abbild eines realen Geschäftsprozesses.
Prozess-Metamodell:	Formales Rahmenwerk für die Abbildung von Geschäftsprozessen.
Prozessschema:	siehe Prozess
Prozess-Typ:	Prozess eines bestimmten Typs (z.B. Leasinggeschäft). Meist analog mit Prozess verwendet
Schema:	siehe Prozess

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang

Anhang A (Änderungsoperationen)

A.1 (Funktionale Beschreibung)

Funktionale Beschreibung der in [KrAc04] vorgestellten Änderungsoperationen:

Funktionale Beschreibung	
Änderungsprimitiven:	Beschreibung: (Die Änderungsprimitiven arbeiten direkt auf den Datenstrukturen durch die ein Schema oder eine Instanz intern repräsentiert wird)
addNode(Node)	Fügt den Knoten Node in die entsprechende Datenstruktur ein
removeNode(Node)	Löscht den Knoten Node in der entsprechenden Datenstruktur
addEdge(Node src, Node dest, Type)	Fügt eine Kante, repräsentiert durch den Startknoten src und den Zielknoten dest, in die entsprechende Datenstruktur ein
removeEdge (Node src, Node dest, Type)	Löscht die Kante (src, dest) in der entsprechenden Datenstruktur
addDataEdge(Node, DataElement, Type) bzw. addDataAccess (im ADEPT2-Datenmodell)	Fügt eine Datenkante, repräsentiert durch den Knoten node und das Datenelement DataElement, vom Typ Type (Lese- oder Schreibkante) in die entsprechende Datenstruktur ein
removeDataEdge(Node, DataElement, Type) bzw. removeDataAccess (im ADEPT2-Datenmodell)	Löscht eine Datenkante (node, DataElement) vom Typ Type aus der entsprechenden Datenstruktur.
addDataElement(Type)	Fügt das Datenelement DataElement in die entsprechende Datenstruktur ein
removeDataElement(DataElement)	Löscht das Datenelement DataElement aus der entsprechenden Datenstruktur
setNodeAttribute(Node, Attribute, Value)	Fügt ein oder ändert das Knotenattribut Attribute des Knotens Node auf den Wert Value
setEdgeAttribute (Node src, Node dest, Type, Attribute, Value)	Fügt ein oder ändert das Kantenattribut Attribute der Kante (src, dest) auf den Wert Value
Einfache und komplexe Änderungsoperationen	Beschreibung:
Kontrollfluss:	
Strukturelle Änderungen:	
Einfache Änderungsoperationen:	
Knoten:	
insertNode(newNode, pred, succ)	Fügt einen Knoten seriell zwischen zwei direkt aufeinander folgenden Knoten ein. pred und succ können auch Verzweigungs- und Vereinigungsknoten einer parallelen oder alternativen Verzweigung sein, dann muss allerdings zwingend ein leerer Teilzweig vorhanden sein (andernfalls sind sie auch nicht direkt aufeinander folgend)
deleteNode(node)	Löscht einen Knoten und schließt die Kanten kurz. Alle in diesen Knoten eingehende und von diesem Knoten ausgehende (Sonder-)Kanten werden gelöscht. Wird der letzte verbleibende Knoten in einem Teilzweig gelöscht, bleibt anschließend ein leerer Teilzweig zurück. Randknoten von Verzweigungs- und Schleifenblöcke können mit dieser Operation nicht direkt gelöscht werden. Hierzu dient die komplexe Änderungsoperation deleteBorderNodes.
moveNodes(first, last, pred, succ)	Verschiebt die durch first und last spezifizierte Knotenmenge (Menge aller Knoten zwischen first und last (inkl.)) zwischen pred und succ; ggf. bleibt ein leerer Teilzweig zurück
assignActivity(node, activity)	Die Operation assignActivity ist lediglich eine Änderung eines Knotenattributs und wird deshalb durch die Operation changeNodeAttribute realisiert.
Kanten:	
insertEmptyBranch(split, join, [sc])	Fügt einen neuen leeren Teilzweig ein, der bei alternativen Verzweigungen mit einem gültigen Auswahlcode versehen wird.

Funktionale Beschreibung	
deleteEmptyBranch(split, join)	Löscht einen leeren Teilzweig aus einer Verzweigung, wobei mind. ein Teilzweig in der Verzweigung übrig bleiben muss.
createSyncEdge(src, dest)	Siehe addEdge(Node src, Node dest, “Sync”)
createPriorityEdge(src, dest)	Siehe addEdge(Node src, Node dest, “Priority”)
createFailureEdge	Siehe addEdge(Node src, Node dest, “Failure”)
deleteSyncEdge(src, dest)	Siehe removeEdge (Node src, Node dest, “Sync”)
deletePriorityEdge	Siehe removeEdge (Node src, Node dest, “Priority”)
deleteFailureEdge	Siehe removeEdge (Node src, Node dest, “Failure”)
nicht-strukturelle Änderungen (Attributänderungen)	
changeNodeAttribute(node, attribute, newValue)	Ändert ein Knotenattribut (siehe auch setNodeAttribute).
changeActivityAttribute(activity, attribute, newValue)	Ändert ein Aktivitätenattribut.
changeEdgeAttribute(src, dest, type, attribute, newValue)	Ändert ein Kantenattribut (siehe auch setEdgeAttribute)
Komplexe Änderungsoperationen:	
insertEmptyBlock(type, pred, succ) (Block mit einem Teilzweig, bei OR ist dies der Default-Zweig)	Fügt einen leeren Verzweigungsblock vom spezifizierten Typ (mit einem leerem Teilzweig) seriell zwischen pred und succ ein
createSurroundingBlock(type, first, last)	Erzeugt um eine Knotenmenge begrenzt durch first und last einen Verzweigungsblock vom spezifizierten Typ
convertBlock(split, join, type, [{sc}])	Ändert die Semantik eines Verzweigungsblocks (AND, OR, AND_OR). Wird dabei zu einer OR-Verzweigung konvertiert, so sind so viele Auswahlcodes anzugeben, wie Kanten von split ausgehen. Die Reihenfolge der Auswahlcodes entspricht der Reihenfolge der Kanten (basierend auf Kanten-IDs)
deleteBorderNodes(split, join)	Löscht zwei zusammengehörige Verzweigungs- und Vereinigungsknoten oder Schleifenanfangs und Endknoten. Eine Verzweigung kann nur dann gelöscht werden, wenn sie nur einen Teilzweig besitzt. Falls das nicht der Fall ist, müssen Teilzweige vor dieser Operation verschoben oder gelöscht werden.
deleteCompleteBranch(first, last)	Löscht einen kompletten Teilzweig aus einer Verzweigung. Im Gegensatz zu deleteNodes verbleibt kein leerer Teilzweig. Wird der letzte Teilzweig gelöscht, werden auch automatisch die entsprechenden Verzweigungs- und Vereinigungsknoten gelöscht.
moveCompleteBranch(first, last, pred, succ)	Verschiebt den durch first und last spezifizierten Teilzweig zwischen pred und succ. Sind pred und succ Verzweigungs- und Vereinigungsknoten, so wird vor dem Einfügen ein neuer (Ziel-)Teilzweig erstellt. Der alte Teilzweig wird gelöscht und damit im Fall des letzten Teilzweigs auch die Verzweigung selbst.
deleteNodes(first, last)	Löscht alle Knoten zwischen first und last, wobei auch Sync- und sonstige Kanten berücksichtigt werden. Der Vorgänger von first und der Nachfolger von last werden direkt verbunden.
insertNodeBetweenNodeSets (node, {beforeNodes}, {afterNodes})	Fügt einen Knoten node parallel zu beforeNodes und afterNodes ein. Dazu werden zuerst die Knoten first und last berechnet, die auf der gleichen Ebene liegen und zwischen denen sich alle Knoten aus beforeNodes und afterNodes befinden. Dann wird der Knoten node parallel zu first und last eingefügt und zwischen jedem Knoten aus beforeNodes und node und zwischen node und afterNodes ein Sync-Kante eingefügt.
Datenfluss:	
createDataElement(newDataElement)	Siehe addDataElement
deleteDataElement(dataElement)	Siehe removeDataElement
createDataEdge(node, dataElement, type)	Siehe addDataEdge
deleteDataEdge(node, dataElement, type)	Siehe removeDataEdge
moveDataEdge(oldNode, oldDataElement, newNode, newDataElement, type)	Das Verschieben von Datenkanten muss atomar möglich sein. Intern wird dies mittels Löschen und Einfügen durchgeführt. Dabei kann der Typ (Lese-/Schreibkante) und damit die Bedeutung einer Kante nicht geändert werden.
Semantische Änderungsoperationen:	
insertActivitySerial(activity, before, after)	Fügt die Aktivität activity seriell zwischen after und before ein

Funktionale Beschreibung	
insertActivityParallel(activity, {activities})	Fügt die Aktivität activity parallel zu activities ein. Dazu wird um die erste und letzte Aktivität von activities ein neuer paralleler Verzweigungsblock erstellt und activity in den noch leeren Teilzweig dieses Verzweigungsblocks eingefügt. Falls um activities bereits ein paralleler Verzweigungsblock existiert, d.h. ist der Vorgänger der ersten Aktivität aus activitites ein paralleler Verzweigungsknoten und der Nachfolger der letzten Aktivität der entsprechende Vereinigungsknoten, so wird lediglich ein neuer Teilzweig erstellt. Dies gilt nicht, wenn bereits eine innere Verzweigung existiert, d.h. sind der erste und letzte Knoten aus activities parallele Verzweigungs- und Vereinigungsknoten, wird um diese (innere) parallele Verzweigung zusätzlich eine (äußere) Verzweigung erzeugt.
insertActivityAlternative(activity, {activities}, sc)	Fügt die Aktivität activity alternativ zu activities ein. Dazu wird um die erste und letzte Aktivität von activities ein neuer alternativer Verzweigungsblock erstellt und activity in den noch leeren Teilzweig dieses Verzweigungsblocks mit dem Auswahlcode sc eingefügt. Ist sc DEFAULT, so muss für die bereits bestehenden Aktivitäten ein weiterer Auswahlcode angegeben werden, andernfalls liegen diese im Vorgabezweig. Auch bei insertActivityAlternative gilt wie bei insertActivityParallel, dass nach außen (bezüglich activities) nach einer vorhandenen alternativen Verzweigung gesucht wird. Ist diese vorhanden, wird nur ein neuer Teilzweig anstelle eines neuen alternativen Verzweigungsblocks erzeugt.
insertActivityBetweenActivitySets(activity, {beforeActivities}, {afterActivities})	Fügt eine Aktivität zwischen zwei Knotenmengen ein. Siehe insertNodeBetweenNodeSets
moveActivitiesSerial(first, last, before, after)	Verschiebt die durch first und last spezifizierte Knotenmenge (Menge aller Knoten zwischen first und last (inkl.)) zwischen after und before. Handelt es sich bei der zu verschiebenden Knotenmenge um einen kompletten Teilzweig, so entspricht diese Operation moveComplete-Branch, d.h. es bleibt kein leerer Teilzweig übrig und die umgebenden Knoten werden beim vorletzten Teilzweig gelöscht (s.a. moveCompleteBranch).
moveActivitiesParallel(first, last, {activities})	Verschiebt die durch first und last spezifizierte Knotenmenge (Menge aller Knoten zwischen first und last (inkl.)) parallel zu activities. Handelt es sich bei der zu verschiebenden Knotenmenge um einen kompletten Teilzweig, so entspricht diese Operation moveCompleteBranch. Wie bei insertActivityParallel wird auch hier kein neuer Verzweigungsblock erstellt, wenn die Menge activities genau die inneren Knoten einer existierenden parallelen Verzweigung umfasst.
moveActivitiesAlternative (first, last, {activities}, sc)	Verschiebt die durch first und last spezifizierte Knotenmenge (Menge aller Knoten zwischen first und last (inkl.)) alternativ zu activities. Das Vorgehen bzgl. der Behandlung von zu verschiebenden Teilzweigen und dem Einfügen in existierende alternative Verzweigungen entspricht dem Vorgehen bei moveActivitiesParallel.

Tabelle 0.1 Funktionale Beschreibung

A.2 (Vor- und Nachbedingungen)

Übersicht über die in dieser Arbeit entwickelten Vor- und Nachbedingungen inklusive der für die Durchführung notwendigen Methoden des ADEPT2-Datemodells:

	Strukturelle Vorbedingung:	**Benötigte Testmethoden: (aus Datenmodell)**
Kontrollfluss:		
Strukturelle Änderungen:		
Einfache Änderungsoperationen:		
Knoten:		
insertNode(newNode, pred, succ)	- newNode ist noch nicht im Schema enthalten - pred und succ sind im Schema enthalten und über eine Kontrollkante direkt miteinander verbunden	- nodeExists(newNode) - getSuccByEdgeType(CONTROL) - getType für pred und succ
deleteNode(node)	- node ist im Schema enthalten - node muss vom Typ NORMAL_NODE sein	- getType(node)

	Strukturelle Vorbedingung:	**Benötigte Testmethoden: (aus Datenmodell)**
moveNodes(first, last, pred, succ)	- pred, succ, first und last sind im Schema enthaltene Knoten - pred und succ sind durch eine Kontrollkante miteinander verbunden (dazu zählen auch Verzweigungs- und Vereinigungsknoten, die durch einen leeren Zweig verbunden sind) - first und last bilden einen gültigen Kontrollblock, d.h. sie befinden sich auf derselben Blockebene - durch die Verschiebung dürfen keine Zyklen durch vorhandene Sync-Kanten entstehen.	- getSuccByEdgeType(CONTROL) - getBranchID für first und last - getType von pred und succ - pred.getSuccByEdgeType (SPLIT_JOIN) = succ - leere Kanten suchen: split.getOutEdges, edge.type = CONTROL, edge.getDestinationNode() = succ - mit isPredecessorOf prüfen, ob von einem Sync- Kantenursprung aus das zugehörige Sync-Kantenziel erreicht werden kann
assignActivity(node, activity)	- node ist im Schema enthalten - die Aktivitätenvorlage „activity“ existiert und darf in dem gegebenen Kontext verwendet werden (Semantik!!)	- Methode, die semantische Aspekte der Aktivitätenvorlage überprüft (für zukünftige Erweiterungen vorsehen)
Kanten:		
insertEmptyBranch(split, join, [sc])	- split ist ein im Schema vorhandener Splitknoten, join der dazugehörige Vereinigungsknoten - es existiert noch kein leerer Teilzweig zwischen split und join Bei alternativen Verzweigungen ist ein Auswahlcode sc anzugeben: - sc darf nicht DEFAULT sein	- split.getSuccByEdgeType (SPLIT_JOIN) = join - split.getSuccByEdgeType (CONTROL) = NULL - split.getType
deleteEmptyBranch(split, join)	- split ist ein im Schema vorhandener Splitknoten, join der dazugehörige Vereinigungsknoten - mindestens 2 Teilzweige zwischen src, dest, davon mindestens einer leer. - ist split vom Typ OR_SPLIT muss ein Auswahlcode sc angegeben werden - der durch sc angegeben Teilzweig muss leer sein - sc darf nicht „DEFAULT“ sein	- s.o. - split.getOutEdges(), mindestens zwei Kanten mit edge.type = CONTROL - mindestens eine davon mit edge.getDestinationNode() = join - split.getType()
createSyncEdge(src, dest)	- dest ist kein Nachfolger von src (vorwärtsgerichtete Sync-Kante) - src ist kein Nachfolger von dest (rückwärtsgerichtete Sync-Kante) - src und dest nicht in unterschiedlichen Teilzweigen einer bedingten Verzweigung oder einer Verzweigung mit finaler Auswahl - es dürfen keine zyklischen Abhängigkeiten entstehen → src/dest nicht Nachfolger von sich selbst	- dest.isPredecessorOf(src, CONTROL) - src.isPredecessorOf(dest, CONTROL) - Process.getMinBlock({src, dest}) - mit isPredecessorOf prüfen, ob vom Kantenursprung aus das Kantenziel erreicht werden kann
createPriorityEdge(src, dest)	- noch nicht abschließend definiert	
createFailureEdge	- noch nicht abschließend definiert	
deleteSyncEdge(src, dest)	- Kante im Schema enthalten und vom Typ Sync-Kante	
deletePriorityEdge	- noch nicht abschließend definiert	
deleteFailureEdge	- noch nicht abschließend definiert	
nicht-strukturelle Änderungen (Attributänderungen)		
changeNodeAttribute (node, attribute, newValue)	- node ist im Schema enthalten - attribute darf kein Attribut sein, das die Struktur des angegebenen Knotens verändert (z.B. Knotentyp)	- Methode checkAttributeType (noch nicht im Datenmodell enthalten)
changeActivityAttribute (activity, attribute, newValue)	- activity existiert im Aktivitätenvorlagen-Repository - activity besitzt das Attribut „attribute“ - es entstehen keine semantischen Konflikte	- Methode, die effizient das Aktivitätenvorlagen-Repository nach Aktivitäten durchsuchen kann - hasAttribute(activity, attribute) (noch nicht im Datenmodell enthalten)

	Strukturelle Vorbedingung:	**Benötigte Testmethoden: (aus Datenmodell)**
changeEdgeAttribute (src, dest, type, attribute, newValue)	- src, dest sind im Schema enthalten und durch den in „type“ angegebenen Kantentyp verbunden - attribute ist ein für diese Kantenart zulässiges Attribut - attribute darf kein Attribut sein, das die Struktur der angegebenen Kante verändert (z.B. Quell-/Zielknoten)	- hasEdgeAttribute(src, dest, attribute) (noch nicht im Datenmodell enthalten)
Komplexe Änderungsoperationen:		
insertEmptyBlock(type, pred, succ) (Block mit einem Teilzweig, bei OR ist dies der Default-Zweig)	- pred und succ sind im Schema enthaltene Knoten - pred und succ sind durch eine Kontrollkante miteinander verbunden (dazu zählen auch Verzweigungs- und Vereinigungsknoten, die durch einen leeren Zweig verbunden sind) - type ist entweder OR, AND, AND_OR oder LOOP	- pred.getSuccByEdgeType (CONTROL) - pred.getType()
createSurroundingBlock (type, first, last)	- first und last sind im Schema enthaltene Knoten - first und last befinden sich auf der selben Blockebene - type ist entweder OR, AND, AND_OR oder LOOP - ist als „type“ LOOP angegeben, - so darf keiner der Knoten zwischen first und last (first, last inklusive) eine ein- oder ausgehende Sync-Kante besitzen - und ist ein Knoten zwischen first und last ein Schleifenanfangs- oder ein Schleifenendknoten, so muss der korrespondierende Anfangs- bzw. Endknoten ebenfalls zwischen first und last liegen	- first.getBranchID() = last.getBranchID() - für alle Knoten zwischen first und last: node.getSuccByEdgeType(SYNC), node.getPredByEdgeType(SYNC)
convertBlock(split, join, type, [{sc}])	- der durch split, join angegeben Block ist nicht vom Typ „type“ - split ist ein im Schema vorhandener Splitknoten, join der dazugehörige Vereinigungsknoten - type ist entweder OR, AND, AND_OR - ist type eine OR-Verzweigung, so sind so viele Auswahlcodes anzugeben wie Teilzweige existieren. Die Reihenfolge der Auswahlcodes entspricht der Reihenfolge der Kanten wobei genau ein Teilzweig den Auswahlcode DEFAULT besitzen muss	- split.getType(), join.getType() - split.getSuccByEdgeType (SPLIT_JOIN) - split.getOutEdges() - edge.getEdgeCode()
deleteBorderNodes(split, join)	- split ist ein im Schema vorhandener Verzweigungs-/Schleifen-anfangsknoten, join der dazugehörige Vereinigungs-/Schleifen-endknoten - zwischen split und join existiert genau ein Teilzweig	- split.getType(), join.getType() - split.getOutEdges()
deleteCompleteBranch(first, last)	- first und last sind im Schema enthaltene Knoten - first ist erster und last letzter Knoten eines Teilzweiges - der DEFAULT-Zweig bei alternativen Verzweigungen darf nur gelöscht werden, wenn er der letzte Teilzweig ist	- pred(first).getType(), succ(last).getType() - pred(first).getOutEdges()
moveCompleteBranch (first, last, pred, succ)	- pred, succ, first und last sind im Schema enthaltene Knoten - first ist erster und last letzter Knoten eines Teilzweiges - bei alternativen Verzweigungen darf der DEFAULT Zweig nur verschoben werden, wenn dieser der letzte Teilzweig der Verzweigung ist	- pred(first).getType(), succ(last).getType() - pred(first).getOutEdges()
deleteNodes(first, last)	- first und last sind im Schema enthaltene Knoten und befinden sich auf derselben Blockebene	- first.getBranchID() = last.getBranchID()

	Strukturelle Vorbedingung:	**Benötigte Testmethoden: (aus Datenmodell)**
insertNodeBetweenNodeSets (node, {beforeNodes}, {afterNodes})	- node und die Knoten aus beforeNodes und afterNodes sind im Schema enthalten - alle Knoten aus beforeNodes sind im Kontrollfluss vor den Knoten aus afterNodes	- isPredecessorOf - getMinBlock (beforeActivities+afterActivities)
Datenfluss:		
createDataElement (newDataElement)	- je nach Implementierung, evtl. Eindeutigkeit des Bezeichners prüfen	
deleteDataElement(dataElement)	- dataElement darf keine ein- oder ausgehenden Datenkanten besitzen (Alternativ: Datenkanten mitlöschen) - zusätzlich auf Instanzebene: Element darf nur gelöscht werden, wenn noch kein Schreibzugriff erfolgt ist	- dataElement.getReadingNodes(), dataElement.getWritingNodes() - Methode, die anzeigt ob ein Schreibzugriff erfolgt ist (noch nicht im Datenmodell enthalten)
createDataEdge (node, dataElement, type)	- node und dataElement müssen im Schema enthalten sein - die Ein- bzw. Ausgabeparameter von node müssen mit den Parametern von dataElement kompatibel sein - type ist entweder Read oder Write	- node.getInputDataElements bzw. Node.getOutputDataElements - dataElement.getType
deleteDataEdge (node, dataElement, type)	- node und dataElement müssen im Schema enthalten sein - zusätzlich auf Instanzebene: die Kante darf nur gelöscht werden, wenn node noch nicht beendet ist	- (InstanceNode)) node.getNodeState()
moveDataEdge (oldNode, oldDataElement, newNode, newDataElement, type)	- oldNode, oldDataElement, newNode, newDataElement müssen im Schema enthalten sein	
Semantische Änderungsoperationen (nicht Bestandteil der Basisänderungsoperationen; können aber auf Grund des Plug-In Interfaces eingefügt werden):		
insertActivitySerial (activity, before, after)	- before und after sind im Schema enthaltende Aktivitäten - before und after sind durch eine Kontrollkante miteinander verbunden - mindestens ein Knoten von before und after ist kein Split bzw. Join-Knoten	- before.getSuccByEdgeType (CONTROL) = after - before.getType(), after.getType()
insertActivityParallel (activity, {activities})	- activities sind im Schema enthaltende Aktivitäten	- first.getType(), last.getType(), pred(first).getType(), succ(last).getType() notwendig für Anwendung der Änderungsoperationen
insertActivityAlternative (activity, {activities}, sc)	- activities sind im Schema enthaltende Aktivitäten	- first.getType(), last.getType(), pred(first).getType(), succ(last).getType() notwendig für Anwendung der
insertActivityBetweenActivitySets (activity, {beforeActivities}, {afterActivities})	- alle Aktivitäten beforeActivities und afterActivities sind im Schema enthalten - die Aktivitäten beforeActivities liegen alle im Kontrollfluß vor den Aktivitäten afterActivities	- process.getMinBlock ({beforeActivities,afterActivities}) Ergebnis: first und last
moveActivitiesSerial (first, last, before, after)	- first, last, before, after sind im Schema enthaltene Aktivitäten - before und after sind durch eine Kontrollkante miteinander verbunden - first und last befinden sich auf der selben Blockebene	- before.getSuccByEdgeType (CONTROL) = after - first.getBranchID() = last.getBranchID()
moveActivitiesParallel (first, last, {activities})	- first, last und activities sind im Schema enthaltene Aktivitäten - first und last befinden sich auf der selben Blockebene - die „erste“ und die „letzte“ Aktivität (firstActivity, lastActivity) aus activities befinden sich auf der selben Blockebene	- first.getBranchID() = last.getBranchID() - firstActivity.getBranchID() = lastActivity.getBranchID() - first.getType(), last.getType(), pred(first).getType(), succ(last).getType() notwendig für Anwendung der Änderungsoperationen

	Strukturelle Vorbedingung:	**Benötigte Testmethoden: (aus Datenmodell)**
moveActivitiesAlternative (first, last, {activities}, sc)	- first, last und activities sind im Schema enthaltene Aktivitäten - first und last befinden sich auf der selben Blockebene - die „erste“ und die „letzte“ Aktivität aus activities befinden sich auf der selben Blockebene	s.o.

Tabelle 0.2 Strukturelle Vorbedingungen

Zustandsbasierte Vorbedingung: **(entfällt bei der Modellierung neuer bzw. Änderung bestehender Schemata)**	
Kontrollfluss:	**Die folgenden Tests benötigen außer der Methode getNodeState() keine weiteren Methoden aus dem Datenmodell.**
Strukturelle Änderungen:	
Einfache Änderungsoperationen:	
Knoten:	
insertNode(newNode, pred, succ)	- succ befindet sich in einem der Zustände Not_Activated, Activated - zusätzlich bei Schemaevolution zulässig: succ befindet sich im Zustand Skipped
deleteNode(node)	- node befindet sich in einem der Zustände Not_Activated, Activated - zusätzlich bei Schemaevolution zulässig: node befindet sich im Zustand Skipped
moveNodes(first, last, pred, succ)	- first und succ befinden sich in einem der Zustände Not_Activated, Activated - zusätzlich bei Schemaevolution zulässig: - first und pred sind Skipped **oder** - pred ist Completed, succ ist Activated oder Not_Activated (Not_Activated ist möglich wenn succ ein Join-Knoten ist), first ist entweder Completed oder Running und es gilt: der Historieneintrag End(pred) steht vor dem Eintrag Start(first) **oder** - pred, last sind im Zustand Completed, succ ist Running oder Completed und es gilt: der Historieneintrag End(pred) steht vor Start(first) und End(last) steht vor Start(succ) (Testmethode nicht im Datenmodell enthalten)
assignActivity(node, activity)	- node befindet sich in einem der Zustände Not_Activated Activated - zusätzlich bei Schemaevolution zulässig: node ist Skipped
Kanten:	
insertEmptyBranch(split, join, sc)	- bei Ad-hoc-Änderungen: split befindet sich in einem der Zustände Not_Activated oder Activated - bei Schemaevolution zulässig: join befindet sich in einem beliebigen Zustand (leere Kante hat keine Auswirkungen!)
deleteEmptyBranch(split, join)	- join befindet sich in einem der Zustände Not_Activated oder Activated - zusätzlich bei Schemaevolution zulässig: join befindet sich in einem beliebigen Zustand (leere Kante hat keine Auswirkungen!)
createSyncEdge(src, dest)	- dest befindet sich in einem der Zustände Not_Activated, Activated oder Skipped - zusätzlich bei Schemaevolution zulässig: - src ist Completed, dest ist Running oder Completed und es gilt: der Historieneintrag End(src) steht vor dem Eintrag Start(dest) oder - src ist Skipped und dest ist Running oder Completed und es gilt: der Historieneintrag End(x) des ersten Vorgängers von src der nicht den Zustand Skipped besitzt (=x) steht vor dem Eintrag Start(dest)
createPriorityEdge(src, dest)	- noch nicht abschließend definiert
createFailureEdge	- noch nicht abschließend definiert
deleteSyncEdge(src, dest)	- dest befindet sich in einem der Zustände Not_Activated oder Activated - zusätzlich bei Schemaevolution zulässig: dest befindet sich in einem beliebigen Zustand
deletePriorityEdge	- noch nicht abschließend definiert
deleteFailureEdge	- noch nicht abschließend definiert

Zustandsbasierte Vorbedingung: **(entfällt bei der Modellierung neuer bzw. Änderung bestehender Schemata)**	
nicht-strukturelle Änderungen (Attributänderungen)	
changeNodeAttribute(node, attribute, newValue)	- node befindet sich in einem der Zustände Not_Activated oder Activated - zusätzlich bei Schemaevolution zulässig: node ist Skipped
changeActivityAttribute(activity, attribute, newValue)	- der Knoten mit dem die Aktivität activity verbunden ist befindet sich in einem der Zustände Not_Activated oder Activated - zusätzlich bei Schemevolution zulässig: der Knoten ist skipped
changeEdgeAttribute(src, dest, type, attribute, newValue)	- dest befindet sich in einem der Zustände Not_Activated oder Activated - zusätzlich bei Schemevolution zulässig: dest ist skipped
komplexe Änderungsoperationen:	
insertEmptyBlock(type, pred, succ) (Block mit einem Teilzweig, bei OR ist dies der Default-Zweig)	- gleiche Bedingungen wie bei insertNode
createSurroundingBlock(type, first, last)	Optimierung: - first befindet sich entweder im Zustand Not_Activated oder Activated - zusätzlich bei Schemaevolution möglich: first ist Skipped
convertBlock(split, join, type, [{sc}])	- alle direkten Nachfolger von split bzgl. Kontrollkanten befinden sich im Zustand Activated, Not_Activated oder Skipped
deleteBorderNodes(split, join)	- split befindet sich im Zustand Not_Activated oder Activated - zusätzlich bei Schemaevolution möglich: split ist Skipped
deleteCompleteBranch(first, last)	- first befindet sich im Zustand Not_Activated oder Activated handelt es sich um den letzten Zweig einer Verzweigung, so darf der Split-Knoten (pred(first)) nur einen der Zustände Activated oder Not-Activated besitzen, da dieser mit gelöscht wird - zusätzlich bei Schemaevolution möglich: wie oben nur mit Skipped
moveCompleteBranch(first, last, pred, succ)	- first und succ befinden sich im Zustand Not_Activated oder Activated handelt es sich um den letzten Zweig einer Verzweigung, so darf der Split-Knoten (pred(first)) nur einen der Zustände Activated oder Not-Activated besitzen, da dieser mit gelöscht wird - zusätzlich bei Schemaevolution möglich: - wie oben nur mit Skipped oder - siehe Kriterien bei moveNodes
deleteNodes(first, last)	- first besitzt einen der Zustände Not_Activated oder Activated - zusätzlich bei Schemaevolution möglich: first ist Skipped
insertNodeBetweenNodeSets(node, {beforeNodes}, {afterNodes})	- der im Kontrollfluss am weitesten vorne liegende Knoten (first) aus „beforeNodes“ besitzt einen der Zustände Not_Activated oder Activated **oder** falls der direkte Vorgänger von first ein AND-Split-Knoten und der direkte Nachfolger von last ein Join-Knoten ist, darf der Join-Knoten nur einen der Zustände Activated oder Not_Activated besitzen - zusätzlich bei Schemaevolution möglich: first bzw. Join ist Skipped
Datenfluss:	
createDataElement(newDataElement)	- immer verträglich
deleteDataElement(dataElement)	- jeder Knoten der das Datenelement dataElement schreibt befindet sich in einem der Zustände Activated oder Not_Activated zusätzlich bei Schemaevolution zulässig: Skipped
createDataEdge(node, dataElement, type)	- handelt es sich um eine Lesekante: der lesende Knoten besitzt einen der Zustände Activated oder Not_Activated - handelt es sich um eine Schreibkante: der schreibende Knoten besitzt einen der Zustände Activated oder Not_Activated zusätzlich bei Schemaevolution zulässig: Skipped
deleteDataEdge(node, dataElement, type)	- siehe createDataEdge
moveDataEdge(oldNode, oldDataElement, newNode, newDataElement, type)	- siehe deleteDataEdge bzw. createDataEdge

Zustandsbasierte Vorbedingung: **(entfällt bei der Modellierung neuer bzw. Änderung bestehender Schemata)**	
Semantische Änderungsoperationen (nicht Bestandteil der Basisänderungsoperationen; können aber auf Grund des Plug-In Interfaces eingefügt werden):	
insertActivitySerial(activity, before, after)	- after besitzt einen der Zustände Not_Activated oder Activated - zusätzlich bei Schemaevolution möglich: after ist Skipped
insertActivityParallel(activity, {activities})	- die im Kontrollfluss am weitesten vorne liegende Aktivität (first) aus „activities“ besitzt einen der Zustände Not_Activated oder Activated **oder** falls der direkte Vorgänger von first ein AND-Split-Knoten und der direkte Nachfolger von last ein Join-Knoten ist, darf der Join-Knoten nur einen der Zustände Activated oder Not_Activated besitzen - zusätzlich bei Schemaevolution möglich: first bzw. Join ist Skipped
insertActivityAlternative(activity, {activities}, sc)	- first besitzt einen der Zustände Not_Activated oder Activated **oder** falls der direkte Vorgänger von first ein OR-Split-Knoten und der direkte Nachfolger von last ein Join-Knoten ist, darf der Join-Knoten nur einen der Zustände Activated oder Not_Activated besitzen - zusätzlich bei Schemaevolution möglich: first bzw. Join ist Skipped
insertActivityBetweenActivitySets(activity, {beforeActivities}, {afterActivities})	- siehe insertActivityParallel
moveActivitiesSerial(first, last, before, after)	- siehe moveNodes **oder** - siehe moveCompleteBranch
moveActivitiesParallel(first, last, {activities})	- first (bzw. pred(first), falls ein ganzer Zweig, der zusätzlich noch der letzte Zweig ist verschoben wird) und firstActivity besitzen einen der Zustände Activated oder Not_Activated - zusätzlich bei Schemaevolution möglich: - first und/oder firstActivity können Skipped sein **oder** - siehe moveNodes bzw. moveCompleteBranch falls der Vorgänge von firstActivity ein AND-Split und der Nachfolger von lastActivity ein AND-Join Knoten ist
moveActivitiesAlternative(first, last, {activities}, sc)	- first (bzw. pred(first), falls ein ganzer Zweig, der zusätzlich noch der letzte Zweig ist verschoben wird) und firstActivity besitzen einen der Zustände Activated oder Not_Activated - zusätzlich bei Schemaevolution möglich: - first und/oder firstActivity können Skipped sein oder - siehe moveNodes bzw. moveCompleteBranch falls der Vorgänge von firstActivity ein OR-Split und der Nachfolger von lastActivity ein OR-Join Knoten ist

Tabelle 0.3 Zustandsbasierte Vorbedingungen

Verwendete Änderungsprimitiven:	
Kontrollfluss:	
Strukturelle Änderungen:	
Einfache Änderungsoperationen:	
Knoten:	
insertNode(newNode, pred, succ)	- addNode(newNode, NORMAL) - Methode removeEdge(src, dest, CONTROL) (Methode - addEdge(pred, newNode, CONTROL) - addEdge(newNode, succ, CONTROL)
deleteNode(node)	- Methode removeEdge(pred(node), node, CONTROL) - Methode removeEdge(node, succ(node), CONTROL) - addEdge(pred(node), succ(node), CONTROL) (ggf. muss der Auswahlcode der gelöschten Kante zwischen pred(node) und node übernommen werden) zusätzlich bei Schemaänderung: - removeEdge(pred(node), node, Kantentyp ≠ CONTROL) - removeEdge(node, succ(node), Kantentyp ≠ CONTROL) - removeNode(node) zusätzlich bei Ad-hoc-Änderung: - setEdgeAttribute(pred(node), node, Kantentyp ≠ CONTROL, "deleted", "true") - setEdgeAttribute(node, succ(node), Kantentyp ≠ CONTROL, "deleted", "true") - setNodeAttribute(node, "deleted", "true")

Verwendete Änderungsprimitiven:	
moveNodes(first, last, pred, succ)	- removeEdge(pred(first), first, CONTROL) - removeEdge(last, succ(last), CONTROL) - addEdge(pred(first), succ(last), CONTROL, Attribute(pred(first)), Attribute von pred(first) Kante übernehmen - removeEdge(pred, succ, CONTROL, [sc]) - addEdge(pred, first, CONTROL, [sc]) - addEdge(last, succ, CONTROL) - addMovedNode für alle Knoten zwischen first und last: addMovedNode(first), ..., addMovedNode(last)
assignActivity(node, activity)	- changeNodeAttribute(node, "ACTIVITY", activity)
Kanten:	
insertEmptyBranch(split, join, sc)	- addEdge(split, join, CONTROL, [sc])
deleteEmptyBranch(split, join)	- removeEdge(pred, succ, CONTROL, [sc])
createSyncEdge(src, dest)	- addEdge(src, dest, SYNC,…)
createPriorityEdge(src, dest)	- noch nicht abschließend definiert
createFailureEdge	- noch nicht abschließend definiert
deleteSyncEdge(src, dest)	- removeEdge(src, dest, SYNC)
deletePriorityEdge	- noch nicht abschließend definiert
deleteFailureEdge	- noch nicht abschließend definiert
nicht-strukturelle Änderungen (Attributänderungen)	
changeNodeAttribute(node, attribute, newValue)	- setNodeAttribute(node, attribute, newValue)
changeActivityAttribute(activity, attribute, newValue)	- setActivityAttribute(activity, attribute, newValue)
changeEdgeAttribute(src, dest, type, attribute, newValue)	- setEdgeAttribute(src, dest, type, attribute, newValue)
komplexe Änderungsoperationen:	
insertEmptyBlock(type, pred, succ) (Block mit einem Teilzweig, bei OR ist dies der Default-Zweig)	- insertNode - evtl. bei “Block in Block” setEdgeAttribute(pred(pred), pred, CONTROL "SELECTION_CODE", sc)
createSurroundingBlock(type, first, last)	- insertEmptyBlock(type, pred(first), first, [sc]) - moveNodes(first, last, pred(pred(first)), pred(first), [sc]), sc = DEFAULT bei OR-Verzweigung
convertBlock(split, join, type, [{sc}])	Änderungsprimitiven (da changeNodeAttribute die Änderung des Typs nicht erlaubt): '- setNodeAttribute(split, "TYPE", type) - setNodeAttribute(join, "TYPE", type) - setEdgeAttribute(split, succ(split), "CONTROL", "SELECTION_CODE", sc)
deleteBorderNodes(split, join)	- deleteNode
deleteCompleteBranch(first, last)	- deleteNodes(first, last) - deleteEmptyBranch(pred(first), succ(last), [sc]) - ggf. deleteBorderNodes für "innere" Verzweigungen bzw. falls (first, last) der letzte Teilzweig ist
moveCompleteBranch(first, last, pred, succ)	- insertEmptyBranch, falls pred und succ Split- und Join-Knoten sind - moveNodes(first, last, pred, succ, [sc]) - deleteBorderNodes(pred(first), succ(last)), falls der verschobene Teilzweig der letzte Teilzweig war
deleteNodes(first, last)	- deleteNode - deleteCompleteBranch, falls "interne" Blöcke existieren, deren Teilzweige Knoten besitzen
insertNodeBetweenNodeSets(node, {beforeNodes}, {afterNodes})	- createSurroundingBlock(AND, first, last), first ist der im Kontrollblock am weitesten vorne liegende Knoten aus „beforeNodes“, last der am weitesten hinten liegende aus afterNodes - insertNode(newNode, split, join, sc), split, join Knoten aus createSurroundingBlock - createSyncEdge(src, dest, type)
Datenfluss:	**Verwendete Primitiven:**
createDataElement(newDataElement)	- addDataElement(newDataElement.Type)
deleteDataElement(dataElement)	- removeDataElement(dataElement) - ggf. removeDataEdge(node, dataElement, type)
createDataEdge(node, dataElement, type)	- addDataEdge(node, dataElement, type)
deleteDataEdge(node, dataElement, type)	- removeDataEdge(node, dataElement, type)
moveDataEdge(oldNode, oldDataElement, newNode, newDataElement, type)	- removeDataEdge(oldNode, oldDataElement, type) - addDataEdge(newNode, newDataElement, type)

Verwendete Änderungsprimitiven:	
Semantische Änderungsoperationen (nicht Bestandteil der Basisänderungsoperationen; können aber auf Grund des Plug-In Interfaces eingefügt werden):	
insertActivitySerial(activity, before, after)	- insertNode(newNode,before, after) - assignActivity(newNode, activity)
insertActivityParallel(activity, {activities})	- createSurroundingBlock(AND, first, last) wenn first.getType(), last.getType() = NORMAL oder SPLIT/JOIN + insertEmptyBranch oder - insertEmptyBranch(pred(first), succ(last)), falls pred(first).getType() = AND_Split, succ(last).getType() = AND_JOIN - insertNode(newNode, split, join) - assignActivity(newNode, activity)
insertActivityAlternative(activity, {activities}, sc)	- createSurroundingBlock(OR, first, last) wenn first.getType(), last.getType() = NORMAL oder SPLIT/JOIN + insertEmptyBranch oder - insertEmptyBranch(pred(first), succ(last)), falls pred(first).getType() = OR_Split, succ(last).getType() = OR_JOIN - insertNode(newNode, split, join, sc) - assignActivity(newNode, activity)
insertActivityBetweenActivitySets(activity, {beforeActivities}, {afterActivities})	- createSurroundingBlock(AND, first, last) - insertNode(newNode, split, join, sc) - assignActivity(newNode, activity)) - createSyncEdge(src, dest, type)
moveActivitiesSerial(first, last, before, after)	- moveNodes(first, last, before, after) **oder** - moveCompleteBranch(first, last, before, after) falls first, last Anfangs- und Endknoten eines Zweiges sind
moveActivitiesParallel(first, last, {activities})	- createSurroundingBlock(AND, firstActivity, lastActivity) + insertEmptyBranch **oder** - insertEmptyBranch(pred(firstActivity), succ(lastActivity)), falls pred(firstActivity).getType() = AND_Split, succ(lastActivity).getType() = AND_JOIN - moveNodes(first, last, firstActivity, lastActivity) oder - moveCompleteBranch falls first, last Anfangs- und Endknoten eines Zweiges sind (dabei entfällt insertEmptyBranch, da dies von moveCompleteBranch ausgeführt wird)

Tabelle 0.4 Verwendete Änderungsprimitiven

	Nachfolgende Tests	Nachbedingung
Kontrollfluss:		
Strukturelle Änderungen:		
einfache Änderungsoperationen:		
Knoten:		
insertNode(newNode, pred, succ)		- Knoten und Kanten erfolgreich eingefügt - topologische Sortierung (falls vorhanden) aktualisiert
deleteNode(node)	- Datenfluss anpassen, falls "node" ein Datenelement schreibt, das ein anderer Knoten obligat liest	- ein- und ausgehende Kontrollkanten von node wurden gelöscht - neue Kante zwischen Vorgänger und Nachfolger des gelöschten Knotens wurde eingefügt zusätzlich bei Schemaänderung: - alle anderen ein- und ausgehenden Kanten wurden ebenfalls physisch gelöscht - der Knoten wurde physisch gelöscht zusätzlich bei Ad-hoc-Änderung: - alle anderen ein- und ausgehenden Kanten wurden als gelöscht markiert - der Knoten wurde als gelöscht markiert

	Nachfolgende Tests	**Nachbedingung**
moveNodes(first, last, pred, succ)	- prüfen, ob die ein- und ausgehenden Sync-Kanten des verschobenen Blockes noch erlaubt sind (Zyklenfreiheit). Evtl. gleiche Testmethoden wie für createSyncEdge verwendbar. Falls durch irgendeine Sync-Kante ein Zyklus entsteht, muss die gesamte Move-Operation rückgängig gemacht werden oder der Benutzer interaktiv zum Löschen entsprechender Kanten aufgefordert werden. - Datenfluss überprüfen: sowohl lesende als auch schreibende Zugriffe von verschobenen Knoten	- pred(first) und succ(last) sind mit einer Kontrollkante verbunden, die die gleichen Attribute besitzt, wie sie die ehemalige Kante (pred(first), first) besaß. - der Block (first, last) wurde zwischen (pred, succ) erfolgreich eingefügt, ggf. mit korrektem Auswahlcode - topologische Sortierung (falls vorhanden) wurde aktualisiert - Es existieren keine Zyklen durch Sync-Kanten - Der Datenfluss bleibt korrekt, d.h. die von den verschobenen Knoten benötigten Daten werden alle noch vor Ausführung der Knoten geschrieben und alle verschobenen Knoten schreiben ihre Daten noch rechtzeitig.
assignActivity(node, activity)	- Datenflusskorrektheit	- Knoten "node" besitzt die Aktivitätenvorlage "activity"
Kanten:		
insertEmptyBranch(split, join, sc)		- zwischen split und join wurde ein leerer Teilzweig eingefügt - ist "split" vom Typ OR_SPLIT oder "join" vom Typ OR_JOIN, so wurde ein gültiger Auswahlcode ≠ DEFAULT hinzugefügt
deleteEmptyBranch(split, join)		- bei paralleler Verzweigung: ein (beliebiger) leerer Zweig wurde gelöscht - bei alternativer Verzweigung: der Teilzweig mit dem Auswahlcode sc wurde gelöscht - es existiert mindestens noch ein Teilzweig
createSyncEdge(src, dest)		- Sync-Kante wurde erfolgreich eingefügt - topologische Sortierung (falls vorhanden) wurde aktualisiert
createPriorityEdge(src, dest)	- noch nicht abschließend	
createFailureEdge	- noch nicht abschließend definiert	
deleteSyncEdge(src, dest)	- Methode, die semantische Aspekte überprüft (für zukünftige Erweiterungen vorsehen) - writerExists Methode - evtl. Interaktion mit dem Benutzer bei inkorrekter Datenversorgung	- Semantik, z.B. durch "interne" Datenflüsse bleibt korrekt - Datenfluss bleibt korrekt; die obligaten Eingabeparamter von "dest" oder dessen Nachfolger (bis zum gemeinsamen Join-Knoten) werden vor deren Ausführung in das entsprechende Datenelement geschrieben
deletePriorityEdge	- noch nicht abschließend definiert	
deleteFailureEdge	- noch nicht abschließend definiert	
nicht-strukturelle Änderungen (Attributänderungen)		
changeNodeAttribute (node, attribute, newValue)		
changeActivityAttribute (activity, attribute, newValue)		
changeEdgeAttribute (src, dest, type, attribute, newValue)	- wurde der Auswahlcode einer Kante auf DEFAULT gesetzt, so muss interaktiv vom Benutzer ein neuer Auswahlcode für den bisherigen DEFAULT-Zweig eingegeben werden	- Kantentattribut wurde auf "newValue" geändert
Komplexe Änderungsoperationen:		

	Nachfolgende Tests	**Nachbedingung**
insertEmptyBlock(type, pred, succ) (Block mit einem Teilzweig, bei OR ist dies der Default-Zweig)		- ein leerer Verzweigungsblock vom Typ "type" mit einem leeren Teilzweig wurde erfolgreich zwischen pred und succ eingefügt. - handelt es sich um einen OR-Block, so hat die leere Kante des Verzweigungsblockes den Auswahlcode DEFAULT
createSurroundingBlock (type, first, last)	- Korrektheit des Datenflusses bei OR-Verzweigung sicherstellen - evtl. Interaktion mit dem Benutzer bei inkorrekter Datenversorgung - writerExists Methode	- die Knoten zwischen first und last sind von einem Block vom Typ "type" umgeben - handelt es sich um einen LOOP-Block, so besitzt keiner der umschlossenen Knoten eine ein- oder ausgehende Sync-Kante - Datenfluss ist korrekt
convertBlock(split, join, type, [{sc}])	- falls auf OR-Verzweigung geändert wurde, Korrektheit des Datenflusses sicherstellen - evtl. Interaktion mit dem Benutzer bei inkorrekter Datenversorgung - writerExists Methode	- Block wurde auf den in "type" angegebenen Typ geändert - ggf. Auswahlcodes sind korrekt - ggf. der Datenfluss wurde korrigiert
deleteBorderNodes(split, join)	- falls split, join oder Knoten des Teilzweiges obligat Datenelemente schreiben, muss durch Interaktion mit dem Benutzer der Datenfluss angepasst werden.	- Block wurde gelöscht - ggf. der Datenfluss wurde korrigiert
deleteCompleteBranch(first, last)	- falls Knoten des Teilzweiges obligat Datenelemente schreiben, muss durch Interaktion mit dem Benutzer der Datenfluss angepasst werden.	- der angebene Teilzweig wurde gelöscht - ggf. der Block wurde gelöscht - ggf. der Datenfluss wurde korrigiert
moveCompleteBranch (first, last, pred, succ)	- falls verschobener Teilzweig bereits einen Auswahlcode besitzt "kompatiblen" Auswahlcode behalten, ansonsten vom Benutzer erfragen - Datenfluss überprüfen (sollte durch moveNodes implizit geschehen)	- Knotenmenge wurde verschoben - ddf. Auswahlcode wurde angepasst - ggf. Datenfluss wurde korrigiert
deleteNodes(first, last)	- Datenfluss (siehe deleteNode)	- alle Knoten zwischen first und last wurden gelöscht bzw. als gelöscht markiert
insertNodeBetweenNodeSets (node, {beforeNodes}, {afterNodes})		- die Aktivität activity wurde so eingefügt, dass die Aktivitäten beforeActivities vor activity und afterActivities nach activity ausgeführt werden
Datenfluss:		
createDataElement (newDataElement)		- ein neues Datenelement wurde eingefügt
deleteDataElement(dataElement)	- Datenfluss überprüfen: - falls auch Kanten mitgelöscht werden: ggf. interaktiv mit User anpassen, bzw. die nicht mehr versorgten Aktivitäten farblich markieren	- das Datenelement wurde gelöscht - ggf. Datenfluss wurde angepasst
createDataEdge (node, dataElement, type)	- Test auf lost updates und paralleles Schreiben	- Datenkante wurde eingefügt - keine Konflikte
deleteDataEdge (node, dataElement, type)	- handelt es sich um eine schreibende Kante, so müssen wie bei deleteDataElement, die lesenden Aktiviäten überprüft werden	- Datenkante wurde eingefügt - keine Konflikte
moveDataEdge (oldNode, oldDataElement, newNode, newDataElement, type)	- type darf nicht verändert werden	- Kante wurde erfolgreich verschoben - keine Datenkonflikte
Semantische Änderungsoperationen (nicht Bestandteil der Basisänderungsoperationen; können aber auf Grund des Plug-In Interfaces eingefügt werden):		
insertActivitySerial (activity, before, after)		- die Aktivität wurde zwischen before und after eingefügt

	Nachfolgende Tests	Nachbedingung
insertActivityParallel (activity, {activities})		- activity wurde parallel zu (mindestens) den in Activities angegebenen Aktivitäten eingefügt
insertActivityAlternative (activity, {activities}, sc)	- falls sc = DEFAULT, muss für den anderen Teilzweig ein gültiger Auswahlcode vom Benutzer erfragt werden - wird activity in einen vorhandenen OR-Teilzweig eingefügt, so ist zu prüfen, ob der angegebene sc gültig ist	- activity wurde alternativ zu activities eingefügt - der Auswahlcode sc, wurde je nach zum Einsatz gekommenen Änderungsoperationen angepasst
insertActivityBetweenActivitySets (activity, {beforeActivities}, {afterActivities})		- die Aktivität activity wurde so eingefügt, dass die Aktivitäten beforeActivities vor activity und afterActivities nach activity ausgeführt werden
moveActivitiesSerial (first, last, before, after)	- Datenfluss und Zyklen durch Sync-Kanten überprüfen (sollte durch moveNodes implizit geschehen)	- die Aktivitäten wurden an die Position zwischen before und after verschoben - ggf. Datenfluss wurde korrigiert und Zyklenfreiheit überprüft
moveActivitiesParallel (first, last, {activities})	- Datenfluss und Zyklen durch Sync-Kanten überprüfen (sollte durch moveNodes implizit geschehen)	- die Aktivitäten zwischen first und last wurden parallel zu activities verschoben - ggf. Datenfluss wurde korrigiert und Zyklenfreiheit überprüft
moveActivitiesAlternative (first, last, {activities}, sc)	s.o.	- die Aktivitäten zwischen first und last wurden alternativ zu activities verschoben - ggf. Datenfluss wurde korrigiert und Zyklenfreiheit überprüft

Tabelle 0.5 Nachbedingungen

Zustandsneubewertung **(unnötig bei der Modellierung neuer bzw. Änderung bestehender Prozesse)**	
Kontrollfluss:	**Es werden keine Kanten neu bewertet, da dass ADEPT2-PMS keine Kantenzustände speichert. Bei Bedarf lassen sich die Kantenzustände direkt aus den Zuständen der Kantenstartknoten bestimmen.**
Strukturelle Änderungen:	
Einfache Änderungsoperationen:	
Knoten:	
insertNode(newNode, pred, succ)	- befindet sich pred im Zustand Activated, so muss dieser auf Not_Activated und newNode auf Activated gesetzt werden.
deleteNode(node)	- befindet sich node im Zustand Activated, so muss dessen Nachfolger auf Activated gesetzt werden
moveNodes(first, last, pred, succ)	- befindet sich succ im Zustand Activated, so muss dieser auf Not_Activated und first auf Activated gesetzt werden - befindet sich first im Zustand Activated, so muss dieser auf Not_Activated gesetzt werden, falls sich pred nicht im Zustand Completed befindet - befindet sich der Vorgänger von first (an der Ursprungsstelle) im Zustand Completed so muss der Nachfolger von last neu bewertet werden.
assignActivity(node, activity)	- entfällt
Kanten:	
insertEmptyBranch(split, join, sc)	- entfällt
deleteEmptyBranch(split, join)	- entfällt
createSyncEdge(src, dest)	- befindet sich dest im Zustand Activated, so muss dieser auf Not_Activated gesetzt werden
createPriorityEdge(src, dest)	- noch nicht abschließend definiert
CreateFailureEdge	- noch nicht abschließend definiert
deleteSyncEdge(src, dest)	- befinden sich alle Vorgänger von dest im Zustand Completed, so muss dieser auf Activated gesetzt werden
deletePriorityEdge	- noch nicht abschließend definiert
deleteFailureEdge	- noch nicht abschließend definiert
nicht-strukturelle Änderungen (Attributänderungen)	
changeNodeAttribute(node, attribute, newValue)	
changeActivityAttribute(activity, attribute, newValue)	- wird das Attribut für die Bearbeiterzuordnung geändert, so muss ein entsprechender Hinweis an die Worklist-Komponente geschickt werden!
changeEdgeAttribute(src, dest, type, attribute, newValue)	- wird der Auswahlcode sc einer Kante geändert, so besteht die Möglichkeit, dass alle Teilzweige einer alternativen Verzweigung neu bewertet werden müssen.

komplexe Änderungsoperationen:	
insertEmptyBlock(type, pred, succ) (Block mit einem Teilzweig, bei OR ist dies der Default-Zweig)	- befindet sich pred im Zustand Completed, so muss der Split-Knoten des eingefügten Blockes auf Activated gesetzt werden - befindet sich succ im Zustand Activated, so muss dieser auf Not_Activated gesetzt werden
createSurroundingBlock(type, first, last)	- befindet sich first im Zustand Activated, so muss der Split-Knoten des eingefügten Blockes auf Activated und first auf Not_Activated gesetzt werden
convertBlock(split, join, type, [{sc}])	'- wird ein And-Split in einen Or-Split gewandelt, muss der Knoten derjenigen Kante, die das Auswahlkriterium erfüllt auf Activated gesetzt und für die anderen Teilzweige eine Deadpath-Elimination durchgeführt werden - wird ein Or-Split in einen And-Split gewandelt, müssen die Skipped Zustände zurückgenommen und der erste Knoten jedes Zweiges mit Activated markiert werden (Sync-Kanten beachten!!)
deleteBorderNodes(split, join)	- befand sich der Vorgänger von split im Zustand Activated, so muss dessen neuer Nachfolger neu bewertet werden
deleteCompleteBranch(first, last)	- der Join Knoten hinter last muss neu bewertet werden. Handelt es sich um den letzten Teilzweig einer Verzweigung, so muss der Nachfolger des Join Knotens neu bewertet werden
moveCompleteBranch(first, last, pred, succ)	- siehe deleteCompleteBranch
deleteNodes(first, last)	- befindet sich first im Zustand Activated, so muss der Nachfolger von last neu bewertet werden
insertNodeBetweenNodeSets(node, {beforeNodes}, {afterNodes})	- befindet sich first (vorderster Knoten aus „beforeNodes“) im Zustand Activated, so muss der Split-Knoten des eingefügten Blockes auf Activated und first auf Not_Activated gesetzt werden
Datenfluss:	nimmt keinen Einfluss auf die Zustandsneubewertung!
Semantische Änderungsoperationen (nicht Bestandteil der Basisänderungsoperationen; können aber auf Grund des Plug-In Interfaces eingefügt werden):	
insertActivitySerial(activity, before, after)	- befindet sich after im Zustand Activated, so muss dieser auf Not_Activated und activity auf Activated gesetzt werden
insertActivityParallel(activity, {activities})	- siehe createSurroundingBlock bzw. insertEmptyBranch (abhängig vom Knotentyp des "ersten" Knotens aus activities)
insertActivityAlternative(activity, {activities}, sc)	- siehe Zustrandsneubewertung bei den verwendeten Änderungsoperationen
insertActivityBetweenActivitySets(activity, {beforeActivities}, {afterActivities})	- siehe Zustrandsneubewertung bei den verwendeten Änderungsoperationen
moveActivitiesSerial(first, last, before, after)	- befindet sich last nicht im Zustand Completed, so muss dessen Nachfolger neu bewertet werden. Befindet sich before im Zustand Completed, so muss after neu bewertet werden.
moveActivitiesParallel(first, last, {activities})	- siehe Zustrandsneubewertung bei den verwendeten Änderungsoperationen

Tabelle 0.6 Zustandsneubewertung

Anhang B (ADEPT2-Datenmodell)

Abbildung 0.1 ADEPT2-Datenmodell (1)

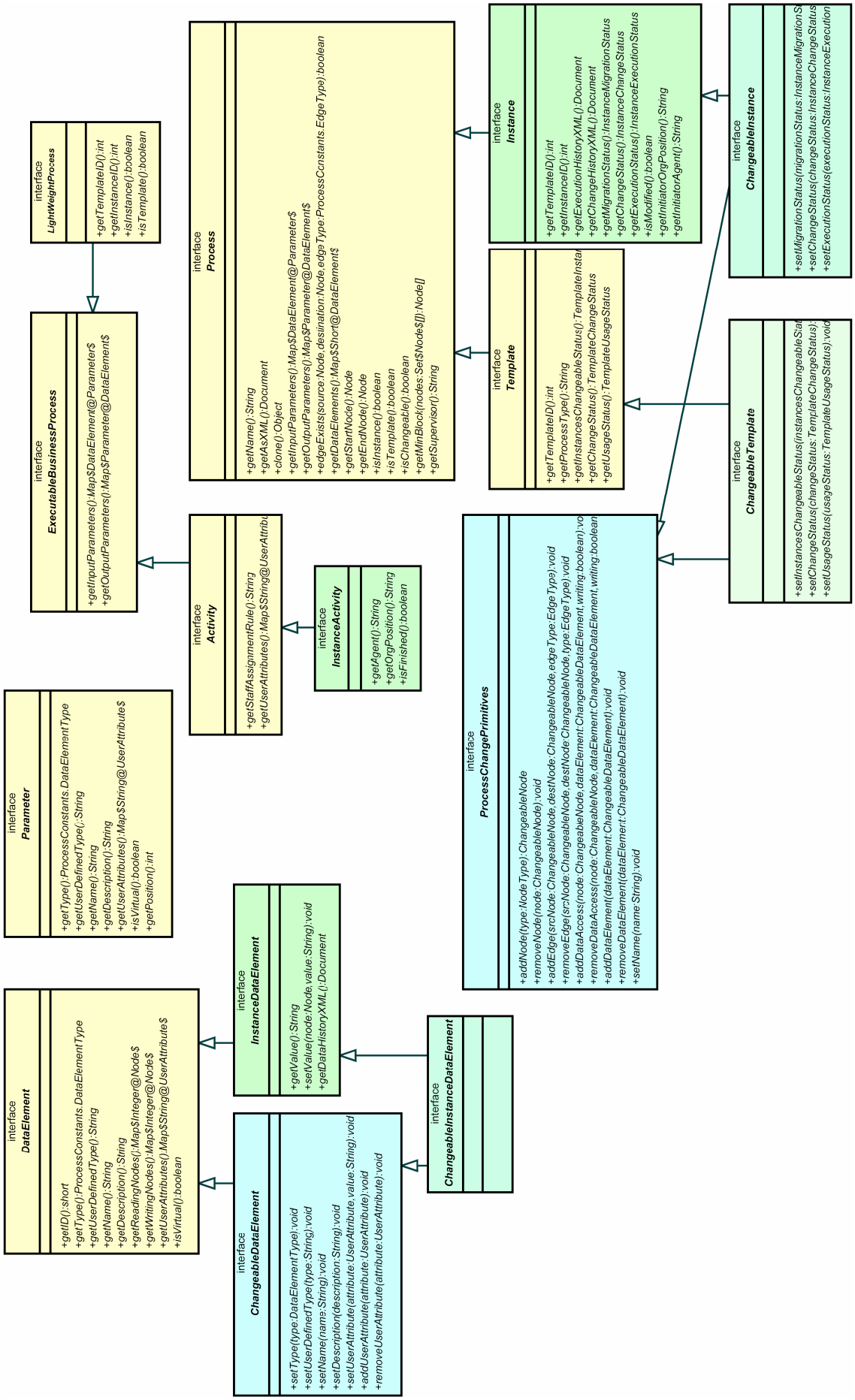


Abbildung 0.2 ADEPT2-Datenmodell (2)

Anhang C (Bereinigen der Deltaschicht)

Von den Änderungsprimitiven durchzuführende Überprüfungen im Rahmen der Erstellung einer korrekten automatisch bereinigten Deltaschicht:

addNode:
Prüfen, ob der neu einzufügende Knoten in *delNodes* enthalten ist. Falls dies so ist, wird er nicht in *newNodes* eingefügt, sondern der bestehende Eintrag in *delNodes* gelöscht.

removeNode:
Prüfen, ob der zu löschende Knoten in *newNodes* enthalten ist. Falls dies so ist, wird er nicht in *delNodes* eingefügt, sondern der bestehende Eintrag in *newNodes* gelöscht. Weiterhin gilt es zu prüfen, ob der zu löschende Knoten in *movedNodes* enthalten ist. Sollte dies der Fall sein, so wird dieser von dort entfernt.

addMovedNode:
Prüfen, ob der zu verschiebende Knoten in *newNodes* enthalten ist. Falls dies so ist, wird er nicht in *movedNodes* eingefügt, sondern der bestehende Eintrag in *newNodes* bleibt erhalten.

addEdge, *addDataAccess*:
Prüfen, ob die neu einzufügende (Daten-)Kante in *deletedEdges* enthalten ist. Falls dies so ist, wird sie nicht in *newEdges* eingefügt, sondern der bestehende Eintrag in *deletedEdges* gelöscht.

removeEdge, *removeDataAccess*:
Prüfen, ob die zu löschende (Daten-)Kante in *newEdges* enthalten ist. Falls dies so ist, wird sie nicht in *deleledEdges* eingefügt, sondern der bestehende Eintrag in *newEdges* gelöscht.

addDataElement:
Prüfen, ob das neu einzufügende Datenelement in *deletedDataElements* enthalten ist. Falls dies so ist, wird es nicht in *newDataElements* eingefügt, sondern der bestehende Eintrag in *deletedDataElements* gelöscht.

removeDataElement
Prüfen, ob das zu löschende Datenelement in *newDataElements* enthalten ist. Falls dies so ist, wird es nicht in *deletedDataElements* eingefügt, sondern der bestehende Eintrag in *newDataElements* gelöscht.

Anhang D (Algorithmen zur Klasseneinteilung)

Algorithmus für *disjoint*:

function *checkInstanceForContextDestroyingChanges*(DifferenceSets$_{S'}$, DifferenceSets$_I$) → exists

```
input
        DifferenceSets_S: the Difference Sets for schema S'
        DifferenceSets_I: the Difference Sets for instance-specific schema S_I
output
       exists: true, if context destroying changes are detected, otherwise false
        false
begin
        //explanation of used attributes:
        //DifferenceSets.XY: one special Difference Set of all Difference //Sets
        //syncE.src, syncE.dest: source respectively dest node of Sync-Edge //dataE.node,
        dataE.data: the node respectively the data element of a //data edge
        //nodeAttrChanged.node: the node whose attribute was changed
        //edgeAttrChanged.edge: the edge whose attribute was changed


        //initialize
        exists := false

        //check if the context of a newly inserted sync-edge from Δ_S is //destroyed by a delete
        or move operation from Δ_I.
        forall (syncE_S ∈ DifferenceSets_S.SyncE_ΔS^add) do
          if ((syncE_S.src OR syncE_S.dest) ∈ (DifferenceSets_I.N_ΔI^del OR
                                              DifferenceSets_I.N_ΔI^move)) then
              exists := true;
          fi
        od

        //check if the context of a newly inserted data-edge from Δ_S is //destroyed by a delete
        or move operation from Δ_I.
        forall (dataE_S ∈ DifferenceSets_S.DataE_ΔS^add) do
          if (((dataE_S.node ∈ (DifferenceSets_I.N_ΔI^del OR DifferenceSets_I.N_ΔI^move))
                        OR (dataE_S.data ∈ DifferenceSets_I.D_ΔI^del)) then
              exists := true;
          fi
        od

        //check if the node whose attribute is changed by an operation from Δ_S //is deleted by
        an operation from Δ_I.
        forall (nodeAttrChanged_S ∈ DifferenceSets_S.NodeAttrChanged_S) do
          if (nodeAttrChanged_S.node ∈ DifferenceSets_I.N_ΔI^del) then
              exists := true;
          fi
        od

        //check if the edge whose attribute is changed by an operation from Δ_S //is deleted by
        an operation from Δ_I.
        forall (edgeAttrChanged_S ∈ DifferenceSets_S.EdgeAttrChanged_S) do
          if (edgeAttrChanged_S.edge ∈ DifferenceSets_I.N_ΔI^del) then
              exists := true;
          fi
        od

        //do the same checks with reversed S and I
        ...

        return exists
```

```
end
```

Algorithmen zur Bestimmung von Ankergruppen

function calculateAnchorIns(Schema$_{S/I}$, $N_{\Delta S/I}^{add}$, $N_{\Delta S/I}^{move}$) → AnchorIns

```
input
        Schema: the schema on which the anchors are to be determined (S' or S_I)
        N_ΔS/I^add: set of newly inserted nodes for S' or S_I
        N_ΔS/I^move: set of moved nodes for S' or S_I
output
        AnchorIns: anchors of newly inserted nodes
begin
        //explanation of used functions:
        //node.getPredByEdgeType(CONTROL): returns all direct predecessors of node 'node'
        //reachable by edges of type CONTROL
        //node.getSuccByEdgeType(CONTROL): returns all direct successors of node 'node'
        //reachable by edges of type CONTROL

        AnchorIns := ∅
        forall (node ∈ N_ΔS/I^add) do
                //get all predecessors which were already contained in S
                preds := node.getPredByEdgeType(CONTROL)
                forall (left ∈ preds) do
                        if (left ∈ (N_ΔS/I^add OR N_ΔS/I^move))
                                //get predecessors of predecessor
                                preds := preds ∪ left.getPredByEdgeType(CONTROL)
                        else
                                leftAnchors := leftAnchors ∪ left
                        fi
                od

                //get all successors which were already contained in S
                succs := node.getSuccByEdgeType(CONTROL)
                forall (right ∈ succs) do
                        if (right ∈ (N_ΔS/I^add OR N_ΔS/I^move))
                                //get successors of successor
                                succs := succs ∪ right.getSuccByEdgeType(CONTROL)
                        else
                                rightAnchors := rightAnchors ∪ right
                        fi
                od

                //create the elements for AnchorIns
                forall (left ∈ leftAnchors) do
                        forall (right ∈ rightAnchors) do
                                AnchorIns := AnchorIns ∪ (left, node, right)
                        od
                od
        od
        return AnchorIns
end
```

function calculateAnchorMove(Schema$_{S/I}$, $N_{\Delta S/I}^{add}$, $N_{\Delta S/I}^{move}$) → AnchorMove

```
input
        s.o.
output
        AnchorMove: anchors of moved nodes
begin
        AnchorMove := ∅
        forall (node ∈ N_ΔS/I^move) do
                s.o.
```

```
            //create the elements for AnchorMove
            forall (left ∈ leftAnchors) do
                  forall (right ∈ rightAnchors) do
                        AnchorMove := AnchorMove ∪ (left, node, right)
                  od
            od
      od
      return AnchorMove
end
```

Anhang E (Algorithmen für biased disjoint Instanzen)

Deadlocks

checkForDeadlockCausingCycles(D_T, D_I, S) → **conflict**

```
input
       D_T: Delta-Layer of process scheme T
       D_I: Delta-Layer of instance I
       S: original process scheme
output
       conflict: true if conflict exists, otherwise false
begin
   //Initialization
   InsertedSyncEdgesOf_D_T := ∅
   InsertedSyncEdgesOf_D_I := ∅
   conflict := false

   //extract all inserted Sync-Edges of D_T
   forall((src, dest) ∈ D_T.newEdges) do
        //check if the current edge is a sync edge
        if (Type = SYNC) then
               if (src ∉ S.Nodes) then
               correspondingNodeInS := reduceGraph(S, D_T, src, 'pred')
               src_T := correspondingNodeInS
               fi
               if (dest ∉ S.Nodes) then
               correspondingNodeInS := reduceGraph(S, D_T, dest, 'succ')
               dest_T := correspondingNodeInS
               fi
               InsertedSyncEdgesOf_D_T = InsertedSyncEdgesOf_D_T ∪ (src_T, dest_T)
        fi
   od

   //extract all inserted Sync-Edges of D_S
   Analogous with input D_I.newEdges, reduceGraph(S, D_I, …) and InsertedSyncEdgesOf_D_I

   //check the conditions of Test 1
   forall ((src_I, dest_I) ∈ InsertedSyncEdgesOf_D_I) do
        forall ((src_T, dest_T) ∈ InsertedSyncEdgesOf_D_T) do
               if ((dest_I ∈ ({src_T} ∪ pred*(S, src_T))) AND
                              (dest_T ∈ {src_I} ∪ pred*(S, src_I)) then
                      conflict := true
               fi
        od
   od

   return conflict
end
```

Überlappende Kontrollblöcke

checkForOverlappingControlBlocks(D_T, D_I, S) → ***conflict***

```
input
       D_T: Delta-Layer of process scheme T
       D_I: Delta-Layer of instance I
       S: original process scheme
output
       conflict: true if conflict exists, otherwise false
begin
   //Initialization
   ContextOfInsertedControlBlocksOf_D_T := ∅
   ContextOfInsertedControlBlocksOf_D_I := ∅
```

```
   conflict := false

   //extract all inserted control-blocks of D_T
   forall((src, dest)∈ D_T.newEdges) do
        if (Type = LOOP ∨ Type = SPLIT_JOIN)) then
               //Note: the loop-edge is backward, therefore dest is the loop-start and src the
               //loop-end-node. As a result, the execution of the Graph Reduction Rules
               //calculates for loop-blocks the successor of the loop-start and the
               //predecessor of the loop-end-node. This is the desired behaviour.
               //get the predecessor in S by using Graph Reduction Rules
               pred_T := reduceGraph(S, D_T, src, 'pred')
               //get the successor in S by using Graph Reduction Rules
               succ_T := reduceGraph(S, D_T, dest, 'succ')

               ContextOfInsertedControlBlocksOf_D_T :=
                                   ContextOfInsertedControlBlocksOf_D_T ∪ (pred_T, succ_T)
        fi
   od

   //extract all inserted control-blocks of D_I
   Analogous with input D_I.newEdges, reduceGraph(S, D_I, …) and
   ContextOfInsertedControlBlocksOf_D_I

   //check the condition of Test 3
   forall ((pred_T, succ_T) ∈ ContextOfInsertedControlBlocksOf_D_T) do
        //get all nodes between pred_T and succ_T
        nodesBetweenPredAndSucc := getAllNodesBetweenPredAndSucc(S, pred_T, succ_T)
        forall ((pred_I, succ_I) ∈ ContextOfInsertedControlBlocksOf_D_I) do
               if ((pred_I ∈ nodesBetweenPredAndSucc AND succ_I ∉ nodesBetweenPredAndSucc) OR
                    (pred_I ∉ nodesBetweenPredAndSucc AND succ_I ∈ nodesBetweenPredAndSucc)) then

                      conflict := true
               fi
        od
   od

   return conflict
end
```

Sync-Kanten in oder aus Schleifenblöcken

checkForSyncEdgesCrossingLoopBoundaries(D_T, D_I, S) → conflict

```
input
       D_T: Delta-Layer of process scheme T
       D_I: Delta-Layer of instance I
       S: original process scheme
output
       conflict: true if conflict exists, otherwise false
begin
   //Initialization
   ContextOfInsertedLoopBlocksOf_D_T := ∅
   InsertedSyncEdgesOf_D_T := ∅
   ContextOfInsertedLoopBlocksOf_D_I := ∅
   InsertedSyncEdgesOf_D_I := ∅
   conflict := false

   //extract all inserted loop-blocks and sync-edges of D_T
   forall((src, dest)∈ D_T.newEdges) do
        if (Type = LOOP OR Type = SYNC)) then
               if (Type = LOOP) then
```

```
                    //get the successor of the loop-start-node in S by using Graph
                    //Reduction Rules. Note, the loop-edge is backward, therefore dest is
                    //the loop-start-node.
                    pred_T := reduceGraph(S, D_T, dest, 'succ')
                    //get the predecessor of the loop-end-node in S by using Graph Reduction
                    //Rules
                    succ_T := reduceGraph(S, D_T, src, 'pred')

            ContextOfInsertedLoopBlocksOf_D_T :=
                                    ContextOfInsertedLoopBlocksOf_D_T ∪ (pred_T, succ_T)

            else
                    if (src ∉ S.Nodes) then
                    correspondingNodeInS := reduceGraph(S, D_T, src, 'pred')
                    src_T := correspondingNodeInS
                    fi
                    if (dest ∉ S.Nodes) then
                    correspondingNodeInS := reduceGraph(S, D_T, dest, 'succ')
                    dest_T := correspondingNodeInS
                    fi
                    InsertedSyncEdgesOf_D_T = InsertedSyncEdgesOf_D_T ∪ (src_T, dest_T)

        fi
    od

    //extract all inserted loop-blocks and sync-edges of D_I
    Analogous with input D_I.newEdges, reduceGraph(S, D_I, …), ContextOfInsertedLoopBlocksOf_D_I
    and InsertedSyncEdgesOf_D_I

    //check the condition of Test 4
    forall ((pred_T, succ_T) ∈ ContextOfInsertedLoopBlocksOf_D_T) do
        //get all nodes between pred_T and succ_T
        nodesBetweenPredAndSucc := getAllNodesBetweenPredAndSucc(S, pred_T, succ_T)
        forall ((src_I, dest_I) ∈ InsertedSyncEdgesOf_D_I) do
                if ((src_I ∈ nodesBetweenPredAndSucc AND dest_I ∉ nodesBetweenPredAndSucc) OR
                     (src_I ∉ nodesBetweenPredAndSucc AND dest_I ∈ nodesBetweenPredAndSucc)) then

                       conflict := true
                       return conflict
                fi
        od
    od
    forall ((pred_I, succ_I) ∈ ContextOfInsertedLoopBlocksOf_D_I) do
        //get all nodes between pred_I and succ_I
        nodesBetweenPredAndSucc := getAllNodesBetweenPredAndSucc(S, pred_I, succ_I)
        forall ((src_T, dest_T) ∈ InsertedSyncEdgesOf_D_I) do
                if ((src_T ∈ nodesBetweenPredAndSucc AND dest_T ∉ nodesBetweenPredAndSucc) OR
                     (src_T ∉ nodesBetweenPredAndSucc AND dest_T ∈ nodesBetweenPredAndSucc)) then

                       conflict := true
                       return conflict
                fi
        od
    od

    return conflict
end
```

Graph Reduction Rules

reduceGraph(S, Delta, node, type) → correspondingNodeInS

input

```
        S: original process scheme
        Delta: Delta-Layer of process scheme S' or I
        node: node of Delta-Layer not contained in S
        type: specifies which anchor of node 'node' is wanted; values 'pred' or 'succ'
output
        correspondingNodeInS: the first predecessor or successor of node 'node' included in S
begin
        if (type = 'pred') then
                do
                        pred := {src | (src, node) ∈ Delta.newEdges}
                while (pred ∉ S)
                correspondingNodeInS := src
        fi
        if (type = 'succ') then
                do
                        succ := {dest | (src, dest) ∈ Delta.newEdges}
                while (succ ∉ S)
                correspondingNodeInS := succ
        fi
        return correspondingNodeInS
end
```

Anhang F (Algorithmen und Tests zu partially equivalent)

F.1 (Berechnung der Änderungsprojektionen aus der Deltaschicht)

Es ist zu beachten, dass die Elemente der erzeugten Änderungsprojektionen zusammengenommen nicht die komplette Deltaschicht ergeben. Bei Δ[ins_Node] werden z.B. nicht die Kanten mit aufgenommen die gelöscht werden mussten, um die Knoten einzufügen. Dies ist aber irrelevant, da zur Bestimmung der Klassenzugehörigkeit die in den Änderungsprojektionen gespeicherte Information ausreicht.

Auswirkungen neu eingefügter Knoten Δ[ins_Node]:
Enthält alle Knoten aus *newNodes* und alle Kanten aus *newEdges* vom Typ *CONTROL*, bei denen entweder *src* oder *dest* in *newNodes* enthalten ist.

Gelöschte Knoten Δ[del_Node]:
Enthält alle Knoten aus *delNodes* und alle Kanten aus *deletedEdges* vom Typ *CONTROL*, bei denen entweder *src* oder *dest* in *delNodes* enthalten ist.

Auswirkungen verschobener Knoten Δ[move_Node]:
Enthält alle Knoten aus *movedNodes* und alle Kanten aus *newEdges* und *deletedEdges* vom Typ *CONTROL*, bei denen entweder *src* oder *dest* in *movedNodes* enthalten ist.

Eingefügte Sync-Kanten Δ[ins_Sync]:
Enthält alle Kanten aus *newEdges* vom Typ *SYNC*.

Gelöschte Sync-Kanten Δ[del_Sync]:
Enthält alle Kanten aus *deletedEdges* vom Typ *SYNC*.

Datenflussänderungen Δ[data]:
Alle Kanten aus *newEdges* und *deletedEdges* vom Typ *READ* bzw. *WRITE* und alle Datenelemente aus *newDataElements* und *delDataElements*.

Attributänderungen Δ[attrChange]:
Alle Einträge aus den entsprechenden Datenstrukturen für Attribute.

F.2 (Konfliktbestimmung)

Es werden die *Difference Sets* verwendet. Bestimmung auf Basis der Deltaschicht analog möglich.

Different Order:
Ein solcher Konflikt lässt sich direkt bei der Klasseneinteilung erkennen. Um genau zu sein, bei der Untersuchung ob eine Instanz zur Klasse *subsumption equivalent* gehört. Bei dem dafür notwendigen Test müssen sowohl die Ankergruppen ($AnchorGroupsAgg(\Delta_S)$ und $AnchorGroupsAgg(\Delta_I)$)[16] als auch

[16] zur Erinnerung: $AnchorGroupsAgg(\Delta) = \{G = \{X_1, \ldots, X_n\} | \forall X_i, X_j \in G: \exists (left_i, X_i, right_i), (left_j, X_j, right_j) \in AnchorIns(\Delta) \cup AnchorMove(\Delta)$ mit $left_i = left_j \wedge right_i = right_j \ (i, j = 1, \ldots, n)\}$

die Reihenfolge (*OrderAgg*(Δ_S) und *OrderAgg*(Δ_I)) der in einer solchen Gruppe enthaltenen Knoten berechnet werden. Mit dieser Information wird dann für jede Gruppe (*leftAnchor*, *rightAnchor*) aus *AnchorGroupsAgg*(Δ) geprüft, ob die Reihenfolge in *OrderAgg*(Δ_S) der Reihenfolge in *OrderAgg*(Δ_I) entspricht. Ist dies für eine betrachtete Reihenfolgebeziehung nicht der Fall, so handelt es sich um einen *Different Order*-Konflikt und somit um eine Instanz der Klasse *partially equivalent*.

Different Anchor:
Auch dieser Konflikt kann bereits bei der Klasseneinteilung erkannt werden. Die notwendige Information erhält man aus den Mengen *AnchorIns*($\Delta_{S \text{ bzw. } I}$) und *AnchorMove*($\Delta_{S \text{ bzw. } I}$). Die Einträge in diesen Mengen haben die Form (*leftAnchor*, *node*, *rightAnchor*). Findet man nun in *AnchorIns*(Δ_S) bzw. *AnchorMove*(Δ_S) einen Knoten *node*, der einen anderen *left*- und oder *rightAnchor* besitzt, wie der gleiche Knoten in *AnchorIns*(Δ_I) bzw. *AnchorMove*(Δ_I), so hat man einen *Different Anchor*-Konflikt gefunden.

Conflicting Target Context:
Die notwendige Information für die Erkennung eines *Conflicting Target Context* erhält man aus den Mengen *AnchorIns*($\Delta_{S \text{ bzw. } I}$), *AnchorMove*($\Delta_{S \text{ bzw. } I}$), $N_{\Delta S \text{ bzw. } I}^{add}$ und $N_{\Delta S \text{ bzw. } I}^{move}$. Man geht folgendermaßen vor:

1. Bestimmung der *AnchorIns*(Δ_S) aller Knoten, die in $N_{\Delta S}^{add}$ nicht aber in $N_{\Delta I}^{add}$ vorhanden sind ($Anc_{\Delta S}^{conc\_context}[ins]$).
   Bestimmung der *AnchorIns*(Δ_I) aller Knoten, die in $N_{\Delta I}^{add}$ nicht aber in $N_{\Delta S}^{add}$ vorhanden sind ($Anc_{\Delta I}^{conc\_context}[ins]$).
   Bestimmung der *AnchorMove*(Δ_S) aller Knoten, die in $N_{\Delta S}^{move}$ nicht aber in $N_{\Delta I}^{move}$ vorhanden sind ($Anc_{\Delta S}^{conc\_context}$[move]).
   Bestimmung der *AnchorMove*(Δ_I) aller Knoten, die in $N_{\Delta I}^{move}$ nicht aber in $N_{\Delta S}^{move}$ vorhanden sind ($Anc_{\Delta I}^{conc\_context}$[move]).
   Form: (*leftAnchor*, *node*, *rightAnchor*)
2. Vergleich der berechneten Mengen. Dabei muss beachtet werden, dass nur die linken und rechten Ankerknoten (*left*- und *rightAnchor*) verglichen werden, nicht die eingefügten bzw. verschobenen Knoten (*node*):
   $(Anc_{\Delta S}^{conc\_context}[ins] \cup Anc_{\Delta S}^{conc\_context}[move]) \cap$
   $(Anc_{\Delta I}^{conc\_context}[ins] \cup Anc_{\Delta I}^{conc\_context}[move])$
   Findet sich ein übereinstimmender Anker, so hat man einen *Conflicting Target Context-Konflikt* gefunden.

Multiple Operations:
Knoten (einfügen, verschieben, löschen): Bestimme alle Knoten aus $N_{\Delta S}^{add}$, $N_{\Delta S}^{del}$ und $N_{\Delta S}^{move}$, die exakt die gleichen Kontextkanten sowohl in $CtrlE_{\Delta S}^{add}$ bzw. $CtrlE_{\Delta S}^{del}$ als auch in $CtrlE_{\Delta I}^{add}$ bzw. $CtrlE_{\Delta I}^{del}$ besitzen. Bei diesen Knoten tritt ein *Multiple Operations*-Konflikt auf.
Sync-Kanten (einfügen, löschen): Konflikt, wenn eine Kante aus $SyncE_{\Delta S}^{add}$ bzw. $SyncE_{\Delta S}^{del}$ mit einer Kante aus $SyncE_{\Delta I}^{add}$ bzw. $SyncE_{\Delta I}^{del}$ übereinstimmt.
Datenkante (einfügen, löschen): Konflikt, wenn eine Kante aus $DataE_{\Delta S}^{add}$ bzw. $DataE_{\Delta S}^{del}$ mit einer Kante aus $DataE_{\Delta I}^{add}$ bzw. $DataE_{\Delta I}^{del}$ übereinstimmt.

Datenelemente (einfügen, löschen): Bestimme alle Datenelemente aus $D_{\Delta S}^{add}$ und $D_{\Delta S}^{del}$, die exakt die gleichen Datenkanten sowohl in $DataE_{\Delta S}^{add}$ bzw. $DataE_{\Delta S}^{del}$ als auch in $DataE_{\Delta I}^{add}$ bzw. $DataE_{\Delta I}^{del}$ besitzen.
Attribute: Gleichheit direkt aus den Attributeinträgen

Context-Destroying Operations:
5 konfliktverursachende Fälle:

1. Kontext wird verschoben:
   Ein Knoten aus $N_{\Delta S}^{move}$ ist *leftAnchor* oder *rightAnchor* in *AnchorIns*(Δ_I) bzw. *AnchorMove*(Δ_I), *src* oder *dest* in *AnchorSync*(Δ_I)[17] oder *node* in *AnchorDataEdge*(Δ_I)[18]. Analog mit vertauschten Δ_S und Δ_I.
2. Kontext wird gelöscht:
   Ein Knoten aus $N_{\Delta S}^{del}$ ist *leftAnchor* oder *rightAnchor* in *AnchorIns*(Δ_I) bzw. *AnchorMove*(Δ_I), *src* oder *dest* in *AnchorSync*(Δ_I) oder *src* in *AnchorDataEdge*(Δ_I). Analog mit vertauschten Δ_S und Δ_I.
3. Aufhebung einer Knotenattributänderung:
   Ein Attribut eines Knotens aus $N_{\Delta S}^{del}$ wird in Δ_I geändert. Analog mit vertauschtem Δ_S und Δ_I.
4. Aufhebung einer Kantenattributänderung:
   Ein Attribut einer Kante aus $CtrlE_{\Delta S}^{del}$ wird in Δ_I geändert. Analog mit vertauschtem Δ_S und Δ_I.
5. Datenkontext wird gelöscht:
   Ein Datenelement aus $D_{\Delta S}^{del}$ ist *data* bei einer neu eingefügten Datenkante aus $DataE_{\Delta I}^{add}$.
   Analog mit vertauschtem Δ_S und Δ_I.

F.3 (Erweiterung zur zustandsbasierten Verträglichkeit)

In Abhängigkeit von der Konfliktklasse hinter den normalen zustandsbasierten Verträglichkeitstest (vgl. Abschnitt 7.5) nachzuschaltende Tests für *partially equivalent* Instanzen.

Different Order:
Zustandsbasierte Verträglichkeit wie bei *Conflicting Target Context*

Different Anchor:
Die Knoten aus $N_{\Delta I}^{add}$ und $N_{\Delta I}^{move}$, für die das Kriterium zutrifft, dürfen sich nicht im bereits durchlaufenen Teil von *I* befinden. Wäre dies der Fall, so ist ein solcher Knoten „fest“ und darf somit nicht mehr geändert werden. Die Schemaänderung, die – bei Auftreten eines solchen Konflikts – den gleichen Knoten einfügen oder verschieben will, kann folglich nicht mehr angewendet werden, ohne dass es zu einem inkorrekten instanzspezifischen Schema (im Falle einer Verschiebung) oder einer doppelten Einfügung kommt.
Der Test ist folgendermaßen durchzuführen. Man prüft für die im Konflikt stehenden Knoten aus $N_{\Delta I}^{add}$ und $N_{\Delta I}^{move}$ den Zustand in der instanzspezifischen Markierung M^I. Befindet sich ein solcher Knoten in einem anderen Zustand als *NOT_ACTIVATED* oder *ACTIVATED*, so ist die Instanz unverträglich.

[17] *AnchorIns*: Sync-Kante aus $SyncE_{\Delta}^{add}$ nach Anwendung von *GraphReductionRules*
[18] *AnchorDataEdge*: Datenkante (*node*, *data*) nach Anwendung der *Graph ReductionRules*

Die Situation, bei der die Schemaänderung eine Knoten in einen bereits durchlaufenen Teil der Instanz einfügen oder verschieben will, muss hier nicht zusätzlich behandelt werden, da dieser Fall bereits vom „normalen“ Verträglichkeitstest abgedeckt wird.

Conflicting Target Context:
Wird für einen bestimmten Anker (*leftAnchor*, *rightAnchor*) ein Konflikt entdeckt, so darf sich der direkte Nachfolger (*succ*) von *leftAnchor* im geänderten instanzspezifischen Schema maximal im Zustand *ACTIVATED* befinden. Dabei ist zu beachten, dass *succ* nicht zwangsweise in $Anc_{\Delta I}^{conc\_context}[ins]$ oder $Anc_{\Delta I}^{conc\_context}[move]$ enthalten sein muss. Das ist immer dann der Fall, wenn *succ* sowohl im geänderten Schema als auch im geänderten instanzspezifischen Schema direkter Nachfolger von *leftAnchor* ist. Da dadurch *succ* keinen Konflikt verursacht, ist er auch nicht in $Anc_{\Delta I}^{conc\_context}[ins]$ bzw. $Anc_{\Delta I}^{conc\_context}[move]$ enthalten.
succ lässt sich über $OrderAgg(\Delta_I)$ für den Anker (X, Y) finden, indem man denjenigen Knoten aus *OrderAgg* nimmt, der nur in vorderer Position eines Ankertupels vorkommt (X, _).

Multiple Operations:
Dieser Konflikt bedarf keines zusätzlichen Tests. Die zustandsbasierte Verträglichkeit wird bereits vom „normalen“ Test abgedeckt.

Context Destroying Operations:
Fall 1 und 2 (Kontext wird verschoben oder gelöscht):
Die Knoten aus $N_{\Delta S}^{move}$ und $N_{\Delta S}^{del}$ dürfen in M^I maximal den Zustand *ACTIVATED* besitzen (wird größtenteils vom „normalen“ Zustandstest abgedeckt; allerdings ist dort u.U. auch ein Verschieben bereits beendeter Knoten möglich, hier nicht).
Vom „normalen“ Zustandstest nicht erkannte Situation:
Auf Instanzebene löschen oder verschieben eines Knotens X, der auf Schemaebene als hinterer Kontextknoten (rechter Anker) eines neu eingefügten oder an diese Stelle verschobenen Knotens fungiert. Hier muss der Zustand des ersten in I vorhandenen Nachfolgers bzw. der Zustand des ersten Knotens nach dem verschobenen Knoten betrachtet werden.
Vorgehen: Suchen einer Kante (*src*, *dest*) in *delEdges* von I bei der X als *dest* vorkommt. Mit *src* in *newEdges* von I suchen. Zustand von *dest* aus *newEdges* prüfen. Zustand maximal *ACTIVATED*.
Sync-Kanten und Datenkanten: Befinden sich *src* und/oder *dest* einer Sync-Kante bzw. *node* einer Datenkante aus Δ_S in $N_{\Delta I}^{del}$ bzw. $N_{\Delta I}^{move}$, so ist dies nur zulässig, wenn sich *src* und *dest* aus $N_{\Delta I}^{del}$ bzw. $N_{\Delta I}^{move}$ nicht in einem bereits durchlaufenen Graphteil (von I) befinden. Prüfen über *delEdges* von I.

Fall 3 und 4 (Aufhebung einer Knoten- bzw. Kantenattributänderung):
Die zu betrachtenden, gelöschten Elemente aus $N_{\Delta I}^{del}$ bzw. $CtrlE_{\Delta I}^{del}$ dürfen sich nicht in einem bereits durchlaufenen Graphteil von I befinden. Anderenfalls ist die Schemaänderung, also das Ändern eines Knoten- oder Kantenattributs, nicht mehr zulässig.

Fall 5 (Datenkontext wird gelöscht): Zulässig, falls sich der Knoten *node* der Datenkante von $DataE_{\Delta I}^{add}$ in einem noch nicht durchlaufenen Graphteil von I befindet. Auch hier gilt wieder, dass die in konfliktstehenden Schemaänderungen nur dann zulässig sind, wenn sich die Instanzänderungen nicht in bereits durchlaufenen Graphteilen befinden.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den 15.03.2006

Martin Jurisch