Synchronisation von Dateibäumen mit Hilfe von Graphtransformationssystemen

Der Technischen Fakultät
der Universität Erlangen–Nürnberg
zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Roman Hodek

Erlangen — Juli 2000

Als Dissertation eingereicht bei
der Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung:	10. Juli 2000
Tag der Promotion:	9. Januar 2001
Dekan:	Prof. Dr. H. Meerkamm
Berichterstatter:	Prof. Dr. H.-J. Schneider
	Prof. Dr. U. Herzog

Zusammenfassung

Die vorliegende Arbeit stellt ein System zum Abgleich von Dateibäumen in einem Rechnernetz vor. Das Ändern, Löschen und Neuanlegen von Dateien wird erkannt und beim nächsten Synchronisationslauf auf alle anderen Rechner des Netzes übertragen. Dabei werden selbstverständlich auch unabhängige Änderungen auf verschiedenen Rechnern erkannt.

Das System kommt ohne zentrale Speicherung und ohne Referenzkopien der Dateien aus. Dadurch eignet es sich besonders für Anwendungen, bei denen manche Rechner keine dauerhafte oder nur eine unzureichende Netzwerkverbindung besitzen. Es ist sogar möglich, daß zwischen Teilen des Netzes gar keine Verbindung im üblichen Sinne besteht und der Austausch zwischen diesen Teilen nur über Rechner erfolgt, die wechselweise mit verschiedenen solchen Inseln verbunden werden. Es wurde außerdem darauf geachtet, daß möglichst wenig Verwaltungsdaten eigens für das Abgleichssystem gespeichert werden müssen.

Das Fundament für den Synchronisationsalgorithmus ist ein Graphtransformationssystem, das auf dem kategoriellen Ansatz beruht. Es werden daher zunächst die theoretischen Grundlagen dieses Verfahrens erörtert und darauf aufbauend Transformationen kategorieller Objekte eingeführt. Anschließend wird ein Zusammenhang zu regelbasierten Systemen hergestellt, mit deren Hilfe Terminierung und Konfluenz des vorgestellten Regelsystems gezeigt werden können. Danach wird erläutert, wie die Synchronisationen im Detail arbeiten und das verwendete Graphtransformationssystem besprochen. Abschließend wird noch auf weiterführende Fragestellungen eingegangen, wie Möglichkeiten zur automatischen Konfliktauflösung, das Wiedereinspielen von Backups, Initialisierung des Abgleichsystems und die praktische Umsetzung.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Verteilte Nutzung von Dateien

Es gibt offensichtlich unzählige Anwendungen dafür, von verschiedenen Rechnern aus auf den gleichen Satz von Dateien zugreifen zu können. In den meisten vernetzten Rechnerumgebungen wird diese Möglichkeit intensiv und so selbstverständlich genutzt, daß nicht tiefer darüber nachgedacht wird.

Es soll nun zunächst näher betrachtet werden, wie dieser verteilte Zugriff aussehen kann. Grundsätzlich muß man zwei alternative Ansätze dazu unterscheiden:

- Die verteilt genutzten Dateien werden nur einmal physikalisch gespeichert. Jeder Zugriff auf die Daten (sowohl lesend als auch schreibend) erfolgt über das Netzwerk. Ein prominentes Beispiel hierfür ist das Network File System (NFS).
- Die Dateien werden auf jedem Rechner, auf dem sie zur Verfügung stehen sollen, eigens physikalisch gespeichert. Änderungen finden zunächst lokal statt und werden von Zeit zu Zeit an die übrigen Rechner verteilt.

Der erste Ansatz ist der naheliegendere und auch der häufiger benutzte. Er gewährleistet inhärent die Konsistenz der Daten, da nur eine einzige physikalische Kopie vorliegt. Allerdings ist der Zugriff auf die Daten an das Netzwerk gebunden und ist ohne dieses nicht möglich. Bei einem Ausfall der Netzwerkverbindung ist weder lesender noch schreibender Zugriff auf die Dateien mehr möglich.

Falls diese Voraussetzung also nicht oder nur unzureichend gegeben ist, etwa bei einer nur zeitweiligen oder zu langsamen Kommunikationsverbindung der Rechner, wird man den zweiten Ansatz in Erwägung ziehen. Da die Dateien bei diesem lokal vorliegen, ist für den Zugriff selbst keine Netzverbindung erforderlich. Diese wird erst benötigt, wenn Änderungen an andere teilnehmende Rechner weitergereicht werden sollen. Dies kann zu einem beliebigen Zeitpunkt nach der Modifikation erfolgen. Auch muß der Benutzer nicht direkt auf die Beendigung der Datenübertragung warten, so daß die Anforderungen an die Geschwindigkeit der Verbindung deutlich niedriger sind.

Man sieht also, daß der zweite Ansatz mit lokaler Speicherung durchaus seine Berechtigung bei gewissen Anwendungen hat, auch wenn zentrale Speicherung zunächst die naheliegendere Methode ist. Allerdings fällt beim Vergleich auch sofort ein Nachteil ins Auge: Es ist möglich, daß die Kopien der gleichen Datei auf verschiedenen Rechnern inkonsistent geändert werden, ohne daß dies zunächst bemerkt wird. Später beim Abgleich fällt der Konflikt zwar auf, muß dann aber trotzdem in der einen oder anderen Weise aufgelöst werden.

Bei näherer Betrachtung bemerkt man allerdings, daß dieses Problem in abgewandelter Form auch bei der Methode des zentralen Zugriffs auftritt. Zum Bearbeiten einer Datei wird normalerweise eine Kopie von ihr im Hauptspeicher angelegt. Wenn mehrere Benutzer die gleiche Datei bearbeiten wollen, so können beim Zurückschreiben der Hauptspeicherkopie in das Dateisystem frühere Modifikationen anderer Benutzer überschrieben werden. Es gibt Möglichkeiten, solche Konflikte frühzeitig zu erkennen, wie zum Beispiel Dateisperren. Leider werden diese in der Praxis aber selten verwendet.

Insofern ist das Konsistenzproblem bei beiden Ansätzen vorhanden. Nur wird es bei lokaler Speicherung subjektiv meist als schwerwiegender beurteilt. Zum einen werden Inkonsistenzen zeitversetzt, dafür aber sicher erkannt. Dies vermittelt den Eindruck, das System des Dateiabgleichs würde zusätzlichen Aufwand verursachen, während bei gegenseitigem Überschreiben von Dateien die Ursache bei den Benutzern und der zufälligen zeitlichen Gleichheit der Bearbeitung zu liegen scheint.

Bei lokaler Speicherung von Kopien der Dateien gibt es mehrere Möglichkeiten, wie der Abgleich modifizierter Daten erfolgen kann:

- Einer der beteiligten Rechner verwaltet eine Referenzversion, die auf alle anderen Rechner verteilt wird. Änderungen müssen zuerst in die Referenzversion übernommen werden.

- Zur Verbesserung der Ausfallsicherheit und zur Lastverteilung in großen Systemen können auch mehrere Rechner mit Referenzkopien vorgesehen werden.
- Alle Rechner sind gleichberechtigt und es gibt keine Referenzkopien von Dateien.

Der Ansatz mit einer Referenzkopie ist leichter zu implementieren, da Vergleiche, ob eine Datei tatsächlich geändert wurde und ob ein Konflikt vorliegt, jeweils nur zwischen zwei Rechnern stattfinden müssen: dem abzugleichenden und dem mit der Referenzkopie. Bei gleichberechtigten Rechnern dagegen müssen alle Rechner in den Vergleich einbezogen werden.

Der zentrale Ansatz mit Referenzkopie weist jedoch ein Problem bezüglich der Ausfallsicherheit auf. Wenn der Referenzrechner ausfällt oder nicht erreichbar ist, bricht das gesamte Abgleichsystem zusammen. Man versucht dies zu kompensieren, indem mehrere Referenzrechner zur Verfügung gestellt werden. Dies bedingt jedoch eine weitere Abgleichsebene zwischen den Referenzrechnern und die Konsistenzproblematik wiederholt sich auf dieser Ebene. Konzeptionell ist dies also nicht grundsätzlich verschieden zu einer einzelnen Referenzkopie.

Dagegen verspricht der gleichberechtigte Ansatz soviel Toleranz gegenüber Rechner- und Netzwerkausfällen wie möglich. Wenn ein Rechner — aus welchen Gründen auch immer — nicht erreichbar ist, so stört dies den Rest des Systems in keiner Weise. Selbstverständlich kann eine modifizierte Datei auf einem solchen Rechner nicht auf die anderen übertragen werden, aber dies liegt außerhalb der Möglichkeiten jeglichen Abgleichsystems.

1.2 Zielsetzungen

In der vorliegenden Arbeit soll ein System zum Abgleich von Dateien vorgestellt werden, das auf lokalen Kopien basiert und auf Referenzkopien verzichtet. Diese Alternative bietet weitgehende Toleranz gegenüber Rechner- und Netzausfällen und eignet sich auch für Rechner, die nur gelegentlich an den Synchronisationen[1] teilnehmen, wie zum Beispiel mobile Computer, die bei Bedarf und an wechselnden Orten an das Netzwerk angeschlossen werden, oder solche, die nur zeitweise über Wählverbindungen Kontakt zu den anderen beteiligten Rechnern aufnehmen.

---

[1] Die Begriffe „Synchronisation“ und „Abgleich“ werden in diesem Zusammenhang synonym gebraucht.

Der Verzicht auf einen zentralen Referenzrechner ermöglicht darüber hinaus den Aufbau von Synchronisationsnetzen, die ausschließlich aus solchen nicht dauerhaft verbundenen Rechnern bestehen. Auch ist es möglich, daß nicht verbundene Gebiete des Synchronisationsnetzes existieren und der Austausch zwischen diesen nur über mobile Rechner oder ähnliches erfolgt, die wechselweise an verschiedene solcher Inseln angebunden werden.

Die Verwendung von Dateisperren oder ähnlichen Mechanismen zur Verhinderung von Konflikten kam nicht in Frage, da eine Sperre eine zentrale Instanz zu ihrer Verwaltung erfordern würde. Ohne diese könnte nicht sicher ermittelt werden, ob eine Sperrung einer Datei vorliegt, und es bestünde doch wieder die Möglichkeit von unabhängigen Änderungen. Und eine zentrale Instanz gefährdet die angestrebte Ausfalltoleranz und verhindert die Bildung von Netzinseln wie oben angesprochen.

Es wurde weiterhin das Ziel verfolgt, den Mechanismus des Dateiabgleichs auf eine fundierte theoretische Grundlage zu stellen. Viele der existierenden Synchronisationssysteme sind ad-hoc-Implementierungen, die zwar funktionieren und ihren Zweck erfüllen, deren Funktionsweise jedoch nie von der theoretischen Seite her betrachtet wurde.

Die Anforderungen an den Dateiabgleich selbst sind offensichtlich: Nach einer Synchronisation sollen die Dateibäume auf allen teilnehmenden Rechnern soweit wie möglich identisch sein. Dies ist ausnahmsweise nicht möglich, wenn unabhängige Modifikationen vorliegen und diese manuell zusammengeführt werden müssen. Die Identität der Dateibäume bezieht sich aber auch auf neu erstellte und gelöschte Dateien. Das heißt, neue Dateien innerhalb des Dateibaums sollen nach dem Abgleich auf allen anderen Rechnern vorhanden sein, und gelöschte Dateien sollen dort auch entfernt werden.

Ein weiteres Ziel für das in dieser Arbeit beschriebene System war es, möglichst wenig Informationen über die hinaus zu benutzen, die direkt vom Betriebssystem zur Verfügung gestellt werden. Bei Systemen mit einem zentralen Rechner mag es noch tragbar sein, dort umfangreiche Metadaten über die abzugleichenden Dateien zu speichern, da diese nur einmal vorliegen müssen. Aber bei gleichberechtigten Rechnern müßten solche Informationen auf allen Knoten vorhanden sein, denn es ist vorab nicht bekannt, welche Rechner an einem Synchronisationslauf teilnehmen werden. Daraus ergibt sich eine Redundanz jeglicher Metadaten, deren Speicherplatzbedarf bei größeren Synchronisationsnetzen zu stark ansteigen würde.

Aus diesen Gründen beschränkt sich das hier vorgestellte System im wesentlichen auf den vom Betriebssystem verwalteten Zeitstempel der letzten Modifikation. Es sind zwar darüber hinaus weitere Daten erforderlich, es wurde jedoch versucht, sie auf ein Minimum zu beschränken.

1.3 Warum mit Graphtransformationen?

Graphen bieten sich bei vielen Anwendungen zur klaren und übersichtlichen Darstellung von Sachverhalten an. Dies ist auch bei der hier vorliegenden Dateisynchronisation der Fall: Die zeitlichen Beziehungen der einzelnen Dateien zueinander und zur letzten Synchronisation lassen sich anschaulich in einem Graphen festhalten. Dieser kann dann durch Graphtransformationen in einen Aktionsgraphen umgeformt werden, der direkt die auszuführenden Aktionen anzeigt.

Diese graphische Darstellung der Beziehungen und der Aktionen ist wesentlich verständlicher als eine Herangehensweise, die beispielweise auf Matrizen beruhen würde. Diese Vorgehensweise ist selbstverständlich auch möglich, erfordert aber etliche komplizierte Umformungen, die sich graphisch fast von selbst aus der Anschauung ergeben.

Auch gibt es mit Graphtransformationssystemen ein theoretisch wohlfundiertes Konzept zur Umformung von Graphen, so daß diese Grundlagen nicht neu entwickelt werden mußten.

1.4 Vergleich mit anderen Systemen

Im folgenden soll der in dieser Arbeit verfolgte Ansatz mit einigen anderen Systemen zur verteilten Nutzung von Dateien verglichen werden. Da sehr viele solcher Systeme existieren, ist die Auswahl sicherlich unvollständig. Es wurden exemplarisch einige von ihnen herausgegriffen.

1.4.1 Andrew File System (AFS)

Das *Andrew File System* (AFS) [HKM$^+$88, Kaz88, ZE96] wurde am Massachusets Institute of Technology (MIT) im Rahmen des Athena-Projekts entwickelt. Es versteht sich als Weiterentwicklung des verbreiteten Network File System (NFS) von Sun Microsystems.

Im Grunde greift AFS genau wie NFS auf Dateien auf einem zentralen Rechner (Fileserver) zu. Allerdings legt es im Gegensatz zu NFS lokale Kopien häufig verwendeter Dateien an und erzielt so Geschwindigkeitsvorteile und kann Ausfälle des Servers bei nur lesenden Zugriffen überbrücken. Schreibzugriffe sind allerdings während solcher Ausfälle nicht möglich und werden mit einer Fehlermeldung abgebrochen.

Vom Standpunkt der Einteilung in Abschnitt 1.1 unterscheidet sich AFS nur wenig von NFS. Das lokale Zwischenspeichern von Dateien kann als Implementationsdetail angesehen werden. Es beschleunigt zwar den Zugriff, eine Netzwerkverbindung zu dem Server ist aber immer noch nötig, um die Aktualität der Daten sicherzustellen. Bei einem Ausfall kann zwar weiter lesend auf die lokale Kopie zugegriffen werden, aber mögliche Inkonsistenzen werden von vornherein durch ein Verbot des Veränderns verhindert. Ein späterer Abgleich modifizierter Daten ist nicht vorgesehen.

1.4.2 `rdist`

`rdist` [Coo92] ist ein bekanntes Werkzeug des Betriebssystems UNIX. Es ist als Hilfe für Systemadministratoren gedacht und implementiert den Ansatz mit lokalen Kopien und einem Referenzrechner.

Bei `rdist` werden die Dateien auf einem Referenzrechner gespeichert. In einer Konfigurationsdatei wird festgelegt, auf welche anderen Rechner diese Dateien verteilt werden sollen. Ein Aufruf des Programms `rdist`, der üblicherweise regelmäßig stattfindet, kopiert schließlich Dateien, die auf dem Referenzrechner geändert wurden, auf alle anderen teilnehmenden Rechner. Auch gelöschte Dateien und weitere Besonderheiten werden berücksichtigt.

Bei `rdist` dürfen Änderungen nur an den Dateien auf dem Referenzrechner vorgenommen werden. Modifiziert man eine der Dateien auf einem anderen Rechner, so wird diese Änderung beim nächsten `rdist`-Lauf überschrieben.

`rdist` eignet sich gut für Anwendungen, bei denen identische Dateibäume auf unterschiedlichen Rechnern benötigt werden, die Benutzung eines Netzwerkdateisystems (NFS o.ä.) aber nicht möglich oder zu ineffizient ist. Es erfüllt jedoch nicht die Anforderung, daß die gemeinsam verwendeten Dateien auf einem beliebigen der beteiligten Rechner verändert werden können und diese Änderungen automatisch an andere Rechner weitergegeben werden.

1.4.3 Concurrent Versions System (CVS)

Das *Concurrent Versions System* (CVS) [Ber90, BL90] ist ein Versionskontrollsystem, das auf dem *Revision Control System* (RCS) [HM85] aufbaut. Im Gegensatz zu diesem erfordert es keine Sperren auf Dateien, die bearbeitet werden sollen.

Die Dateien werden auf einem zentralen Referenzrechner in einem sogenannten *Repository* gespeichert. Jeder Benutzer hat seine eigene Kopie des Dateibaums, in der er nach Belieben Änderungen vornehmen kann. Er stößt dann schließlich ein Zurückschreiben modifizierter Versionen in das Repository an ("check-in"), bei dem ein Test stattfindet, ob sich dort bereits eine neuere Version befindet als die, von der die Änderungen ausgingen. In diesem Fall liegt dann ein Konflikt mit einer konkurrierenden Modifikation eines anderen Benutzers vor. Wenn sich dessen Änderungen nicht überschneiden, kann sie CVS teilweise automatisch in die Arbeitsversion des zurückschreibenden Benutzers übernehmen. Im übrigen muß dieser aber selbst für die Auflösung des Konflikts sorgen und eine korrigierte Fassung an das Repository liefern.

Obwohl CVS als Versionskontrollsystem eigentlich nicht Dateisynchronisation als Hauptaufgabe hat, läßt es sich doch aufgrund seines Verzichts auf explizite Sperren für diesen Zweck einsetzen. Die Übertragung modifizierter Dateien zum Referenzrechner und von dort zu den anderen Rechnern findet immer nur auf Anforderung statt, was sich aber bei Bedarf automatisieren läßt.

Allerdings findet sich auch hier ein zentraler Referenzrechner, von dem das Funktionieren des gesamten Systems abhängt. Auch speichert dieser Referenzrechner umfangreiche Metadaten zu den einzelnen Dateien, nämlich die gesamte Modifikationshistorie (was dem Sinn eines Versionskontrollsystems entspricht) und andere Begleitdaten wie Kommentare zu den Änderungen. Mit der Zeit kann die Menge der Metainformationen auf ein Vielfaches der eigentlich verwalteten Daten ansteigen. Dies kann störend sein, wenn man diese Informationen gar nicht benötigt, weil man nur an einem Abgleich der Dateien interessiert ist.

1.5 Übersicht

Bisher wurden in **Kapitel 1** die Grundideen und Zielsetzungen dieser Arbeit vorgestellt.

In **Kapitel 2** werden die theoretischen Grundlagen für Graphtransformationssysteme behandelt. Es werden Kategorien und etliche damit zusammenhängende

Begriffe und Konstruktionen einführt. Darauf basierend werden allgemeine Transformationssysteme für kategorielle Objekte definiert. Es werden einige Beispiele für Kategorien gegeben und insbesondere auf die später verwendeten Graphen eingegangen.

Kapitel 3 führt in die Theorie der Reduktionssysteme ein, mit deren Hilfe nachgewiesen werden kann, ob ein Transformationssystem wie das im folgenden Kapitel verwendete eindeutig terminiert.

Kapitel 4 beschreibt dann die Grundkonzepte, auf denen die Dateisynchronisation beruht und stellt das verwendete Graphtransformationssystem vor. Aus den vorgefundenen Beziehungen zwischen den Dateien wird ein Zustandsgraph aufgebaut, der in einen Aktionsgraphen umgeformt wird. Es wird auch gezeigt, daß das verwendete Transformationssystem terminiert und konfluent ist.

In **Kapitel 5** werden schließlich einige weitere Fragen behandelt, die im Zusammenhang der Dateisynchronisation auftreten, wie die Initialisierung des Systems vor dem ersten Gebrauch, dem Umgang mit Änderungskonflikten und dem Einspielen von Datensicherungen. Des weiteren wird kurz geschildert, wie das System in der Praxis umgesetzt werden kann.

Abschließend enthält **Kapitel 6** eine Zusammenfassung der vorgestellten Lösung und einen Ausblick auf mögliche Weiterentwicklungen.

Kapitel 2

Graphtransformationssysteme

2.1 Motivation

Graphen sind in der Informatik eine weit verbreitete Datenstruktur. Sie werden auf vielfältige Weise zur anschaulichen Darstellung von Verbindungen zwischen Objekten, Netzen irgendwelcher Art, von Abhängigkeiten etc. benutzt.

Aufgrund dieses weiten Einsatzfeldes kam schon früh der Wunsch auf, Graphen nach bestimmten Regeln verändern zu können. Es gab viele Ansätze zu diesem Thema [PR69a, EPS73, RM72], bei denen jedoch ein gemeinsames Grundprinzip erkennbar ist: Man sucht in dem Graph nach einem bestimmten Teilgraphen und ersetzt ihn durch einen neuen. Die Analogie zu Produktionen von Chomsky-Grammatiken ist offensichtlich: Es gibt eine linke Seite, die in einem größeren Zusammenhang enthalten ist, entfernt diesen Teil und ersetzt ihn nach bestimmten Regeln durch die rechte Seite der Produktion.

Bei Graphen kann man sich eine Regel darstellen als

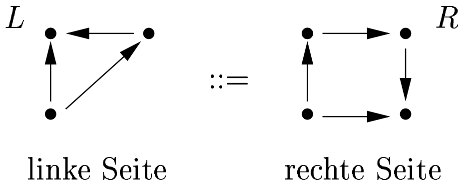


Abbildung 2.1: Eine einfache Graphersetzungsregel

Die Anwendung findet dann in folgenden Schritten statt:

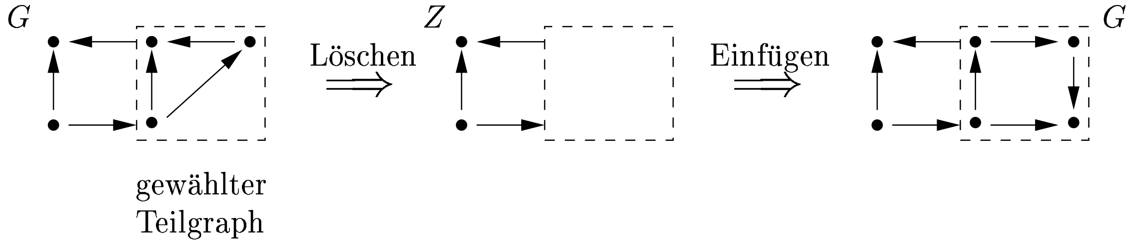


Abbildung 2.2: Anwendung der einfachen Ersetzungsregel

Zuerst wird in der rechten Hälfte des Graphen die linke Seite der Produktion gefunden. Dieser Bereich ist in der Abbildung durch das gestrichelte Rechteck markiert. Dann wird die linke Seite aus dem Graphen entfernt. Man bemerke, daß dabei mehrere nicht verbundene Kanten verbleiben, das Ergebnis dieses Zwischenschrittes ist also kein Graph. Anschließend wird die rechte Seite der Produktion eingefügt.

Aus dieser Beschreibung ergeben sich bereits mehrere Fragen: Soll es tatsächlich zugelassen werden, daß das Zwischenergebnis nicht der Definition eines Graphen entspricht? Und wie genau ist die rechte Seite einzufügen? Die bildliche Darstellung suggeriert zwar eine bestimmte Orientierung, diese ist jedoch nicht ohne weiteres gegeben. Genauso könnte die rechte Seite um 90 ° gedreht eingefügt werden. Und an welchen Stellen sollen unverbundene Kanten des Zwischenergebnisses mit der rechten Seite verknüpft werden?

Um all diese Probleme zu lösen, führt man einen *Klebegraphen* ein, der Verbindungsstellen zwischen der linken und rechten Seite einer Produktion definiert. Ein Klebegraph besteht meist nur aus Knoten, und es existiert eine Abbildung von ihm zu beiden Seiten der Produktion, die angibt, welche Knoten oder auch Kanten der beiden Seiten miteinander verknüpft sein sollen.

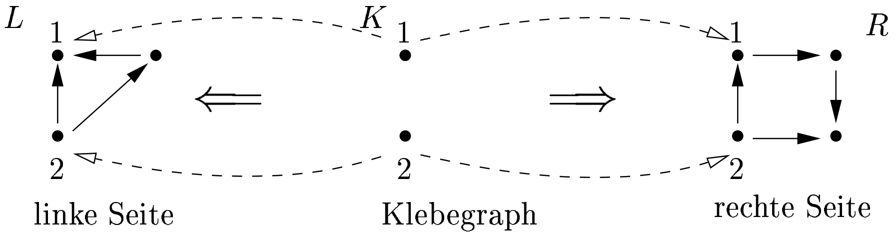


Abbildung 2.3: Ersetzungsregel mit Klebegraph

Die vom Klebegraphen ausgehenden Abbildungen werdem zum einen durch die gestrichelten Pfeile dargestellt und zum anderen durch die Zahlen 1 und 2 ausgedrückt, die eine Verbindung gleich numerierter Knoten in den verschiedenen

Graphen ausdrücken sollen. In Zukunft wird auf die Pfeile in derartigen Abbildungen verzichtet, um die Diagramme übersichtlicher zu halten.

Bei der Anwendung einer solchen Produktion mit einem Klebegraphen werden nur die Teile der linken Seite gelöscht, die nicht mit dem Klebegraphen verknüpft sind. Auf diese Weise entfällt auch die Notwendigkeit, unverbundene Kanten im Zwischenergebnis zu akzeptieren. Im Gegenteil, wenn dort nicht verbundene Kanten verboten werden, kann durch den Klebegraphen festgelegt werden, ob weitere Kanten außer denen der linken Seite zu in der Produktion auftauchenden Knoten existieren dürfen. Die Klebestellen und ihre Abbildung erlauben auch die Auswahl, welche Teile der linken und der rechten Seite als „gleich“ betrachtet werden sollen. Zusammenfassend gibt der Klebegraph also die Stellen der Ersetzung an, die mit dem Rest des einbettenden Graphen verbunden sind und bleiben (daher auch die Namensgebung).

Die Anwendung der neuen Produktion mit Klebegraphen auf das Beispiel sieht dann wie folgt aus:

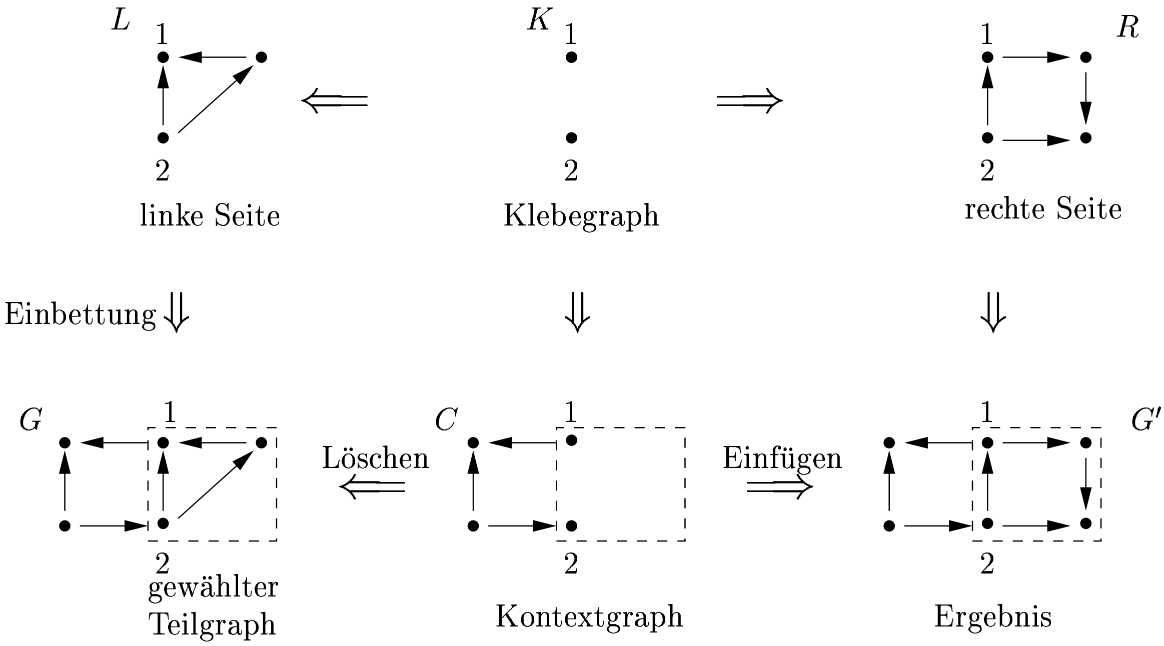


Abbildung 2.4: Anwendung der Ersetzungsregel mit Klebegraph

Der erste Schritt besteht wiederum darin, die linke Seite in dem Ausgangsgraphen G zu finden. Dann wird ein Kontextgraph C konstruiert, in dem die linke Seite mit Ausnahme der Klebestellen entfernt wurde. Anschließend wird die rechte Seite eingefügt, wobei die Klebestellen aufeinandergelegt werden.

Interessant ist an dieser Stelle ein Vergleich mit Chomsky-Grammatiken, die auf Zeichenketten arbeiten. Diese ersetzen auch eine linke durch eine rechte Seite, nur benötigen sie dazu kein Äquivalent zu einem Klebegraphen. Die Antwort liegt in der Tatsache, daß Zeichenketten sozusagen eindimensionale Strukturen sind. Es gibt gar keine andere sinnvolle Möglichkeit, die Teilstücke anders als an ihren beiden Enden zu verkleben (Abb. 2.5).

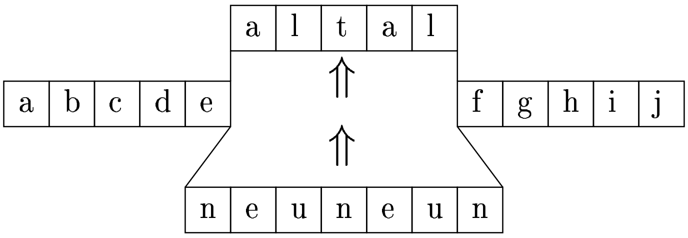


Abbildung 2.5: Ersetzung bei Zeichenketten

Man kann eine Analogie zu einem Graphen herstellen, indem man jedes Zeichen durch einen Knoten repräsentiert und (außer beim letzten) eine Kante von jedem Knoten auf den Knoten seines Folgezeichens einführt. Eine Chomsky-Produktion entspricht dann der Ersetzung einer solchen Knotenkette durch eine andere. Man kann beliebige Knoten und Kanten in den Klebegraphen aufnehmen, würde damit aber nur eine Mindestlänge der rechten Seite festlegen. Im Normalfall ist dies jedoch nicht gewünscht und auch nicht sinnvoll. Andererseits müssen die beiden Endpunkte Klebeknoten sein, denn sonst wären die Kanten, die zu dem zu ersetzenden Teilstück hinführen bzw. aus ihm herauskommen, im Kontext nicht verbunden. Auch eine andere Abbildung der Klebeknoten als auf jeweils den Anfang bzw. das Ende der Teilstücke ist unsinnig, da das Ergebnis nach der Ersetzung keiner gültigen Zeichenkette entsprechen würde.

Der Klebegraph kann also als implizit festgelegt angesehen werden und muß daher nicht gesondert angegeben werden. Aus diesem Grund erscheint die Erfordernis eines Klebegraphen bei Graphoperationen zunächst fremd und ungewohnt, vor allem wenn man versucht, Analogien zu Zeichenkettenoperationen herzustellen. Bei näherer Betrachtung jedoch ist es ein Sonderfall, daß man die Angabe, wo alte und neue Teile mit dem Rest verknüpft werden sollen, vernachlässigen kann.

2.2 Kategorien

Die Kategorientheorie beschäftigt sich mit abstrakten *Objekten*, die ebenso abstrakt durch sogenannte *Morphismen* miteinander in Beziehung stehen. Es werden keine weiteren Annahmen über das Wesen oder die Interpretation von Objekten und Morphismen gemacht. Alle Definitionen und Sätze bewegen sich nur auf deren Ebene. Durch diesen hohen Abstraktionsgrad können die Erkenntnisse der Kategorientheorie auf viele Anwendungsfelder in der Informatik übertragen werden, wie zum Beispiel Automatentheorie, Typtheorie, Design von Programmiersprachen usw. Die Kategorientheorie dient aber auch als Grundlage für Graphtransformationssysteme. Mit ihrer Hilfe kann exakt definiert werden, wie die informell im letzten Abschnitt eingeführten Transformationsregeln arbeiten.

2.2.1 Grundbegriffe

Definition 2.1: (Kategorie)
Eine Kategorie $\mathcal{C}$ ist ein Sextupel $(Obj, Mor, dom, codom, \cdot, id)$ mit:

- Obj ist eine Klasse von *Objekten*
- Mor ist eine Klasse von *Morphismen*
- $dom : Mor \rightarrow Obj$ gibt das Objekt an, von dem ein Morphismus ausgeht ("domain")
- $codom : Mor \rightarrow Obj$ gibt das Objekt an, zu dem ein Morphismus hinführt ("codomain")
- $\cdot : Mor \times Mor \rightarrow Mor$ ist die *Kompositionsoperation* auf Morphismen. Sie ordnet einem Paar $f, g \in Mor$ mit $codom(f) = dom(g)$ einen Morphismus $g \cdot f \in Mor$ zu, der die Bedingung

  $$dom(g \cdot f) = dom(f) \quad \wedge \quad codom(g \cdot f) = codom(g)$$

  erfüllt.

  Außerdem wird von der Komposition $\cdot$ verlangt, daß sie *assoziativ* ist, d.h.

  $$h \cdot (g \cdot f) = (h \cdot g) \cdot f$$

- $id : Obj \rightarrow Mor$ gibt für jedes Objekt $o \in Obj$ einen *Identitätsmorphismus* id_o an, der

  $$dom(id_o) = codom(id_o) = o$$

  erfüllt. Die Identitäten sind neutrale Elemente der Komposition, d.h. wenn $dom(f) = a$ und $codom(f) = b$, so gilt:

  $$f \cdot id_a = f \quad \wedge \quad id_b \cdot f = f$$

◇

Objekte und Morphismen werden gerne in Diagrammen dargestellt, in denen die Knoten für Objekte stehen und Pfeile für Morphismen. Zum Beispiel würde man einen Morphismus f mit $dom(f) = a$ und $codom(f) = b$, $a, b \in Obj$ als

$$a \xrightarrow{f} b$$

veranschaulichen. Eine andere gebräuchliche Schreibweise ist — angelehnt an Funktionen — $f : a \to b$.

Gerne verwendet werden auch sog. kommutierende Diagramme, in denen das = in der Mitte bedeuten soll, daß beide Wege kommutieren, d.h. $f \cdot g = h \cdot k$

$$\begin{tikzcd}
\bullet \arrow[r, "g"] \arrow[d, "k"'] \arrow[dr, phantom, "="] & \bullet \arrow[d, "f"] \\
\bullet \arrow[r, "h"'] & \bullet
\end{tikzcd}$$

Es gibt einige Morphismen mit besonderen Eigenschaften:

Definition 2.2: (Epi-, Mono- und Isomorphismen)

- Ein Morphismus $f : a \to b$ heißt *Epimorphismus* genau dann, wenn gilt

$$\forall g, h : b \to c : \quad g \cdot f = h \cdot f \Rightarrow g = h$$

- Ein Morphismus $f : b \to a$ heißt *Monomorphismus* genau dann, wenn gilt

$$\forall g, h : c \to b : \quad f \cdot g = f \cdot h \Rightarrow g = h$$

- Ein Morphismus $f : a \to b$ heißt *Isomorphismus* genau dann, wenn es einen Morphismus $g : b \to a$ gibt mit der Eigenschaft

$$f \cdot g = id_b \quad \wedge \quad g \cdot f = id_a$$

$\diamond$

Zu Isomorphismen existieren also Umkehrmorphismen. Es läßt sich aber leicht zeigen, daß die Umkehrung auch eindeutig ist. Angenommen es gäbe ein g' mit den gleichen Eigenschaften wie g aus der Definition des Isomorphismus. Dann gilt:

$$g' = g' \cdot id_b = g' \cdot (f \cdot g) = (g' \cdot f) \cdot g = id_a \cdot g = g$$

Bei Isomorphismen ist also die Schreibweise f^{-1} für den Umkehrmorphismus sinnvoll.

Viele Eindeutigkeitsaussagen der Kategorientheorie gelten nur „bis auf Isomorphie", da Objekte, zwischen denen Isomorphismen existieren, in vielen Fällen identische Eigenschaften bezüglich Kommutativiät und Epi-/Monomorphismen besitzen.

Es läßt sich einfach nachweisen, daß ein Isomorphismus f auch Mono- und Epimorphismus ist. Angenommen es gilt $f \cdot g = f \cdot h$:

$$\begin{array}{rcll} f \cdot g & = & f \cdot h & \Rightarrow \\ f^{-1} \cdot f \cdot g & = & f^{-1} \cdot f \cdot h & \Rightarrow \\ id_{dom(f)} \cdot g & = & id_{dom(f)} \cdot h & \Rightarrow \\ g & = & h & \Rightarrow \\ & & f \text{ ist Monomorphismus} & \end{array}$$

Der gleiche Beweis läßt sich analog für Epimorphismus durchführen:

$$\begin{array}{rcll} g \cdot f & = & h \cdot f & \Rightarrow \\ g \cdot f \cdot f^{-1} & = & h \cdot f \cdot f^{-1} & \Rightarrow \\ g \cdot id_{codom(f)} & = & h \cdot id_{codom(f)} & \Rightarrow \\ g & = & h & \Rightarrow \\ & & f \text{ ist Epimorphismus} & \end{array}$$

An diesen beiden sehr ähnlichen Beweisen läßt sich ein grundlegendes Prinzip der Kategorientheorie erkennen: das *Dualitätsprinzip*. Es besagt, daß man durch Vertauschen von *dom* und *codom* aller Morphismen duale Definitionen (co-Definitionen) und Sätze gewinnen kann. In der graphischen Darstellung entspricht dies dem Umdrehen aller Pfeile. Verdeutlicht sei dies an der Definition von Epi- und Monomorphismus, die duale Begriffe sind (Abb. 2.6).

Der Begriff des Isomorphismus ist selbstdual, d.h. nach dem Umdrehen aller Pfeile erhält man die gleiche Definition. Die beiden obigen kurzen Beweise waren ebenfalls dual, daher ihre Ähnlichkeit. Im folgenden wird zur Abkürzung öfters auf das Dualitätsprinzip verwiesen werden.

Eine weitere Klasse von besonderen Morphismen sind die Retraktionen und Coretraktionen:

Definition 2.3: (Retraktion, Coretraktion)
In einer Kategorie $\mathcal{C}$ heißt ein Morphismus $f : a \to b$ Retraktion, wenn es ein $g : b \to a$ gibt mit $f \cdot g = id_b$.

$$\begin{tikzcd}[column sep=small] f \text{ Epimorphismus wenn} & a \arrow[r, "f"] \arrow[dr, "f"'] & b \arrow[r, "g"] & c & \Longrightarrow g = h \\ & & b \arrow[ur, "h"'] & & \end{tikzcd}$$

$$\begin{tikzcd} & b \arrow[dl, "f"'] & \\ a & = & c \arrow[ul, "g"'] \arrow[dl, "h"] \\ & b \arrow[ul, "f"] & \end{tikzcd}$$

Abbildung 2.6: Duale Definitionen von Epi- und Monomorphismus

> Dual dazu ist $f : b \to a$ eine Coretraktion, wenn ein $g : a \to b$ existiert mit $f \cdot g = id_b$. ◇

Eine Coretraktion ist also ein umkehrbarer Morphismus, einer von dem ein Weg zurück zu $dom(f)$ existiert. Diese Umkehrung muß nicht eindeutig sein, sie muß nur existieren. Dual dazu sind Retraktionen die Umkehrung zu mindestens einem anderen Morphismus. Es ist trivial zu sehen, daß jeder Isomorphismus sowohl Retraktion als auch Coretraktion ist. Aber es gilt auch die Umkehrung:

$$\begin{array}{lcl} f \text{ Retraktion} & \Rightarrow & \exists g : f \cdot g = id_b \\ f \text{ Coretraktion} & \Rightarrow & \exists g' : g' \cdot f = id_a \end{array}$$

$$g = id_a \cdot g = (g' \cdot f) \cdot g = g' \cdot (f \cdot g) = g' \cdot id_b = g'$$

(Es genügt sogar, wenn f Retraktion und Monomorphismus ist.)

Auch kann man zeigen, daß jede Coretraktion Monomorphismus ist: Sei $f : a \to b$ Coretraktion, d.h. $\exists f' : b \to a : f' \cdot f = id_a$.

$$f \cdot g = f \cdot h \quad \Rightarrow \quad f' \cdot f \cdot g = f' \cdot f \cdot h \quad \Rightarrow \quad id_a \cdot g = id_a \cdot h \quad \Rightarrow \quad g = h$$

Dual dazu ist also auch jede Retraktion ein Epimorphismus. Zusammenfassend kann man eine Hierarchie von ausgezeichneten Morphismen wie in Abbildung 2.7 entwerfen.

2.2.2 Konstruktionen

In der Kategorientheorie lassen sich auch Konstruktionen ausführen. Das bedeutet, daß bestimmte Objekte oder Morphismen durch Bedingungen auf anderen

$$\begin{tikzcd}
 & f \text{ Isomorphismus} \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & \\
f \text{ Coretraktion} \arrow[r, Rightarrow] \arrow[dd, Rightarrow] & \wedge \arrow[u, Rightarrow] & f \text{ Retraktion} \arrow[l, Rightarrow] \arrow[dd, Rightarrow] \\
 & & \\
f \text{ Monomorphismus} & & f \text{ Epimorphismus}
\end{tikzcd}$$

Abbildung 2.7: Hierarchie der Morphismen

Objekten/Morphismen beschrieben und so definiert werden. Auch hier wird die den Kategorien eigene Abstraktion erhalten und diese Bedingungen können nur auf der Ebene der Beziehungen zwischen Objekten und Morphismen formuliert werden. Sie können sich also nur auf die Existenz und/oder Gleichheit von Objekten/Morphismen stützen.

Die wichtigsten Konstruktionsschemata sind die der *Limites* und *Colimites*. Auch hier taucht wieder das Dualitätsprinzip auf und es gibt zu jeder Limeskonstruktion einen dualen Colimes. Für die späteren Anwendungen sind die Colimites weit wichtiger als die Limites, so daß diese zuerst eingeführt werden.

Die Basis jeder Colimes-Konstruktion ist ein Diagramm mit Objekten O_i und Morphismen. Von diesen ausgehend betrachtet man alle Objekte T_j, die von den O_i mit Morphismen erreicht werden können. Wenn es einen Morphismus zwischen zwei Objekten O_i und O_j in dem ursprünglichen Diagramms gab, so muß das resultierende Dreieck kommutativ sein:

$$\begin{tikzcd}
O_1 \arrow[rr] \arrow[dr] & {=} & O_2 \arrow[dl] \\
 & T_j &
\end{tikzcd}$$

Das „minimale“ Objekt T_0, das auf diese Weise von allen O_i erreicht werden kann, ist dann der Colimes zu dem Diagramm. „Minimal“ heißt in diesem Zusammenhang, daß alle anderen Objekte, die die gleiche Eigenschaft wie für T_0 beschrieben haben, von T_0 aus durch einen eindeutigen Morphismus u erreicht werden können:

$$\begin{tikzcd}
O_1 \arrow[rr] \arrow[dr] \arrow[ddr] & {} \arrow[d, phantom, "="] & O_2 \arrow[dl] \arrow[ddl] \\
{} \arrow[r, phantom, "="] & T_0 \arrow[d, "u"] & {} \arrow[l, phantom, "="] \\
& T_1 &
\end{tikzcd}$$

Doch nun zu einigen konkreten Beispielen für Colimites. Das einfachste Basisdiagramm ist ein leeres Diagramm. Da keine Ausgangsobjekte vorhanden sind, ist nur die letzte Bedingung zu erfüllen, daß zu jedem anderen Objekt ein eindeutiger Morphismus existiert. Dies führt zu folgender Definition:

Definition 2.4: (Initiales Objekt)
Ein Objekt $a \in Obj$ in einer Kategorie $\mathcal{C}$ heißt initiales Objekt, wenn es zu jedem $b \in Obj$ einen eindeutigen Morphismus $f : a \rightarrow b$ gibt. Das initiale Objekt ist bis auf Isomorphie eindeutig. $\diamond$

Der duale Begriff dazu ist das terminale Objekt:

Definition 2.5: (Terminales Objekt)
Ein Objekt $a \in Obj$ in einer Kategorie $\mathcal{C}$ heißt terminales Objekt, wenn es von jedem $b \in Obj$ einen eindeutigen Morphismus $f : b \rightarrow a$ gibt. Das terminale Objekt ist ebenfalls bis auf Isomorphie eindeutig. $\diamond$

Wenn das Basisdiagramm nur aus zwei Objekten besteht, erhält man die Definition des Coproduktes:

Definition 2.6: (Coprodukt)
Seien in einer Kategorie $\mathcal{C}$ $a, b \in Obj$. Ein Objekt $c \in Obj$ heißt Coprodukt zu a und b genau dann, wenn es Morphismen $f : a \rightarrow c$ und $g : b \rightarrow c$ gibt und wenn es für alle anderen Objekte $c' \in Obj$, zu denen Morphismen $f' : a \rightarrow c$ und $g' : b \rightarrow c$ existieren, einen eindeutigen Morphismus u mit der Eigenschaft $f' = u \cdot f$ und $g' = u \cdot g$ gibt. $\diamond$

Im Diagramm läßt sich das Coprodukt folgendermaßen darstellen:

$$\begin{tikzcd}
a \arrow[r, "f"] \arrow[dr, "f'"'] \arrow[r, phantom, "=", shift right=4ex] & c \arrow[d, "u"] \arrow[r, phantom, "=", shift right=4ex] & b \arrow[l, "g"'] \arrow[dl, "g'"] \\
& c' &
\end{tikzcd}$$

Abbildung 2.8: Definition des Coproduktes

Auch Coprodukte sind eindeutig bis auf Isomorphie. Man sagt auch, eine Kategorie besitzt Coprodukte, wenn zu jedem Paar von Objekten ein Coprodukt existiert.

Man kann auch von zwei Objekten und einem Morphismus ausgehen und den anderen Morphismus zur Vervollständigung eines Coprodukts suchen:

Definition 2.7: (Coproduktkomplement)
Ist in einer Kategorie $\mathcal{C}$ $f : a \to c$ gegeben, so heißt ein Morphismus $g : b \to c$ Coproduktkomplement genau dann, wenn $f : a \to c, g : b \to c$ Coprodukt zu a, b ist. ◇

(Bemerkung: Das Coproduktkomplement muß nicht eindeutig sein.)

Während das Coprodukt von zwei Objekten ausgeht, erhält man mit zwei Morphismen als Basis den Coegalisator:

Definition 2.8: (Coegalisator)
Sind in einer Kategorie $\mathcal{C}$ zwei Morphismen $f, g : a \to b$ gegeben, so heißt ein Morphismus $h : b \to c$ Coegalisator zu f, g, wenn $h \cdot f = h \cdot g$ und wenn es für alle anderen $h' : b \to c'$ mit dieser Eigenschaft ($h' \cdot f = h' \cdot g$) ein eindeutiges $u : c \to c'$ mit $h' = u \cdot h$ gibt. ◇

$$\begin{tikzcd}
a \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & b \arrow[r, "h"] \arrow[dr, "h'"'] & c \arrow[d, "u"] \arrow[dl, phantom, "=", shift right=2ex] \\
& & c'
\end{tikzcd}$$

Abbildung 2.9: Definition des Coegalisators

Coegalisatoren sind ebenfalls bis auf Isomorphie eindeutig und analog zu Coproduken spricht man davon, daß eine Kategorie Coegalisatoren besitzt, wenn es zu jedem Paar von Morphismen mit gleichem Domain und Codomain einen Coegalisator gibt.

Die nun folgende Definition des Pushouts ist die Basis der Transformationsregeln, die auf der Basis der Kategorien aufgebaut werden. Das Ausgangsdiagramm für das Pushout besteht aus drei Objekten und zwei Morphismen zwischen diesen.

Definition 2.9: (Pushout)
Seien in einer Kategorie $\mathcal{C}$ zwei Morphismen $f : a \rightarrow b$ und $g : a \rightarrow c$ gegeben. Das Objekt $d \in Obj$ und die zwei Morphismen $p : c \rightarrow d$ und $q : b \rightarrow d$ heißen Pushout zu f, g, wenn $p \cdot g = q \cdot f$ und für alle anderen $p' : c \rightarrow d'$ und $q' : b \rightarrow d'$ mit dieser Eigenschaft $(p' \cdot g = q' \cdot f)$ ein eindeutiger Morphismus u mit $p' = u \cdot p$ und $q' = u \cdot q$ existiert. ◇

$$\begin{tikzcd}
a \arrow[r, "f"] \arrow[d, "g"'] \arrow[rd, phantom, "="] & b \arrow[d, "q"] \arrow[rdd, "q'"] & \\
c \arrow[r, "p"] \arrow[rrd, "p'"'] \arrow[rd, phantom, "="] & d \arrow[rd, "u"] \arrow[r, phantom, "="] & {} \\
 & {} & d'
\end{tikzcd}$$

Abbildung 2.10: Definition des Pushouts

Auch Pushouts sind eindeutig bis auf Isomorphie und man sagt, eine Kategorie besitze Pushouts, wenn zu allen Paaren von Morphismen mit gleichem Domain ein Pushout existiert. Ein Diagramm, das ein Pushout ist, wird mit einem PO in der Mitte notiert:

$$\begin{tikzcd}
a \arrow[r, "f"] \arrow[d, "g"'] \arrow[rd, phantom, "\text{PO}"] & b \arrow[d, "q"] \\
c \arrow[r, "p"] & d
\end{tikzcd}$$

Das Pushout kann mit Hilfe von Coprodukt und Coegalisator konstruiert werden. Zuerst konstruiert man das Coprodukt i_1, i_2 zu b und c. Dann faßt man $j_1 = i_1 \cdot g$ und $j_2 = i_2 \cdot f$ zusammen und bildet den Coegalisator h zu j_1 und j_2. Die gesuchten Morphismen p und q des Pushouts sind dann $p = h \cdot i_1$ und $q = h \cdot i_2$.

$$\begin{tikzcd}
a \arrow[rr, "f"] \arrow[dd, "g"'] \arrow[dr, shift right, "j_1"'] \arrow[dr, shift left, "j_2"] & & b \arrow[dd, "q"] \arrow[dl, "i_2"'] \\
& e \arrow[dr, "h"] & \\
c \arrow[rr, "p"'] \arrow[ur, "i_1"] & & d
\end{tikzcd}$$

Abbildung 2.11: Konstruktion des Pushouts aus Coprodukt und Coegalisator

Wie zum Coprodukt kann auch zum Pushout eine komplementäre Operation betrachtet werden:

Definition 2.10: (Pushoutkomplement)
Seien in einer Kategorie $\mathcal{C}$ zwei Morphismen $f : a \to b$ und $q : b \to d$ gegeben. Das Objekt $c \in Obj$ und die zwei Morphismen $g : a \to c$ und $p : c \to d$ heißen Pushoutkomplement zu f, q, wenn $q \cdot f = p \cdot g$ ein Pushout ist. $\diamond$

Das Pushoutkomplement ist wie das Coprodukt nicht zwingend eindeutig. In [Sch99b] findet sich eine Konstruktion für das Pushoutkomplement auf Basis des Coproduktkomplements (Abb. 2.12): Bei gegebenen Morphismen $f : a \to b$ und $q : b \to d$ konstruiert man zunächst das Coproduktkomplement $q' : c' \to d$ zu q. Danach bildet man das Coprodukt zu a und c'. Es ergeben sich ein Objekt c und Morphismen $g : a \to c$ und $g' : c' \to c$. Dann ist $q \cdot f = p \cdot g$ Pushout.

Die Konstruktion kann offensichtlich nur durchgeführt werden, wenn zu q ein Coproduktkomplement existiert.

Zum Pushout soll nun auch wieder die duale Definition, das Pullback, vorgestellt werden[1], da dieses später einmal benötigt wird.

---

[1] Der Systematik nach müßte das Pushout eigentlich Co-Pullback heißen.

$$\begin{tikzcd}
a \arrow[r, "f"] \arrow[d, "g"'] & b \arrow[d, "q"] \\
c \arrow[r, "p"] & d \\
c' \arrow[u, "g'"] \arrow[ur, "q'"'] &
\end{tikzcd}$$

Abbildung 2.12: Konstruktion des Pushoutkomplements

Definition 2.11: (Pullback)
Seien in einer Kategorie $\mathcal{C}$ zwei Morphismen $p : d \rightarrow c$ und $q : d \rightarrow b$ gegeben. Das Objekt $a \in Obj$ und die zwei Morphismen $f : b \rightarrow a$ und $g : c \rightarrow a$ heißen Pullback zu p, q, wenn $f \cdot q = g \cdot p$ und für alle anderen $f' : b \rightarrow a'$ und $g' : c \rightarrow a'$ mit dieser Eigenschaft $(f' \cdot q = g' \cdot p)$ ein eindeutiger Morphismus u mit $f' = u \cdot f$ und $g' = u \cdot g$ existiert. ◇

$$\begin{tikzcd}
a' \arrow[drr, phantom, "="] \arrow[ddr, phantom, "="] & & \\
& a \arrow[ul, "u"] \arrow[dr, phantom, "="] & b \arrow[l, "f"'] \arrow[ull, "f'"'] \\
& c \arrow[u, "g"] \arrow[uul, "g'"] & d \arrow[l, "p"] \arrow[u, "q"']
\end{tikzcd}$$

Abbildung 2.13: Definition des Pullbacks

Die Existenz beliebiger Colimites wird durch die Eigenschaft der Covollständigkeit beschrieben:

Definition 2.12: (Covollständigkeit)
Eine Kategorie heißt endlich covollständig, wenn für alle endlichen Diagramme zugehörige Colimites existieren. Dies ist der Fall genau dann, wenn sie entweder

Pushouts und ein initiales Objekt besitzt oder alternativ Coprodukte und Coegalisatoren. ⋄

2.3 Konkrete Beispiele von Kategorien

In diesem Abschnitt sollen einige Beispiele für konkrete Kategorien gegeben werden. Die wichtigste von diesen ist die Kategorie $\mathcal{S}et$, auf der fast alle anderen Beispiele aufbauen. Daneben ist die Kategorie $\mathcal{G}raph$ von entscheidender Bedeutung für die späteren Anwendungen der Transformationsregeln.

2.3.1 Mengen

Die Kategorie $\mathcal{S}et$ ist wie folgt definiert:

- Die Objekte sind die Mengen.
- Die Morphismen sind die (totalen) Funktionen auf Mengen.
- Die Komposition $\cdot$ der Kategorie ist die Komposition der zugrundeliegenden Mengenfunktionen.
- Der Identitätsmorphismus für eine Menge $A \in Obj$ ist die Identitätsfunktion auf dieser Menge, also $\forall a \in A : id(a) = a$.

Beispielsweise gibt es für die Objekte $a = \{1, 4, 6\}$ und $b = \{2, 5, 7\}$ den Morphismus $f : a \to b$, wobei $f(x) = x + 1$. Dieser Morphismus ist auch ein Isomorphismus, denn mit $f^{-1} : b \to a, f^{-1}(x) = x - 1$ existiert eine Umkehrung.

In $\mathcal{S}et$ sind die Monomorphismen die injektiven Funktionen und die Epimorphismen die Surjektionen. Coretraktionen und Retraktionen fallen ebenfalls fast mit diesen Begriffen zusammen, allerdings mit einer Ausnahme: Die (leeren) Abbildungen aus der leeren Menge in eine beliebige nichtleere Menge sind keine Coretraktionen. Diese Abbildungen sind zwar injektiv (kein Bildelement wird mehr als einmal getroffen), aber sie sind dennoch nicht umkehrbar.

Initiales Objekt in $\mathcal{S}et$ ist die leere Menge, von der genau eine Abbildung zu allen anderen Mengen (nämlich die leere Abbildung) und sonst keine weitere existiert. Terminales Objekt ist jede einelementige Menge. Zu dieser gibt es von jeder anderen Menge genau eine Abbildung, nämlich die, die alle Elemente auf das eine Element abbildet. Hier sieht man deutlich, daß die Eindeutigkeit nur bis auf Isomorphie gegeben ist. Alle einelementigen Mengen sind isomorph zueinander, denn jede Funktion zwischen ihnen kann nur das eine Element der einen Menge auf das eine Element der anderen Menge abbilden.

Das Coprodukt entspricht in $\mathcal{S}et$ der disjunkten Vereinigung der beiden Ausgangsmengen. Bei der disjunkten Vereinigung werden die Elemente jeweils mit einem Index versehen, so daß Elemente im Durchschnitt im Coprodukt nicht nur einmal auftauchen, sondern jedes Element genau ein Ziel besitzt. Betrachtet man die beiden Mengen $A = \{a, b, c\}$ und $B = \{a, d, e\}$, so ist das Coprodukt $C = \{a_1, b_1, c_1, a_2, d_2, e_2\}$ oder vereinfachend $C = \{a_1, b, c, a_2, d, e\}$.

Gibt es eine weitere Menge C', zu der von den beiden Ausgangsmengen Funktionen $f'_i(i \in \{1, 2\})$ existieren, so läßt sich der von der Definition des Coprodukts geforderte eindeutige Morphismus $u : C \to C'$ als $f'(x) = y \Rightarrow u(x_i) = y$ $(i \in \{1, 2\})$ konstruieren.

Das Coproduktkomplement (s. Definition 2.7) in $\mathcal{S}et$ existiert offensichtlich nur für injektive Abbildungen. Für eine solche Funktion $f : A \to C$ ist das Coproduktkomplement $B = C - f[A]$, also die Differenz aus der Zielmenge und dem Bild der Menge A in C. Bei dieser Konstruktion hat jedes Element der Mengen A und B genau ein Bild in C, es fallen keine Bilder zusammen und es gibt in C auch keine weiteren Elemente. C ist also Coprodukt zu A und B. Ist der Abbildung f dagegen nicht injektiv, so ergibt sich folgendes Problem:

$$\begin{tikzcd}
\{a, a', b\} \arrow[r, "f"] \arrow[dr, "f'"'] & \{a, b, c, d\} \arrow[d, phantom, "\nexists u"] & \{c, d\} \arrow[l, "g"'] \arrow[dl, "g'"] \\
& \{a, a', b, c, d\} &
\end{tikzcd}$$

Abbildung 2.14: Nicht-Existenz des Coproduktkomplements bei nichtinjektiver Funktion

Sei $f(a) = f(a') = a$ aber $f'(a) = a$ und $f'(a') = a'$. Dann gibt es keinen Morphismus u zwischen C und C' mit $f' = u \cdot f$, weil die einmal zusammengeworfenen Elemente a und a' von u nicht mehr getrennt werden können. C erfüllt somit nicht mehr die Definition des Coproduktes (Abb. 2.14).

Coegalisatoren in $\mathcal{S}et$ bilden in ein Restklassensystem bezüglich der Elemente ab, die unter den beiden Morphismen gemeinsames Urbild haben. Zur näheren

Erläuterung seien zwei Funktionen $f, g : A \to B$ gegeben und $\sim$ definiert als folgende Relation:

$$x \sim y \quad \Leftrightarrow \quad \exists a \in A : f(a) = x \wedge g(a) = y$$

$x \sim y$ gilt also, wenn $x, y \in B$ gemeinsames Urbild in A haben. Sei dann $\approx$ die von $\sim$ induzierte Äquivalenzrelation, also der reflexive, transitive und symmetrische Abschluß von $\sim$. Bekanntlich erzeugen Äquivalenzrelationen Restklassensysteme, in unserem Fall das System $B/_{\approx} = \{[x] \mid x \in B\}$. ($[x]$ wird wie üblich als Notation für die Äquivalenzklasse benutzt, in der x enthalten ist.) Der Coegalisator h ist nun die natürliche Abbildung von B nach $B/_{\approx}$: $h(x) = [x]$.

Der Coegalisator führt also Elemente wieder zusammen, die von f und g verschieden abgebildet wurden. Erläutert sei dies an einem Beispiel. Seien

$$f, g : \{a, b, c, d\} \to \{i, j, k, l, m, n\}$$

definiert als

	f	g
a	i	i
b	j	k
c	l	m
d	m	n

Es ergeben sich also folgende Beziehungen für die Relation $\sim$:

$$j \sim k, \quad l \sim m, \quad m \sim n$$

und das Restklassensystem

$$\begin{array}{rcl} [i] & = & \{i\} \\ [j] & = & \{j, k\} \\ [l] & = & \{l, m, n\} \end{array}$$

Der Coegalisator h zu f und g ist dann

$$\begin{array}{rcl} & h & \\ i & \mapsto & [i] \\ j & \mapsto & [j] \\ k & \mapsto & [j] \\ l & \mapsto & [l] \\ m & \mapsto & [l] \\ n & \mapsto & [l] \end{array} \qquad \begin{array}{rcl} & h \cdot f = h \cdot g & \\ a & \mapsto & [i] \\ b & \mapsto & [j] \\ c & \mapsto & [l] \\ d & \mapsto & [l] \end{array}$$

Elemente von A, die von f und g auf verschiedene Elemente von B abgebildet wurden, gehen also nach dem Anfügen des Coegalisators wieder zu einem gemeinsamen Ziel. Darüber hinaus werden auch solche Elemente von A bei $h \cdot f$ identifiziert, bei denen die Bilder unter f und g zusammenfielen (im Beispiel c und d).

Das Pushout kann — wie im vorherigen Abschnitt bereits erwähnt — aus Coprodukt und Coegalisator konstruiert werden. Hier ein Beispiel:

$$\begin{tikzcd}
\mathrm{A} = \{a\} \arrow[rr] \arrow[dd] \arrow[dr, shift left] \arrow[dr, shift right] & & \{a,b,d\} = \mathrm{B} \arrow[dl] \arrow[dd] \\
& \{a_1,a_2,b_1,b_2,c,d\} \arrow[dr] & \\
\mathrm{B} = \{a,b,c\} \arrow[ur] \arrow[rr] & & \{[a],[b_1],[b_2],[c],[d]\} = \mathrm{D}
\end{tikzcd}$$

Abbildung 2.15: Konstruktion eines Mengen-Pushouts aus Coprodukt und Coegalisator

Zuerst wird das Coprodukt zu $\{a, b, d\}$ und $\{a, b, c\}$ gebildet; dies ist die disjunkte Vereinigung $\{a_1, a_2, b_1, b_2, c, d\}$, bei der die as und bs auf getrennte Bilder a_i und b_i $(i \in \{1, 2\})$ abgebildet werden. Anschließend wird der Coegalisator konstruiert, der die Elemente a_1 und a_2, die gemeinsames Urbild a in A haben, wieder auf eine gemeinsame Restklasse $[a]$ abbildet. b_1 und b_2 bleiben in D jedoch getrennt, da sie kein Urbild in A besitzen.

Das Pushout kann also bei Mengen anschaulich als eine Mischung aus normaler und disjunktiver Vereinigung angesehen werden. Elemente in B und C, die kein Urbild in A haben, werden disjunkt vereinigt. D.h., falls B und C gemeinsame Elemente haben sollten, die nicht aus A stammen, so werden diese auf getrennte Elemente in der Vereinigung abgebildet. Solche Elemente von B und C jedoch, die ein Urbild in A haben, werden im Ergebnis D auf das gleiche Element abgebildet.

Das Pushoutkomplement (s. Definition 2.10) in $\mathcal{S}et$ kann wie folgt konstruiert werden [Sch95a], wenn f injektiv ist und somit ein Coproduktkomplement existiert:

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] & B \arrow[d, "q"] & A' \arrow[l, "f'"'] \arrow[dl, "p'"] \\
C \arrow[r, "p"] & D &
\end{tikzcd}$$

Abbildung 2.16: Konstruktion des Pushoutkomplements in $\mathcal{S}et$

- Konstruiere A' als Coproduktkomplement zu $f : A \to B$, d.h. $A' = B - f[A]$.
- $p' : A' \to D := q \cdot f$
- Konstruiere $C := D - p'[A']$ und $p : C \to D$ als die Projektion von C nach D.
- Das noch fehlende $g : A \to C$ kann dann als $g := p^{-1} \cdot q \cdot f$ konstruiert werden, wobei p^{-1} die eindeutige Umkehrung der Einschränkung $p : C \to p[C] \subseteq D$ ist.

Dieses Konstruktionsverfahren ist leider nicht immer anwendbar. Dafür muß noch die sogenannte *Identifikationsbedingung* erfüllt sein, die besagt, daß zwei verschiedene Elemente $x, x' \in B$, die ein gemeinsames Bild in D haben, ein Urbild in A haben müssen. Andernfalls ergibt sich folgende Situation:

$$\begin{tikzcd}
\{1\} \arrow[r, "f"] & \{1,2\} \arrow[d, "q"] & \{2\} \arrow[l, "f'"'] \arrow[dl, "p'"] \\
\{\} \arrow[r, "p"] & \{1 = 2\} &
\end{tikzcd}$$

Abbildung 2.17: Notwendigkeit der Identifikationsbedingung für das Pushoutkomplement

Die Abbildung q wirft die beiden Elemente 1 und 2 zusammen, was im Diagramm als Zielelement $1 = 2$ veranschaulicht wird und das obige Bedingung verletzt (2 hat kein Urbild in A). Die als $D - p'[A']$ konstruierte Menge C ist damit leer und es gibt keine Funktion $g : A \to C$. Formal kann die Bedingung als

$$\forall x, x' \in B : \quad q(x) = q(x') \to x = x' \vee (x \in f[A] \wedge x' \in f[A])$$

geschrieben werden.

2.3.2 Natürliche Zahlen und Teilbarkeit

Um zu demonstrieren, daß sich die Konzepte der Kategorientheorie nicht nur auf ein natürlich scheinendes Anwendungsgebiet wie Mengen übertragen lassen, wird im folgenden noch eine Kategorie $\mathcal{N}at$ mit einigen besonderen Eigenschaften definiert:

- Die Objekte sind die natürlichen Zahlen.
- Es existiert ein Morphismus zwischen a und b genau dann, wenn a ein Teiler von b ist. Es gibt also höchstens einen Morphismus zwischen zwei Objekten.
- Zwei existierende Morphismen können immer miteinander komponiert werden, da wenn a Teiler von b ist und b Teiler von c, so ist auch a Teiler von c. Die Komposition existiert also und ist eindeutig, weil es nur einen Morphismus von a nach c geben kann.
- Der Identitätsmorphismus zu a ist der Morphismus von a nach a. Dieser existiert, da jede natürliche Zahl sich selbst teilt.

In $\mathcal{N}at$ sind alle Morphismen Mono- und Epimorphismen, denn wenn überhaupt zwei Morphismen zwischen zwei Objekten existieren, so müssen diese sowieso identisch sein. Die Folgerung in der Definition von Epi- und Monomorphismen wird daher unabhängig von f.

Andererseits gibt es keine Retraktionen und Coretraktionen außer den Identitäten, die sogar Isomorphismen sind. Da Teilbarkeit nur in einer Richtung möglich ist (von der kleineren zur größeren Zahl), kann es keine Umkehrungen geben, außer die Zahlen sind gleich.

Die Kategorie $\mathcal{N}at$ besitzt ein initiales Objekt, nämlich die 1. Von ihr gibt es einen (wie immer eindeutigen) Morphismus zu jedem anderen Objekt, denn 1 ist Teiler jeder natürlichen Zahl. Allerdings gibt es kein terminales Objekt, denn es gibt keine Zahl, die von jeder anderen geteilt werden könnte.

Das Coprodukt zu zwei Objekten a, b von $\mathcal{N}at$ ist das kleinste gemeinsame Vielfache. Dies ist definitionsgemäß die kleinste Zahl, die von a und b geteilt wird. Von kgV(a,b) gibt es (eindeutige) Morphismen zu allen anderen gemeinsamen Vielfachen von a und b. Das duale Produkt ist der größte gemeinsame Teiler.

Das Coproduktkomplement in $\mathcal{N}at$ existiert immer, ist aber nicht unbedingt eindeutig. Sucht man zu $f : a \to c$ das Komplement $g : b \to c$, so benötigt man eine

Zahl b mit der Eigenschaft $c = \mathrm{kgV}(a, b)$. Betrachtet man die Primzahlzerlegung der beteiligten Zahlen

$$\begin{aligned} a &= p_1^{a_1} \cdot p_2^{a_2} \cdot p_3^{a_3} \cdot \ldots \\ b &= p_1^{b_1} \cdot p_2^{b_2} \cdot p_3^{b_3} \cdot \ldots \\ c &= p_1^{c_1} \cdot p_2^{c_2} \cdot p_3^{c_3} \cdot \ldots \\ & \quad (p_1, p_2, p_3, \ldots \text{ sei die Folge der Primzahlen}) \end{aligned}$$

so muß für das kleinste gemeinsame Vielfache

$$c_i = \max(a_i, b_i) \quad \forall i \in \mathbb{N}$$

gelten. Weil bekannt ist, daß a Teiler von c ist, gilt auch

$$a_i \leq c_i \quad \forall i \in \mathbb{N}$$

Ist $a_i < c_i$, so muß $b_i = c_i$ gesetzt werden, damit die Maximumsbedingung des kgV erfüllt ist. Ist dagegen $a_i = c_i$, so kann man b_i aus dem Intervall $[0, c_i]$ frei wählen, ohne daß die Bedingung verletzt wird.

Der Coegalisator zu $f, g : a \to b$ in $\mathcal{N}at$ ist immer id_b. Das es nur einen Morphismus zwischen a und b geben kann, sind f und g sowieso identisch und der einfachste Morphismus $h : b \to c$ mit $h \cdot f = f \cdot g$ ergibt sich, wenn man $c = b$ setzt. Zu allen anderen Zahlen, die von b mit $h' : b \to c'$ geteilt werden, gibt es ein eindeutiges u mit $u \cdot c = c'$, nämlich c' selbst.

Aus der Konstruktion des Pushouts mit Coprodukt und Coegalisator leitet sich ab, daß in $\mathcal{N}at$ das Pushout mit dem Coprodukt zusammenfällt und unabhängig von dem Objekt a ist. Dies ergibt sich aus der Tatsache, daß der Coegalisator der Identitätsmorphismus ist.

2.3.3 Graphen

Es gibt zwei Möglichkeiten, Graphen zu definieren: Die eine beschreibt Kanten als Paare von Knoten, wobei die erste Komponente der Quellknoten und die zweite der Zielknoten ist. Eine Alternative ist es, zwei Funktionen einzuführen, die zu einer Kante den Quell- bzw. Zielknoten angeben. Für die Betrachtung von Graphen als Kategorien eignet sich die zweite Möglichkeit besser, weil dann die Knoten- und Kantenmengen separat als Objekte der Kategorie $\mathcal{S}et$ betrachtet und abgebildet werden können.

Definition 2.13: (Graph)
Ein Graph ist ein Quadrupel (V, E, s, t) mit

- V ist die Menge der Knoten ("vertices")
- E ist die Menge der Kanten ("edges")
- $s : V \rightarrow E$ ("source") gibt den Quellknoten einer Kante an und
- $t : V \rightarrow E$ ("target") gibt den Zielknoten einer Kante an

◇

Für die Definition einer Kategorie $\mathcal{G}raph$ werden dann noch Graphmorphismen als Abbildungen zwischen Graphen benötigt:

Definition 2.14: (Graphmorphismus)
Ein Graphmorphismus f zwischen zwei Graphen $G_1 = (V_1, E_1, s_1, t_1)$ und $G_2 = (V_2, E_2, s_2, t_2)$ besteht aus zwei Morphismen der Kategorie $\mathcal{S}et$ $f = (f_V, f_E)$ wobei $f_V : V_1 \rightarrow V_2$ und $f_E : E_1 \rightarrow E_2$. Die Abbildungen müssen strukturverträglich sein, d.h.

$$\forall e \in E : \quad f_V(s_1(e)) = s_2(f_E(e)) \quad \wedge \quad f_V(t_1(e)) = t_2(f_E(e))$$

◇

Die letzte Bedingung kann graphisch auch so formuliert werden, daß die folgenden Diagramme kommutieren müssen:

$$\begin{tikzcd}
E_1 \arrow[r, "f_E"] \arrow[d, "s_1"'] \arrow[dr, phantom, "="] & E_2 \arrow[d, "s_2"] \\
V_1 \arrow[r, "f_V"'] & V_2
\end{tikzcd}
\qquad
\begin{tikzcd}
E_1 \arrow[r, "f_E"] \arrow[d, "t_1"'] \arrow[dr, phantom, "="] & E_2 \arrow[d, "t_2"] \\
V_1 \arrow[r, "f_V"'] & V_2
\end{tikzcd}$$

Ein Beispiel für einen injektiven Graphmorphismus findet sich in Abb.2.18. f_V bildet jeweils Knoten mit gleichen Nummern aufeinander ab. Die Kantenabbildung f_E ergibt sich unmittelbar aus der Strukturverträglichkeitsbedingung und ist nicht gesondert dargestellt.

Der gezeigte Graphmorphismus ist offenbar injektiv, denn es werden keine verschiedenen Knoten oder Kanten des Ausgangsgraphen auf gleiche Knoten/Kanten des Zielgraphen abgebildet. Man kann definieren, daß ein Graphmorphismus genau dann injektiv heißen soll, wenn die zugrundeliegenden Knoten- und Kantenabbildungen injektiv sind. Analog definieren sich surjektive Graphmorphismen.

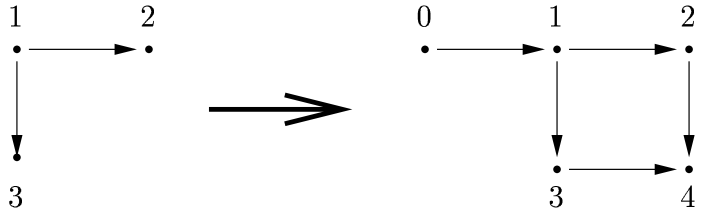


Abbildung 2.18: Ein injektiver Graphmorphismus

Abb.2.19 zeigt einen nicht-injektiven Morphismus, der die Knoten 1 und 3 auf den gleichen Knoten $1 = 3$ abbildet. Die Kante von 1 nach 3 wird dabei zu einer Schleife an $1 = 3$.

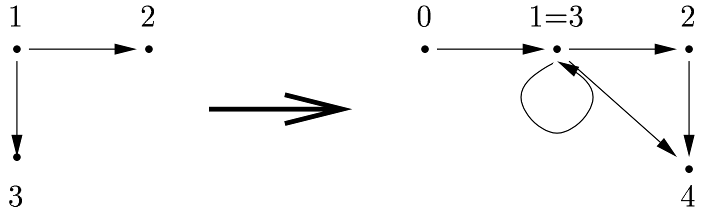


Abbildung 2.19: Ein nicht-injektiver Graphmorphismus

Man sieht auch, daß Graphmorphismen als Einbettung eines Graphen in einen anderen betrachtet werden können. Dies ergibt sich unmittelbar daraus, daß jeder Knoten und jede Kante des Ursprungsgraphen ein Bild im Zielgraphen haben muß (alle Abbildungen sind totale Abbildungen, wenn nicht explizit von partiellen Funktionen die Rede ist). Andererseits kann der Zielgraph natürlich größer als der Ausgangsgraph sein, denn keine der Funktionen muß surjektiv sein. Die Einbettung wirkt etwas unnatürlich, wenn sie nicht injektiv ist, daher verwenden die meisten Graphtransformationsregeln in der Praxis injektive Abbildungen. Aber auch nicht-injektive Einbettungen haben gewisse Anwendungen.

Nach diesen Festlegungen läßt sich nun eine Kategorie der Graphen $\mathcal{G}raph$ als eine Art Verdopplung von $\mathcal{S}et$ definieren. Die Knoten- und Kantenmenge sind jeweils Objekte von $\mathcal{S}et$ und die Knoten- und Kantenabbildungen sind Morphismen in $\mathcal{S}et$.

Definition 2.15: (Kategorie $\mathcal{G}raph$)

- Die Objekte sind Graphen wie aus Definition 2.13.
- Die Morphismen sind die Graphmorphismen wie in Definition 2.14.

- Die Komposition zweier Graphmorphismen $f = (f_V, f_E)$ und $g = (g_V, g_E)$ ist die komponentenweise Verknüpfung der zugrundeliegenden Mengenmorphismen:
$$f \cdot g = (f_V \cdot g_V, f_E \cdot g_E)$$
- Die Identitätsmorphismen setzen sich aus den Identitäten auf Knoten und Kanten zusammen:
$$id_G = (id_V, id_E)$$

◇

Die verschiedenen Morphismeneigenschaften (Epi-, Mono-, Isomorphismus, Retraktion, Coretraktion) übertragen sich direkt von den $\mathcal{S}et$-Morphismen auf Morphismen in $\mathcal{G}raph$.

Auch die Konstruktionen der Colimites können direkt von $\mathcal{S}et$ abgeleitet werden, indem sie zweimal getrennt für Knoten und Kanten ausgeführt werden. Die Funktionen s und t zwischen Kanten und Knoten des konstruierten Objekts ergeben sich zwangsläufig aus dem geforderten eindeutigen Morphismus u, der von dem Colimes zu allen anderen Objekten führt, mit denen ein kommutierendes Diagramm entsteht.

$$\begin{tikzcd}
A_E \arrow[rr, "f_E"] \arrow[dr, "s_A"] \arrow[dd, "g_E"'] & & B_E \arrow[dr, "s_B"] \arrow[dd, dashed, "q_E"'] & \\
& A_V \arrow[rr, "f_V"] \arrow[dd, "g_V"'] & & B_V \arrow[dd, "q_V"'] \\
C_E \arrow[rr, dashed, "p_E"] \arrow[dr, "s_C"] & & D_E \arrow[dr, "s_D"] & \\
& C_V \arrow[rr, "p_V"] & & D_V
\end{tikzcd}$$

Abbildung 2.20: Pushout-Konstruktion in $\mathcal{G}raph$

Exemplarisch wird dies für das Pushout in Abbildung 2.20 demonstriert. Bei gegebenen $f : A \to B$ und $g : A \to C$ ergeben sich alle Objekte und Morphismen bis auf s_D bei der Konstruktion der Pushouts $q_E \cdot f_E = p_E \cdot g_E$ und $q_V \cdot f_V = p_V \cdot g_V$ in $\mathcal{S}et$. Nun ist aber $q_E \cdot f_E = p_E \cdot g_E$ ein Pushout in $\mathcal{S}et$ und es gibt neben D_E ein weiteres Objekt $D_V \in Obj_{\mathcal{S}et}$, zu dem ein kommutierendes Diagramm über f_E und g_E existiert: $(q_V \cdot s_B) \cdot f_E = (p_V \cdot s_C) \cdot g_E$. Daher muß es aufgrund der Pushouteigenschaft einen eindeutigen Morphismus $u : D_E \to D_V$ geben, so daß $(q_V \cdot s_B) \cdot f_E = u \cdot q_E \cdot f_E$ und $(p_V \cdot s_C) \cdot g_E = u \cdot p_E \cdot g_E$. Aufgrund der Eindeutigkeit von u ergibt sich $s_D = u$.

Die gesuchte Quellkantenfunktion ergibt sich also in eindeutiger Weise aus der Konstruktion. Selbstverständlich gelten diese Überlegungen genauso für t_D, so daß $D = (D_V, D_E, s_D, t_D)$ wieder ein vollständiger Graph ist. Mit dem gleichen Beweisschema lassen sich die fehlenden s und t bei allen Colimeskonstruktionen in $\mathcal{G}raph$ finden, die alle in ihrer Definition die Existenz eines eindeutigen u fordern, die ebenfalls die Diagramme kommutierend machen.

Das Coproduktkomplement existiert in $\mathcal{G}raph$ genauso wie in $\mathcal{S}et$ nur für injektive Morphismen. Durch die getrennte Berechnung der Mengendifferenz für Knoten und Kanten kann es dazu kommen, daß manche Kanten im Komplement keine Quell- oder Zielknoten mehr besitzen. Dies ist jedoch für die Anwendungen in dieser Arbeit unerheblich, da das Coproduktkomplement nur als Hilfsmittel zur Konstruktion des Pushoutkomplements benötigt wird. Letzteres verdient jedoch nähere Betrachtung.

Zunächst muß die in 2.3.1 erwähnte Identifikationsbedingung sowohl für die Knoten- als auch die Kantenmenge erfüllt sein. Danach ergibt sich noch das Problem, daß nach einer getrennten Konstruktion des Pushoutkomplements in $\mathcal{S}et$ das Ergebnis nicht in allen Fällen ein Graph ist. Es gibt Fälle, in denen Quell- oder Zielknoten von Kanten fehlen können. In dem Beispiel in Abb. 2.21 (aus [Sch99b]) wurde die komponentenweise Konstruktion bei Knoten und Kanten wie in 2.3.1 beschrieben durchgeführt.

Da $q : B \to D$ injektiv ist, kann das Coproduktkomplement $q' : B' \to D$ konstruiert werden. Das Ergebnis ist kein Graph, was aber bei diesem Zwischenergebnis nicht erforderlich ist. Beim nächsten Schritt wird B als Coprodukt zu A und B' gebildet. Dieses Ergebnis ist wieder kein Graph, da die Kante zum Knoten 4 keinen Quellknoten besitzt.

Um solche Situationen zu vermeiden, ist die *Klebebedingung* zu erfüllen [EPS73]:

Satz 2.16: (Klebebedingung)
Die komponentenweise Konstruktion des Pushoutkomplements mit Hilfe der Coprodukte kann dann und nur dann zu einem Pushoutdiagramm in $\mathcal{G}raph$

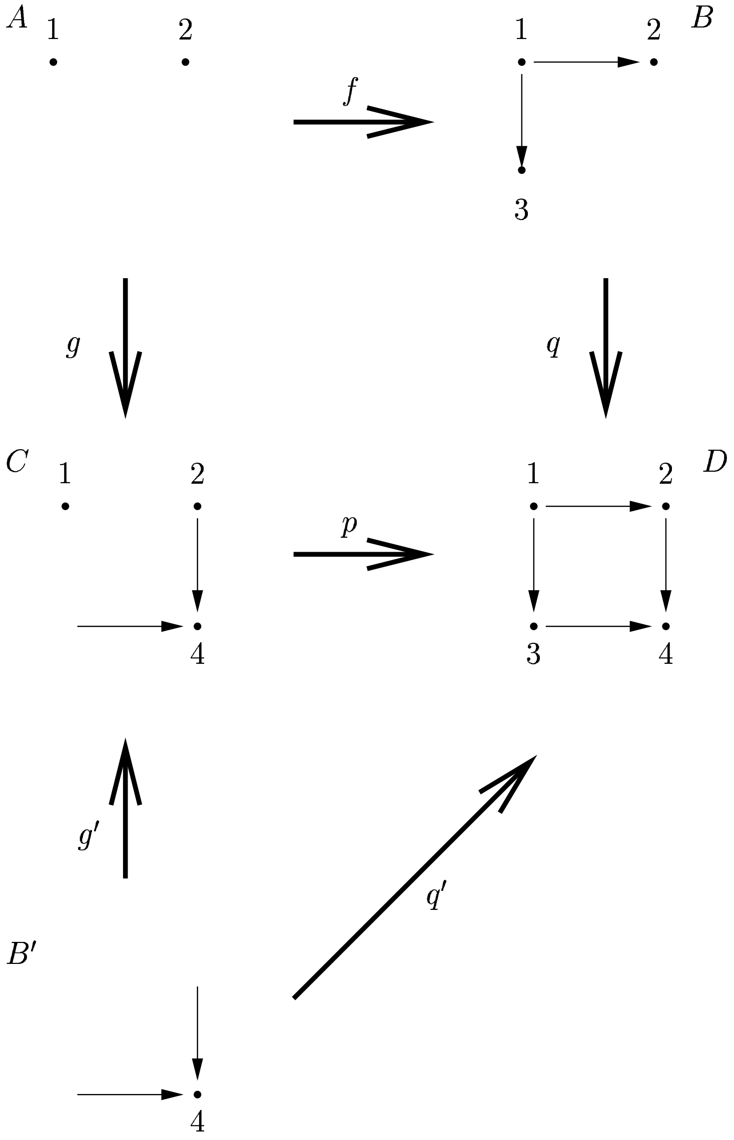


Abbildung 2.21: Notwendigkeit der Klebebedingung

ergänzt werden, wenn gilt:

$$\begin{aligned} s_D\, q_E[B'_E] &\subseteq q'_V[B'_V] \cup q_V f_V[A_V] \\ t_D\, q_E[B'_E] &\subseteq q'_V[B'_V] \cup q_V f_V[A_V] \end{aligned}$$

Korollar dazu ist, daß, wenn f und q Monomorphismen in $\mathcal{G}raph$ sind und die Klebebedingung erfüllt ist, ein eindeutiges Pushoutkomplement existiert. $\diamond$

Im Beispiel wird die Klebebindung dadurch verletzt, daß $3 \in s_D\, q_E[B'_E] = \{2, 3\}$, aber $3 \notin q'_V[B'_V] = \{4\}$ und $3 \notin q_V f_V[A_V] = \{1, 2\}$.

2.3.4 Markierte Graphen

In der Praxis sind Graphen alleine meistens nicht ausreichend, da sie nur eine Struktur darstellen, jedoch nicht die Daten, die strukturiert werden. Um diese mit aufzunehmen, *markiert* man die Knoten und Kanten des Graphen.

Allgemein ist es möglich, Knoten und Kanten mit kategoriellen Objekten zu markieren [PEM87, Sch93, Fis99]. Dies eröffnet weit gefächerte Möglichkeiten, wie zum Beispiel Markierungen mit Mengen, Graphen, Algebren etc. In dieser Arbeit werden keine komplexen Markierungen benötigt, daher soll an dieser Stelle nur kurz das Prinzip der kategoriellen Markierung vorgestellt werden. Anschließend wird die einfache Markierung mit einem endlichen Alphabet näher ausgeführt, die im weiteren Verlauf Anwendung findet.

Definition 2.17: (Kategoriell markierte Graphen)
Sei $L = (\mathcal{L}_V, \mathcal{L}_E)$ ein Paar von Markierungskategorien. Ein markierter Graph ist dann ein Sextupel $G = (V, E, s, t, l_V, l_E)$, wobei (V, E, s, t) den zugrundeliegenden Graphen definieren und $l_V : V \to \mathcal{L}_V$ und $l_E : E \to \mathcal{L}_E$ Abbildungen sind, die Knoten und Kanten mit Objekten aus $\mathcal{L}_V$ bzw. $\mathcal{L}_E$ markieren.

Ein Morphismus in der Kategorie der kategoriell markierten Graphen besteht aus einem Graphmorphismus für den zugrundeliegenden unmarkierten Graphen und Morphismen $\in \mathrm{Mor}_{\mathcal{L}_V}$ für jeden Knoten und Morphismen $\in \mathrm{Mor}_{\mathcal{L}_E}$ für jede Kante zur Abbildung der Markierungen. ◇

Diese Objekte und Morhpismen bilden eine Kategorie, die covollständig ist (Def. 2.12), wenn $\mathcal{L}_V$ und $\mathcal{L}_E$ covollständig sind. Alle Konstruktionen übertragen sich dann also auf solchermaßen markierte Kategorien.

Bei einfach markierten Graphen besteht die Markierung nur aus Elementen aus festgelegten Markierungsalphabeten:

Definition 2.18: (Markierter Graph)
Ein markierter Graph ist ein Sextupel $G = (V, E, s, t, l_V, l_E)$, wobei (V, E, s, t) ein Graph ist. $l_V : V \to L_V$ und $l_E : E \to L_E$ sind Abbildungen, die jedem Knoten und jeder Kante Elemente der jeweiligen Markierungsalphabete L_V bzw. L_E zuordnen. L_V bzw. L_E sind endliche Mengen.

Ein Morphismus f zwischen zwei markierten Graphen G und H besteht aus $f_V : V_G \to V_H$ und $f_E : E_G \to E_H$, die wie in Definition 2.14 strukturverträglich sind:

$$f_V \cdot s_G = s_H \cdot f_E \quad \wedge \quad f_V \cdot t_G = t_H \cdot f_E$$

Zusätzlich muß f mit den Markierungsfunktionen verträglich sein:

$$l_{V_H} \cdot f_V = l_{V_G} \quad \wedge \quad l_{E_H} \cdot f_E = l_{E_G}$$

Auf diese Weise markierte Graphen bilden eine Kategorie $\mathcal{LG}raph$ und das Pushout kann genau wie bei unmarkierten Graphen komponentenweise konstruiert werden. ◇

$$\begin{tikzcd}
 & G_E \arrow[dl, "l_{E_G}"'] \arrow[d, "f_E"'] \arrow[r, "s_G"] \arrow[dr, phantom, "="] & G_V \arrow[d, "f_V"] \arrow[dr, "l_{V_G}"] & \\
L_E \arrow[r, phantom, "="] & {} & {} & L_V \arrow[l, phantom, "="] \\
 & H_E \arrow[ul, "l_{E_H}"] \arrow[r, "s_H"'] & H_V \arrow[ur, "l_{V_H}"'] &
\end{tikzcd}$$

Abbildung 2.22: Morphismus zwischen markierten Graphen

Objekte, die durch den Graphmorphismus aufeinander abgebildet werden, müssen also gleich markiert sein. Dies ist in vielen Fällen zu restriktiv, so daß man auch strutkurierte Alphabete einführen kann. Bei diesen sind gewisse Markierungsänderungen möglich.

Ein *strukturiertes Alphabet* ist ein Alphabet L zusammen mit einer reflexiven und transitiven Relation $\sqsubseteq$ auf seinen Elementen.

Definition 2.19: (Strukturiert markierte Graphen)
Ein strukturiert markierter Graph ist ein markierter Graph wie aus Definition 2.18, wobei die Markierungsalphabete L_V und L_E durch Relationen $\sqsubseteq_V$ und $\sqsubseteq_E$ strukturiert sind.

Für einen Morphismus zwischen solchen Graphen lautet die Bedingung für die Markierungsverträglichkeit

$$l_{V_H} \cdot f_V \sqsupseteq l_{V_G} \quad \wedge \quad l_{E_H} \cdot f_E \sqsupseteq l_{E_G}$$

Die strukturiert markierten Graphen bilden mit ihren Morphismen eine Kategorie $\mathcal{SLG}raph$. ◇

$$\begin{tikzcd}
L_E \arrow[d, "\sqsubseteq_E"'] & G_E \arrow[l, "l_{E_G}"'] \arrow[r, "s_G"] \arrow[d, "f_E"'] \arrow[rd, phantom, "="] & G_V \arrow[r, "l_{V_G}"] \arrow[d, "f_V"] & L_V \arrow[d, "\sqsubseteq_V"] \\
L_E & H_E \arrow[l, "l_{E_H}"] \arrow[r, "s_H"'] & H_V \arrow[r, "l_{V_H}"'] & L_V
\end{tikzcd}$$

Abbildung 2.23: Morphismus zwischen strukturiert markierten Graphen

Für die Existenz des Pushouts in $\mathcal{SLG}raph$ ist zusätzlich notwendig, daß es eine kleinste obere Schranke bezüglich $\sqsubseteq_V$ bzw. $\sqsubseteq_E$ gibt. Werden im Pushoutobjekt vorher getrennte Knoten oder Kanten aufeinander abgebildet, so werden diese mit der kleinsten oberen Schranke der Markierungen der Urbilder versehen.

Problematisch in $\mathcal{SLG}raph$ ist die Konstruktion des Coproduktkomplements, das für das Pushoutkomplement benötigt wird, da hier Mehrdeutigkeiten möglich sind. Man betrachte ein Alphabet $\{a, b, c, d\}$ und als Strukturierungsrelation den reflexiven und transitiven Abschluß von

$$a \sqsubseteq d, \quad b \sqsubseteq d, \quad c \sqsubseteq d$$

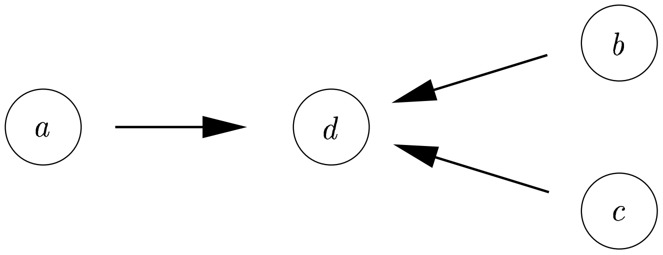


Abbildung 2.24: Mehrdeutigkeit des Coproduktkomplements in $\mathcal{SLG}raph$

In Abbildung 2.24 bestehen die Graphen jeweils nur aus einem markierten Knoten. Ist der linke Morphismus gegeben, so sind die beiden rechten Morphismen als Coproduktkomplement möglich, denn d ist die kleinste obere Schranke sowohl von $\{a, b\}$ als auch von $\{a, c\}$. Diese Mehrdeutigkeit kann bei der späteren Anwendung von Graphtransformationsregeln durchaus zu unangenehmen Situationen führen, da der Kontext und damit das Ergebnis eines Ableitungsschrittes nicht immer eindeutig ist.

2.4 Transformation kategorieller Objekte

In diesem Abschnitt sollen nun Transformationen für kategorielle Objekte vorgestellt werden. Diese beruhen auf Morphismen und Pushouts der zugrundeliegenden Kategorie. Entsprechend der späteren Anwendung werden sich die Beispiele auf Graphen beziehen, die Definitionen und Konstruktionen ermöglichen aber ein analoges Vorgehen in beliebigen Kategorien, die die genannten Bedingungen erfüllen.

2.4.1 Definitionen

Definition 2.20: (Produktion)
Eine Produktion p in einer Kategorie $\mathcal{C}$ ist ein Paar von Morphismen dieser Kategorie (p_l, p_r) mit $dom(p_l) = dom(p_r)$. ◇

Eine Produktion wird üblicherweise in der Form

$$B_l \xleftarrow{p_l} K \xrightarrow{p_r} B_r$$

geschrieben. B_l wird die linke Seite genannt, B_r die rechte Seite der Produktion. K ist das Klebeobjekt.

Definition 2.21: (Direkte Ableitbarkeit)
In einer Kategorie $\mathcal{C}$ heißt ein Objekt $G_r \in Obj_{\mathcal{C}}$ mittels einer Produktion $p = (p_l, p_r)$ direkt ableitbar aus $G_l \in Obj_{\mathcal{C}}$ genau dann, wenn ein Morphismus $g : K \to C$ existiert, so daß $q_l \cdot g = g_l \cdot p_l$ und $q_r \cdot g = g_r \cdot p_r$ Pushouts in $\mathcal{C}$ sind. Die direkte Ableitbarkeit wird geschrieben als $G_l \xrightarrow{p} G_r$. ◇

$$\begin{tikzcd}
B_l \arrow[d, "g_l"'] \arrow[dr, phantom, "\text{PO}"] & K \arrow[l, two heads, "p_l"'] \arrow[r, two heads, "p_r"] \arrow[d, "g"'] \arrow[dr, phantom, "\text{PO}"] & B_r \arrow[d, "g_r"] \\
G_l & C \arrow[l, two heads, "q_l"] \arrow[r, two heads, "q_r"'] & G_r
\end{tikzcd}$$

Abbildung 2.25: Ableitungsschritt

Wie von dem Ableitbarkeitsbegriff bei Chomsky-Grammatiken gewohnt, kann man $G_l \longrightarrow G_r$ schreiben, wenn die verwendete Produktion nicht von Bedeutung ist.

Sind p und g gegeben, so sind G_l und G_r eindeutig. Sind p und g_l (und damit G_l) gegeben, so ist die Existenz von C von der Existenz eines Pushoutkomplements zu p_l und g_l abhängig. Da die Definition die Existenz von C erfordert, ist ohne Pushoutkomplement im konkreten Fall keine Ableitbarkeit gegeben und die Produktion ist mit der Einbettung g_l auf G_l nicht anwendbar. Auch wenn C existiert, ist aufgrund der Eigenschaften des Pushoutkomplements C und damit G_r nicht notwendigerweise eindeutig.

Definition 2.22: (Ableitbarkeit)

In einer Kategorie $\mathcal{C}$ heißt G_n (in beliebig vielen Schritten) ableitbar von G_0 genau dann, wenn

$$\exists G_1, \ldots, G_{n-1} : \forall i \in \{1, ..., n\} : \quad G_{i-1} \longrightarrow G_i$$

Die Ableitbarkeit wird geschrieben als $G_0 \overset{*}{\longrightarrow} G_n$. ◇

Definition 2.23: (Grammatik)

Eine Grammatik in einer Kategorie $\mathcal{C}$ ist ein Tripel (T, P, S) wobei $T \subseteq Obj_\mathcal{C}$ eine Menge von terminalen Objekten ist, $P \subseteq Mor_\mathcal{C} \times Mor_\mathcal{C}$ eine endliche Menge von Produktionen und $S \in Obj_\mathcal{C}$ das Startobjekt ist.

Die von der Grammatik erzeugte Sprache L ist die Menge aller Objekte in der Terminalmenge, die mit Produktionen aus P aus dem Startsymbol abgeleitet werden können:

$$L = \{G \in T \mid S \overset{*}{\longrightarrow} G\}$$

◇

All diese Definitionen sind sehr ähnlich zu den bekannten Definitionen bei Chomsky-Grammatiken. Dies ist beabsichtigt, da die Grammatiken auf kategorieller Basis eine Erweiterung der Konzepte von Chomsky-Grammatiken darstellen. Sie können auf Objekten aus beliebigen Kategorien operieren anstatt auf Zeichenketten.

Man beachte, daß die Unterscheidung zwischen terminalen und nichtterminalen Symbolen bei Chomsky-Grammatiken für kategorielle Grammatiken unüblich ist. Sie dient dort zur Vereinfachung der Definition der erzeugten Sprache, während bei kategoriellen Grammatiken meist über die Terminierung der Ableitungsfolge oder einem Prädikat auf den Objekten geschieht.

Auch wenn in Anlehnung an Chomsky-Grammatiken von Produktionen und Ableitungen die Rede ist, so können Ableitungsschritte auch als allgemeine Transformationen auf den Objekten der Kategorie aufgefaßt werden. Wenn es die Anwendung nicht erfordert, kann man auf eine Terminalmenge als Ziel der Transformationen aus dem Startobjekt verzichten. Eine Produktionenmenge P kann dann als Satz von Transformationsregeln aufgefaßt werden, deren Ziel anderweitig bestimmt wird oder sich eindeutig ergibt.

2.4.2 Konstruktion von Ableitungsschritten

Bei der Definition der direkten Ableitbarkeit (Def. 2.21) wurde bereits erwähnt, daß bei der normalen Anwendung einer Produktion auf ein gegebenes Objekt ein Kontext C zu konstruieren ist. Dieser Kontext muß nicht immer existieren oder eindeutig sein.

Die Konstruktion eines Ableitungsschrittes geht folgendermaßen vonstatten:

1. Die Produktion und eine Einbettung der linken Seite in das Ausgangsobjekt G_l sind gegeben:

$$\begin{tikzcd}
B_l \arrow[d, "g_l"'] & K \arrow[l, two heads, "p_l"'] \arrow[r, two heads, "p_r"] & B_r \\
G_l & &
\end{tikzcd}$$

2. Der Kontext C wird als Pushoutkomplement zu p_l und g_l konstruiert:

$$\begin{tikzcd}
B_l \arrow[d, "g_l"'] \arrow[dr, phantom, "\text{PO}"] & K \arrow[l, two heads, "p_l"'] \arrow[r, two heads, "p_r"] \arrow[d, "g"'] & B_r \\
G_l & C \arrow[l, two heads, "q_l"] &
\end{tikzcd}$$

3. Abschließend bildet man das Pushout zu p_r und g und erhält damit G_r:

$$\begin{tikzcd}
B_l \arrow[d, "g_l"'] \arrow[dr, phantom, "\text{PO}"] & K \arrow[l, two heads, "p_l"'] \arrow[r, two heads, "p_r"] \arrow[d, "g"'] \arrow[dr, phantom, "\text{PO}"] & B_r \arrow[d, "g_r"] \\
G_l & C \arrow[l, two heads, "q_l"] \arrow[r, two heads, "q_r"'] & G_r
\end{tikzcd}$$

Schritt 3 wirft keine Probleme auf, da alle anwendungsrelevanten Kategorien Pushouts besitzen und ein Pushout (bis auf Isomorphie) eindeutig ist.

Schritt 2 dagegen beruht auf dem Pushoutkomplement, das abhängig von der Kategorie nicht immer existiert oder mehrdeutig sein kann. Für die Beispielkategorien $\mathcal{S}et$ und $\mathcal{G}raph$ wurde das Pushoutkomplement in den jeweiligen Abschnitten (2.3.1 und 2.3.3) näher betrachtet. Insbesondere ist bei $\mathcal{S}et$ die Identifikationsbedingung und bei $\mathcal{G}raph$ die Klebebedingung zu beachten.

Für den Rest dieser Arbeit kann zur Vereinfachung jedoch angenommen werden, daß p_l und g_l Monomorphismen sind. Für diesen Fall kann bewiesen werden, daß das Pushoutkomplement immer eindeutig existiert, sofern zusätzlich die Klebebedingung erfüllt ist [Sch99b].

Auch für den Fall, daß die erwähnten Morphismen nicht monomorph sind, kann charakterisiert werden, wann eine Ableitung möglich ist und wann Mehrdeutigkeiten auftreten können. Dazu muß jeder Morphismus in einen Mono- und einen Epimorphismus zerlegbar sein (sog. $\mathcal{E}$-$\mathcal{M}$-*faktorisierbare Kategorie*). Dann kann die Konstruktion des Komplements in vier Teile zerlegt werden. Näheres findet sich ebenfalls in [Sch99b].

Das Beispiel in Abb. 2.26 demonstriert die Konstruktion eines Ableitungsschrittes für Graphen. Die Morphismen zwischen den Graphen werden wie üblich durch die Nummerierung der Knoten angegeben.

Zuerst wird das Coproduktkomplement links unten konstruiert. Es besteht aus den Knoten- und Kantendifferenzmengen der linken Seite der Produktion und des Ausgangsgraphen. Das Komplement ist kein Graph, da einige Kanten keine Quell- oder Zielknoten besitzen. Bildet man dann das Coprodukt mit dem Klebeobjekt, erhält man den Kontext des Ableitungsschrittes. Dieser muß nun wieder ein korrekter Graph sein und ist es im Beispiel auch, da die fehlenden Knoten aus dem Klebegraphen hinzugefügt werden.

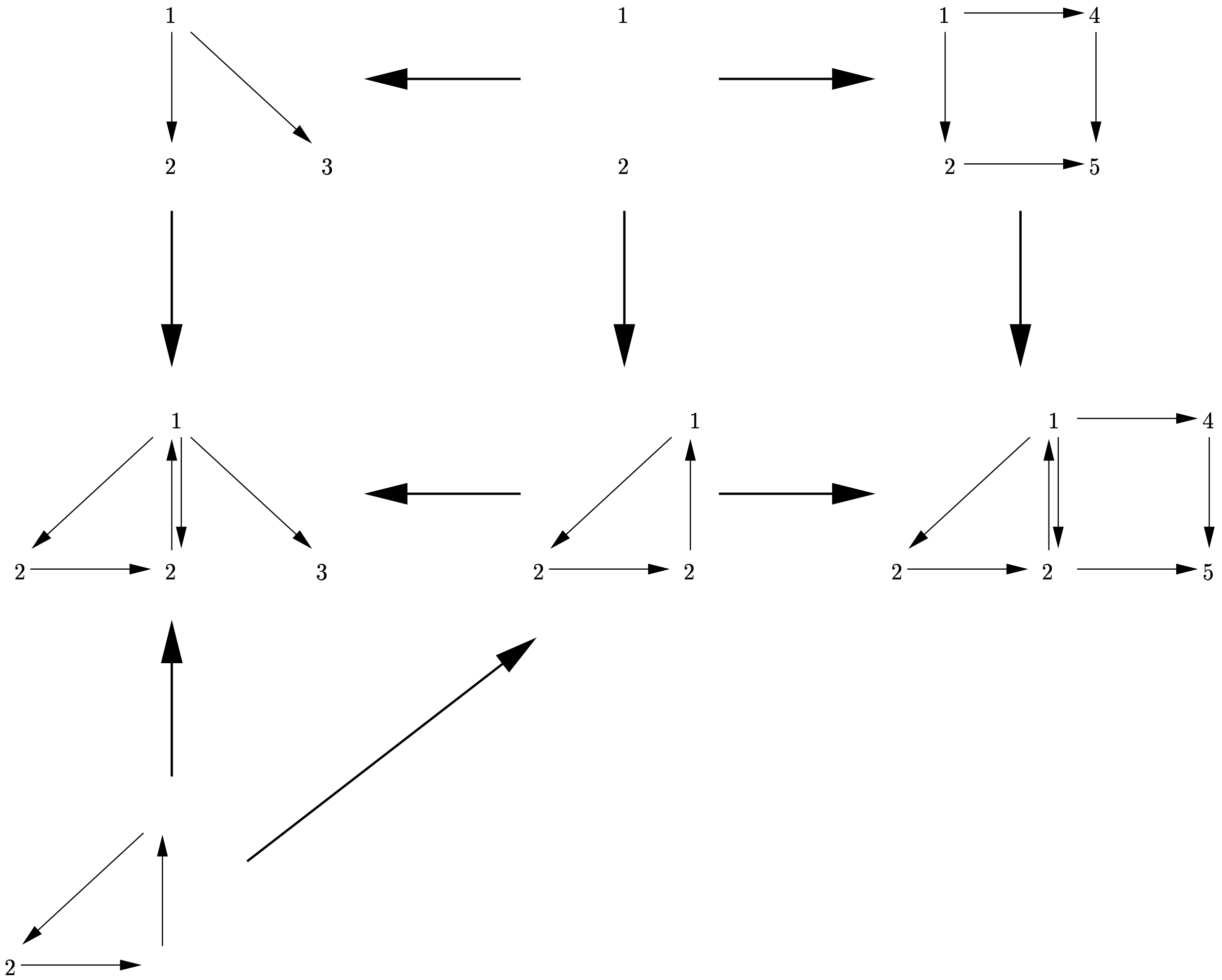


Abbildung 2.26: Beispiel eines Ableitungsschrittes mit Graphen

Schließlich folgt die Konstruktion des Ergebnisses als Pushout zur rechten Diagrammhälfte. Entsprechend den Eigenschaften des Pushouts werden die rechte Seite und der Kontext disjunkt vereinigt, wobei sie allerdings entlang der Klebestellen „verklebt“ werden, indem Bestandteile, die ein Urbild im Klebegraphen haben, aufeinander abgebildet werden.

2.5 Parallele Unabhängigkeit

Eine interessante Eigenschaft von kategoriellen Transformationssytemen ist die *parallele Unabhängigkeit* von Transformationsschritten. Diese besagt, daß bei Anwendung des einen Schrittes die zweite Transformationsregel immer noch anwendbar ist und umgekehrt.

Definition 2.24: (Parallele Unabhängigkeit)
Zwei Ableitungsschritte $G_l \xrightarrow{pr} G_r$ und $G_l \xrightarrow{pr'} G'_r$ heißen parallel unabhängig, wenn

$$\exists \alpha : B_l \longrightarrow C', \alpha' : B'_l \longrightarrow C : \; g_l = q'_l \cdot \alpha \;\wedge\; g'_l = q_l \cdot \alpha'$$

◇

Anschaulich bedeutet die Existenz von α und α', daß die linke Seite B_l der Produktion pr noch im Kontext C' der Produktion pr' enthalten sein muß und umgekehrt. Damit ist sichergestellt, daß pr dann auch noch auf das Ergebnis G'_r vom pr' angewendet werden kann. Die Frage ist, ob nach Anwendung der zweiten Produktion auch das gleiche Endergebnis steht wie nach umgekehrter Anwendung der Schritte.

Zur Klärung dieser Frage wird zunächst noch die Definition einer PIT-Kategorie (“parallel independence theorem”) benötigt. Für die Klasse $\mathcal{M}$ von Morphismen werden meist die Monomorphismen eingesetzt.

Definition 2.25: (PIT-Kategorie)
Eine Kategorie heißt PIT-Kategorie bezüglich einer Klasse $\mathcal{M}$ von Morphismen, wenn sie folgende Bedingungen erfüllt:

- Sind zwei Morphismen $f, g \in \mathcal{M}$, so existiert ein Pushout $q \cdot f = p \cdot g$ und $p, q \in \mathcal{M}$.
- Zu allen $p, q \in \mathcal{M}$ existiert ein Pullback $q \cdot f = p \cdot g$ mit $f, g \in \mathcal{M}$.

$$\begin{tikzcd}
B_l \arrow[dd, "g_l"'] \arrow[dddr, bend left, "\alpha"] & K \arrow[l, two heads, "p_l"'] \arrow[r, two heads, "p_r"] \arrow[d, "g"'] & B_r \arrow[d, "g_r"] \\
& C \arrow[r, two heads, "q_r"'] \arrow[dl, "q_l"'] & G_r \\
G_l & & \\
& C' \arrow[r, two heads, "q'_r"] \arrow[ul, "q'_l"] & G'_r \\
B'_l \arrow[uu, "g'_l"] \arrow[uuur, bend right, "\alpha'"] & K' \arrow[l, two heads, "p'_l"] \arrow[r, two heads, "p'_r"'] \arrow[u, "g'"] & B'_r \arrow[u, "g'_r"']
\end{tikzcd}$$

Abbildung 2.27: Parallele Unabhängigkeit

- Sind in dem Diagramm

$$\begin{tikzcd}
& K_2 \arrow[r, two heads] \arrow[d] \arrow[dr, phantom, "(2)"] & B_2 \arrow[d] \\
K_1 \arrow[d] \arrow[dr, phantom, "(1)"] & C_0 \arrow[l, two heads] \arrow[r, two heads] \arrow[d] \arrow[dr, phantom, "(3)"] & C_1 \arrow[d] \\
B_1 & C_2 \arrow[l, two heads] \arrow[r, two heads] & G
\end{tikzcd}$$

  alle Morphismen in $\mathcal{M}$, die Diagramme (1+3) und (2+3) Pushouts und (3) ein Pullback, so sind alle Teildiagramme (1), (2) und (3) Pushouts.

◇

Damit kann man folgenden Satz aufstellen:

Satz 2.26: (Unabhängigkeit von Ableitungsschritten)
In PIT-Kategorien gilt, daß bei zwei parallel unabhängigen Ableitungsschritten

$$G_0 \xrightarrow{p_1} G_1 \text{ und } G_0 \xrightarrow{p_2} G_1'$$

ein G_2 existiert, so daß

$$G_1 \xrightarrow{p_2} G_2 \text{ und } G_1' \xrightarrow{p_1} G_2$$

◇

(Beweis in [Sch99a, Sch99b].) In [Sch95a] findet sich auch der Nachweis, daß $\mathcal{S}et$ und $\mathcal{G}raph$ mit den Injektionen als Klasse $\mathcal{M}$ die PIT-Bedingungen erfüllen.

2.6 Anwendbarkeitsbedingungen

Bei vielen Anwendungen von Transformationssystemen auf der Basis von Kategorien benötigt man zusätzliche Bedingungen für die Anwendbarkeit von Produktionen. In der Literatur finden sich viele Ansätze, wie solche Bedingungen formuliert werden können [PR69b, EH86, Sch91, HW95]. Für die Zwecke dieser Arbeit ist eine allgemein gehaltene Definition [Sch99b] ausreichend:

Definition 2.27: (Produktionen mit Anwendbarkeitsbedingung)
Eine Anwendbarkeitsbedingung für eine Produktion p in einer Kategorie $\mathcal{C}$ mit linker Seite B_l ist eine entscheidbare Menge

$$A(p) \subseteq \bigcup_{G \in Obj_\mathcal{C}} \{f \in Mor_\mathcal{C} \mid dom(f) = B_l \wedge codom(f) = G\}$$

Eine Produktion mit Anwendbarkeitsbedingung ist nur dann auf ein Objekt G anwendbar, wenn für die Einbettung $g_l : B_l \longrightarrow G$ nach G $g_l \in A(p)$ gilt. ◇

Im weiteren Verlauf werden nur zwei Typen von Anwendbarkeitsbedingungen benötigt: Monomorphe Einbettung und daß bestimmte Knoten der linken Seite in G keine weiteren Kanten in einer bestimmten Richtung besitzen dürfen, selbstverständlich ausgenommen den in B_l vorhandenen. Die Forderung nach Injektivität ist einfach zu realisieren, indem $A(p)$ mit der Menge aller injektiven Morphismen geschnitten wird.

Das Verbot von weiteren Kanten kann auf folgende Weise erreicht werden:

$$A(p) = \{f : B_l \longrightarrow G \mid f \text{ monomorph} \wedge \nexists f' : B_l' \longrightarrow G\}$$

wobei B_l' in geeigneter Weise definiert wird. Sowohl die Monomorphie ist entscheidbar, als auch die Nichtexistenz einer bestimmten Einbettung, da die verwendeten Graphen endlich sind.

Beispielsweise soll in der folgenden Produktion gefordert werden, daß Knoten 1 keine einlaufenden Kanten besitzen darf. Dies wird durch eine Bemerkung $|in| = 0$ dargestellt, wobei $|in|$ für die Anzahl hinführender Kanten steht ("in-count"). Entsprechend wäre $|out|$ ("out-count") die Zahl der auslaufenden Kanten.

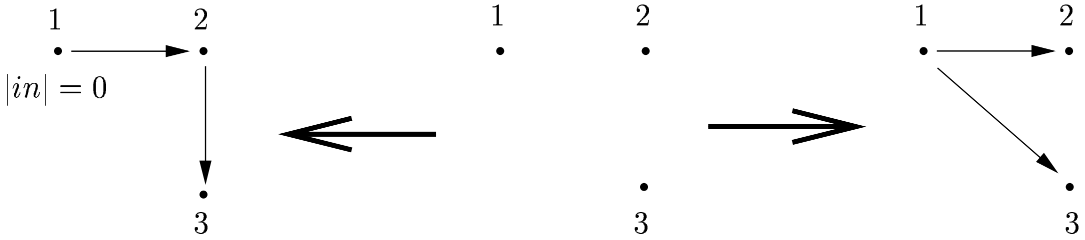


Abbildung 2.28: Produktion mit Anwendbarkeitsbedingung

Diese Bedingung läßt sich realisieren, indem man ein B_l' definiert, das keine Einbettung in G besitzen darf:

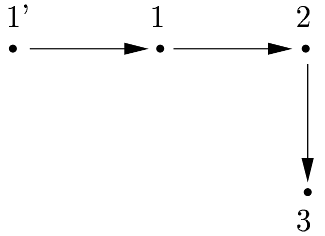


Abbildung 2.29: B_l' für die Anwendbarkeitsbedingung

Man sieht leicht, daß immer, wenn ein $f' : B_l' \longrightarrow G$ existiert, der Knoten 1 in G Ziel mindestens einer Kante ist, was ausgeschlossen werden soll. Auf ähnliche Weise kann man alle Bedingungen der Form $|in| < n$ und $|out| < n \quad (n \in \mathbb{N})$ realisieren.

Bei markierten Graphen ist auch die Markierung zu beachten. Bei strukturierten Alphabeten, die ein kleinstes Element besitzen, kann dies zur Markierung verwendet werden und es gibt einen Graphmorphismus zu allen Graphen mit gleicher Struktur und beliebigen Markierungen an Kanten und Knoten.

Wird kein strukturiertes Alphabet eingesetzt, so muß man, um etwa alle einlaufenden Kanten zu verbieten, die oben gegebene Bedingung $A(p)$ leicht modifizieren und einen Allquantor für die Markierung des Knotens $1'$ und der Kante $1' \longrightarrow 1$ in B_l' einführen. Eine weitere mögliche Variation ist es, nur Kanten mit bestimmter Markierung zu verbieten, indem man $A(p)$ entsprechend gestaltet.

Kapitel 3

Reduktionssysteme

Im vorangegangenen Kapitel wurden Transformationsregeln auf kategoriellen Objekten eingeführt. Will man nun ein System von solchen Regeln praktisch anwenden, ergeben sich Fragen nach Eigenschaften des Gesamtsystems:

- Terminiert eine Transformationsfolge oder gibt es unendliche Ketten von Transformationsschritten?
- Wenn bei einem Objekt zwei verschiedene Transformationen möglich sind, können die beiden Linien später wieder zusammengeführt werden? Diese Eigenschaft wird *Konfluenz* genannt.
- Wenn ein System terminiert und konfluent ist, gibt es dann ein eindeutiges Ziel jeder Transformationsfolge von diesem Objekt aus? Existieren also zu bestimmten oder allen Objekten Normalformen?

Alle diese Fragen werden von der Theorie der Reduktionssysteme behandelt [Klo87, DJ90, Löw92, Der93, Ave95]. Hierbei wird von den konkreten Transformationen abstrahiert und nur von einer gegebenen Relation zwischen den Objekten ausgegangen. Dies ist für die Behandlung obiger Fragestellungen ausreichend. Wie ein Objekt bei einem tatsächlichen System aus einem anderen abgeleitet wird ist unerheblich.

3.1 Grundlagen

Reduktionssysteme oder regelbasierte Systeme (in der englischen Literatur sind die Begriffe "rewriting systems", "replacement systems" oder auch "reduction systems" gebräuchlich) werden von verschiedenen Autoren leicht unterschiedlich definiert. Hier soll die Definition aus [Sch95b] übernommen werden:

Definition 3.1: (Reduktionssystem)
Ein Reduktionssystem ist durch einen Zustandsraum Z und eine Relation $\longrightarrow \subseteq Z \times Z$ gegeben. (Z muß nicht endlich sein.) ⋄

Es werden folgende Schreibweisen benutzt:

$$\begin{array}{lll} x \xrightarrow{0} y & :\Leftrightarrow & x = y \\ x \xrightarrow{i+1} y & :\Leftrightarrow & \exists z \in Z : \; x \xrightarrow{i} z \wedge z \longrightarrow y \quad (i \geq 0) \\ x \xrightarrow{*} y & :\Leftrightarrow & \exists i \geq 0 : \; x \xrightarrow{i} y \\ x \xrightarrow{+} y & :\Leftrightarrow & \exists i > 0 : \; x \xrightarrow{i} y \end{array}$$

Definition 3.2: (noethersch)
Ein Reduktionssystem $(Z, \longrightarrow)$ heißt noethersch genau dann, wenn es in ihm keine unendlichen Ketten $z_1 \longrightarrow z_2 \longrightarrow z_3 \longrightarrow \ldots \quad (z_i \in Z)$ gibt. ⋄

In einem noetherschen System terminieren offensichtlich alle Reduktionsfolgen. Daher ist diese Eigenschaft bei praktischer Anwendung eines Systems oft erforderlich.

Definition 3.3: (Konfluenz)
Ein Reduktionssystem $(Z, \longrightarrow)$ heißt konfluent genau dann, wenn

$$\forall x, u, v \in Z : \; x \xrightarrow{*} u \wedge x \xrightarrow{*} v \Rightarrow \exists z \in Z : \; u \xrightarrow{*} z \wedge v \xrightarrow{*} z$$

⋄

Die Eigenschaft der Konfluenz (Abb. 3.1) besagt also, daß, wenn man ausgehend von einem x zu verschiedenen u, v gelangt, diese getrennten Ableitungslinien wieder zu einem z zusammengeführt werden können. Man hat also durch die Entscheidung, von x zu u oder v zu gehen, nicht die Möglichkeit vergeben, schließlich zu z zu gelangen.

$$\begin{tikzcd}
& x \arrow[dl, "*"'] \arrow[dr, "*"] & & & & x \arrow[dl, "*"'] \arrow[dr, "*"] & \\
u & & v & \Longrightarrow & u \arrow[dr, "*"'] & & v \arrow[dl, "*"] \\
& & & & & \exists z &
\end{tikzcd}$$

Abbildung 3.1: Konfluenz

Eine alternative und kürzere Schreibweise für die Konfluenz ist [Sch95b]:

$$\overset{*}{\longleftarrow} \cdot \overset{*}{\longrightarrow} \subseteq \overset{*}{\longrightarrow} \cdot \overset{*}{\longleftarrow}$$

Der rückwärts gerichtete Pfeil $\longleftarrow$ steht für die Anwendung der Umkehrrelation zu $\longrightarrow$. Auf der linken Seiten der Inklusion gelangt man von u rückwärts zu x und dann vorwärts zu v. Die Inklusion besagt, daß es dann auch möglich sein muß, von u aus vorwärts zu gehen und anschließend mit umgekehrt angewandeten Regeln auch zu v. Das Element am Ende des Vorwärtspfades ist dann z.

Eine eingeschränkte Variante ist die lokale Konfluenz:

Definition 3.4: (Lokale Konfluenz)
Ein Reduktionssystem $(Z, \longrightarrow)$ heißt lokal konfluent genau dann, wenn

$$\forall x, u, v \in Z: \; x \longrightarrow u \wedge x \longrightarrow v \Rightarrow \exists z \in Z: \; u \overset{*}{\longrightarrow} z \wedge v \overset{*}{\longrightarrow} z$$

oder

$$\longleftarrow \cdot \longrightarrow \subseteq \overset{*}{\longrightarrow} \cdot \overset{*}{\longleftarrow}$$

$\diamond$

Der Unterschied ist also nur, daß bei lokaler Konfluenz von x zu u und v nur ein Schritt erlaubt ist. Daher ist diese Eigenschaft oft einfacher nachzuweisen.

Definition 3.5: (Überführbarkeit)
$x, y \in Z$ heißen ineinander überführbar $x \longleftrightarrow y :\Leftrightarrow x \longrightarrow y \;\vee\; y \longrightarrow x$. $\diamond$

Die Überführbarkeit ist also die Anwendung der Regeln in beliebigen Richtungen. Es gelten die gleichen Schreibweisen $\overset{i}{\longleftrightarrow}$, $\overset{*}{\longleftrightarrow}$, usw. wie bei $\longrightarrow$. Es ist auch leicht zu sehen, daß $\overset{*}{\longleftrightarrow}$ eine Äquivalenzrelation darstellt. Die Symmetrie ergibt sich direkt aus der Definition von $\longleftrightarrow$ und Reflexivität sowie Transitivität ergeben sich aus der Definition von $\overset{*}{\longleftrightarrow}$.

3.2 Terminierung

Es gibt in der Literatur viele Ansätze, die Terminierung von Reduktionssystemen zu beweisen [Löw92, Der86, Der87a, Der87b]. Diese beruhen aber meist auf den speziellen Eigenschaften bestimmter Zustandsmengen wie zum Beispiel Termen oder Graphen.

Für allgemeine Reduktionssysteme ist auch nicht entscheidbar, ob sie noethersch sind. Dies kann man sich leicht verdeutlichen, indem man zu einer beliebigen Turingmaschine (oder einem anderen Berechenbarkeitsmodell) ein äquivalentes Reduktionssystem konstruiert, indem man Zustand der Maschine und Bandinhalt in den Zuständen des Reduktionssystems zusammenfaßt. Die erlaubten Arbeitsschritte bilden die Relation $\longrightarrow$ und gehen zu einem anderen Zustand/Bandinhalt über. Das Reduktionssystem ist genau dann noethersch, wenn die Turingmaschine bei jedem Eingabewort hält, was bekanntlich nicht entscheidbar ist.

Bei gegebenem Reduktionssystem kann man jedoch nach folgendem Prinzip versuchen zu zeigen, daß es noethersch ist: Sei M eine beliebige Menge, auf der eine wohlfundierte Partialordnung $\leq$ definiert ist. Man ordnet mit einer Funktion $\varphi : Z \rightarrow M$ jedem Zustand des Systems ein Element aus einer Menge M zu und fordert

$$x \longrightarrow y \;\Rightarrow\; \varphi(x) > \varphi(y).$$

($a > b$ sei wie üblich definiert als $b \leq a \wedge b \neq a$.)

Gelingt dies, so ist das Reduktionssystem noethersch. Angenommen es wäre nicht noethersch, so gäbe es eine unendliche Ableitungskette $x_1 \longrightarrow x_2 \longrightarrow x_3 \longrightarrow \ldots$ und damit auch $\varphi(x_1) > \varphi(x_2) > \varphi(x_3) > \ldots$. Dies steht aber im Widerspruch dazu, daß $\leq$ wohlfundiert ist.

Zum Nachweis der Terminierung eines Reduktionssystems ist also eine Abbildung in eine geeignete Menge mit wohlfundierter Partialordnung zu finden, zum Beispiel in die natürlichen Zahlen.

Hier ein Beispiel: Man betrachte ein Reduktionssystem mit den Graphen als Zustandsmenge und der einzigen Ableitungsregel, daß Doppelkanten (d.h. zwei Kanten mit gleichen Quell- und Zielknoten) durch eine einfache Kante ersetzt werden dürfen. Die Produktion dazu ist

Abbildung 3.2: Produktion des Beispiels

Ein geeignetes φ für einen Graphen $G = (V, E, s, t)$ ist dann $\varphi(G) = |E|$. Bei jeder Anwendung der Produktion sinkt die Anzahl der Kanten, so daß die Bedingung des Beweisschemas erfüllt ist. Auch ist $\leq$ auf den natürlichen Zahlen wohlfundiert, so daß das System noethersch folglich noethersch sein muß. Alternativ könnte man auch die Anzahl der Paare von Doppelkanten zählen (bei n gleichgerichteten Kanten zwischen zwei Knoten gibt es $\binom{n}{2}$ Paare). Dann ist φ von irreduziblen Graphen tatsächlich 0.

3.3 Konfluenz

Von großer Bedeutung für den Nachweis der Konfluenz eines Systems ist der folgende Satz von Newman/Huet.

Satz 3.6: (Newman/Huet)
Eine noethersche Relation ist genau dann konfluent, wenn sie lokal konfluent ist. ⋄

Dies reduziert den Aufwand zum Nachweis der Konfluenz erheblich. Es muß nur noch betrachtet werden, welche Paare von Ableitungsregeln auf den gleichen Zustand angewendet werden können. Gibt es solche, so muß für diese beiden Regeln die (lokale) Konfluenz gezeigt werden. Es ist jedoch nicht mehr erforderlich, alle Paare (u, v) zu betrachten, die in beliebig vielen Schritten aus einem Zustand ableitbar sind.

Zum Beweis des Satzes [Sch95b] wird das Prinzip der *noetherschen Induktion* benötigt. Dieses erlaubt es ähnlich der vollständigen Induktion bei natürlichen Zahlen, eine Aussage für alle Objekte zu beweisen, wenn man von der Gültigkeit der Aussage für die direkten Vorgänger auf die Gültigkeit für das Objekt selbst schließen kann. Ist diese Schlußkette fundiert, d.h. gibt es ein Anfangsobjekt, für das die Aussage gilt, so gilt sie für alle Nachfolger. Ist die Menge der Nachfolger die Gesamtheit der Objekte, so ist die Aussage global gültig.

Sei in einem noetherschen System ein Prädikat P auf Z gegeben. Kann man dann für alle $x \in Z$ den Induktionsschritt

$$\forall y \in Z : x \xrightarrow{\geq 1} y : P(y) \;\Rightarrow\; P(x)$$

machen, so gilt $\forall x \in Z : P(x)$. Die Schlußkette ist bei dieser Art der Induktion fundiert, da aufgrund der Eigenschaft noethersch des Systems die Kette der

Vorgänger y (im Sinne der Induktion) endlich ist. Am Ende der Kette gibt es keine y mehr und die Voraussetzung der Implikation ist leer und damit wahr und somit muß $P(x)$ gelten.

Nun zum Beweis des Satzes [Sch95b]: Die Richtung von Konfluenz zu lokaler Konfluenz ist trivial. In der anderen Richtung wählt man als Prädikat $P(x)$ die Eigenschaft, daß dieses Objekt konfluent ist:

$$P(x) \;:\Longleftrightarrow\; \forall u, v \in Z : \; x \xrightarrow{*} u \land x \xrightarrow{*} v \;\Longrightarrow\; \exists z \in Z : \; u \xrightarrow{*} z \land v \xrightarrow{*} z$$

Es ist zu zeigen, daß $P(x)$ unter der Induktionsannahme gilt, daß $P(x')$ für alle $x' : x \xrightarrow{\geq 1}$ wahr ist. Zusätzlich ist die Voraussetzung gegeben, daß das System lokal konfluent ist.

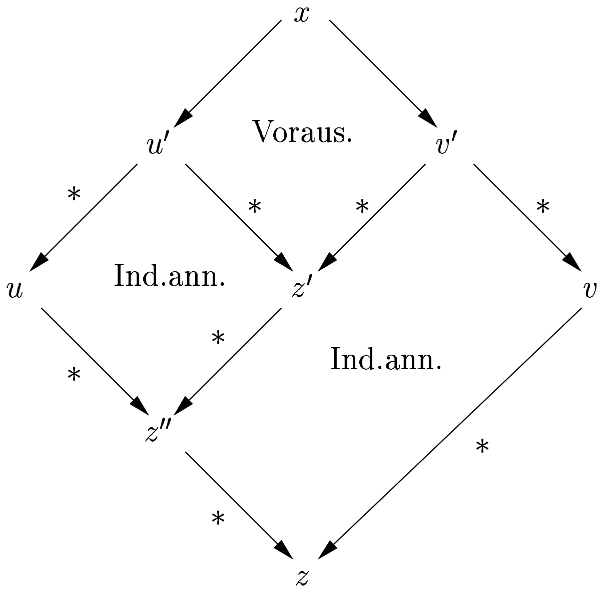


Abbildung 3.3: Zum Beweis des Satzes von Newman/Huet

Wenn die Ableitungen $x \xrightarrow{*} u$ und $x \xrightarrow{*} v$ in mindestens einem Schritt stattfinden, so kann man den ersten Schritt zu u' und v' abtrennen. Weil das System lokal konfluent ist, existiert dann ein z', das von u' und v' aus erreichbar ist. u' und v' sind aber auch Nachfolger von x, so daß über die Induktionsannahme deren Konfluenz angenommen werden kann. Man gelangt damit zu z'' und schließlich dem gesuchten z.

Besteht eine der beiden Ableitungen von x aus 0 Schritten, so ist der gesuchte gemeinsame Nachfolger z einfach das Objekt in der anderen Richtung. Ähnlich ist $z = x$, wenn beide Ableitungen aus 0 Schritten bestehen.

$$\begin{tikzcd}
u = x \arrow[dr] & \\
& v = z
\end{tikzcd}$$

Will man die Konfluenz des Doppelkanten-Beispiels aus dem vorherigen Abschnitt zeigen, so muß man also nur die lokale Konfluenz nachweisen. Dazu betrachtet man alle Graphen, bei denen die einzige Regel in verschiedener Weise angewendet werden kann, d.h. solche mit Doppelkanten zwischen mehr als einem Knotenpaar oder mit ≥ 3 Kanten zwischen einem Knotenpaar. In allen diesen Fällen ist leicht einzusehen, daß nach dem Löschen einer der in Frage kommenden Doppelkanten die andere noch existiert und in einem weiteren Schritt auch entfernt werden kann (Abb. 3.4 auf Seite 56). Das Ergebnis nach zwei Schritten ist das gleiche, egal in welcher Reihenfolge die Doppelkanten gelöscht wurden. Damit sind die getrennten Linien zusammengeführt und das System ist lokal konfluent und damit insgesamt konfluent.

3.4 Normalformen

Ist ein konfluentes und noethersches System gegeben, so ist auch noch von Interesse, wieviele verschiedene Objekte am Ende der Ableitungskette stehen können. Solche Objekte heißen Normalformen:

Definition 3.7: (Normalform)
y heißt Normalform von x, geschrieben $x \xrightarrow{!} y$, genau dann, wenn

$$x \xrightarrow{*} y \;\wedge\; \nexists z \in Z :\; y \longrightarrow z$$

Von y aus dürfen also keine weiteren Schritte möglich sein. Es wird dann auch *irreduzibel* genannt. ◇

Es ist offensichtlich, daß in jedem noetherschen System jeder Zustand mindestens eine Normalform besitzen muß, denn beginnt man dort eine Ableitungskette, so endet diese irgendwann und man hat eine Normalform gefunden. In Verbindung mit Konfluenz ergibt sich folgender Satz:

Satz 3.8: (Eindeutige Normalformen)
Ist ein Reduktionssystem noethersch und konfluent, so besitzt jedes $z \in Z$ genau eine Normalform. ◇

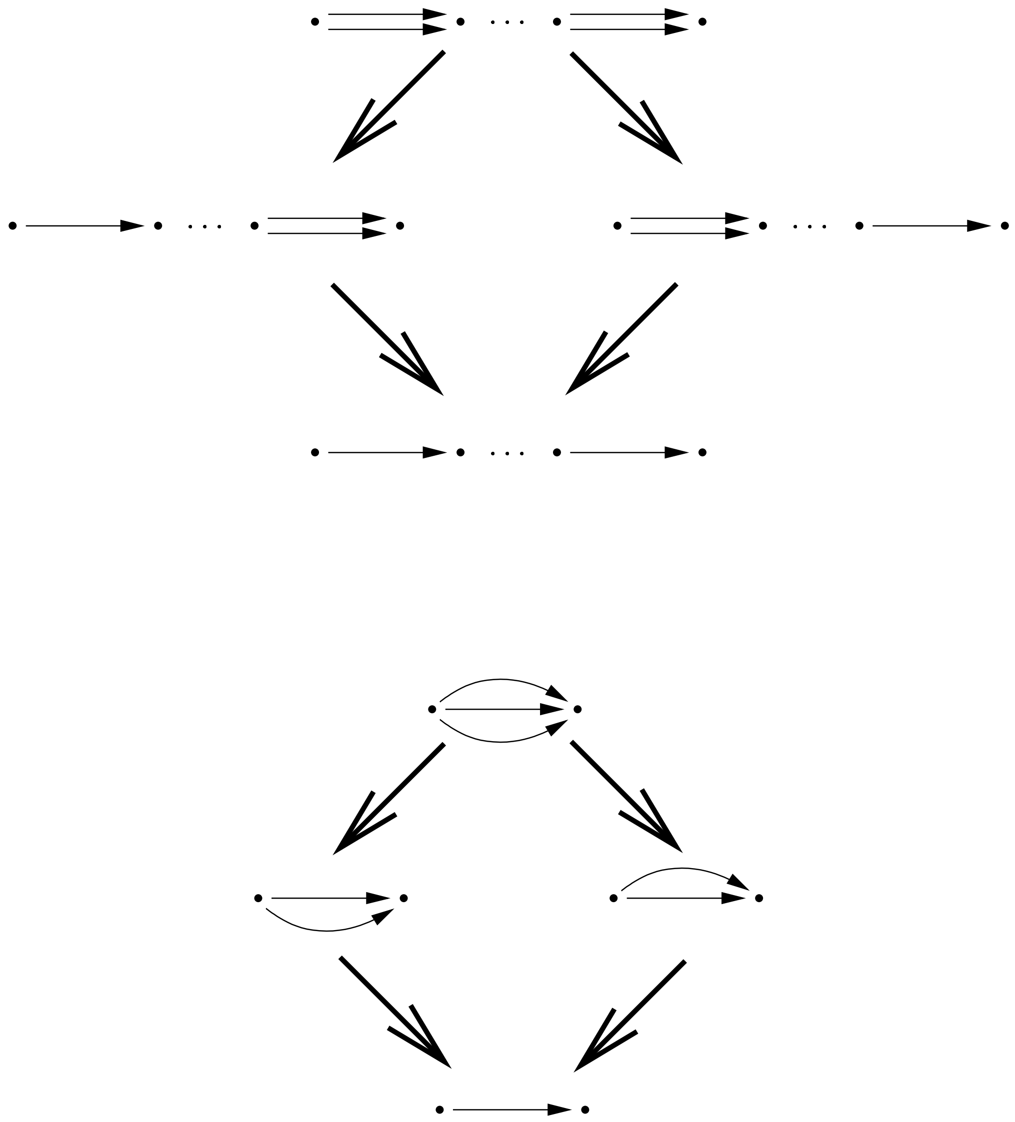


Abbildung 3.4: Lokale Konfluenz beim Doppelkanten-Beispiel

Beweis: Angenommen, es gäbe zwei Normalformen z_1 und z_2. Aufgrund der Konfluenzeigenschaft lassen sich beide Ableitungslinien zu einem z zusammenführen. Da aber z_1 und z_2 Normalformen und damit irreduzibel sind, muß $z_1 = z_2 = z$ sein.

Solche Systeme heißen daher auch *eindeutig terminierend* und haben in der Praxis die angenehme Eigenschaft, daß man ausgehend von einem bestimmten Objekt einen beliebigen möglichen Ableitungsschritt wählen kann und unabhängig von dieser Entscheidung immer beim gleichen Endergebnis ankommen wird. Ebenso gilt:

Satz 3.9: (Überführbarkeit und Normalform)
Ist ein Reduktionssystem noethersch und konfluent, so gilt

$$\forall x, y \in Z:\ x \xleftrightarrow{*} y \ \Rightarrow\ \exists z \in Z:\ x \xrightarrow{!} z \wedge y \xrightarrow{!} z$$

◇

Ineinander überführbare Zustände besitzen also eine gemeinsame (eindeutige) Normalform.

$$\begin{tikzcd}
z_0 \arrow[dr, "*"] & & z_2 \arrow[dl, "*"'] \arrow[dr, "*"] & & z_4 \arrow[dl, "*"'] \arrow[dr, "*"] & & z_6 \arrow[dl, "*"'] & \cdots \\
& z_1 \arrow[d, "!"'] & & z_3 \arrow[d, "!"'] & & z_5 \arrow[d, "!"'] & & \\
& z_1' \arrow[dr, "0"'] & = & z_3' \arrow[dl, "0"] \arrow[dr, "0"'] & = & z_5' \arrow[dl, "0"] & & \\
& & z_2' & = & z_4' & & &
\end{tikzcd}$$

Abbildung 3.5: Zum Beweis des Satzes zu Überführbarkeit und Normalform

Beweis (s. Abb. 3.5): $z_0 \xleftrightarrow{*} z_n$ läßt sich in eine Folge von $z_0 \xrightarrow{*} z_1 \xleftarrow{*} z_2 \xrightarrow{*} z_3 \xleftarrow{*} \ldots$ zerlegen. Zu $z_1, z_3, z_5, \ldots$ bildet man die eindeutigen Normalformen $z_1', z_3', z_5', \ldots$. Aufgrund der Konfluenz müssen die getrennten Pfade von den $z_{2i} (i \in \mathbb{N})$ zu z_{2i-1} und z_{2i+1} zu einem z_{2i}' zusammenlaufen. Da $z_1', z_3', z_5', \ldots$ Normalformen sind, muß $z_1' = z_2' = z_3' = z_4', \ldots$ gelten.

3.5 Kategorielle Transformationssysteme als Reduktionssysteme

Es ist leicht zu sehen, daß die in Kapitel 2 vorgestellten Transformationen auf Basis von Kategorien ein regelbasiertes System entsprechend Definition 3.1 bilden. Man wählt Obj_C als Zustandsmenge Z und die direkte Ableitbarkeit gemäß Definition 2.21 als Relation $\longrightarrow$. Damit kann man die oben angeführten Überlegungen zu Terminierung, Konfluenz und Normalformen auch auf diese Transformationssysteme anwenden.

Ein besonderer Zusammenhang besteht zwischen lokaler Konfluenz und paralleler Unabhängigkeit. Satz 2.26 (Seite 45) besagt, daß bei parallel unabhängigen Transformationsschritten in PIT-Kategorien ein Objekt existiert, zu dem die Ableitungslinien wieder zusammengeführt werden können. Dies ist aber genau die Forderung der lokalen Konfluenz, so daß man formulieren kann:

Satz 3.10: (Parallele Unabhängigkeit und Konfluenz)
Sind in einem kategoriellen Transformationssystem, das die PIT-Bedingungen erfüllt, alle Paare (p_1, p_2) von Produktionen für alle möglichen Einbettungen g_l der linken Seite in ein Objekt parallel unabhängig, so ist das System lokal konfluent. ⋄

Kapitel 4

Grundlagen der Dateisynchronisation

In diesem Kapitel soll nun der Mechanismus vorgestellt werden, der der Dateisynchronisation nach dem Prinzip lokaler Kopien und ohne Referenzkopie zugrunde liegt. Nach einigen Betrachtungen zu den möglichen Beziehungen der Dateien untereinander folgen Abschnitte, in denen dargelegt wird, wie aus den Beziehungen ein Zustandsgraph aufgebaut wird und später in einen Aktionsgraphen umgeformt wird. Danach werden Terminierung und Konfluenz des Systems gezeigt.

4.1 Grundlagen

4.1.1 Voraussetzungen

Die Dateisynchronisation geht von den folgenden Voraussetzungen aus:

- Ein Dateibaum soll auf mehreren Rechnern vorgehalten werden und — abgesehen von zeitlichen Verzögerungen bis zu einem Abgleich — auf allen diesen Rechnern identisch sein.
- Auf allen Rechnern sollen Modifikationen an den Dateien vorgenommen werden können. Solche Änderungen sollen bei der Synchronisation an alle anderen Rechner übertragen werden.

- Auch das Neuerstellen und Löschen von Dateien kann als Modifikation aufgefaßt werden. In diesen Fällen soll die neue Datei auch auf den anderen Rechnern angelegt bzw. die gelöschte auch dort entfernt werden.

- Das zugrundeliegende Betriebssystem verwaltet für jede Datei einen Zeitstempel für den Zeitpunkt der letzten Modifikation. Dieser wird beim Anlegen und bei jeder Änderung immer auf die gerade aktuelle Systemzeit gesetzt.

- Die Zeitstempel des Betriebssystems sind verläßlich. Darunter ist vor allem zu verstehen, daß es keine Dateien mit zukünftigen Modifikationszeiten gibt, d.h. solchen, die größer als die aktuelle Zeit sind.

- Die Uhren der beteiligten Rechner sind synchronisiert, d.h. sie liefern zu einem Zeitpunkt (in gewissem Rahmen[1]) die gleiche Zeit.

Für die anschließenden Betrachtungen kann man folgende Vereinfachungen einführen:

- Es genügt, jeweils nur eine einzelne Datei zu betrachten. Die Überlegungen gelten analog auch für alle anderen Dateien des Dateibaums. Das Aufbauen des Zustandsgraphen und die Umwandlung in einen Aktionsgraphen wird also getrennt für alle vorhandenen Dateien des Baums durchgeführt.

- Man kann die Menge der betrachteten Rechner auf diejenigen einschränken, die zum Zeitpunkt des Abgleichs präsent sind, d.h. die funktionstüchtig sind und über eine nutzbare Netzwerkverbindung verfügen. Für welche der beteiligten Rechner dies gilt, wird zu Beginn einer Synchronisation festgestellt.

- Rechner, die diese Voraussetzung nicht erfüllen, werden bei dem vorliegenden Dateiabgleich ignoriert und können an einem späteren Abgleich teilnehmen. Bei diesem wird dann der Dateibaum auch bei ihnen auf den aktuellen Stand gebracht beziehungsweise Dateien, die mittlerweile auf ihnen modifiziert wurden, dem Rest des Systems bekannt gemacht.

---

[1] Exakt synchronisierte Uhren sind in der Praxis leider nicht möglich. Für den Dateiabgleich ist aber ein Gleichlauf innerhalb gewisser Genauigkeitsgrenzen durchaus ausreichend. Die meisten Betriebssysteme verwalten die Dateizeitstempel nur mit einer Genauigkeit von einer Sekunde, so daß Abweichungen unterhalb 0.5 Sekunden für diese Zwecke vernachlässigbar sind. Vgl. auch Kapitel 6.2.

4.1.2 Einfache Dateibeziehungen

Das Verfahren der Dateisynchronisation beruht im wesentlichen auf den Modifikationszeiten der Dateien, die vom Betriebssystem verwaltet werden. Bei jedem schreibenden Zugriff auf eine Datei wird dieser Zeitstempel aktualisiert und auf die aktuelle Zeit gesetzt. Praktisch alle geläufigen Betriebssysteme kennen einen derartigen Zeitstempel.

Durch einen Vergleich der Modifikationszeiten kann man also einfach erkennen, ob eine Datei neuer als eine andere ist. Abbildung 4.1 verdeutlicht diesen trivialen Sachverhalt.

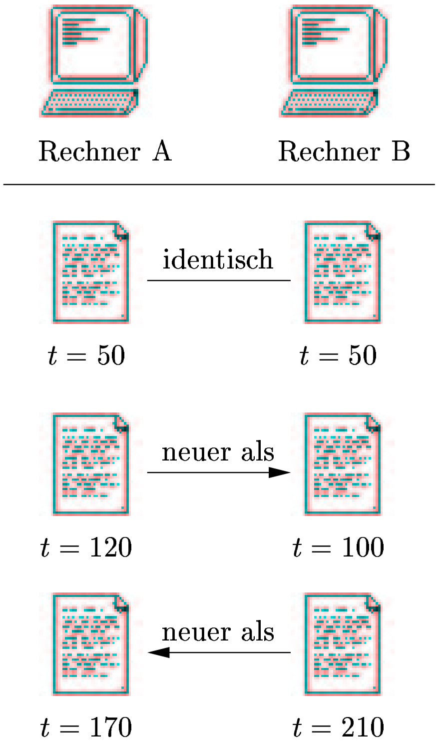


Abbildung 4.1: Beziehungen zweier Dateien nur mit Modifikationszeiten

4.1.3 Erkennen neuer und gelöschter Dateien

Bezieht man nun aber den Fall nicht vorhandener Dateien ein, so ergibt sich eine unklare Situation. Eine nicht existierende Datei hat keine Modifikationszeit, dementsprechend kann nichts verglichen werden. Ist eine bestimmte Datei auf Rechner A vorhanden, auf Rechner B dagegen nicht, so läßt sich nicht unterscheiden, ob diese Datei nun kürzlich auf A angelegt wurde oder ob sie vorher auf beiden existierte und auf B gelöscht wurde (Abb. 4.2). In letzterem Fall müßte der Abgleich sie auch auf A entfernen, andernfalls nach B kopieren.

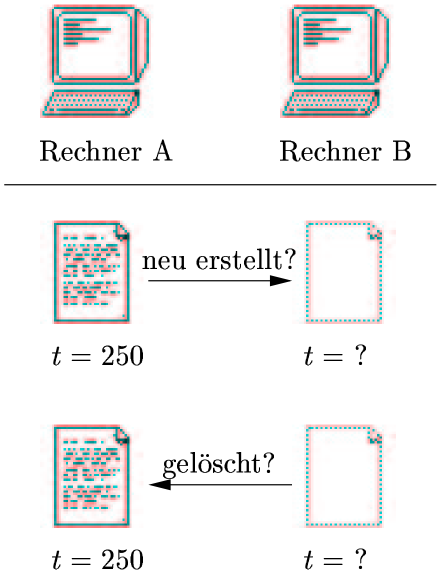


Abbildung 4.2: Beziehungen mit einer gelöschten Datei

Diese Mehrdeutigkeit läßt sich auflösen, indem man sich zusätzlich den Zeitpunkt der letzten Sychronisation merkt. Dann ergibt sich aus dem Zeitstempel der existierenden Datei, ob diese neu angelegt wurde. Dies ist genau dann der Fall, wenn ihre Modifikationszeit größer (neuer) als der Zeitstempel des letzten Abgleichs ist.

Die Korrektheit obiger Aussage läßt sich einfach damit nachweisen, daß nach der letzten Synchronisation die Datei auf beiden Rechnern in gleichem Zustand war. Nach einem erfolgreichen Abgleich existierte die Datei entweder auf keinem Rechner, oder sie existierte auf allen und dann mit einem Zeitstempel vor dem letzten Synchronisationszeitpunkt.

Fall 1: Nach dem letzten Abgleich existierte die Datei nicht.
Wenn nun beim nächsten Abgleich eine Datei existiert, kann diese nur nach der Synchronisation angelegt worden sein und muß folglich einen neueren Zeitstempel tragen.

Fall 2: Die Datei existierte nach dem letzten Abgleich.
Dann muß ihr Zeitstempel kleiner oder gleich der Zeit dieses Abgleichs gewesen sein. Laut allgemeiner Voraussetzung sind Zeitstempel verläßlich und es können also keine „zukünftigen“ Zeiten auftreten. Eine solche Datei mit Zeit $\leq$ der letzten Synchronisation muß also bei dieser vorhanden gewesen sein und war durch den Abgleich auf allen Rechnern vorhanden.

Wenn sie nun bei der nächsten Synchronisation auf einem Rechner fehlt, muß sie dort in der Zwischenzeit gelöscht worden sein, und die Löschoperation soll durch den Abgleich übertragen werden.

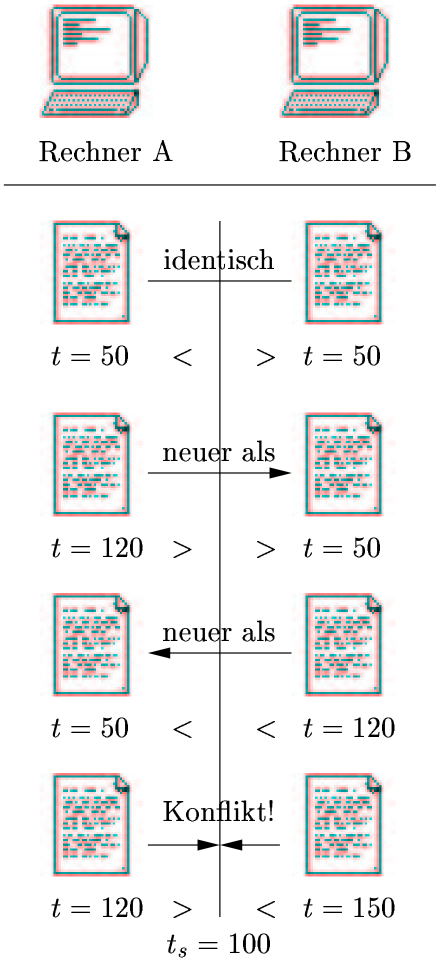


Abbildung 4.3: Dateibeziehungen unter Berücksichtigung der letzten Synchronisationszeit

Auch ohne nicht existierende Dateien ist der Zeitpunkt der letzten Synchronisation von Nutzen: Mit seiner Hilfe kann man feststellen, ob unabhängige Änderungen auf verschiedenen Rechnern stattgefunden haben. In diesem Fall werden mehrere Modifikationszeiten größer als die letzte Synchronisation sein (Abb. 4.3).

Mit der Speicherung des Zeitpunktes der letzten Synchronisation wurde eine Information in das System aufgenommen, die über die Betriebssystemdaten hinausgeht. Nach den selbstgesteckten Zielsetzungen (siehe 1.2) sollen diese auf ein Minimum beschränkt bleiben.

Allerdings muß auf jedem Rechner nur ein Zeitstempel für jeden anderen Rechner gespeichert werden. Die Anzahl der gespeicherten Daten ist also linear proportional zur Anzahl der beteiligten Rechner und ist völlig unabhängig von der Anzahl der abzugleichenden Dateien. Der Speicherplatzbedarf für die Zeitstempel hält sich also in vertretbaren Grenzen.

Der Fall, daß eine Datei zwischen zwei Synchronisationen gelöscht und dann wieder neu angelegt wird, ist leider für den Abgleichsmechanismus unsichtbar. Für diesen erscheint es so, als ob die Datei einfach modifiziert worden wäre. Im Endeffekt ist aber auch gar nichts anderes geschehen: Der ursprüngliche Inhalt wurde durch einen neuen ersetzt. Daß die Datei zwischenzeitlich gar nicht mehr existierte, ist unerheblich. Auch wäre das Endergebnis das gleiche, wenn zwischen dem Löschen und dem Neuanlegen eine Synchronisation stattgefunden hätte. Dann wird die Datei zwar temporär auch auf den anderen Rechnern gelöscht, später aber die neu angelegte Datei auch zu diesen übertragen.

Ein weiterer Sonderfall ist es, wenn eine Datei auf einem Rechner gelöscht, auf einem anderen jedoch modifiziert wird. In diesem Fall wird die Modifikationszeit der geänderten Datei größer als die Zeit der letzten Synchronisation sein. Dies führt zu der Fehlinterpretation, daß die geänderte Datei als neu angesehen wird.

Die Frage ist, was das korrekte Vorgehen wäre. Im Grunde gesehen liegt ein Konflikt vor: Auf verschiedenen Rechnern wurden unterschiedliche Operationen an der Datei vorgenommen, einmal eine Änderung und einmal das Löschen. Dies sollte so behandelt werden wie andere Konflikte (s. 5.1), nur daß hier keine automatische Bereinigung möglich ist.

Probleme bereitet es allerdings, diesen Fall zu erkennen. Dazu wäre es nötig zu wissen, ob die Datei bereits bei der letzten Synchronisation existiert hat. Nur dann läßt sich die Änderung von einer neuen Datei unterscheiden. Um dies zu implementieren, müßte man allerdings auf jedem Rechner eine Liste der Dateien ablegen, die bei der letzten Synchronisation existierten. Diese Liste würde nun aber einiges an Speicherplatz beanspruchen. Die Anzahl der Einträge entspricht der Anzahl Dateien im Dateibaum und kann durchaus recht groß werden. Auch muß man — wenn man nicht auf betriebssystemspezifische Indexnummern o.ä. zurückgreifen will — den Namen jeder Datei speichern, was den Speicherbedarf zusätzlich in die Höhe treibt.

Aus diesen Gründen wurde in der aktuellen Implementierung auf einen solchen Test verzichtet. Die Konsequenz ist, daß das Löschen einer Datei ignoriert wird, wenn sie gleichzeitig auf einem anderen Rechner geändert wurde. Die ausgeführte Aktion ist das Verbreiten der geänderten Datei. Insofern gehen wenigstens keine

Daten verloren, wie dies bei fälschlichem Löschen der Datei auf allen Rechnern der Fall wäre.

4.2 Aufbau des Zustandsgraphen

Der Dateiabgleich funktioniert über die Ableitung von Aktionen (Kopieren/Löschen) aus den vorgefundenen Relationen zwischen den Dateien untereinander und dem Zeitpunkt der letzten Synchronisation. Das Mittel hierzu sind Graphen, die aus den vorhandenen Daten aufgebaut und dann mit einem Graphtransformationssystem umgeformt werden.

Es wird jeweils ein eigener Graph für jede Datei in dem zu synchronisierenden Dateibaum aufgebaut. Man kann diese Einzelgraphen auch als Teile eines großen, nicht zusammenhängenden Graphen für den gesamten Dateibaum interpretieren. Im folgenden soll jedoch zur Vereinfachung der Darstellungen die Sicht auf jeweils eine Datei beschränkt werden.

Ausgehend von den Beziehungen korrespondierender Dateien auf jeweils zwei Rechnern wie in Abb. 4.3 auf Seite 63 kann man einen *Zustandsgraphen* aufbauen, der die Verhältnisse der Modifikationszeiten auf allen Rechnern angibt. Die Informationen über die Beziehungen werden in einer Phase vor der Erstellung des Graphen gesammelt.

Es gibt Rechner, auf denen die betrachtete Datei existiert, und solche, auf denen sie nicht existiert. Deshalb werden im Zustandsgraphen die den Rechnern entsprechenden Knoten mit **ex** bzw. **nex** markiert. Das Alphabet zur Knotenbeschriftung ist dementsprechend

$$L_V = \{\mathbf{ex}, \mathbf{nex}\}$$

In den Abbildungen werden die Knotenarten manchmal anschaulicher durch Dateisymbole dargestellt, die bei nicht existierenden Dateien grau und ohne Text sind.

Die teilweise in den Abbildungen vermerkten Modifikationszeiten sind Anmerkungen zum leichteren Verständnis. Sie sind nicht Bestandteil des Graphen und stellen keine Markierung dar. Alle benötigten Informationen werden bereits durch die Kantenmarkierung ausgedrückt, eine weitere Speicherung der tatsächlichen Zeitstempel nach dem Aufbau des Zustandsgraphen ist nicht erforderlich.

Bei den Kanten gibt es zwei verschiedene Markierungen, eine für Kanten, die eine „neuer als“-Relation repräsentieren, und eine für Konflikte. Das Kantenalphabet im Zustandsgraphen kann man also als

$$L_E = \{\textbf{neuer}, \textbf{Konflikt}\}$$

festlegen.

Da ein Konflikt symmetrisch ist (er besteht in beiden Richtungen), werden für ihn zwei Kanten in unterschiedliche Richtungen in den Zustandsgraphen eingetragen. In den Abbildungen werden die antiparallelen Doppelkanten als ungerichtete Kante dargestellt, die sie auch modellieren sollen. Diese Ersetzung hat keine weiteren Auswirkungen auf die Darstellung der Graphen oder die später folgenden Transformationsregeln, da auch dort für eine ungerichtete Kante zwei entgegengesetzt gerichtete Kanten mit gleicher Markierung zu lesen sind.

Der Aufbau des Zustandsgraphen erfolgt nun so, daß für jeden beteiligten Rechner ein Knoten eingeführt und entsprechend der Existenz der Datei auf ihm markiert wird. Danach wird für jedes Paar von unterschiedlichen Rechnern eine oder zwei Kanten mit entsprechender Markierung angelegt, wenn sie nicht in der „identisch“-Relation stehen.

4.3 Analysephase

Im weiteren Verlauf werden folgende Informationen verwendet, die aus der Analysephase stammen:

Definition 4.1: (Bezeichnungen)

$$\begin{aligned}
R &= \text{Menge der teilnehmenden Rechner}\\
ex(r) &= \begin{cases} 0 & \text{falls die Datei auf } r \in R \text{ nicht existiert}\\ 1 & \text{falls die Datei auf } r \in R \text{ existiert}\end{cases}\\
modt(r) &= \begin{cases} \text{Modifikationszeit der Datei auf } r \in R & \text{wenn } ex(r) = 1\\ \text{undefiniert} & \text{sonst}\end{cases}\\
ls(r_1, r_2) &= \begin{cases} \text{letzte Synchronisationszeit von } r_1 \text{ und } r_2 & \text{wenn } r_1 \neq r_2\\ \text{undefiniert} & \text{sonst}\end{cases}\\
& (r_1, r_2 \in R)
\end{aligned}$$

◇

Die Definition von ls ist symmetrisch und es gilt $\forall r_i, r_j \in R, r_1 \neq r_2 : \; ls(r_i, r_j) = ls(r_j, r_i)$. Damit kann formal der markierte Zustandsgraph (V, E, s, t, l_V, l_E) definiert werden:

Definition 4.2: (Zustandsgraph)

$$\begin{aligned}
V &= R \\
E &= \{(r_1, r_2) \in V \times V \mid r_1 \neq r_2 \;\wedge\; kb(r_1, r_2)\} \\
kb(r_1, r_2) &= (ex(r_1) = 1 \;\wedge\; modt(r_1) > ls(r_1, r_2)) \vee \\
& \;\; (ex(r_1) = 0 \;\wedge\; ex(r_2) = 1 \;\wedge\; modt(r_2) < ls(r_1, r_2))) \\
s(r_1, r_2) &= r_1 \\
t(r_1, r_2) &= r_2 \\
l_V(r) &= \begin{cases} \textbf{nex} & \text{wenn } ex(r) = 0 \\ \textbf{ex} & \text{wenn } ex(r) = 1 \end{cases} \\
l_E(r_1, r_2) &= \begin{cases} \textbf{Konflikt} & \text{wenn } kb(r_1, r_2) \;\wedge\; ex(r_2) = 1 \;\wedge\; modt(r_2) > ls(r_1, r_2) \\ \textbf{neuer} & \text{wenn } kb(r_1, r_2) \;\wedge\; (ex(r_2) = 0 \vee modt(r_2) \leq ls(r_1, r_2)) \\ \text{undefiniert} & \text{sonst} \end{cases}
\end{aligned}$$

◇

Die Kanten werden als geordnete Paare von Knoten repräsentiert, obwohl die Darstellung eines Graphen mit Quell- und Zielkantenfunktionen s und t verwendet wird. Dies geschieht, um konsistent mit Definition 2.13 und den darauf aufbauenden Konstruktionen aus Kapitel 2 zu bleiben. Die Kantenfunktionen s und t werden damit zu einfachen Projektionen.

Der Aufbau von E und l_E erfolgt anhand der Entscheidungstabelle 4.1. Dort erkennt man, daß drei der vier Fälle, in denen eine Kante (r_1, r_2) zu generieren ist, von der Bedingung $ex(r_1) = 1 \wedge modt(r_1) > ls(r_1, r_2)$ abgedeckt sind. Der dritte Fall kann hier subsummiert werden, weil auf $ex(r_2) = 1$ verzichtet wird. Man beachte, daß das damit eventuell undefinierte $modt(r_2)$ in dieser Alternative nicht verwendet wird. Der vierte Fall wird separat in die Kantenbedingung kb aufgenommen.

Die Markierung einer erzeugten Kante ist in fast allen Fällen **neuer**, außer wenn $ex(r_2) = 1 \wedge modt(r_2) > ls(r_1, r_2)$. Dies wird in der Bedingung in der Definition von l_E so ausgedrückt. Die Kanten in umgekehrter Richtung werden jeweils bei der Betrachtung des Knotenpaares (r_2, r_1) behandelt und müssen in der Tabelle nicht berücksichtigt werden. Erwähnenswert ist aber, daß Konfliktkanten in

$ex(r_1)$	$ex(r_2)$				Kante (r_1, r_2)	Markierung
1	1	$t_1 < ls$,	$t_2 < ls$	$\Rightarrow$	nein	–
1	1	$t_1 > ls$,	$t_2 < ls$	$\Rightarrow$	ja	**neuer**
1	1	$t_1 < ls$,	$t_2 > ls$	$\Rightarrow$	nein	–
1	1	$t_1 > ls$,	$t_2 > ls$	$\Rightarrow$	ja	**Konflikt**
1	0	$t_1 < ls$	–	$\Rightarrow$	nein	–
1	0	$t_1 > ls$	–	$\Rightarrow$	ja	**neuer**
0	1	–	$t_2 < ls$	$\Rightarrow$	ja	**neuer**
0	1	–	$t_2 > ls$	$\Rightarrow$	nein	–
0	0	–	–	$\Rightarrow$	nein	–

Tabelle 4.1: Entscheidungstabelle für Einführung von Kanten (Abkürzungen: $t_1 := modt(r_1), t_2 := modt(r_2), ls := ls(r_1, r_2)$)

beiden Richtungen generiert und mit **Konflikt** markiert werden, weil die Bedingungen dafür symmetrisch in r_1 und r_2 sind.

Hier ein einfaches Beispiel: Angenommen, das Sammeln der benötigten Informationen bei vier teilnehmenden Rechnern $R = \{A, B, C, D\}$ ergibt die Daten von Tabelle 4.2. Eine graphische Darstellung findet sich in Abb. 4.4. Der Zeitpunkt der letzten Sychronisation zweier Rechner wurde jeweils an den Balken in Mitte der Verbindungslinie vermerkt. In der Abbildung wurden bereits die Dateibeziehungen gemäß Abb. 4.3 eingetragen, ein Pfeil steht also für eine „neuer als“-Beziehung.

ex	Modifikationszeiten	letzte Synchronisation
		$ls(A, B) = 5$
1	$modt(A) = 6$	$ls(A, C) = 5$
1	$modt(B) = 1$	$ls(A, D) = 5$
1	$modt(C) = 3$	$ls(B, C) = 6$
1	$modt(D) = 1$	$ls(B, D) = 2$
		$ls(C, D) = 2$

Tabelle 4.2: Datentabelle für erstes Beispiel

Baut man gemäß den oben angeführten Regeln einen Zustandsgraphen auf, so ergibt sich für alle Rechner die Markierung **ex**, da die betrachtete Datei auf allen existiert. Es werden auch für fast alle Rechnerpaare Kanten eingeführt, da zwischen ihnen eine „neuer als“-Relation besteht. Nur zwischen B und D entfällt die Kante. Das Ergebnis ist in Abb. 4.5 dargestellt.

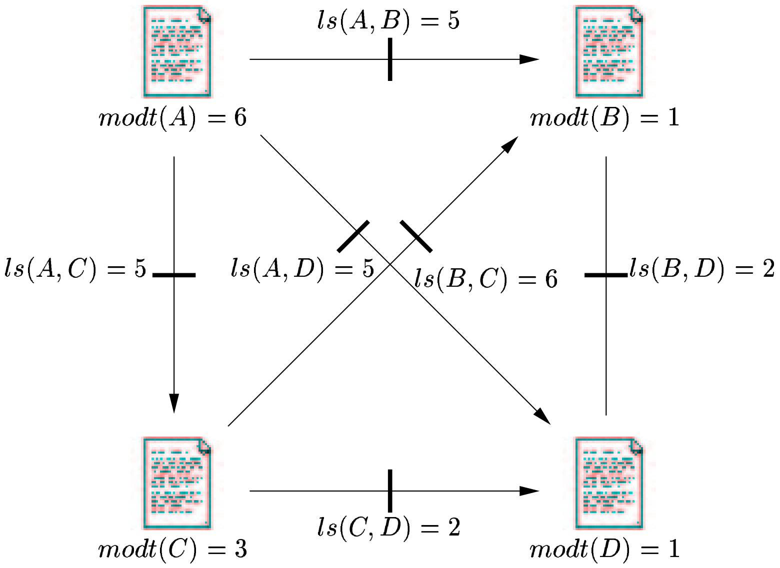


Abbildung 4.4: Dateibeziehungen beim ersten Beispiel

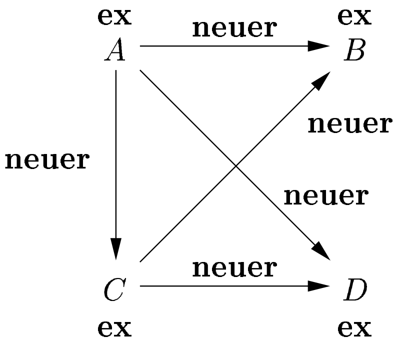


Abbildung 4.5: Zustandsgraph des ersten Beispiels

Ein weiteres Beispiel soll einen Rechner beinhalten, auf dem die betrachtete Datei nicht existiert. Die Informationssammlung für die drei Rechner $R = \{A, B, C\}$ ist in Tabelle 4.3 angegeben.

ex	Modifikationszeiten	letzte Synchronisation
1	$modt(A) = 1$	$ls(A,B) = 2$
1	$modt(B) = 1$	$ls(A,C) = 2$
0	$modt(C) = \text{undef}$	$ls(B,C) = 2$

Tabelle 4.3: Datentabelle für zweites Beispiel

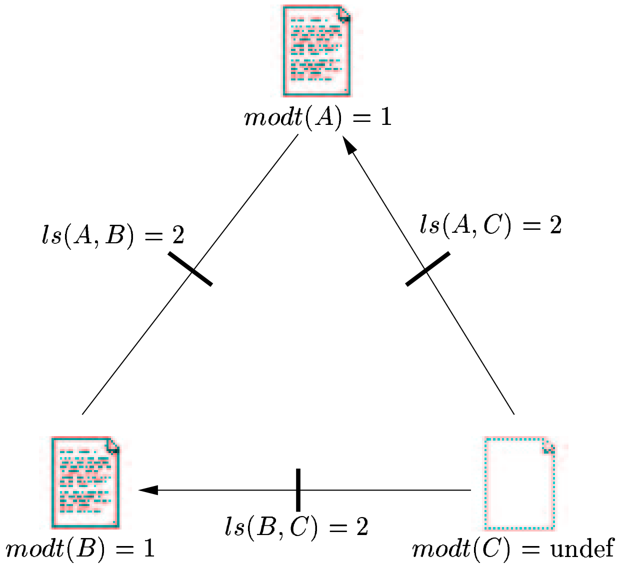


Abbildung 4.6: Dateibeziehungen beim zweiten Beispiel

Die letzte Synchronisation aller beteiligten Rechner fand zum Zeitpunkt 2 statt. Zu diesem hatte die Datei auf allen Rechnern die Modifikationszeit 1. Beim aktuellen Abgleich existiert die Datei auf Rechner C nicht mehr. Aufgrund der Definitionen ergeben sich **neuer**-Kanten von C zu A und B.

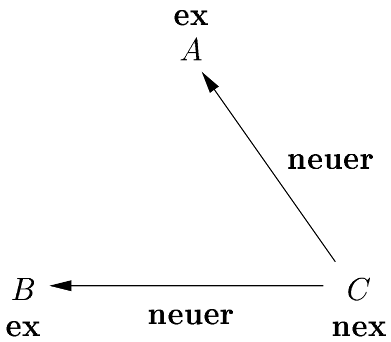


Abbildung 4.7: Zustandsgraph des zweiten Beispiels

4.4 Umformung in einen Aktionsgraphen

Nachdem der Zustandsgraph wie im vorangegangenen Abschnitt beschrieben erstellt wurde, kann er mit einem Graphtransformationssystem in einen Aktionsgraphen umgeformt werden. Ein solches Transformationssystem wird in diesem Abschnitt vorgestellt.

Im Aktionsgraphen gibt es erweiterte Markierungen von Knoten und Kanten:

$$\begin{aligned} L_V &= \{\mathbf{ex}, \mathbf{nex}, \mathbf{del}\} \\ L_E &= \{\mathbf{neuer}, \mathbf{kopieren}, \mathbf{Konflikt}, \mathbf{equiv}\} \end{aligned}$$

Ein mit **del** markierter Knoten bedeutet, daß die Datei auf dem entsprechenden Rechner gelöscht werden soll. Eine **kopieren**-Kanten steht für das Kopieren der Datei vom Quell- zum Zielknoten.

equiv-Kanten sind wie **Konflikt**-Kanten ihrer Natur nach ungerichtet und werden mittels zweier entgegengesetzt gerichteter und gleich markierter Kanten modelliert. Sind Knoten über eine solche Kante verbunden sagt dies aus, daß von allen solchen Knoten **kopieren**-Kanten zu einem weiteren Knoten existieren werden und es gleichgültig ist, von welchem Rechner die Datei kopiert wird (vgl. Anschnitt 4.4.3).

Der im letzten Abschnitt gewonnene Zustandsgraph wird mit den im folgenden beschriebenen Transformationsregeln so lange umgeformt, bis keine weiteren Regeln mehr anwendbar sind. In Abschnitt 4.5 wird gezeigt werden, daß nicht unendlich viele Transformationen möglich sind und sie aufgrund der Konfluenz des Systems auch zu einem eindeutigen Endzustand führen (vgl. auch Abschnitt 3.4)

Ein Aktionsgraph befindet sich in einem angestrebten gültigen Finalzustand,

- wenn er entweder Konfliktkanten enthält; dann wird die in Abschnitt 5.1 beschriebene Konfliktbehandlung angestoßen.
- oder wenn er keine **Konflikt**- und **neuer**-Kanten mehr enthält, sondern nur noch mit **kopieren** oder **equiv** markierte Kanten. In diesem Fall ist der Abgleich für die betrachtete Datei möglich.

Der Finalzustand des Aktionsgraphen existiert eindeutig, wie sich aus der in Abschnitt 4.5 bewiesenen Terminierung und Konfluenz ergibt (vgl. auch Abschnitt 3.4).

Am Ende noch vorhandene **ex**- und **nex**-Knoten könnten mit zusätzlichen Regeln in eine **notdel**-Markierung umgewandelt werden, was jedoch unnötig ist. Beide Markierungen werden einfach als „auf diesem Rechner nicht löschen “ interpretiert.

Nach der Umformung in einen konfliktfreien Aktionsgraphen können die in ihm angegebenen Aktionen (kopieren und löschen) ausgeführt werden. Zu beachten ist noch die Modifikationszeit, die kopierte Dateien erhalten sollen. Dies wird in Abschnitt 4.6 näher behandelt. In 5.4.2 wird auch das genauere Vorgehen zum Ausführen der Aktionen beschrieben, insbesondere die Implikationen, die sich aus der Tatsache ergeben, daß der Synchronisationsvorgang selbst Zeit beansprucht, in der weitere Änderungen an den Dateien vorgenommen werden könnten.

Die Abbildungen 4.8 und 4.9 auf den nächsten Seiten zeigen alle Regeln des Transformationssystems im Überblick. Sie sollen im folgenden einzeln vorgestellt und genauer erläutert werden. In der Darstellung der Regeln definieren gleiche Knotennummern in verschiedenen Graphen die Morphismen zwischen diesen.

Es ist bei den Regeln nicht besonders vermerkt, daß alle eine Anwendbarkeitsbedingung besitzen, die injektive Einbettung fordert. Weitere Bedingungen werden als $|in| < n$ oder $|out| < n \quad (n \in \mathbb{N})$ an den betroffenen Knoten geschrieben. Sie bedeuten, daß der Knoten weniger als n ein- bzw. auslaufende Kanten besitzen muß, wobei aber mit **equiv** markierte Kanten ausgenommen sind. Auf die explizite Erwähnung von **equiv** in den Bedingungen wurde der Übersichtlichkeit halber verzichtet. Die Realisierung derartiger Anwendungsbedingungen wurde in Abschnitt 2.6 beschrieben. Formulierungen wie $|in| = 0$ oder $|in| \leq 1$ können trivial in die oben angegebene Standardform der Bedingungen umgesetzt werden.

4.4.1 Regeln (1a) und (1b): Transitivität von neuer-Kanten

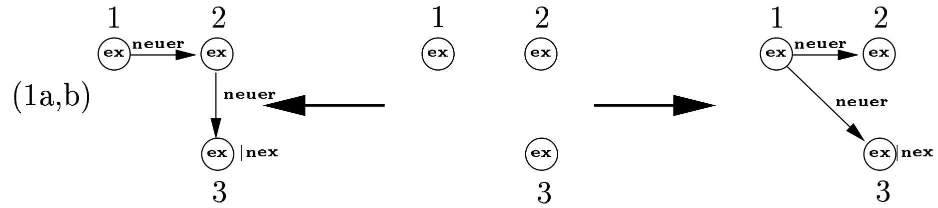


Abbildung 4.10: Regeln (1a) und (1b)

Wenn die Datei bei Rechner 1 neuer als die auf Rechner 2 ist und diese wiederum neuer als die von 3, so muß insgesamt die Datei von Rechner 1 die neueste sein und sollte auch nach 3 kopiert werden. Daher wird der Quellknoten der Kante von 2 nach 3 zu 1 umgelegt. Anschaulich bedeutet dies, daß später die Datei von 1 nach 3 und nicht von 2 nach 3 kopiert werden soll.

Die gleichen Überlegungen gelten für den Fall, daß die Datei auf Rechner 3 nicht existiert. Auch dann soll von 1 nach 3 und nicht von 2 nach 3 kopiert werden.

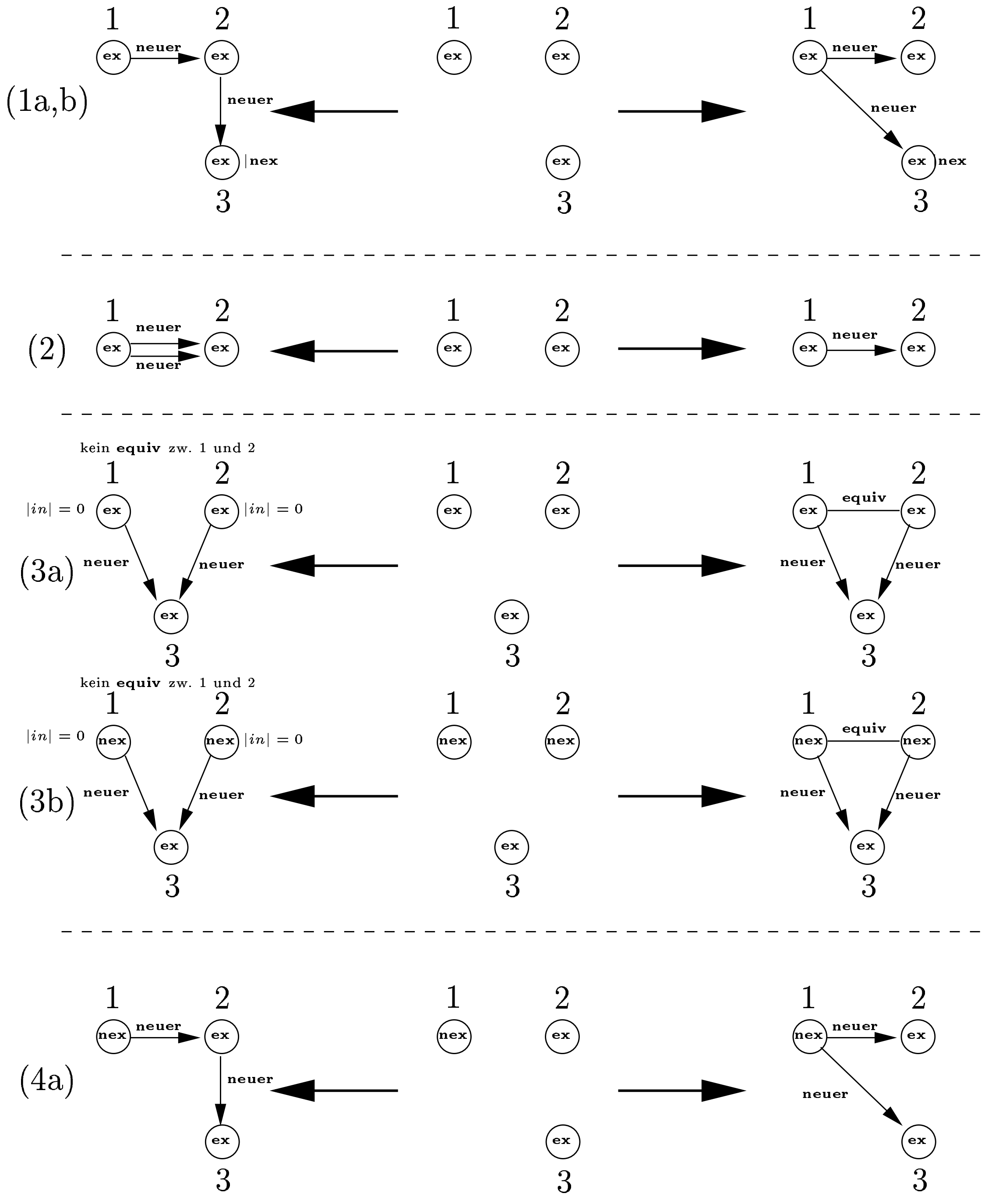


Abbildung 4.8: Transformationsregeln im Überblick (Teil 1)

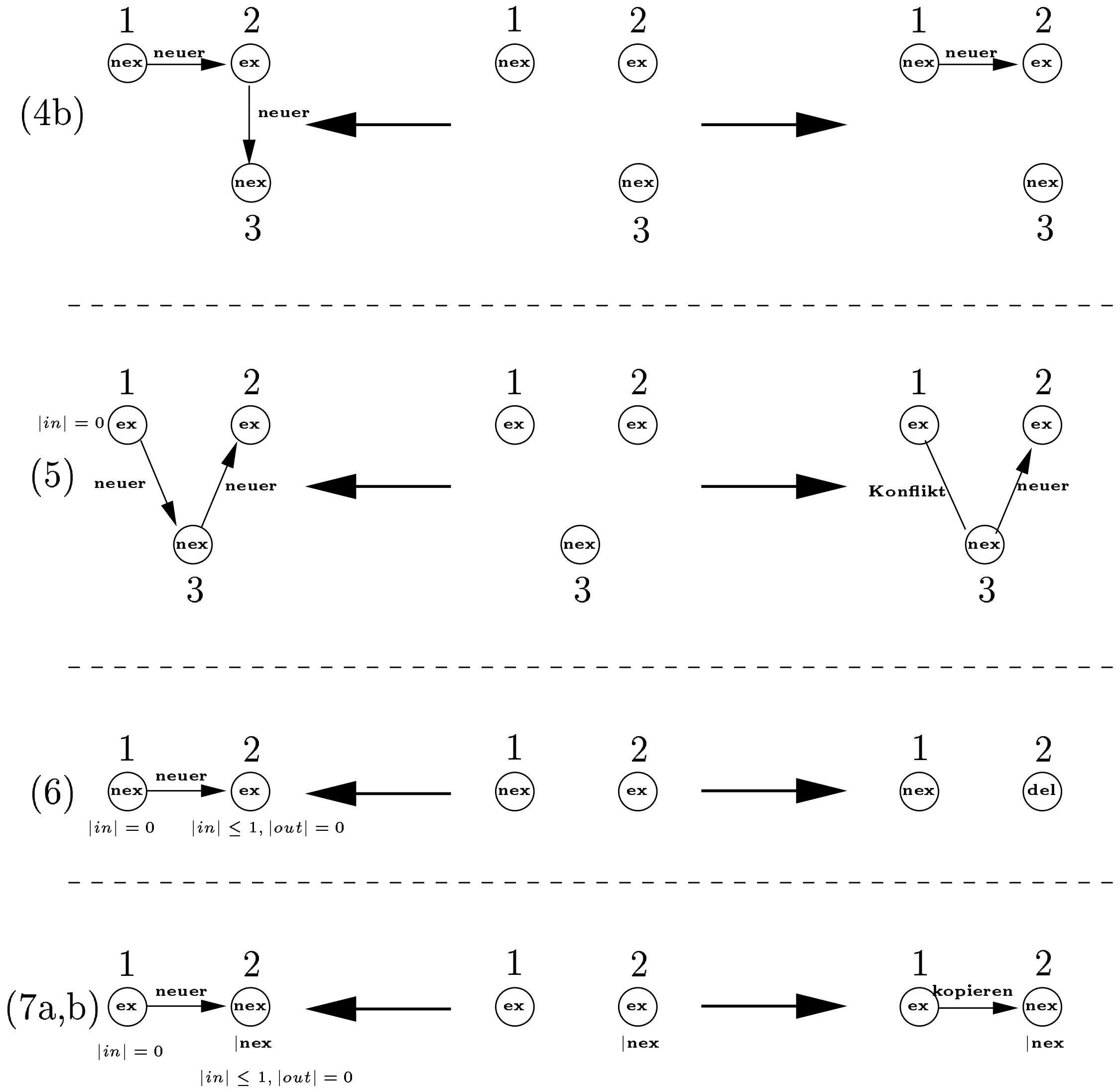


Abbildung 4.9: Transformationsregeln im Überblick (Teil 2)

Regel (1b) ist strukturell die gleiche wie (1a), nur daß Knoten 3 eine andere Markierung besitzt. Im Diagramm ist dies als „|**nex**“ angedeutet.

Mit einem strukturierten Markierungsalphabet (s. Abschnitt 2.3.4) wäre es möglich, die Markierung von Knoten 3 beliebig zu gestalten und damit die beiden Regelteile zusammenzufassen. Das geschah nicht, um Mehrdeutigkeiten bei der Markierung durch das Bilden einer kleinsten oberen Schranke zu vermeiden. Das gesamte Transformationssystem kommt ohne strukturiertes Alphabet aus, so daß dies nicht in Kauf genommen werden sollte, wenn nur Regelvarianten eingeführt werden müssen.

4.4.2 Regel (2): Löschen von Doppelkanten

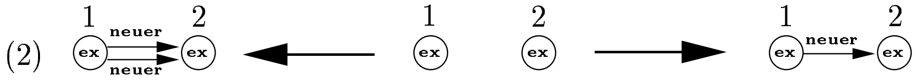


Abbildung 4.11: Regel (2)

Bei Anwendung von Regel (1) können doppelte **neuer**-Kanten entstehen, nämlich wenn sich zwischen den Knoten 1 und 3 der Regel (1) bereits eine solche befand. Die doppelten Kanten sind redundant und können somit durch eine einzelne ersetzt werden.

4.4.3 Regeln (3a) und (3b): Äquivalente Dateiversionen

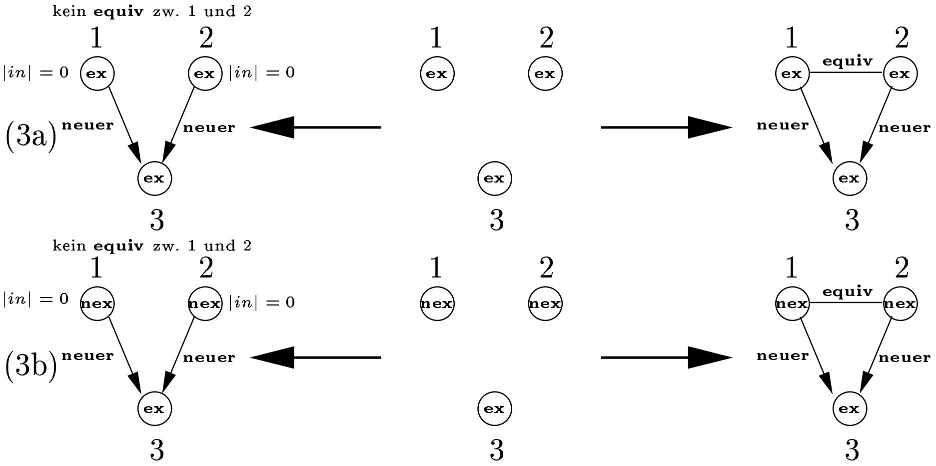


Abbildung 4.12: Regeln (3a) und (3b)

Wenn zwei **neuer**-Kanten von verschiedenen Rechnern 1 und 2 bei Knoten 3 zusammenlaufen und diese Rechner nicht mit einer weiteren Kante korreliert sind, so bedeutet das, daß 1 und 2 erst kürzlich abgeglichen wurden und damit die Datei von 1 nach 2 oder umgekehrt übertragen wurde. Bei dieser Synchronisation nahm Rechner 3 aber nicht teil, so daß nun jeweils auf 1 und 2 die neuere Dateiversion vorliegt. Es ist im Grunde genommen gleichgültig, von welchem der Rechner die Datei nach 3 kopiert wird.

Würde man aber einfach eine der beiden **neuer**-Kanten löschen, so wäre das System nicht mehr konfluent. Um das zu vermeiden, wird eine **equiv**-Kante zwischen Knoten 1 und 2 eingeführt, die später signalisiert, daß ein beliebiger der so verbundenen Knoten ausgewählt werden kann, um die Datei nach Rechner 3 zu kopieren.

Durch dieses Verfahren wird also die nichtdeterministische Auswahl zwischen den Rechern aus dem Transformationssystem herausgenommen und in die spätere Interpretation des Aktionsgraphen verschoben, um weiterhin eindeutige Normalformen zu erhalten.

Eine alternative Lösung wäre gewesen, doch eine beliebige der **neuer**-Kanten zu löschen, anhand der **equiv**-Kanten eine Äquivalenzrelation $\sim$ zwischen Aktionsgraphen zu definieren und dann nur noch Konfluenz modulo $\sim$ anzustreben. Dies hätte den Beweis der Konfluenz jedoch noch umfangreicher gemacht, da in allen Fällen auch äquivalente Graphen betrachtet werden müßten.

Auch kann es Gründe zur Auswahl bestimmer Rechner geben, die nicht in den Graphen repräsentiert sind, wie etwa die Entfernung zum Zielknoten in der Netztopologie, was Auswirkungen auf die Laufzeiteffizienz hat. Die endgültige Entscheidung, von welchem Rechner kopiert wird, bleibt also besser höheren Ebenen des Synchronisationssystems überlassen.

Eine ähnliche Situation wie bei Regel (3a) liegt vor, wenn die Datei auf den Rechnern 1 und 2 nicht mehr existiert. Es wurde beim Abgleich von 1 und 2 gleicher Zustand hergestellt, nur auf Rechner 3, der an diesem Abgleich nicht teilnahm, existiert noch eine alte Dateiversion, die ebenfalls gelöscht werden soll. Dieser Fall unterscheidet sich nur durch die Knotenmarkierungen und wird durch Regel (3b) behandelt.

Die Anwendbarkeitsbedingungen stellen sicher, daß diese Regel nicht angewandt wird, wenn es noch neuere Versionen der Datei gibt. Durch die Regeln (1) und (4) werden irgendwann die entsprechenden **neuer**-Kanten von einer anderen Quelle auf 3 zeigen lassen, zu der es keine einlaufenden Kanten gibt. Regel (3) findet also erst dann Verwendung, wenn die **neuer**-Kanten von dem Knoten mit den

„neuesten“ Versionen der Datei kommen. Zwischen diesen wird dann die **equiv**-Kante generiert.

Es muß noch erwähnt werden, daß bei den Regeln (3a) und (3b) über eine Anwendbarkeitsbedingung auch ausgeschlossen wird, daß sich zwischen Knoten 1 und 2 bereits eine **equiv**-Kante befindet. Dies ist notwendig, damit zwischen zwei Knoten nicht beliebig viele solcher Kanten generiert werden können.

4.4.4 Regeln (4a) und (4b): Löschen ist neueste Operation

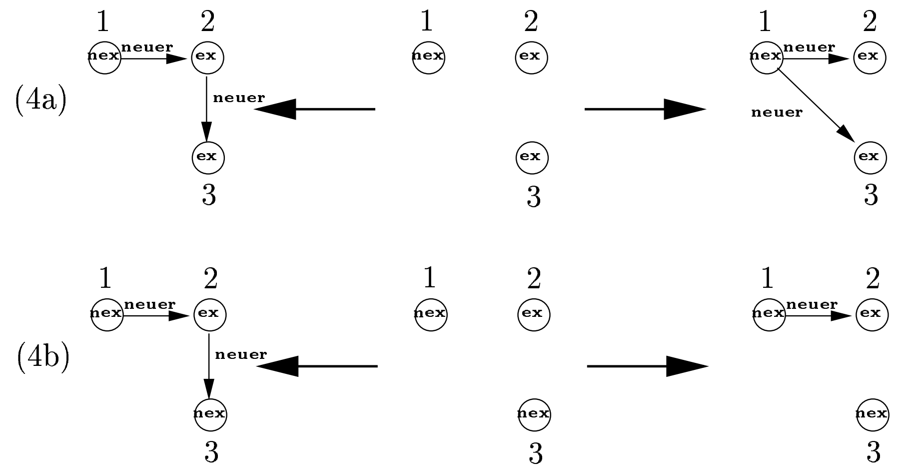


Abbildung 4.13: Regeln (4a) und (4b)

Beide Regeln drücken eine Transitivität von **neuer**-Kanten aus, die bei einem Knoten mit nicht existierender Datei starten. Sie sind analog zu Regel (1), nur daß die betrachtete Datei auf Rechner 1 nicht existiert und der zugehörige Knoten dementsprechend mit **nex** markiert ist.

Regel (4a) überträgt genauso wie (1) die **neuer**-Kante zwischen 2 und 3 zu 1 und 3. Bei Regel (4b), wo auch Knoten 3 mit **nex** markiert ist, wird nur die Kante von 2 nach 3 gelöscht, aber keine neue eingeführt. **neuer**-Kanten zwischen **nex**-Knoten hätten keine praktische Funktion (die Datei hat bereits identischen Zustand auf beiden Rechnern), und es wird später manchmal vorausgesetzt, daß keine solchen Kanten zwischen **nex**-Knoten existieren können. Sie treten aufgrund der Unvergleichbarkeit von **nex**-Knoten im Zustandsgraphen niemals auf und sollten deshalb nicht unnötig eingeführt werden.

4.4.5 Regel (5): Konflikt zwischen Neuerstellen und Löschen von Dateien

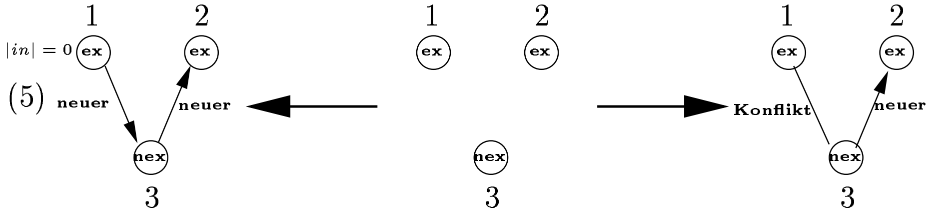
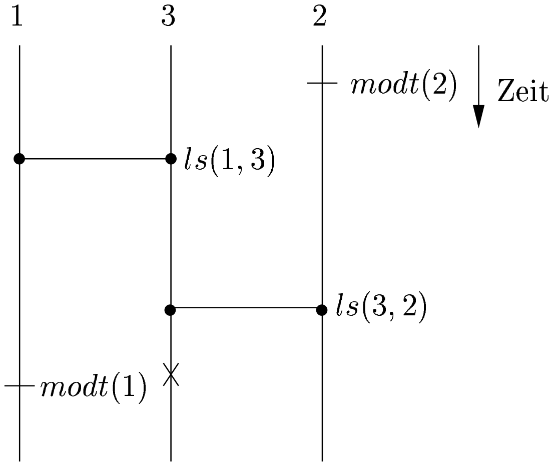


Abbildung 4.14: Regel (5)

Diese Regel behandelt einen indirekten Konflikt zwischen einer Änderung und einem Löschen, das ja auch eine Operation auf der Datei darstellt. Ein solcher Konflikt kann (wie in Abschnitt 4.1.3 auf Seite 64 bereits dargelegt) auf direkte Weise leider nicht erkannt werden. Wenn jedoch ein weiterer Rechner existiert, zu dem die Löschoperation noch hätte übertragen werden müssen, dies jedoch nicht geschah, weil keine Synchronisation in der Zwischenzeit stattfand, so kann man den indirekten Konflikt erkennen.

Man betrachte das zugehörige Zeitdiagramm:



Zum Zeitpunkt $ls(3,2)$ muß die Datei noch auf Rechner 3 existiert haben, denn sonst wäre sie von 2 nach 3 kopiert worden[2]. Wenn sie aber zu dieser Zeit auf beiden Rechnern vorhanden war, so muß sie nach der Synchronisation in gleichem Zustand gewesen sein und hatte $modt(3) \leq modt(2)$. Das Löschen der Datei auf 3

---

[2] Das gilt auch für den Fall eines unerkannten direkten Konflikts zwischen Ändern und Löschen (Abschnitt 4.1.3), da dabei die neue Datei bevorzugt wird.

muß also nach $ls(3,2)$ erfolgt sein und stellt folglich eine unabhängige Operation zu der Modifikation auf Rechner 1 dar.

Die Forderung der Anwendbarkeitsbedingung, daß keine weitere einlaufende Kante zu Knoten 1 existieren darf, stellt sicher, daß nach Möglichkeit erst Regeln (1) und (4) angewandt werden. Ist dies nicht mehr möglich, so wird Knoten 1 der mit der neuesten Version der Datei sein, und mit diesem soll der Konflikt signalisiert werden.

4.4.6 Regeln (6) und (7): Umwandlung in Aktionen

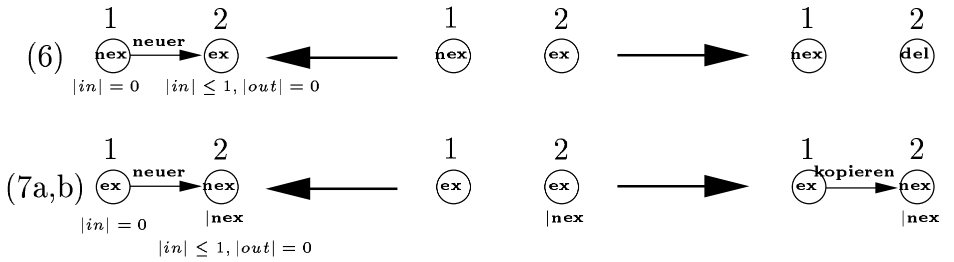


Abbildung 4.15: Regeln (6) und (7)

Dieser Satz von Regeln wandelt schließlich den transformierten Zustandsgraphen in den endgültigen Aktionsgraphen um. Die Anwendungsbedingungen fordern in allen Fällen, daß zu Knoten 1 keine einlaufenden Kanten existieren und von 2 keine auslaufenden. Die Regeln (6) und (7) werden also nur auf Pfade der Länge 1 angewandt. Dies verhindert verfrühte Anwendung dieser Regeln. Existieren weitere Kanten, so kann der Graph noch weiter transformiert werden und **neuer**-Kanten dürfen nicht gelöscht oder in **kopieren** umgewandelt werden. Sonst wäre eventuell die Anwendung anderer Regeln nicht mehr möglich.

Die Bedingung $|in| \leq 1, |out| = 0$ hätte auch auf andere Weise realisiert werden können. Wenn man den Knoten 2 im Klebegraphen wegläßt, so tritt er im Kontextgraphen nicht mehr auf. Wären dann weitere Kanten außer der von der linken Seite der Produktion mit ihm verbunden, so wäre der Kontextgraph kein gültiger Graph mehr und die Produktion könnte nicht angewendet werden. Beide Methoden sind im Ergebnis äquivalent. Die Alternative der Anwendbarkeitsbedingungen wurde nur gewählt, um ein einheitliches Aussehen der Regeln zu erreichen.

Regel (6) bearbeitet das Übertragen einer Löschoperation. Ist eine nicht existierende Datei **neuer** als eine existierende, so soll die Datei auf dem Zielrechner gelöscht werden. Die **neuer**-Kante wird anschließend nicht mehr benötigt, da die Aktion mit der neuen Knotenmarkierung vermerkt ist.

Die Regeln (7a) und (7b) verwandeln **neuer**-Kanten in **kopieren**-Kanten. Die Markierung des Zielknotens (**ex** oder **nex**) ist dabei unerheblich, daher zwei Ausführungen der Regel mit unterschiedlichen Markierungen.

4.5 Vollständigkeit, Terminierung und Konfluenz des Systems

4.5.1 Vollständigkeit

Die Vollständigkeit des Regelsystems ist die Eigenschaft, daß alle möglichen Zustandsgraphen in einen Aktionsgraphen umgeformt werden können. Sie ergibt sich aus einer vollständigen Fallunterscheidung über die Bestandteile des gültigen Graphen. Abbildung 4.16 zeigt die Gruppen der existierenden Fälle.

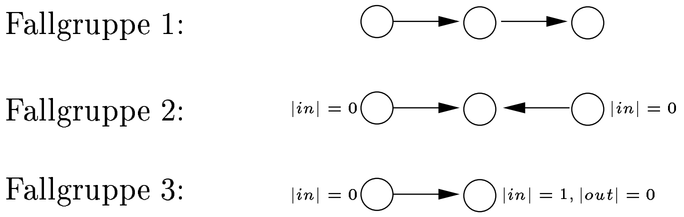


Abbildung 4.16: Fallgruppen für Vollständigkeitsnachweis

Folgendes läßt sich dabei von vornherein ausschließen: Zum einen können Fälle weggelassen werden, bei denen eine Kante zwei **nex**-Knoten verbindet. Solche Kanten werden laut der Definition des Zustandsgraphen in diesem nicht erzeugt und werden auch durch Regelanwendungen nicht eingeführt (siehe auch Abschnitt 4.4.4).

Des weiteren müssen nur **neuer**-Kanten betrachtet werden. Gibt es **Konflikt**-Kanten, so ist das Ergebnis immer ein gültiger Aktionsgraph, denn in diesem Fall werden keine weiteren Anforderungen gestellt. Ist ein Konflikt vorhanden, so werden später keine Aktionen ausgeführt, sondern eine Konfliktbehandlung. Kanten mit der Markierung **kopieren** können ebenso vernachlässigt werden, denn sie sind im finalen Aktionsgraphen erlaubt und nicht weiter transformierbar. Analoge Überlegungen gelten für **del**-Knoten. Auch **equiv**-Kanten werden von den

Regeln außer (3) ignoriert, und dort sieht die Anwendbarkeitsbedingung nur vor, daß keine mehrfachen **equiv**-Kanten erzeugt werden können.

Es bleiben also Diagramme übrig, die nur aus **neuer**-Kanten und **ex**- und **nex**-Knoten bestehen. Da die Kantenmarkierung eindeutig ist, wird sie im folgenden weggelassen, und innerhalb der Fallgruppen müssen nur noch die verschiedenen Kombinationen von Knotenmarkierungen betrachtet werden.

Zunächst zur Fallgruppe 1: In diese fallen Verkettungen von Kanten. Die möglichen Fälle sind:

$$\begin{array}{ccccccl}
\mathbf{ex} & \longrightarrow & \mathbf{ex} & \longrightarrow & \mathbf{ex} & \Rightarrow & \text{Regel (1a)} \\
\mathbf{ex} & \longrightarrow & \mathbf{ex} & \longrightarrow & \mathbf{nex} & \Rightarrow & \text{Regel (1b)} \\
\mathbf{ex} & \longrightarrow & \mathbf{nex} & \longrightarrow & \mathbf{ex} & \Rightarrow & \text{Regel (5)} \\
\mathbf{ex} & \longrightarrow & \mathbf{nex} & \longrightarrow & \mathbf{nex} & \Rightarrow & \text{(tritt nicht auf)} \\
\mathbf{nex} & \longrightarrow & \mathbf{ex} & \longrightarrow & \mathbf{ex} & \Rightarrow & \text{Regel (4a)} \\
\mathbf{nex} & \longrightarrow & \mathbf{ex} & \longrightarrow & \mathbf{nex} & \Rightarrow & \text{Regel (4b)} \\
\mathbf{nex} & \longrightarrow & \mathbf{nex} & \longrightarrow & \mathbf{ex} & \Rightarrow & \text{(tritt nicht auf)} \\
\mathbf{nex} & \longrightarrow & \mathbf{nex} & \longrightarrow & \mathbf{nex} & \Rightarrow & \text{(tritt nicht auf)}
\end{array}$$

Tabelle 4.4: Fallunterscheidung für Fallgruppe 1

In der zweiten Gruppe werden die Fälle behandelt, in denen zwei Kanten bei einem Knoten zusammenlaufen. Die Anwendbarkeitsbedingungen schreiben vor, daß beim linken und/oder rechten Knoten keine Kanten eintreffen dürfen. Gibt es solche Kanten, so liegt ein Pfad mit Mindestlänge 2 vor und es trifft ein Fall der Gruppe 1 zu. Auch kann die Kombination **ex** ⟶ **ex** ⟵ **nex** hier nicht auftreten. Rechner mit nicht existierender Datei stehen immer mit Kanten in Relation zu allen **ex**-Knoten, weil sich immer eine „neuer als"-Beziehung in der einen oder anderen Richtung ergibt. Und diese Kante zwischen den beiden äußeren Knoten kann nur mit anderen Regeln entfernt werden, wenn es eine weitere einlaufende Kanten gibt. Diese sind aber aufgrund der Anwendbarkeitsbedingungen nicht möglich.

$$\begin{array}{ccccccl}
\mathbf{ex} & \longrightarrow & \mathbf{ex} & \longleftarrow & \mathbf{ex} & \Rightarrow & \text{Regel (3a)} \\
\mathbf{ex} & \longrightarrow & \mathbf{nex} & \longleftarrow & \mathbf{ex} & \Rightarrow & \text{Regel (5)} \\
\mathbf{ex} & \longrightarrow & \mathbf{nex} & \longleftarrow & \mathbf{nex} & \Rightarrow & \text{(tritt nicht auf)} \\
\mathbf{nex} & \longrightarrow & \mathbf{ex} & \longleftarrow & \mathbf{nex} & \Rightarrow & \text{Regel (3b)}
\end{array}$$

Tabelle 4.5: Fallunterscheidung für Fallgruppe 2

Die anderen vier möglichen Kombinationen von Knotenmarkierungen sind symmetrisch zu den eben beschriebenen. Es verbleibt noch die Möglichkeit, daß zu einem Knoten genau eine eingehende Kante und keine abgehende existiert, was in Gruppe 3 fällt. Die Anwendbarkeitsbedingungen schreiben ebenso wie bei Gruppe 2 keine einlaufende Kante zu dem linken Knoten vor. Wie dort läge dann eine Verkettung und ein Gruppe-1-Fall vor.

ex	⟶	**ex**	⇒	Regel (7a)
ex	⟶	**nex**	⇒	Regel (7b)
nex	⟶	**ex**	⇒	Regel (6)
nex	⟶	**nex**	⇒	(tritt nicht auf)

Tabelle 4.6: Fallunterscheidung für Fallgruppe 3

Zusätzlich zu den obigen Überlegungen erzeugen alle diese Regeln außer (6) und (7) wieder einen Graphen, der der Definition eines Zustandsgraphen entspricht. Das heißt, es gibt keine Markierungen, die dort nicht erlaubt sind und es gibt weiterhin keine **neuer**-Kanten zwischen **nex**-Knoten. Damit ist die Voraussetzung für die Fallunterscheidung wieder gegeben und der Graph kann entsprechend weiter umgeformt werden.

Die weiteren Regeln greifen aufgrund ihrer Anwendbarkeitsbedingungen erst, wenn keine anderen Regeln mehr anwendbar sind. Sie dienen dazu, die restlichen **neuer**-Kanten zu entfernen, die im Aktionsgraphen nicht gestattet sind. Die Anwendbarkeitsbedingungen fordern, daß es keine einlaufenden Kanten beim Quellknoten geben darf. Dies kann jedoch durch Anwendung der anderen Regeln erreicht werden. Und oben wurde gezeigt, daß das für alle Kombinationen von Verkettungen möglich ist.

Natürlich kann es geschehen, daß in einem Fall wie in Abbildung 4.17 von einem Knoten Einzelkanten ausgehen, die in Gruppe 3 fallen, aber auch Ketten der Gruppe 1. Die obige Fallunterscheidung kann dann getrennt für die einzelnen Pfade ab dem obersten Knoten stattfinden. Die Umformungen können unabhängig voneinander stattfinden, da sie nur lokal wirken und auch keine Knoten löschen. Es ist auch unschädlich, wenn Einzelkanten wie im linken Teil bereits in **kopieren**-Kanten umgewandelt werden, bevor alle Kantenketten bearbeitet sind. Es ist also möglich, die Regeln (6) und (7) erst in einer zweiten Phase zu verwenden, wenn alle anderen Regeln nicht mehr anwendbar sind. Die Vollständigkeit der Regeln ist aber auch dann noch gegeben, wenn keine solche Einteilung in Phasen stattfindet.

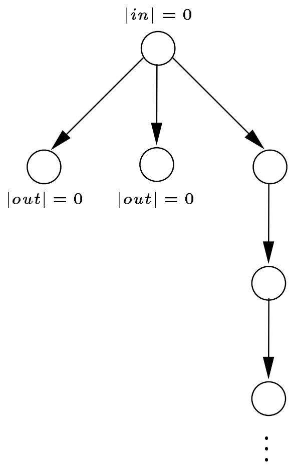


Abbildung 4.17: Beispiel einer Fallkombination

4.5.2 Terminierung

Die Überlegungen des vorherigen Abschnitts leiten bereits zu der Idee für einen Terminierungsnachweis hin: Die Regeln (1), (4) und (5) wandeln verkettete Kanten in nicht verkettete um. Die Summe aller Pfadlängen in dem Graphen sinkt daher bei jeder Anwendung. Bei den Regeln (2), (6) und (7) wird jeweils ein **neuer**-Kante entfernt.

Eine besondere Behandlung benötigt Regel (3): Hier werden keine Kanten gelöscht und auch kein Pfad verkürzt. Allerdings wird eine **equiv**-Kante eingeführt und dies kann aufgrund der Anwendbarkeitsbedingungen nur geschehen, wenn eine solche nicht bereits existiert. Da der Graph und somit die Knotenanzahl endlich sind, kann es nicht unendlich viele **equiv**-Kanten geben und Transformationen gemäß Regel (3) können nicht beliebig oft stattfinden.

All dies kann formalisiert werden, indem man — wie in Abschnitt 3.2 dargelegt — eine Abbildung $\varphi : Obj_{\mathcal{G}raph} \longrightarrow \mathbb{N}_0$ definiert. $\mathbb{N}_0$ besitzt mit $\leq$ eine wohlfundierte Ordnung, so daß es als Zielmenge geeignet ist. Kann man $\varphi(G_l) > \varphi(G_r)$ für jeden möglichen Ableitungsschritt $G_l \longrightarrow G_r$ zeigen, so vermindert sich die φ-Bewertung der Graphen in einer Ableitungsfolge bei jedem Schritt. Da φ nicht unter 0 sinken kann, ist der Fall unendlicher Ableitungsketten ausgeschlossen und das Transformationssystem muß folglich terminieren.

Sei die Menge aller Pfade P in einem Graphen $G = (V, E, s, t)$ definiert als

$$\begin{aligned} P \;=\; & \{(v_1, v_2, \ldots, v_n) \in V^* \mid n \in \mathbb{N} \backslash \{1\} \quad \wedge \\ & \quad (\forall 1 \leq i \leq n-1 \colon \exists e \in E \colon s(e) = v_i \wedge t(e) = v_{i+1}) \quad \wedge \end{aligned}$$

$$(\nexists e \in E : t(e) = v_0) \ \wedge \ (\nexists e \in E : s(e) = v_n) \ \}$$

und die Länge eines Pfades $p \in P$ als

$$l(p) = |p| - 1$$

Es ist einfach, die obige Definition in einem markierten Graphen $(V, E, l_V : V \longrightarrow L_V, l_E : E \longrightarrow L_E, s, t)$ auf Kanten mit einer bestimmten Markierung einzuschränken. Man erhält $P(m), m \in L_E$ durch die zusätzliche Forderung $l_E(e) = m$ nach dem Existenzquantor von e.

Für die Behandlung der **equiv**-Kanten wird noch ein Maß benötigt, das angibt, wieviele dieser potentiellen Kanten *nicht* vorhanden sind. Dafür muß man bestimmen, wieviele solcher Kanten maximal möglich sind. Es gibt keine **equiv**-Schleifen, d.h. Kanten von einem Knoten zu sich selbst, da solche nicht im Zustandsgraphen eingetragen werden und aufgrund der injektiven Einbettungen auch bei Transformationen nicht erzeugt werden. Alle anderen Kanten, auch Zyklen, sind möglich. Insgesamt kann es also höchstens soviele Kanten geben wie es ungerichtete Kanten in einem vollständig vernetzten Graphen mit n Knoten gibt. Die maximale Anzahl von **equiv**-Kanten in einem Graphen mit n Knoten ist also

$$mek(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{1}{2}(n^2 - n)$$

und die Anzahl nicht vorhandener solcher Kanten

$$nvek(n) = mek(n) - |\{e \in E \mid l_E(e) = \mathbf{equiv}\}|$$

Dieses Maß *nvek* wird nun bei Anwendung der Regel (3) steigen, da die Knotenanzahl bei jeder Regel unverändert bleibt.

Eine geeignete Definition für φ ist dann:

$$\varphi(G) = \sum_{p \in P(\mathbf{neuer})} l(p) + |\{e \in E \mid l_E(e) = \mathbf{neuer}\}| + nvek(|V|)$$

Betrachtet man nun die Auswirkungen einer Regelanwendung auf den Aktionsgraphen, so stellt man fest, daß bei der vorausgesetzten injektiven Einbettung die folgenden Komponenten von φ verändert werden, während die nicht erwähnten konstant bleiben:

Regeln (1a) und (1b): ein **neuer**-Pfad wird um 1 verkürzt

Regel (2): es gibt eine **neuer**-Kante weniger

Regel (3): es gibt eine nicht vorhandene **equiv**-Kante weniger

Regel (4a): ein **neuer**-Pfad wird um 1 verkürzt

Regel (4b): ein **neuer**-Pfad wird um 1 verkürzt und es gibt eine **neuer**-Kante weniger

Regel (5): es gibt eine **neuer**-Kante weniger

Regel (6): es gibt eine **neuer**-Kante weniger

Regeln (7a) und (7b): es gibt eine **neuer**-Kante weniger

In jedem Fall gilt also bei einer Ableitung $G_l \longrightarrow G_r$, daß $\varphi(G_l) > \varphi(G_r)$, wenn man $|V|$ als konstant annehmen kann. Letzteres ergibt sich unmittelbar aus den Regeln, durch die keine Knoten gelöscht oder hinzugefügt werden.

Damit ist bewiesen, daß das Regelsystem terminiert.

4.5.3 Konfluenz

Um zusätzlich zur Terminierung die Konfluenz des gegebenen Transformationssystems zu beweisen, muß man gemäß Satz 3.6 nur lokale Konfluenz nachweisen. Das bedeutet, daß immer, wenn bei einem Graphen zwei unterschiedliche Regelanwendungen möglich sind, für jede dieser Ableitungen weitere Transformationen existieren müssen, so daß die beiden Ableitungslinien schließlich bei einem gemeinsamen Objekt zusammenlaufen (siehe Definition 3.4).

Es wäre nun aber äußerst mühsam, diese Eigenschaft für *alle* möglichen Aktionsgraphen nachzuweisen. Tatsächlich genügt es aber, alle Paare von linken Seiten zu betrachten, die so in einen Graphen eingebettet werden können, daß die Bilder der Einbettungsmorphismen g_l und h_l mindestens eine Kante in ihrem Durchschnitt haben. Formal bedeutet dies, daß für zwei Produktionen P_1 und P_2 mit linken Seiten $B_i (i \in \{1, 2\})$ gilt: $g_{l_E}[B_1] \cap h_{l_E}[B_2] \neq \{\}$. Man nennt dies *kritische Paare*. Dabei ist auch der Fall eingeschlossen, daß die beiden linken Seiten zu ein und derselben Regel gehören, aber an unterschiedlichen Stellen eingebettet werden. Der Fall der Einbettung an gleicher Stelle ist trivial, da man bereits nach einem Schritt das gleiche Ergebnis erhält.

Die Gültigkeit obiger Aussage ergibt sich aus folgenden Überlegungen: Haben in einem Graphen die Bilder der linken Seiten zweier Produktionen keine Kanten gemeinsam, so kann B_l der einen Produktion immer noch im Kontext des anderen

Ableitungsschrittes eingebettet werden, denn bei allen gegebenen Regeln sind alle Knoten Klebeknoten und sind daher im Kontext vorhanden. In diesem Fall sind die Produktionen also parallel unabhängig (Def. 2.24), so daß die Ableitungen auch schon wegen Satz 2.26 konfluent sind.

Gibt es aber ein Paar von Regeln, bei denen die Bilder der linken Seiten mindestens eine gemeinsame Kante aufweisen, so genügt es, einen Graphen zu betrachten, der nur aus genau den Bildern dieser B_l besteht. Für jeden größeren Graphen, zu dem eine Einbettung existiert, sind dann die gleichen Ableitungssequenzen möglich, da dann auch alle Kontexte und rechte Seiten in dem größeren Graphen wiederzufinden sind. Anschaulich formuliert kann beliebig viel hinzugefügt werden, ohne daß dies die Anwendbarkeit der Ableitungsschritte gefährdet. Ein formaler Beweis hierfür findet sich in [Sch99b].

Um die Konfluenz des gegebenen Regelsystems zu beweisen, müssen alle kritischen Paare betrachtet werden. Für jedes Paar müssen zwei Ableitungsfolgen angegeben werden, die schließlich wieder zu einem gemeinsamen Ergebnis zusammenführen.

Bei insgesamt 11 Produktionen[3] gibt es die nicht geringe Anzahl von $\sum_{i=1}^{11} i = 66$ potentiellen Paaren von linken Seiten. Allerdings gibt es nur 16 Kombinationen, bei denen tatsächlich eine Überlappung an einer Kante möglich ist und wobei auch die Anwendbarkeitsbedingungen erfüllt werden. Diese kritischen sind in Tabelle 4.7 auf Seite 91 aufgelistet, zusammen mit der Angabe einer Abbildung, in der die lokale Konfluenz gezeigt wird. Manche ähnliche Paare wurden in derselben Abbildung zusammengefaßt.

4.6 Modifikationszeit kopierter Dateien

Bisher wurde noch die Frage ausgeklammert, welche Modifikationszeit eine Datei erhalten soll, die im Rahmen der Durchführung der ermittelten Aktionen auf einen anderen Rechner kopiert wird. Es gibt hier auf den ersten Blick zwei Möglichkeiten: die gleiche Zeit wie die Originaldatei oder die Zeit, die auch als Synchronisationszeitpunkt vermerkt wird. Beide Alternativen werfen allerdings Probleme auf.

---

[3] Die sonst oft gemeinsam behandelten a/b-Varianten müssen hier getrennt gesehen werden.

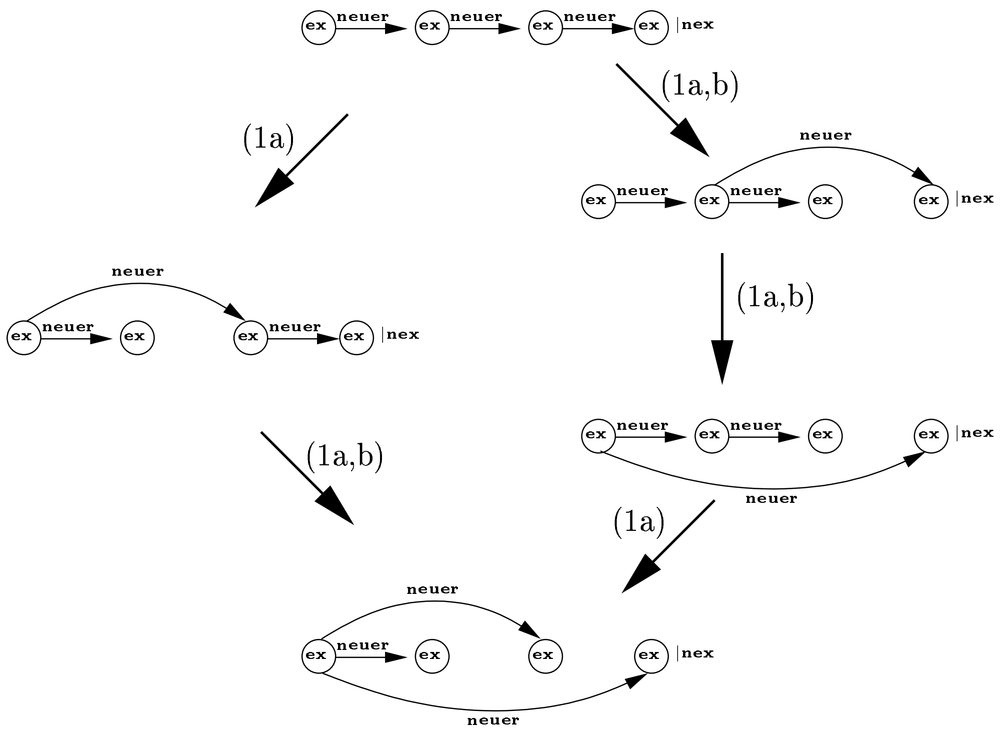


Abbildung 4.18: Kritische Paare (1a), (1a,b) I

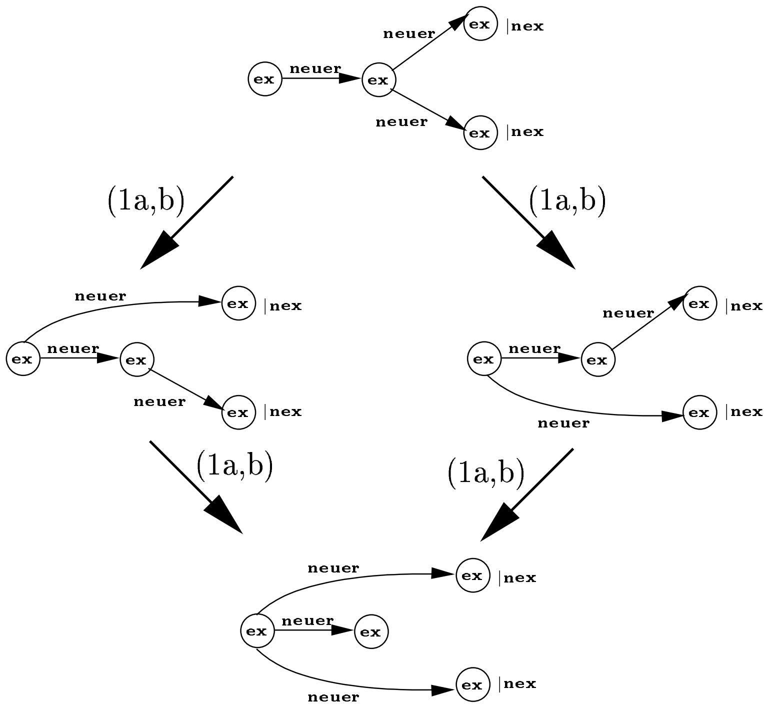


Abbildung 4.19: Kritische Paare (1a), (1a,b) II und (1b), (1b)

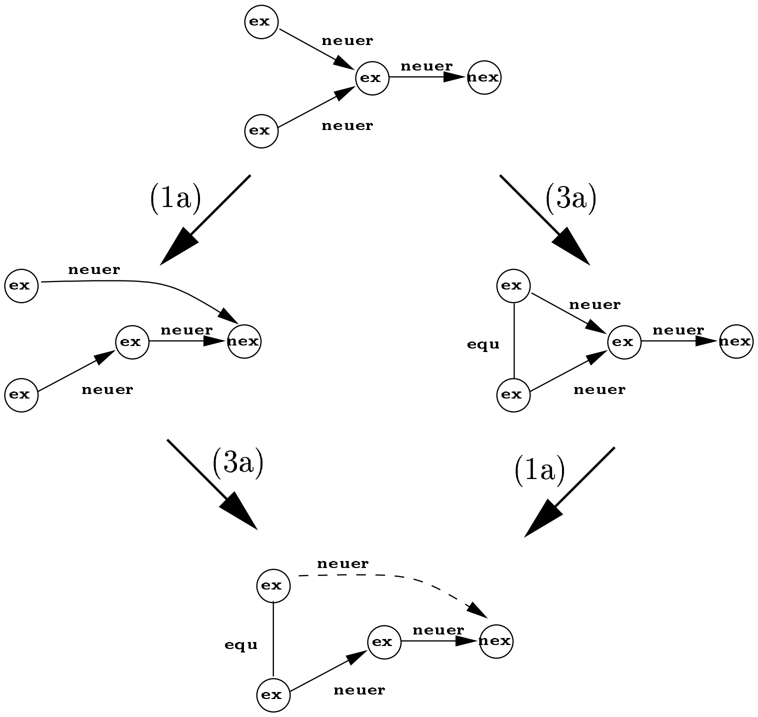


Abbildung 4.20: Kritisches Paar (1a), (3a)

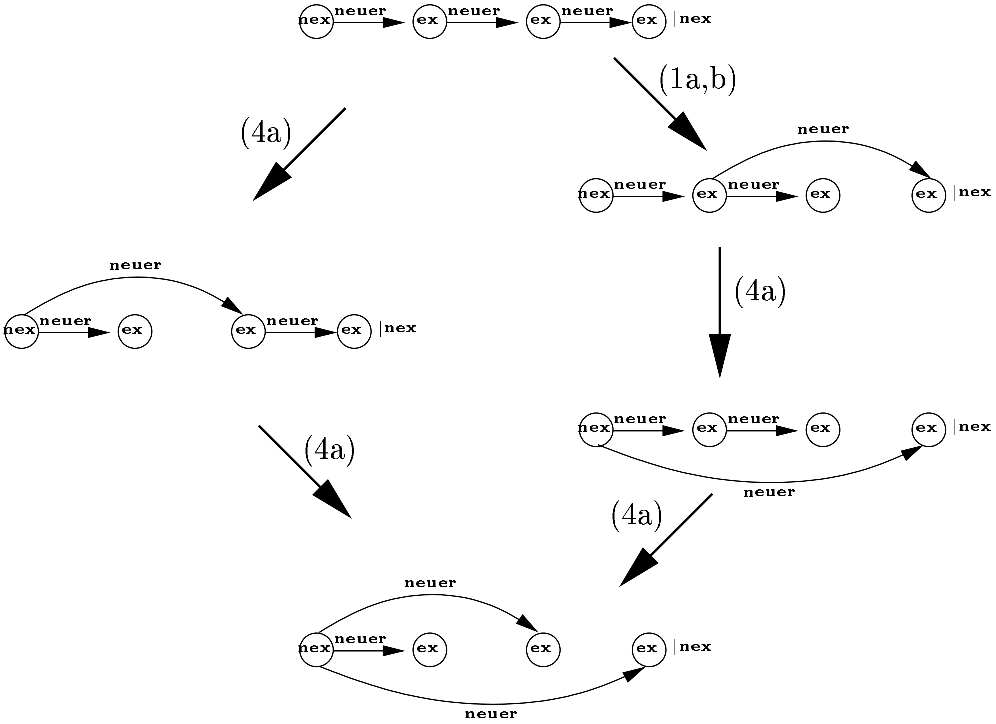


Abbildung 4.21: Kritische Paare (1a,b), (4a)

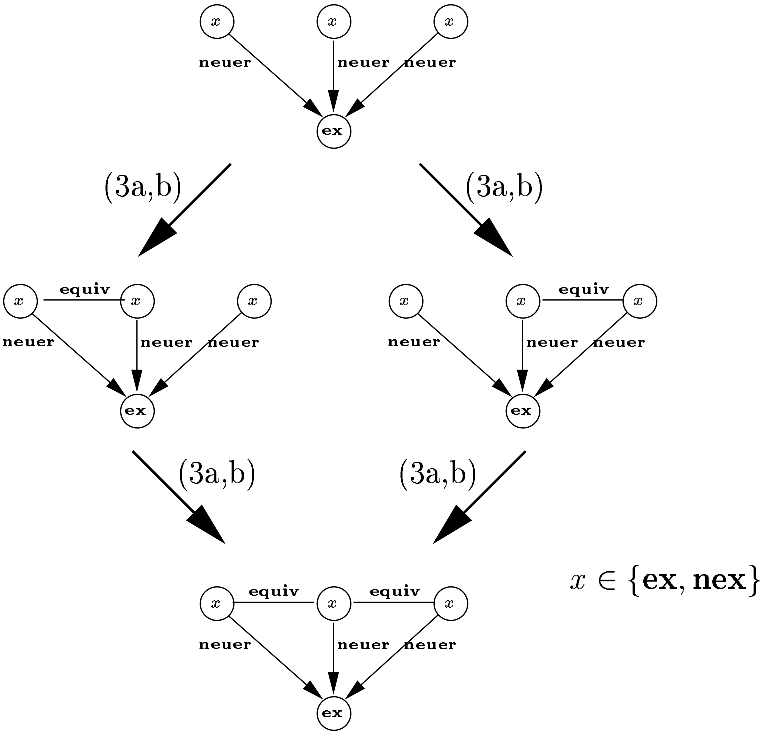
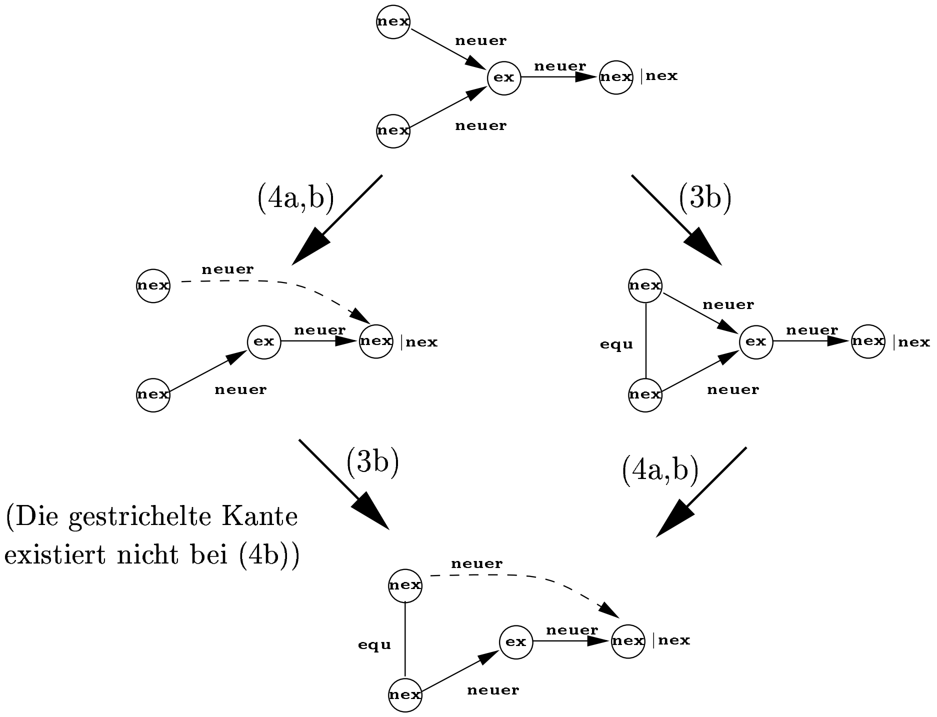


Abbildung 4.22: Kritische Paare (3a,b), (3a,b)



(Die gestrichelte Kante existiert nicht bei (4b))

Abbildung 4.23: Kritische Paare (3b), (4a,b)

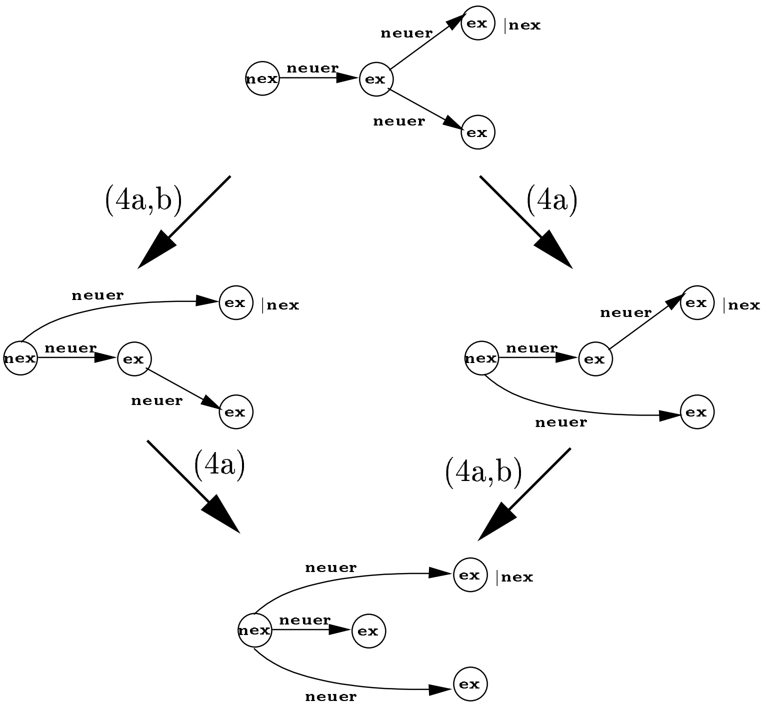


Abbildung 4.24: Kritische Paare (4a), (4a,b)

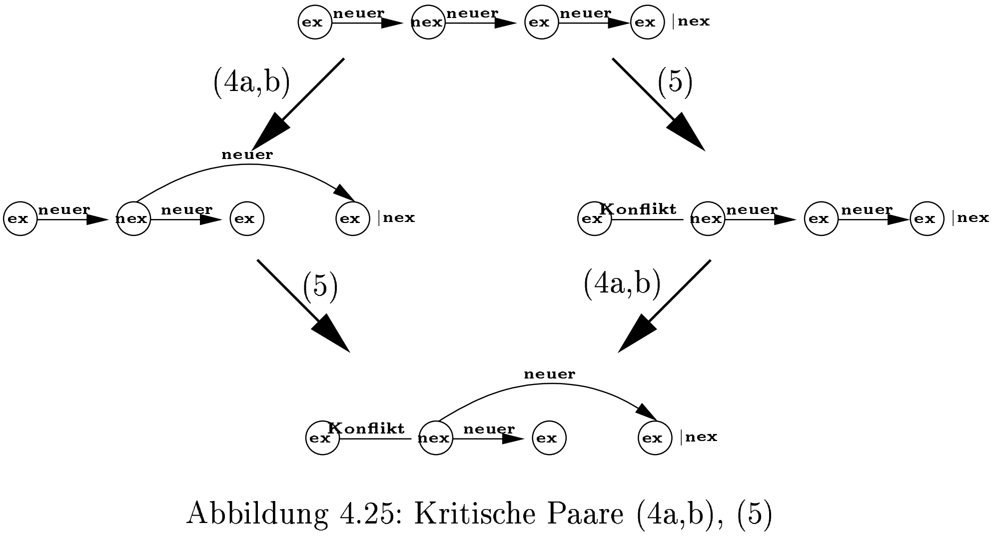


Abbildung 4.25: Kritische Paare (4a,b), (5)

Regelpaar	Abbildung
(1a), (1a)	Abb. 4.18 (erste Überlappungsvariante)
(1a), (1a)	Abb. 4.19 (zweite Überlappungsvariante)
(1a), (1b)	Abb. 4.18 (erste Überlappungsvariante)
(1a), (1b)	Abb. 4.19 (zweite Überlappungsvariante)
(1a), (3a)	Abb. 4.20
(1a), (4a)	Abb. 4.21
(1b), (1b)	Abb. 4.19
(1b), (4a)	Abb. 4.21
(3a), (3a)	Abb. 4.22
(3b), (3b)	Abb. 4.22
(3b), (4a)	Abb. 4.23
(3b), (4b)	Abb. 4.23
(4a), (4a)	Abb. 4.24
(4a), (4b)	Abb. 4.24
(4a), (5)	Abb. 4.25
(4b), (5)	Abb. 4.25

Tabelle 4.7: Kritische Paare des Transformationssystems

4.6.1 Originalzeit als Modifikationszeit

Im ersten Fall, dem Beibehalten der ursprünglichen Modifikationszeit, geschieht folgendes:

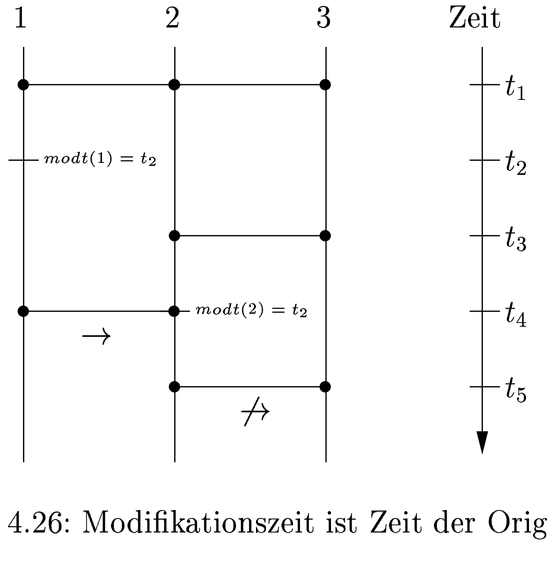


Abbildung 4.26: Modifikationszeit ist Zeit der Originaldatei

Angenommen zum Zeitpunkt t_1 sei eine Datei auf allen Rechnern identisch, was durch die Synchronisation zwischen allen ausgedrückt wird. Dann wird diese Datei auf Rechner 1 zu t_2 geändert. Bei der nächsten Synchronisation mit 2 wird sie dorthin kopiert und die Modifikationszeit dort auf $modt(2) = t_2$ festgelegt. Wenn nun Rechner 2 das nächste Mal mit 3 abgeglichen wird, dann ist $modt(2) = t_2 < ls(2,3) = t_3$, denn nach der Änderung auf 1 hat eine Synchronisation zwischen 2 und 3 stattgefunden, die keine Änderungen an der Datei festgestellt hat. Die Datei wird also *nicht* von 2 nach 3 kopiert, was im Diagramm mit dem durchgestrichenen Pfeil dargestellt ist. Dieses Verhalten ist nicht korrekt und soll natürlich vermieden werden.

4.6.2 Abgleichszeit als Modifikationszeit

Ändert man die gewählte Modifikationszeit der Kopie auf den Zeitpunkt des Abgleichs, so tritt dieser Effekt nicht mehr auf. Es wird dann korrekt erkannt, daß die Datei eine neuere Version ist als die ursprünglich auf dem Zielrechner vorhandene. Dadurch wird diese Version auch bei Synchronisationen mit dritten Rechnern an diese weitergegeben wie in Abbildung 4.27 dargestellt.

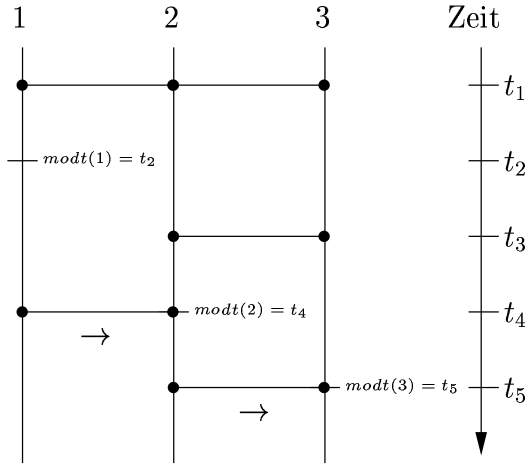


Abbildung 4.27: Modifikationszeit ist Zeit des Abgleichs

Es ist leicht einzusehen, daß bei dieser Wahl der Modifikationszeit die Datei bei allen Abgleichen mit Rechnern, die noch die ursprüngliche Version der Datei besitzen, als neuer angesehen wird. Auch Konflikte mit solchen Rechnern werden weiterhin korrekt erkannt. Wenn man es genauer betrachtet, könnte die gewählte Modifikationszeit der Datei im allgemeinen auch kleiner sein, solange sie nur echt größer als die Zeit des letztes Abgleichs des Zielrechners mit einem beliebigen anderen Rechner bleibt (vgl. auch 4.6.4). Auch dann wird die Datei als neuer im Vergleich zu anderen Rechnern erkannt werden.

Was aber geschieht, wenn eine weitere Synchronisation mit einem Rechner stattfindet, der diese Version der Datei schon besitzt (siehe Abb. 4.28)? Eine geänderte Datei wird zum Zeitpunkt t_3 von Rechner 2 nach 3 kopiert. Dort erhält sie gemäß der neuen Konvention den Zeitstempel t_3. Später wird die gleiche Datei bei einem Abgleich zu Rechner 1 übertragen und erhält dort $modt(1) = t_4$. Bei der folgenden Synchronisation zwischen Rechner 1 und 3 liegt scheinbar ein Konflikt vor, der in Wirklichkeit aber gar nicht vorliegt, da es sich um die gleiche Datei handelt.

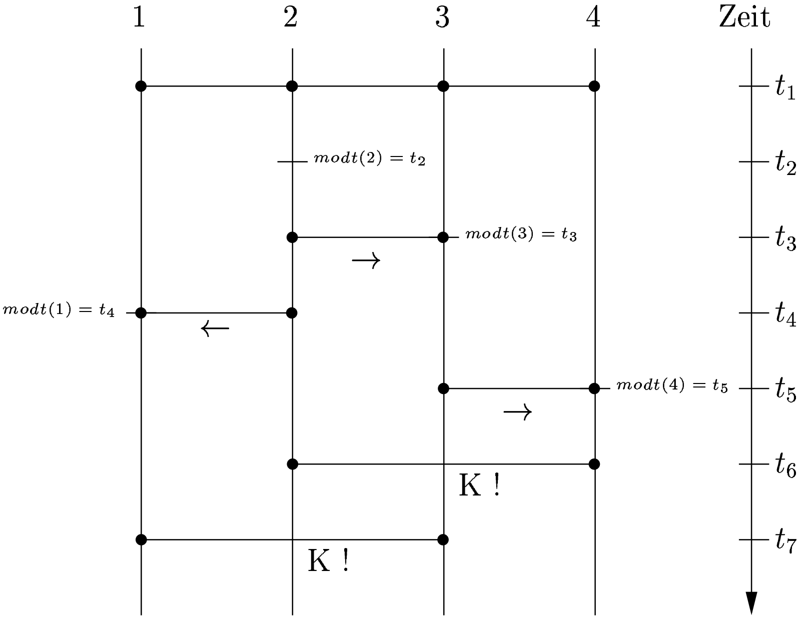


Abbildung 4.28: Scheinbare Konflikte bei Verwendung der Synchronisationszeit

Der gleiche Effekt tritt auch mit dem Ursprungsrechner auf, wenn die Datei von Rechner 3 an Rechner 4 weitergegeben wurde. Bei dem Abgleich zwischen diesen beiden wird der Zeitstempel t_5 vergeben, und bei der folgenden Synchronisation zwischen 2 und 4 sind die Modifikationszeiten t_2 und t_5 beide größer als $ls(2,4) = t_1$.

Die entstehenden Fehlinterpretationen beschränken sich auf Konflikte. Es kann nicht vorkommen, daß die Datei auf einem Rechner n als neuer angesehen wird als die Version auf $m \neq n$, obwohl tatsächlich eine identische Version der Datei auf beiden Knoten vorliegt. Denn dazu müßte der Zeitstempel auf n echt größer als $ls(n,m)$ sein und auf m nicht kleiner oder gleich dieser Marke sein. $ls(n,m)$ markiert aber einen Abgleich, so daß danach beide Zeitstempel $\leq ls(n,m)$ waren und ein $modt(n) > ls(n,m)$ nur durch eine weitere Synchronisation von n mit einem dritten Rechner n' entstanden sein kann.

Hier ist nun eine Fallunterscheidung notwendig. Sei t_0 der Zeitpunkt des letzten Abgleichs von n und n' vor $t_1 := ls(n,m)$ und t_x die Zeit des nächsten Abgleichs der gleichen Rechner nach $ls(n,m)$.

1. Hatte n' zum Zeitpunkt t_0 bereits die gleiche Version der Datei, die zu t_1 nach n kopiert werden soll, so wurde sie hier bereits nach n kopiert und bei t_1 wird ein Konflikt zwischen n und m festgestellt. Dies steht im Widerspruch zu der Annahme, daß zu t_1 von m nach n kopiert wird.

2. Andernfalls wird diese Version zu einer Zeit $t_2 > t_0$ nach n' kopiert und dort mit Zeitstempel t_1 versehen. Die Dateiversion kann nicht vor t_1 nach n gelangen, da für t_0 der letzte Abgleich zwischen den Rechnern vor t_1 angenommen wurde.

   Anschließend wird zu t_1 die Version von m nach n kopiert und erhält dort Zeitstempel $t_1 > t_0$. Da $t_2 > t_0 \wedge t_1 > t_0$ und t_0 der letzte Abgleich zwischen n' und n war, wird beim folgenden Abgleich zwischen ihnen zu t_x ein Konflikt erkannt.

   Bei einem Konflikt wird die Datei aber nicht kopiert, was zu einem Widerspruch zu der Annahme führt, die Datei wäre nach $ls(n,m) = t_1$ von n' nach n kopiert worden.

4.6.3 Pfadverfolgung

Das Ziel ist es nun, diese tatsächlich nicht vorhandenen Konflikte als solche zu erkennen. Das Problem wird dadurch verursacht, daß sich eine Dateiversion schrittweise und auf verschiedenen Pfaden auf den teilnehmenden Rechnern verbreitet. Dabei erhöht sich die Modifikationszeit monoton. Treffen verschiedene solcher Pfade bei einer Synchronisation aufeinander, so kommt es zu dem scheinbaren Konflikt. Es gilt also zu erkennen, ob es sich beim $modt(r_1) > ls(r_1,r_2) \wedge modt(r_2) > ls(r_1,r_2)$ um das Zusammenlaufen solcher Pfade mit gemeinsamen Ursprung handelt, oder ob sie von unabhängigen Modifikationen stammen und somit verschiedene Ausbreitungspfade aufeinanderstoßen.

Eine Lösung wäre also, zu jeder Datei zu speichern, von welchem Rechner die Dateiversion ursprünglich kommt und wann sie angelegt wurde. Eine Originalversion kann daran erkannt werden, daß sich die Modifikationszeit seit der letzten Synchronisation verändert hat, d.h.

$$modt(r) > \max_{r' \in R \setminus \{r\}} ls(r,r')$$

Wird dies während der Ermittlung der Modifikationszeiten direkt vor einem Abgleich auf einem Rechner $r \in R$ festgestellt, so wird $u(r) := (r, modt(r))$ gesetzt. Wird eine Datei während einer Abgleichsaktion von einem Rechner r zu r' kopiert, so werden der Ursprungsort und -zeit mitgegeben, d.h. $u(r') := u(r)$ gesetzt.

So wird zu jeder Datei auf allen Rechnern die Originalversion vermerkt. Ein Konflikt zwischen r und r' liegt dann nur noch vor, wenn $u(r) \neq u(r')$, andernfalls treffen gerade Pfade mit gleichem Ursprung aufeinander und die Dateien sind bereits identisch. Der Zeitstempel der Datei muß dabei auch transportiert werden, um mehrmalige Änderungen auf dem gleichen Rechner unterscheiden zu können.

Leider steht dieser Ansatz im Widerspruch zu der Zielsetzung, daß das Synchronisationssystem möglichst wenig zusätzliche Daten speichern soll (siehe 1.2). Denn es müßte auf jedem Rechner und für *jede* Datei ein zusätzliches Datum abgelegt werden. Daher soll nun eine Alternative zu obigem Konzept entwickelt werden.

4.6.4 Virtuelle Modifikationszeiten

Wie bereits in Abschnitt 4.6.2 erwähnt wurde, ist es nicht unbedingt nötig, kopierte Dateien mit dem Zeitstempel des Abgleichs zu versehen, sondern es genügt auch ein Zeitpunkt davor, sofern dieser noch nach allen bisher stattgefundenen Synchronisationen des Zielrechners liegt. Das Abgleichssystem mißt nur in Zeiträumen zwischen einzelnen Abgleichen, unterscheidet aber nicht verschiedene Zeitpunkte innerhalb eines solchen Intervalls. Daher ist eine freie Wahl des Zeitstempels innerhalb des letzten Intervalls auf dem Zielrechner möglich (Abb.4.29).

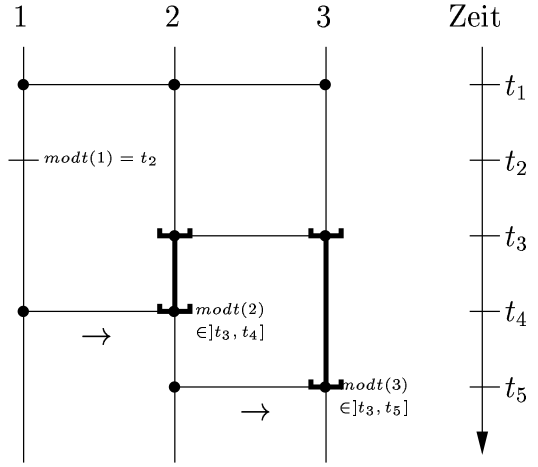


Abbildung 4.29: Wahlmöglichkeiten für Modifikationszeit

Es muß also die Bedingung

$$ls_{max}(r) := \max_{r \in R \setminus \{r_{\text{Ziel}}\}} ls(r, r_{\text{Ziel}}) \ < \ modt_{\text{neu}}(r_{\text{Ziel}}) \ \leq \ ls_{\text{aktuell}}(r_{\text{Quelle}}, r_{\text{Ziel}})$$

gelten. Die obere Grenze ergibt sich aus der Forderung, daß keine zukünftigen Zeitstempel auftreten dürfen.

Selbstverständlich wird man nicht wünschen, daß die Datei beim Kopieren zurückdatiert wird, so daß man auch $modt_{\text{neu}}(r_{\text{Ziel}}) \geq modt(r_{\text{Quelle}})$ fordern wird. Wählt man nun die kleinste Zeit mit diesen Eigenschaften, so ergibt sich

$$modt_{\text{neu}}(r_{\text{Ziel}}) := \max\{s_{max}, modt(r_{\text{Quelle}})\}$$

Es ist zu erwarten, daß in vielen Fällen die gewählte Zeit mit $modt(r_{\text{Quelle}})$ übereinstimmen wird, denn in der Praxis werden die meisten Synchronisationen mit größeren Rechnergruppen stattfinden und die Änderungen an Dateien zwischen diesen Abgleichen liegen.

Eine weitere Überlegung in diesem Zusammenhang ist, daß die Vergabe eines abweichenden Zeitstempels bei der Kopie einen Verlust von Information darstellt. Wenigstens aus Benutzersicht wäre es wünschenswert, wenn auch innerhalb synchronisierter Dateibäume die bei Dateien angegebene Modifikationszeit dem wahren Änderungszeitpunkt entspricht und nicht ein Artefakt der Abgleiche ist.

Es bietet sich also an, die Kopien mit dem Zeitstempel der Originaldatei zu versehen und für den Fall, daß das Synchronisationssystem für sein korrektes Funktionieren ein $modt_{\text{neu}}(r_{\text{Ziel}}) > modt(r_{\text{Quelle}})$ erfordert, diese Zeit an anderer Stelle zu speichern. Dies stellt also eine *virtuelle Modifikationszeit* dar, die nur innerhalb des Abgleichssystems benutzt wird und nicht im Dateisystem selbst auftritt.

Dies führt zu folgendem Konzept einer *Ausnahmeliste*:

- Jeder Rechner r des Systems verwaltet für jeden synchronisierten Dateibaum eine Ausnahmeliste, die für die Dateien virtuelle Modifikationszeiten angeben kann, aber nicht muß. Für die bei einem Abgleich betrachtete Datei sei der Eintrag in dieser Liste auf Rechner r $al(r)$. Ist die Datei nicht in der Liste enthalten, so ist $al(r)$ undefiniert.
- Für alle Zwecke des Abgleichssystems ersetzt die virtuelle Modifikationszeit die reale, wenn sie für die betrachtete Datei definiert ist. Die Definition von $modt(r)$ aus Abschnitt 4.2 muß also wie folgt abgeändert werden:

$$modt(r) := \begin{cases} al(r) & \text{wenn } al(r) \text{ definiert} \\ \text{reale Modifikationszeit auf } r & \text{sonst} \end{cases}$$

- Ein Eintrag in die Ausnahmeliste erfolgt, wenn beim Übertragen einer Datei im Rahmen einer Synchronisation gemäß obiger Definition von $modt_{\text{neu}}(r_{\text{Ziel}})$ ein Zeitstempel $> modt(r_{\text{Quelle}})$ erforderlich ist. Der Eintrag ist dann die Zeit der letzten Synchronisation des Zielrechners (abgesehen von der gerade laufenden), das weiter oben definierte s_{max}.

- Ein Eintrag in der Ausnahmeliste kann wieder gelöscht werden, wenn der Rechner mit allen anderen Rechnern abgeglichen wurde. Dann hat die Information ihrem Zweck erfüllt, nämlich die Datei als neu zu kennzeichnen. Es gilt dann

  $$modt_{\text{real}}(r) < al(r) < \min_{r' \in R \setminus \{r\}} ls(r, r')$$

  und für die Dateibeziehungen ist es gleichgültig, wieviel kleiner der Zeitstempel auf einem Rechner ist, wenn er nur kleiner als ein $ls(r, r')$ ist. Es kann also bei Erfüllung dieser Bedingung auch wieder $modt_{\text{real}}(r)$ als $modt(r)$ verwendet werden.

  Nach dem Abschluß einer Synchronisation kann also das neue Minimum der $ls(r, r')$ bestimmt werden und alle Einträge der Ausnahmeliste, die älter als dieses Minimum sind, entfernt werden.

Auch diese Ausnahmeliste stellt natürlich zusätzliche Daten dar, die gespeichert werden müssen. Allerdings muß nicht zu jeder Datei eines Baums ein Datum gespeichert werden, sondern nur dann, wenn dies tatsächlich notwendig ist. Der Speicherplatzbedarf kann also nicht größer sein als der einer vollständigen Liste. Für den praktischen Einsatz ist auch zu erwarten, daß Einträge in die Liste tatsächlich Ausnahmen sein werden, denn, wie oben schon angedeutet, wird in vielen Fällen die reale Modifikationszeit ausreichend sein. Zusätzlich ist die Lebensdauer von Listeneinträgen begrenzt, so daß deren Umfang auch wieder abnehmen kann[4].

4.6.5 Virtuelle Modifikationszeiten und Konflikterkennung

Damit ist aber das Problem der fälschlicherweise erkannten Konflikte noch nicht gelöst. Nur hat man, wenn man die Modifikationszeit der Originaldatei auf allen Rechnern erhält, bereits eine gewisse Art von Identifikation für die Ursprungsdatei: ihren Zeitstempel. Wird ein Konflikt erkannt und die realen Modifikationszeiten der Dateien sind gleich, so stellt dies einen Hinweis dar, daß es sich um

---

[4] Dies setzt voraus, daß mit allen Rechnern zu irgendeinem Zeitpunkt in der Zukunft wieder Abgleiche stattfinden werden. Andernfalls würde das Minimum der letzten Synchronisationszeiten unverändert bleiben. Daher müssen also Rechner „abgemeldet“ werden, wenn sie aus dem Synchronisationssystem herausgenommen werden sollen.

identische Dateien handeln könnte. Sind sie dagegen unterschiedlich, so kann es sich sicher nicht um die gleiche Originaldatei handeln und es liegt ein wahrer Konflikt vor.

Dieser Hinweis ist nur noch nicht ausreichend. In der Praxis ist es zwar unwahrscheinlich, es kann aber dennoch durchaus vorkommen, daß unabhängige Änderungen fast gleichzeitig durchgeführt werden. Durch die beschränkte Auflösung der Zeitstempel (beispielweise in Sekunden) können daher auch in diesem Fall gleiche Zeitstempel auftreten.

Bei gleichen realen Modifikationszeiten muß daher noch geprüft werden, ob die Dateiinhalte tatsächlich identisch sind. Dies kann durch einen vollständigen Vergleich erfolgen, der aber im allgemeinen einen Transfer beider Dateiinhalte über das Netzwerk erfordert, nämlich von den Rechnern, die die jeweiligen Dateiversionen speichern, zu dem, der den Abgleich durchführt. Letzterer kann mit einem der beiden anderen identisch sein, was einen Netzwerktransfer erspart, doch ist dies ein Ausnahmefall.

Will man eine sehr kleine aber dennoch von 0 verschiedene Fehlerrate akzeptieren, so kann man anstelle eines vollständigen Inhaltsvergleichs ein Prüfsummenverfahren verwenden, ähnlich dem, das bei der Initialisierung des Systems beschrieben wird (siehe 5.3). Damit reduziert sich die für die Vergleiche transportierte Datenmenge auf wenige Bytes.

Die Gleichheitsüberprüfung bei der Konflikterkennung hat noch einen weiteren positiven Nebeneffekt: Ändert ein Benutzer eine Datei und kopiert sie dann manuell auf einen anderen Rechner des Synchronisationssystems und erhält dabei die Kopie den Zeitpunkt des Kopiervorgangs als Zeitstempel, so wird ebenso wie bei einer unabhängigen Änderung ein Konflikt erkannt werden. Werden dann jedoch die Dateiinhalte verglichen, ergibt sich, daß es sich um die gleiche Version handelt, und eine Konfliktbehandlung erübrigt sich. Eigentlich ist es ein Fehlverhalten des Benutzers, wenn er unnötigerweise Dateien am Synchronisationssystem vorbei kopiert und dabei zusätzlich die Modifikationszeit verändert. In dem beschriebenen Fall kann dies aber kompensiert werden.

Kapitel 5

Weitere Einzelfragen der Dateisynchronisation

In diesem Kapitel sollen einige weitere Themen behandelt werden, die nicht zu den Grundlagen des Synchronisationssystems zählen. Es wird genauer auf die Behandlung von erkannten Konflikten eingegangen und der Fall behandelt, daß Dateien von einer Datensicherung restauriert werden und damit Modifikationszeiten zurück in die Vergangenheit springen. Auch wird dargelegt, was beim erstmaligen Abgleich von Dateibäumen zu tun ist, und wie eine Implementation des beschriebenen Synchronisationssystems aussehen kann.

5.1 Konfliktbehandlung

Wenn während einer Synchronisation Konflikte entdeckt werden, so muß der Abgleich für die betroffenen Dateien abgebrochen werden, wie es im vorangegangenen Kapitel bereits mehrfach angesprochen wurde. Im allgemeinen müssen solche Konflikte zwischen unabhängigen Änderungen manuell aufgelöst werden. Es erfolgt daher eine Benachrichtigung der Eigentümer der in Konflikt stehenden Dateien und/oder einen Administrator. Weiterhin wird die Datei als konfliktbehaftet markiert, indem ihr Name oder eine andere eindeutige Identifikation innerhalb des Dateibaums zusammen mit der Zeit des fehlgeschlagenen Abgleichs in einer *Konfliktliste* auf allen Knoten vorgenommen werden, die mit einer **Konflikt**-Kante im Aktionsgraphen verbunden sind. Bei späteren Abgleichen können anhand dieser Liste die Dateien ausgeschlossen werden, zu denen noch nicht bereinigte Konflikte existieren.

Das Auflösen eines Konflikts geschieht dadurch, daß eine Person die unabhängigen Änderungen begutachtet und in eine gemeinsame Version zusammenführt. Er kann sich dabei selbstverständlich auch für eine bestimmte Version entscheiden und die anderen verwerfen. Er muß dann die bereinigte Version auf einem der Rechner des Synchronisationsnetzes ablegen und mit einem aktuellen Zeitstempel versehen, was beim Abspeichern einer zusammenführenden Version von selbst geschieht. Danach benachrichtigt er das Synchronisationssystem auf diesem Rechner von der Bereinigung des Konflikts, das daraufhin den Eintrag in der lokalen Konfliktliste löscht.

Bei Abgleichen nach diesem Zeitpunkt hat dieser Rechner eine Dateiversion mit einer Modifikationszeit, die größer als alle vorangegangenen Synchronisationszeiten ist. Durch Vergleich dieses Zeitstempels mit der Konfliktzeit können die teilnehmenden Rechner erkennen, daß der Konflikt aufgelöst wurde. Die Datei ist daraufhin nicht mehr von Abgleichen ausgeschlossen und der Konfliktlisteneintrag wird gelöscht. Da die Modifikationszeit auch größer als alle letzten Synchronisationen ist, wird die neuerstellte Dateiversion als die neueste angesehen und auf andere Rechner verteilt.

Unter bestimmten Bedingungen kann das Synchronisationssystem sogar eine automatische Auflösung von Konflikten vornehmen. Diese Voraussetzungen sind:

- Es muß sich um eine Klartextdatei handeln.
- Es dürfen nur zwei Dateien an dem Konflikt beteiligt sein.
- Die unabhängigen Änderungen dürfen sich nicht überlappen.
- Es muß noch eine gemeinsame Vorgängerversion der Datei auf einem Knoten des Synchronisationsnetzes vorhanden sein.

Sind alle diese Voraussetzungen gegeben, so kann man die getrennt vorgenommenen Änderungen relativ zu der gemeinsamen Vorgängerversion bestimmen und wechselseitig auf die jeweils andere Datei anwenden. Das Ergebnis ist eine Dateiversion, die beide Modifikationen integriert und damit in einem Versionsgraphen gemeinsamer Nachfolger der in Konflikt stehenden Versionen ist (Abb. 5.1).

Zum Finden von Änderungen zwischen Klartextdateien existieren bereits Werkzeuge wie `diff` [MM85, Ukk85, Mye86] und `diff3`. Letzteres besitzt genau die Funktionalität, die für den vorliegenden Anwendungsfall benötigt wird: Es vergleicht zwei Dateien mit einer gemeinsamen Vorgängerversion. Die Ausgabe dieses Werkzeugs gliedert sich in Abschnitte, die jeweils nicht überlappende Zeilenbereiche mit Modifikationen beschreiben. Dabei wird zu jedem solchen Abschnitt

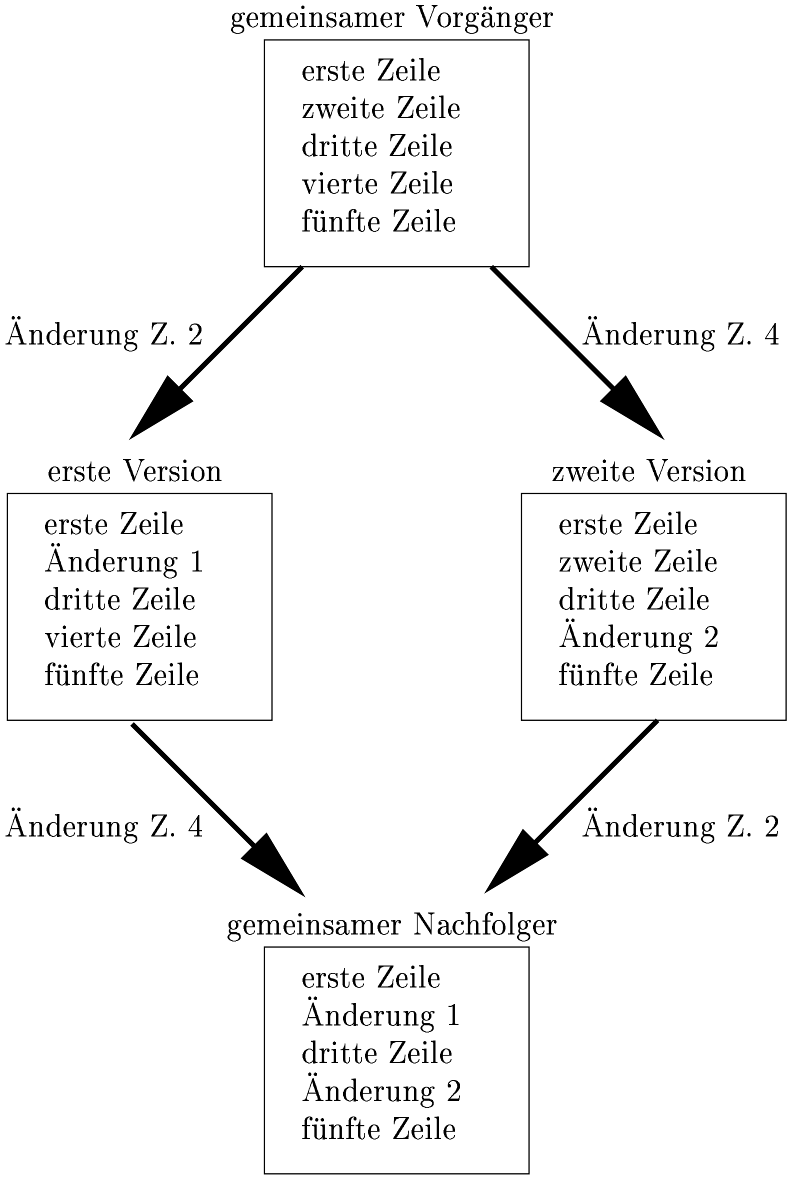


Abbildung 5.1: Zusammenführen unabhängiger Änderungen

angegeben, in welcher der beiden neueren Versionen Änderungen im Vergleich zum Original vorgenommen wurden, bzw. daß in beiden Modifikationen in diesem Bereich stattfanden und welche.

Für die automatische Konfliktbereinigung muß die `diff3`-Ausgabe also nur daraufhin analysiert werden, ob alle Änderungen in nicht zusammenhängenden Zeilenbereichen stattfanden. Ist dies der Fall, so können die unabhängigen Modifikationen zusammengeführt werden. `diff3` kann sogar dies selbständig erledigen, wenn es mit einer dafür vorgesehenen Option (`-m`) aufgerufen wird.

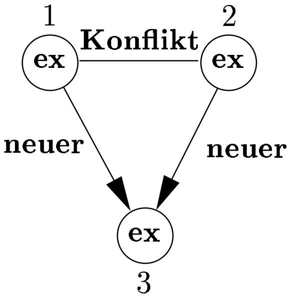


Abbildung 5.2: Finden eines gemeinsamen Vorgängers

Auf der Seite des Synchronisationssystems ist also noch zu klären, ob es sich um Textdateien handelt und ob eine gemeinsame Vorgängerversion existiert. Ersteres kann leider nur durch Heuristiken erfolgen, die auch von der benutzten Systemumgebung abhängig sein können. Bei Betriebssystemen der UNIX-Familie ist übliche Vorgehensweise, den Anfang der Datei auf Zeichen zu durchsuchen, die in Textdateien normalerweise nicht vorkommen, und sicherzustellen, daß die Zeilen eine angemessen erscheinende Länge haben.

Von mehr Interesse ist allerdings das Auffinden eines gemeinsamen Vorgängers, von dem die getrennten Änderungen ausgingen. Hier kommt die Tatsache zu Hilfe, daß das Synchronisationssystem genau dazu dient, Änderungen an Dateien zu erkennen. Es kann also auf die Informationen im Aktionsgraphen zurückgegriffen werden. Führen von den beiden Knoten, die den Konflikt aufweisen, **neuer**-Kanten zu einem weiteren Knoten und sind alle diese Knoten mit **ex** markiert (Abb. 5.2), so ist dies ein gemeinsamer Vorgänger.

Aus der Definition von **neuer**-Kanten ist offensichtlich, daß die Datei auf den Knoten 1 und 2 neuer sein muß als die auf 3. Zu den Zeitpunkten $ls(1,3)$ und $ls(2,3)$ der Synchronisationen zwischen diesen Rechnerpaaren war die Datei jeweils in gleichem Zustand und auf 3 wurde sie seitdem nicht mehr verändert. Ist $ls(1,3) = ls(2,3)$, so ist unmittelbar klar, daß die aktuelle Datei auf 3 der gesuchte gemeinsame Vorgänger ist. Sind $ls(1,3)$ und $ls(2,3)$ unterschiedlich, so bestünde noch die Möglichkeit, daß die Datei in dem Zeitintervall dazwischen auf 3 modifiziert wurde, entweder manuell oder durch einen Abgleich, an dem 1 und 2 nicht teilnahmen. Wenn man ohne Beschränkung der Allgemeinheit $ls(1,3) < ls(2,3)$ annimmt, dann ist danach $modt(3) > ls(1,3)$ und wegen der vorhandenen **Konflikt**-Kante $modt(1) > ls(1,3)$. Damit läge aber ein Konflikt auch zwischen 1 und 3 vor, der im Widerspruch zu der existierenden **neuer**-Kante steht.

Das Synchronisationssystem kann also mit seinen Mitteln einfach gemeinsame

Vorgänger finden unter der Voraussetzung, daß mindestens drei Rechner an dem momentanen Abgleich teilnehmen und der Graph von Abb. 5.2 im Aktionsgraph gefunden werden kann.

Wenn nun alle Vorbedingungen für eine automatische Konfliktbereinigung gegeben sind, kann eine solche vorgenommen werden. Die Dateieigentümer und/oder ein Administrator sollte aber dennoch davon benachrichtigt werden. Zum einen kann allein die Tatsache von Bedeutung sein, daß unabhängige Änderungen stattgefunden hatten. Zum anderen könnte das Zusammenmischen der erfolgten Modifikationen bei bestimmten Typen von Klartextdateien unbeabsichtigte Änderungen der Semantik zur Folge haben. Es sollte daher auf alle Fälle einer Person die Möglichkeit zur Kontrolle der vorgenommenen Zusammenführung ermöglicht werden.

5.2 Einspielen von Datensicherungen

Es ist eine besondere Situation für das Synchronisationssystem, wenn Dateien aus synchronisierten Dateibäumen im Rahmen von Datensicherungsmaßnahmen archiviert werden und dann später ein solches Archiv teilweise oder vollständig wieder in das Dateisystem zurückgeschrieben wird. Denn dabei erhalten die so angelegten oder überschriebenen Dateien normalerweise wieder die Modifikationszeit, die sie zum Zeitpunkt der Archivierung hatten. Beim Zurückspielen eines Archivs können sich Modifikationszeiten also zurück in die Vergangenheit ändern, was über die bisherigen Annahmen hinausgeht.

Werden Dateien aus einem Datensicherungsarchiv restauriert, wird man sie als die jeweils neuesten Versionen betrachten wollen, damit sie auch auf die anderen Rechner des Synchronisationsnetzes verbreitet werden. Um dies zu erreichen, bieten sich die virtuellen Modifikationszeiten aus Abschnitt 4.6.4 an. Mit ihrer Hilfe kann man — unabhängig von der realen, im Dateisystem angegebenen Modifikationszeit — die zurückgespielten Dateien als neu kennzeichnen. Dazu wird einfach die virtuelle Modifikationszeit in der Ausnahmeliste auf den Zeitpunkt gesetzt, zu dem das Einspielen des Archivs beendet ist. Dies geschieht selbstverständlich für alle Dateien, die von der Operation betroffen wurden. Das Synchronisationssystem muß also nur eine geeignete externe Schnittstelle für solche Einträge in die Ausnahmeliste vorsehen.

Bei dieser Vorgehensweise werden Konflikte auftreten, wenn sich betroffene Dateien zum Zeitpunkt des Einspielens des Archivs nicht auf allen Knoten des Netzes im gleichen Zustand befanden. Dieses Verhalten kann als durchaus nützlich angesehen werden, denn es erlaubt während der Konfliktbehandlung (siehe Abschnitt 5.1) die Entscheidung, ob eine solche geänderte Version einer Datei weiterhin verwendet werden soll oder die Variante aus dem Archiv.

5.3 Initialisierung

In den bisherigen Ausführungen wurde immer vorausgesetzt, daß in der Vergangenheit eine letzte Synchronisation stattgefunden hat und sich nach diesem Zeitpunkt korrespondierende Dateien auf allen beteiligten Rechnern in gleichem Zustand befanden. Wie wird jedoch am Anfang die erste Synchronisation durchgeführt, wenn das System zum erstenmal auf einen Dateibaum angewandt wird?

Um die Bedingung des gleichen Zustands sicherzustellen, muß eine Initialisierung durchgeführt werden, zu deren erfolgreichem Abschluß ein Zeitstempel für eine letzte Synchronisation gesetzt wird. Der Initialisierungslauf bekommt eine Menge von teilnehmenden Rechnern als Parameter und zu jedem dieser Rechner jeweils den Startpunkt (Verzeichnisname) des zu betrachtenden Dateibaums in seinem Dateisystem. Von diesen Informationen ausgehend werden folgende Tests und Aktionen durchgeführt:

- Die Dateibäume müssen auf allen Rechnern die gleichen Dateien enthalten. Fehlt eine Datei auf einem Knoten, so muß sie entweder dort angelegt werden, oder die Datei muß auf allen Rechnern entfernt werden.
- Der Inhalt aller korrespondierenden (und existierenden) Dateien muß identisch sein. Ist dies nicht der Fall, so muß eine Referenzdatei ausgewählt und auf die anderen Rechner verteilt werden.

Der Initialisierungslauf kennt folgende Arbeitsmodi, um die oben beschriebenen Entscheidungen so weit wie möglich automatisch treffen zu können:

Referenzrechner: Es wird vom Benutzer ein Rechner angegeben, der einen Referenzdateibaum enthält. Sofern bei einem anderen Rechner Abweichungen im Vergleich zu der jeweiligen Referenzdatei auftreten, so wird diese dorthin dupliziert, bzw. wenn die Referenzdatei nicht existiert, so muß die Datei dort gelöscht werden.

Referenzrechner ohne Löschen: Wie oben werden bei Unterschieden die Dateien des Referenzrechners verwendet. Existiert eine Datei auf diesem nicht, aber auf anderen Rechnern und ist auf allen, auf denen sie vorhanden ist, identisch, so wird sie auch auf den Referenzrechner kopiert. Dieser Modus vermeidet also das Löschen von Dateien und legt sie bei Bedarf auch auf dem Referenzrechner an.

Neueste Datei bevorzugen: Werden unterschiedliche Versionen der Datei gefunden, so wird anhand der Modifikationszeiten die neueste ausgewählt und auf alle Rechner verteilt.

Kann anhand der Regeln für den jeweiligen Arbeitsmodus keine Entscheidung für eine vorliegende Situation getroffen werden, so wird der Initialisierungslauf mit einer Fehlermeldung abgebrochen. Beispielsweise könnte beim Modus „Referenzrechner ohne Löschen“ eine Datei nur auf Nicht-Referenzrechnern vorhanden sein, dort aber in unterschiedlichen Varianten. Dann trifft die Bedingung dort, daß die Datei überall identisch sein muß, wenn sie existiert, nicht zu. In diesem Fall erfolgt also ein Abbruch.

Finden sich während des Initialisierungslaufs keine Fehlersituationen, so werden die nötigen Kopieraktionen ausgeführt und dann der Zeitstempel für die letzte Synchronisation auf die aktuelle Zeit gesetzt. Da angenommen wurde, daß keine Modifikationszeiten neuer als die aktuelle Zeit auftreten können (siehe 4.1.1), sind damit die Zeiten aller Dateien kleiner als $ls(A,B) (\forall A,B \in R, A \neq B)$. Es wurde auch sichergestellt, daß der Inhalt aller korrespondierenden Dateien identisch ist. Somit ist die Voraussetzung für die Annahme gegeben, daß man jede Veränderung einer Datei an einer größeren Modifikationszeit als der letzten Synchronisationszeit erkennen kann.

Dafür wird auch die Annahme gemacht, daß sich während der Initialisierung keine Modifikationszeiten ändern, was für diese nur einmal stattfindende Prozedur akzeptiert werden kann.

Man beachte, daß in der Initialisierungsphase die tatsächliche Identität der Dateien geprüft werden muß, nicht etwa nur identische Modifikationszeiten. Ein Byte-für-Byte-Vergleich aller Paarungen korrespondierender Dateien wäre allerdings sehr aufwendig, und zusätzlich müßten sehr viele Daten über das Netzwerk transportiert werden. Um dies zu vermeiden, setzt man Prüfsummenverfahren ein, die mit sehr großer Wahrscheinlichkeit bei unterschiedlichen Dateiinhalten eine unterschiedliche Prüfsumme liefern. Zum Beipiel liegt beim MD5-Verfahren [Riv92] die Wahrscheinlichkeit, daß zwei Dateien zufällig eine gleiche Prüfsumme aufweisen, bei 2^{-64}. Diese Wahrscheinlichkeit läßt sich weiter reduzieren, indem mehrere Verfahren gleichzeitig eingesetzt werden. Sind die Algorithmen der Verfahren tatsächlich unabhängig voneinander, so multiplizieren sich die Fehlerquoten zur Gesamtquote, die damit sehr schnell weiter gegen 0 sinkt. Für den normalen praktischen Einsatz dürfte eine Fehlerquote von $2^{-100} \approx 10^{-31}$ mehr als ausreichend sein und ist bereits durch die Kombination von zwei guten Verfahren ähnlich MD5 zu erreichen.

Auch müssen nicht alle Dateipaarungen miteinander verglichen werden. Im Modus „Referenzrechner“ sind gar keine Vergleiche erforderlich, bei „Referenzrechner mit Löschen“ ist nur interessant, ob die vorhandenen Dateien alle untereinander gleich sind. Sobald also eine erste unterschiedliche Variante gefunden wurde, kann der Test abgebrochen und die Bedingung verneint werden. Auch ist die Gleichheit natürlich transitiv, so daß der Aufwand linear mit der Anzahl der beteiligten Rechner ist. Auch im Modus „neueste Datei bevorzugen“ ist die Frage nur, ob alle korrespondierenden Dateien identisch sind. Trifft man auf die erste unterschiedliche Datei, so kann bereits die neueste anhand der Modifikationszeit ausgewählt werden.

5.4 Praktische Umsetzung

5.4.1 Implementation des Transformationssystems

Für die Implementation des in dieser Arbeit beschriebenen Transformationssystems wurde aus Effizienzgründen kein allgemein anwendbares Graphtransformationswerkzeug wie z.B. PROGRESS [Sch89] eingesetzt. Stattdessen wurden die Regeln fest codiert.

Dies ist relativ einfach möglich, da die linken Seiten nur aus drei Grundtypen bestehen: Einzelne Kanten, eine Verkettung von zwei Kanten und zwei Kanten, die an einem Knoten zusammenlaufen. Auch kann man die Erkenntnis verwenden, daß es genügt, wenn die auf Einzelkanten operierenden Regeln (6) und (7) erst gegen Ende zum Einsatz kommen (siehe Abschnitt 4.5). Man kann die Transformation also in zwei Phasen unterteilen: Zuerst Regeln (1)–(5) und dann, wenn diese nicht mehr anwendbar sind, (6) und (7).

Man kann sich also zunächst darauf beschränken, für eine linke Seite ein Paar von **neuer**-Kanten zu finden. Je nach deren Orientierung, der Markierung der Knoten und den Anwendbarkeitsbedingungen wird dann eine Regel ausgewählt. Da sich die Bedingungen auf Anzahl ein- bzw. ausgehender Kanten beschränken, ist deren Überprüfung bei geeigneter Repräsentation[1] des Graphen einfach. Daß das Auffinden einer Regel zu einem Kantenpaar immer möglich ist, wurde in Abschnitt 4.5.1 dargelegt. Auch eine injektive Einbettung ergibt sich aus der Art der Suche der linken Seite von selbst.

Werden keine Paare von **neuer**-Kanten mehr gefunden, kann in die nächste Phase übergegangen werden und mit den Regeln (6) und (7) die verbliebenen **neuer**-Kanten umgewandelt werden. Regel (2) muß nicht explizit implementiert werden, wenn die zugrundeliegende Datenstruktur für den Graphen nicht mehr als eine **neuer**-Kante von einem Knoten zum anderen repräsentieren kann.

5.4.2 Rahmen für die Synchronisation

Auf allen beteiligten Rechnern laufen Hintergrundprozesse, die folgende Aufgaben erfüllen:

- Sie stoßen nach Ablauf einer gewissen Zeitspanne nach dem letzten Abgleich eines Dateibaums eine neue Synchronisation an.
- Sie erstellen auf Anfrage eines anderen Prozesses eine Liste von Informationen über den Dateibaum auf ihrem Rechner. Diese Liste enthält vor allem die Modifikationszeiten aller Dateien in dem betrachteten Dateibaum, gegebenenfalls die virtuellen Modifikationszeiten aus der Ausnahmeliste. Daneben umfassen die gelieferten Informationen auch die aktuelle Konfliktliste (vgl. 5.1).
- Sie nehmen Dateien entgegen, die auf ihren Rechner kopiert werden sollen und speichern sie in ihrem Dateibaum ab. Dies wird weiter unten noch näher ausgeführt.
- Sie löschen auf Anforderung Dateien in dem synchronisierten Dateibäumen.
- Sie senden auf Aufforderung eine ihrer Dateien an eine Menge von Rechnern, die als Parameter mitgegeben wurde.
- Falls sie feststellen, daß auf einem der teilnehmenden Rechner noch kein Hintergrundprozeß zur Kommunikation läuft, so versuchen sie diesen zu starten. Dies ist möglich, wenn es entsprechende Mittel zum Starten von Prozessen auf anderen Rechnern gibt[2].

---

[1] „Geeignet“ bedeutet also, daß auch die zu einem Knoten hingerichteten Kanten ohne Durchlaufen aller Knoten gefunden werden können. Kanten werden also doppelt gespeichert, einmal bei ihrem Quellknoten und dem Zielknoten.

[2] Beispielweise `rsh` oder `ssh` bei UNIX-Systemen.

Der Ablauf einer Synchronisation sieht dann wie folgt aus:

- Einer der beteiligten Rechner beginnt eine Synchronisation, nachdem er festgestellt hat, daß die letzte Synchronisation eines Dateibaums lange genug zurückliegt.
- Er stellt fest, welche der Rechner des Synchronisationsnetzes erreichbar sind und sendet an diese eine Anforderung für die Modifikationszeitenliste.
- Es kann vorkommen, daß mehrere Rechner unabhängig voneinander eine Synchronisation beginnen wollen, es darf aber nur einer von ihnen den gerade begonnenen Abgleich durchführen. Dieser Fall kann von den anstoßenden Rechnern dadurch erkannt werden, daß sie eine Anforderung für die Modifikationszeitenliste erhalten.

  Man kann hier so vorgehen, daß jeder Rechner, der einen Abgleich beginnt, dafür seine aktuelle lokale Zeit als Anfangszeit festlegt. Die Anforderungen enthalten diesen Zeitstempel und der empfangende Rechner kann sie mit seinem Anfangszeitpunkt vergleichen. Er wird seinen Abgleichsversuch abbrechen, wenn der empfangene Zeitstempel kleiner ist als sein eigener Anfangszeitpunkt.

  Dieses Vorgehen ist unabhängig von der Synchronisation der Rechneruhren, benötigt also keine gemeinsame Zeitbasis. Notwendig ist nur, daß die festgelegten Anfangszeiten alle verschieden sind und eine Ordnungsrelation $\leq$ auf ihnen existiert. Für den Abgleich ist nicht wichtig, welcher Rechner ihn durchführt, es kann also aufgrund einer ad hoc definierten Ordnung ausgewählt werden.

  Leider kann es vorkommen, daß zwei Rechner gleiche Anfangszeiten auswählen und damit keine Entscheidung zwischen diesen beiden möglich ist. Für diesen Fall muß noch eine weitere Ordnung auf den Rechnern definiert sein, wie zum Beispiel der alphabetische Vergleich der Rechnernamen oder ähnliches.
- Ist der abgleichende Rechner festgelegt und sind alle Informationen bei ihm eingetroffen, so kann er wie in Kapitel 4 beschrieben für jede Datei den Zustandsgraphen aufbauen und anschließend in den Aktionsgraphen umformen.
- Enthält ein Aktionsgraph Konflikte, so wird für die entsprechende Datei eine Konfliktbehandlung ausgelöst.
- Enthält ein Aktionsgraph **equiv**-Kanten, so wird die Menge der Rechner bestimmt, die durch sie verbunden sind. Gibt es Knoten, zu denen mehrere **kopieren**-Kanten aus dieser Menge hinlaufen, so wird ein Rechner aus

der Menge ausgewählt, von dem aus die Datei kopiert werden soll. Die anderen **kopieren**-Kanten werden ignoriert. Zur Auswahl können eventuell Kriterien wie topologische Nähe im Netzwerk, Güte der Kommunikationsverbindung und ähnliches herangezogen werden. Auch der Rechner, der den Abgleich durchführt, ist eine bessere Wahl als andere, da dann eine Sendeaufforderung eingespart werden kann.

- Jetzt stehen die auszuführenden Operationen fest. Zur ihrer Durchführung verschickt der abgleichende Rechner eine Sendeaufforderung an den Rechner, der jeweils die Datei besitzt, die Ausgangsknoten von **kopieren**-Kanten im Aktionsgraphen ist. Diese Aufforderung enthält auch eine Liste von Rechnern, wohin die Datei transferiert werden soll. Analog wird zu Rechnern mit **del**-Knoten eine Löschungsanforderung verschickt.
- Sind alle Aktionen ausgeführt und bestätigt worden, so ist die Synchronisation abgeschlossen und die Warteperiode bis zum nächsten Abgleich für diesen Dateibaum beginnt zu laufen.

Es bleibt noch zu klären, wie damit umgegangen wird, wenn *während* der Durchführung einer Synchronisation Dateien verändert werden, denn es ist nicht möglich, Modifikationen während dieser Zeitspanne zu unterbinden.

Für diesen Fall wird bei den Aufforderungen zum Entgegennehmen und Abspeichern einer Datei eine *erwartete Modifikationszeit* mitgeschickt. Dieser Parameter ist der Zeitpunkt, den der annehmende Rechner für die Datei in seiner Modifikationszeitenliste angegeben hat. Vor dem Abspeichern prüft der Knoten nun, ob die aktuelle Modifikationszeit noch damit übereinstimmt. Wenn dieser Test positiv ausfällt, kann die empfangene Datei ins Dateisystem übertragen werden. Andernfalls wird sie verworfen, denn es fand in der Zwischenzeit eine Änderung an ihr statt. Es sind keine weiteren Maßnahmen erforderlich, denn die Modifikationszeit der geänderten Datei ist dann größer als die letzte Modifikationszeit und sie wird somit beim nächsten Synchronisationslauf behandelt werden.

Damit dies wie beschrieben funktioniert, muß der Zeitpunkt der aktuellen Synchronisation, der später als letzte Synchronisationszeit ls gespeichert wird, richtig gewählt werden. Dieser wird festgelegt auf das Maximum aus dem Beginn der Informationssammelphase und aller dabei ermittelten Modifikationszeiten. Damit ist zum einen sichergestellt, daß alle verwendeten Zeiten beim nächsten Abgleich tatsächlich kleiner oder gleich dem letzten Synchronisationszeitpunkt angenommen werden können. Zum anderen werden dann auch nachträgliche Änderungen sicher erkannt, da sie einen größeren Zeitstempel tragen. Daß eine Datei, die Ziel einer Kopieraktion ist, überschrieben wird, wird durch die vorher besprochene erwartete Modifikationszeit verhindert. Ist sie Quelle für Kopien zu anderen Rechnern, so ist es unschädlich, daß sie nochmals verändert wurde.

Kapitel 6

Schlußbemerkungen

6.1 Zusammenfassung der Ergebnisse

Zum Abschluß sollen hier die wichtigsten Ergebnisse zusammengefaßt werden:

- Das vorgestellte Synchronisationssystem leistet effektiv den Abgleich paralleler Dateibäume auf verschiedenen Rechnern. Dabei ist keine permanente Kommunikationsverbindung der Rechner erforderlich.
- Auch das Löschen und Neuanlegen von Dateien wird berücksichtigt und diese Operationen ebenso wie Änderungen auf andere Rechner übertragen.
- Unabhängige Änderungen auf verschiedenen Rechnern werden erkannt (eine Ausnahme siehe nächster Punkt) und soweit wie möglich sinnvoll damit umgegangen.
- Es wurde eine größere praktische Anwendung für Graphtransformationssysteme vorgestellt.
- Es hat sich als durchaus sinnvoll erwiesen, ein Graphtransformationssystem zur Lösung des gestellten Problems zu verwenden. Sicher sind hierfür auch andere Ansätze denkbar, doch die Darstellung der Dateibeziehungen als Graphen ist natürlich und anschaulich, ebenso wie die Regeln, mit denen daraus der Aktionsgraph abgeleitet wird.
- An manchen Stellen wurden Kompromisse zugunsten der Speicherplatzeffizienz eingegangen. So werden direkte Konflikte zwischen Ändern und

Löschen von Dateien nicht erkannt (Abschn. 4.1.3), diese können aber unter Umständen auf indirekte Weise doch noch festgestellt werden (siehe 4.4.5).

Auch wurde keine vollständige Pfadverfolgung (Abschn. 4.6.3) zum Erkennen identischer Dateiversionen implementiert, sondern es wird auf einen nachträglichen Identitätstest mit Hilfe von Prüfsummen zurückgegriffen. Dies impliziert eine — wenn auch sehr kleine — Fehlerquote, so daß in seltenen Fällen Konflikte gemeldet werden können, die in der Realität nicht vorhanden sind.

6.2 Ausblick

Ein verbleibendes Problem ist noch die Voraussetzung, daß die Uhren der beteiligten Rechner, aus denen auch die lokalen Modifikationszeiten abgeleitet werden, in gewissem Rahmen synchron laufen müssen (4.1), damit Zeitstempel vergleichbar sind. Hier verläßt sich das Abgleichssystem bisher auf externe Maßnahmen, beispielsweise das Network Time Protocol [Mil91, Mil92].

Es wäre jedoch möglich, Zeitunterschiede auch innerhalb des Systems zu kompensieren. Dazu müßte man zu Beginn eines Abgleichs die Zeitdifferenz der Knoten zu dem Rechner, der den Abgleich durchführt, messen. Dies erweist sich jedoch aufgrund der von Null verschiedenen und variablen Nachrichtenlaufzeiten als nichttriviales Problem.

Eine Alternative dazu wäre, für jeden Rechner ein eigenes Zeitsystem anzunehmen, in dem das Zeitmaß zwar monoton, aber unabhängig von den anderen Rechnern voranschreitet. Die lokalen Modifikationszeiten werden innerhalb eines solchen Systems vergeben. Es dürften dann aber keine Vergleiche von Zeitstempeln aus unterschiedlichen Zeituniversen stattfinden. Diese Vorgehensweise ist im Grundsatz möglich, wenn man die Forderung nach Symmetrie der letzten Synchronisationszeit ls aufgibt. $ls(r_1, r_2)$ wäre dann im System von r_1 angegeben und $ls(r_2, r_1)$ ein äquivalenter absoluter Zeitpunkt im System von r_2. Bei diesem Ansatz wäre zu prüfen, ob alle Annahmen und Beweise unter der neuen Voraussetzung noch gelten. Auch gibt es bisher Fälle, in denen Zeitstempel die Grenzen ihres Systems verlassen, beispielweise die Modifikationszeit kopierter Dateien, die unter anderem vom Zeitstempel der Originaldatei abgeleitet wird. Es wäre zu klären, wie hiermit sinnvoll umgegangen werden kann.

Bei den Kompromissen, die zugunsten der Speicherplatzeffizienz eingegangen wurden, könnte noch nach weiteren Lösungsmöglichkeiten gesucht werden. Während das Verfolgen von Pfaden, die von einer gemeinsamen Originaldatei ausgehen (4.6.3), für die Praxis zufriedenstellend gelöst ist, stört es doch, daß direkte Konflikte zwischen Löschen und Ändern von Dateien nicht erkannt werden.

Eine weitere denkbare Erweiterung des Systems ist eine Art kontinuierlicher Abgleich ohne Unterteilung in Phasen der Synchronisation und abgleichsfreie Zeiten. Dafür wäre es wünschenswert, die Granularität der Abgleichseinheiten feiner zu gestalten. Eine Synchronisation ist zwar auch mit dem vorliegenden System auf Dateiebene möglich, jedoch muß dann für jede Datei ein eigener letzter Synchronisationszeitpunkt gespeichert werden, was der Speicherplatzeffizienz zuwiderläuft. Eine sinnvolle größere Gruppierung wären Abgleiche auf Verzeichnisebene. Auch das dynamische Hinzufügen oder Herausnehmen von Rechnern aus dem Synchronisationsnetz dürfte keine grundlegenden Probleme aufwerfen, müßte jedoch näher betrachtet werden.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Ave95] J. Avenhaus. *Reduktionssysteme. Rechnen und Schließen in gleichungsdefinierten Strukturen.* Springer Verlag, Berlin, 1995.

[Ber90] Brian Berliner. CVS II: Parallelizing software development. In *Proc. USENIX Winter 1990*, Washington, D.C., 1990.

[BL90] Y. Bernard and P. Lavency. A process-oriented approach to configuration management. In *11th Int. Conf. on Software Engineering*, pages 320–330, Los Alamitos, CA, 1990. IEEE Computer Society Press.

[CEH+97] Andrea Corradini, Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, and Anika Wagner. Algebraic approaches to graph transformation - part I: Single pushout approach and comparison with double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*, chapter 4, pages 247–312. World Scientific, 1997.

[Coo92] Michael A. Cooper. Overhauling rdist for the '90s. In *Proc. USENIX LISA VI*, Long Beach, CA, 1992.

[Der86] N. Dershowitz. Termination. In J. P. Jouannaud, editor, *Rewriting Techniques and Applications*, number 202 in Lecture Notes in Computer Science, pages 180–224. Springer Verlag, 1986.

[Der87a] N. Dershowitz. Corrigendum to termination of rewriting. *Journal of Symbolic Computing*, 4:409–410, 1987.

[Der87b] Nachum Dershowitz. Termination of rewriting. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, pages 69–115. Academic Press, 1987. Reprinted from *Journal of Symbolic Computation.*

[Der93] N. Dershowitz. A taste of rewrite systems. In P. E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 199–228. Springer Verlag, 1993.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewriting systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 243–320. Elsevier Publishers, Amsterdam, 1990.

[EH86] Hartmut Ehrig and Annegret Habel. Graph grammars with application conditions. In G. Rozenberg and A. Salomaa, editors, *The Book of L*, pages 87–100. Springer-Verlag, Berlin, 1986.

[Ehr79] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69, 1979.

[Ehr87] Hartmut Ehrig. Tutorial introduction to the algebraic approach of graph-grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph Grammars and Their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science*, pages 3–14, 1987.

[EKL91] Hartmut Ehrig, Martin Korff, and Michael Löwe. Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts. In H. Ehrig, Hans-Jörg Kreowski, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 24–37, 1991.

[EPS73] H. Ehrig, M. Pfender, and H.-J. Schneider. Graph grammars: An algebraic approach. In *IEEE Proc. Conf. Switching and Automata Theory*, pages 167–180, Iowa City, 1973.

[Fis99] Ingrid Fischer. *Describing Neural Networks with Graph Transformations.* PhD thesis, Universität Erlangen-Nürnberg, 1999.

[HKM$^+$88] John H.Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6-1:51–81, 1988.

[HM85] H. Herold and M. Meyer. *SCCS und RCS.* Addison-Wesley, Reading, MA, 1985.

[HW95] R. Heckel and A. Wagner. Ensuring consistency of conditional graph grammars — A constructive approach. Electronic Notes in Theoretical Computer Science, pages 95–104, Pisa, Italy, 1995.

[Kaz88] Michael L. Kazar. Synchronisation and caching issues in the andrew file system. In *Proc. USENIX Winter 1988*, Dallas, TX, 1988.

[Klo87] J. W. Klop. Term rewriting systems. In *EATCS Bulletin 32*. 1987.

[Löw92] Michael Löwe. Theorie funktionaler Ersetzungssysteme. Lecture Notes, Technical University of Berlin, October 1992.

[Mil91] David L. Mills. Internet time synchronization: The Network Time Protocol. *IEEE Trans. Communications COM*, (39, 10):1482–1493, Oktober 1991.

[Mil92] David L. Mills. *Network Time Protocol (Version 3) specification, implementation and analysis.* Network Working Group Report. University of Delaware, März 1992. RFC 1305.

[MM85] Webb Miller and Eugene W. Myers. A file comparison program. In *Software—Practice and Experience*, volume 15 No. 11, pages 1025–1040. 1985.

[Mye86] Eugene W. Myers. An O(ND) difference algorithm and its variations. In *Algorithmica*, volume 1 No. 2, pages 251–266. 1986.

[PEM87] Francesco Parisi-Presicce, Hartmut Ehrig, and Ugo Montanari. Graph rewriting with unification and composition. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph Grammars and Their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science*, pages 496–514, 1987.

[PR69a] J.L Pfaltz and A. Rosenfeld. Web grammars. In *Proc. Int. Joint Conference on Artificial Intelligence*, pages 609–619, 1969.

[PR69b] John L. Pfaltz and Azriel Rosenfeld. Web grammars. In *Int. Joint Confenence on Artificial Intelligence*, pages 609–619, 1969.

[Riv92] R. Rivest. The MD5 Message-Digest Algorithm. MIT and RSA Data Security, Inc., RFC 1321, April 1992. `http://www.cis.ohio-state.edu/rfc/rfc1321.txt`.

[RM72] A. Rosenfeld and D. Milgram. Web automata and web grammars. In *Machine Intelligence*, pages 307–324, 1972.

[Sch89] A. Schürr. Introduction to progress, an attribute graph grammar based specification language. In *Graph Theoretic Concepts in Computer Science*, volume 411 of *Lecture Notes in Computer Science*, Berlin, 1989. Springer Verlag.

[Sch91] A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen: Formale Definitionen, Anwendungen und Werkzeuge.* Deutscher Universitätsverlag, Braunschweig, 1991. Dissertation, RWTH Aachen.

[Sch93] H.-J. Schneider. On categorical graph grammars: Integrating structural transformations and operations on labels. *Theoretical Computer Science*, 109, 1993.

[Sch95a] H.-J. Schneider. Programmiersprachen zur Beschreibung paralleler Prozesse. In K. Waldschmidt et al., editor, *Parallelrechner: Architekturen, Systeme, Werkzeuge.* Teubner Verlag, Stuttgart, 1995.

[Sch95b] H.J. Schneider. Algorithmische Grundlagen Regelbasierter Systeme. Lecture Notes IMMD2, Universität Erlangen-Nürnberg, 1995.

[Sch99a] H.-J. Schneider. Application of Double-Pushout Approach to Describing Systems of Processes. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. III: Concurrency.* World Scientific, 1999.

[Sch99b] H.-J. Schneider. Graphtransformationssysteme. Lecture Notes IMMD2, Universität Erlangen-Nürnberg, 1999.

[Ukk85] E. Ukkonen. Algorithms for approximate string matching. In *Information and Control*, volume 64, pages 100–118. 1985.

[ZE96] Ed Zayas and Craig Everhart. Design and specification of the cellular andrew environment. Technical Report CMU-ITC-070, Information Technology Center, 1996.

Stichwortverzeichnis

Lebenslauf

Persönliche Daten

Name	Roman Hodek
geboren am	24.09.1969 in Ingolstadt/Do.
Eltern	Rüdiger Hodek, geb. am 11.03.1943 in Falkenau Ulrike Hodek geb. Brey, geb. am 06.07.1945 in Ingolstadt/Do.
Staatsangehörigkeit	deutsch

Ausbildung

09/1975 – 07/1978	Besuch der Volksschule an der Münchener Straße, Ingolstadt
09/1978 – 06/1988	Besuch des Apian-Gymnasium Ingolstadt Abschluß: allgemeine Hochschulreife
11/1990 – 06/1995	Studium der Informatik an der Friedrich-Alexander-Universität Erlangen-Nürnberg Abschluß: Diplom-Informatiker Univ.

Sonstiges

02/1989 – 09/1990	Zivildienst am Pflegeheim Ingolstadt
08/1995 – 07/2000	Beschäftigung als hauptberuflicher wissenschaftlicher Mitarbeiter am Lehrstuhl für Programmier- und Dialogsprachen (IMMD II) der Friedrich-Alexander-Universität Erlangen-Nürnberg
seit 08/2000	Tätigkeit bei Caldera (Deutschland) GmbH