Universität Ulm
Abt. Datenbanken und Informationssysteme
Leiter: Prof. Dr. P. Dadam

Effiziente Verarbeitung von Produktstrukturen in weltweit verteilten Entwicklungsumgebungen

DISSERTATION
zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Informatik
der Universität Ulm

vorgelegt von

ERICH MÜLLER
aus Krumbach (Schwaben)

April 2003

Amtierender Dekan:	Prof. Dr. F. W. von Henke
Gutachter:	Prof. Dr. P. Dadam Prof. Dr. M. Weber
Tag der Promotion:	17. Juli 2003

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter in der Abteilung Datenbanken und Informationssysteme der Universität Ulm. Den Anstoß für die Bearbeitung des Themas „Effiziente Verwaltung von Produktstrukturen in weltweit verteilten Entwicklungsumgebungen“ gaben mehrere Projekte, die im Rahmen einer Forschungskooperation mit der Abteilung RIC/ED des DaimlerChrysler-Forschungszentrums in Ulm durchgeführt wurden. In diesen Projekten konnte ich wertvolle Erfahrungen und Erkenntnisse gewinnen, die diese Arbeit maßgeblich beeinflussten.

Mein Dank gilt zunächst meinem Doktorvater, Herrn Professor Dr. Peter Dadam, der mich durch alle Höhen und Tiefen während der vergangenen Jahre begleitet und auch in schwierigen Situationen stets unterstützt und vorangebracht hat. Danken möchte ich auch Herrn Professor Dr. Michael Weber für seine Unterstützung und die Anfertigung des Zweitgutachtens. Allen Kolleginnen und Kollegen der Abteilung DBIS danke ich herzlich für die gute Zusammenarbeit und die immer freundschaftliche Atmosphäre am Arbeitsplatz. Ebenso danke ich den Kollegen des DaimlerChrysler-Forschungszentrums, die mir oft mit Hintergrundinformationen zum Thema Produktentwicklung geholfen und mir damit die Praxis ein gutes Stück näher gebracht haben.

Ein besonderer Dank gilt meinem Kollegen und Freund Jost Enderle, der mich in vielen, oft stundenlangen Diskussionen unterstützt und weitergebracht hat. Ich möchte mich bei ihm und bei Stefanie Rinderle für das sorgfältige Korrekturlesen der Arbeit bedanken. Mein Dank gilt auch Rudi Seifert für die verlässliche technische Betreuung.

Ich danke meinen Eltern, die mich auch in privaten Angelegenheiten sehr unterstützt und mir damit viel Zeit für die Erstellung der Arbeit verschafft haben. Schließlich danke ich besonders meiner Frau Katja, ohne deren liebevolle Unterstützung, Ermutigungen und Entlastung diese Arbeit wohl nicht fertiggestellt worden wäre.

Kettershausen, im April 2003 *Erich Müller*

Kurzfassung

Die Produktentwicklung ist ein zeitaufwändiger, kostenintensiver Prozess. Unterstützung bieten die so genannten Produktdatenmanagement-Systeme (PDM-Systeme), die Informationen über alle produktrelevanten Daten verwalten. Dazu setzen diese Systeme typischerweise auf relationalen Datenbankmanagementsysteme auf, die sie jedoch mehr oder weniger als primitives Dateisystem verwenden. In lokalen Entwicklungsumgebungen werden PDM-Systeme bereits zum Teil sehr erfolgreich eingesetzt, in geographisch verteilten Szenarien jedoch verhindern die extrem langen Antwortzeiten einiger Benutzeraktionen einen profitablen Einsatz.

Gegenstand dieser Arbeit ist die Entwicklung einer Architektur kombiniert mit einer Auswertungsstrategie für die problematischen Aktionen, so dass auch in weltweit verteilten Entwicklungsumgebungen akzeptable Antwortzeiten zu erwarten sind. Der Schlüssel dazu liegt in der Optimierung der Systeme hinsichtlich ihres Kommunikationsverhaltens, das an die spezifischen Eigenschaften von Weitverkehrsnetzen anzupassen ist.

Heute verfügbare PDM-Systeme übertragen oft Daten, auf die der anfragende Benutzer keine Zugriffsrechte besitzt. Mit einer möglichst frühzeitigen Auswertung der Zugriffsrechte kann dies verhindert werden. Dazu wird in der Arbeit ein Ansatz beschrieben, der die Zugriffsbedingungen bereits beim Zugriff auf das Datenbankmanagementsystem überprüft, indem die Bedingungen auf SQL-Sprachkonstrukte abgebildet werden, und damit eine deutliche Senkung des übertragenen Datenvolumens ermöglicht.

Ein weiterer Kritikpunkt ist die Häufigkeit, mit der heutige verteilt installierte PDM-Systeme kommunizieren. Durch Bereitstellung von geeigneten Informationen über die Verteilung der Produktdaten sowie gezieltes Parallelisieren von Aufgaben unter Ausnutzung dieser Information lässt sich die Anzahl der Kommunikationen minimieren. In der Arbeit wird dazu eine Art Index definiert, bezeichnet als Object-Link-and-Location-Katalog. Dieser Katalog ist standortspezifisch und enthält Verweise auf Objekte an entfernten Standorte, die direkt oder indirekt (transitiv) mit lokalen Produktdaten verknüpft sind. Bei der Suche nach Daten zu einem gegebenen Teilprodukt ist auf Grund dieser Kataloginformation unmittelbar bekannt, welche entfernten Standorte in die Suche einbezogen werden müssen. Indem komplexe Suchanfragen an diese Standorte parallel versendet werden, können enorme Einsparungen hinsichtlich der Antwortzeit erzielt werden.

Die Tauglichkeit der in der Arbeit vorgestellten Ansätze wird durch mehrere Simulationen verschiedener praxisrelevanter Szenarien nachgewiesen.

Inhaltsverzeichnis

Teil I

Motivation, Problemwelt und Grundlagen

Kapitel 1

Einleitung

Die konsistente, effiziente und durchgängige Verwaltung von produktbezogenen Daten ist eine große Herausforderung für viele Unternehmen des produzierenden Gewerbes. Ein vielversprechender Ansatz auf dem Weg zur Erreichung dieses Ziels sind die so genannten *Produktdatenmanagement-Systeme* (PDM-Systeme) [CIM97, Hew93, VDI99]. Heute erhältliche PDM-Systeme, die hauptsächlich in der Produktentwicklung und -fertigung eingesetzt werden, sind jedoch für den Einsatz in Unternehmen mit weltweit verteilten und miteinander kooperierenden Standorten nicht geeignet. Antwortzeiten für Benutzeraktionen, die im lokalen Kontext noch im akzeptablen Bereich liegen, können im „interkontinentalen“ Umfeld um Größenordnungen ansteigen. Gegenstand dieser Arbeit ist die Entwicklung eines Verfahrens, welches eine effiziente Verarbeitung von Produktstrukturinformationen in derartigen weltweit verteilten Entwicklungsumgebungen ermöglicht. Dabei werden die Nachteile, die sich auf Grund des Einsatzes von Weitverkehrsnetzen ergeben, durch eine Optimierung des Kommunikationsverhaltens der PDM-Systeme weitgehend kompensiert.

1.1 Produktdatenmanagement

Die Produktentwicklung ist ein teurer und zeitaufwändiger Prozess. Besonders in technologisch fortschrittlichen Wirtschaftsbereichen, wie Automobilbau, Luft- und Raumfahrt und Elektronikgerätebau, zwingt der ständig wachsende Konkurrenzkampf die Unternehmen dazu, diesen Prozess mehr und mehr zu rationalisieren und dabei gleichzeitig die Qualität der Produkte zu steigern, um am Markt überleben zu können. Die Rahmenbedingungen hierzu werden durch die fortschreitende Globalisierung der Märkte definiert: Die Anzahl an Produkt- und Ausstattungsvarianten, und damit auch die Komplexität der Produkte, steigt immer

mehr an. Gleichzeitig ist eine Verkürzung der Produktlebenszyklen zu beobachten, weshalb neue Ideen möglichst schnell umgesetzt werden müssen, um die ohnehin geringen Möglichkeiten der Gewinnerzielung noch nutzen zu können.

In den vergangenen Jahren wurden enorme Anstrengungen unternommen, um die in den Entwicklungsprozess involvierten Disziplinen zu optimieren. CAx-Systeme (Computer Aided Design, Computer Aided Engineering, Computer Aided Manufacturing etc.) werden eingesetzt, um die Zeitspanne zwischen Produktidee und Markteinführung (*time-to-market*) zu verkürzen und sich dadurch einen Vorteil gegenüber den Mitbewerbern zu schaffen.

Diese *intra*-disziplinäre Optimierung, mit welcher durchaus bemerkenswerte Anfangserfolge erzielt werden konnten, führt nicht automatisch zu einer Verbesserung des gesamten Entwicklungsprozesses – die Verbesserungen beschränken sich vielmehr auf einzelne Teilbereiche des Ablaufs. Die Ursachen dieser Einschränkung sind vielfältig: Die genannten Systeme stellen „Insellösungen“ für spezielle Aufgaben dar, wobei typischerweise jedes System einen eigenen Datenspeicher zur Verwaltung der Anwendungsdaten verwendet. Eine Integration ist zunächst nicht gegeben. Folglich wird zum Beispiel das Suchen nach Daten nur unzureichend unterstützt, Korrektheit und Konsistenz gemeinsam genutzter Daten kann kaum garantiert werden, ein systemübergreifendes Änderungs- und Konfigurationsmanagement sucht man vergebens, und schließlich sind auch die Möglichkeiten des kontrollierten parallelen Arbeitens (*concurrent engineering*) nur rudimentär vorhanden. Diese Schwächen können nur durch eine *inter*-disziplinäre Optimierung beseitigt werden (vgl. [JRR91]).

Produktdatenmanagement-Systeme greifen diese Idee auf: Alle produktrelevanten Daten, die während des Produktlebenszyklus, beginnend vom Produktdesign über Entwicklung und Produktion bis hin zu Verkauf, Wartung und Verwertung, anfallen, sollen über eine „zentrale“ Stelle erreichbar sein. Zu diesen Daten zählen Strukturinformationen, die den Aufbau des Produktes wiedergeben, und produktbeschreibende Informationen wie Spezifikationen, CAD-Zeichnungen etc. Des Weiteren sollen alle Prozesse, die während des Zyklus durchlaufen werden, über dieses System koordiniert und gesteuert werden. Dazu zählt im Besonderen das Änderungsmanagement, welches eine konsistente Weiterentwicklung und Anpassung des Produktes gewährleisten soll.

Damit stellt ein PDM-System das *Informationszentrum für alle Personen* dar, die eine Aufgabe im Produktlebenszyklus wahrnehmen. Zwei typische, in den verschiedensten Prozessen häufig ausgeführte Aktionen der Benutzer sind beispielsweise die Suche nach Objekten mit gegebenen Eigenschaften und die Navigation in der Produktstruktur („Aus welchen Bauteilen besteht das Produkt?“ oder die umgekehrte Richtung „In welchem Produkt wird ein Bauteil verwendet?“). Wei-

tere Aktionen sind das Anlegen neuer Versionen von Bauteilen und Dokumenten, Freigeben von Konstruktionszeichnungen für die Produktion und vieles mehr.

Die Anwender erwarten selbstverständlich, dass diese Aktionen jederzeit auf den aktuellen Daten konsistent und performant ausgeführt werden. Zu berücksichtigen sind dabei die Zugriffsberechtigungen der Anwender, d. h. nicht jeder Anwender darf alle Operationen auf allen Objekten ausführen.

Weitergehende allgemeine Ausführungen zum Produktdatenmanagement und den PDM-Systemen können [Hew93, Pel00, Sch99, VDI99] entnommen werden.

1.2 Weltweit verteilte Produktentwicklung

In den letzten Jahren hat sich der Trend zur Globalisierung der Unternehmen verstärkt. Unternehmenszusammenschlüsse und Kooperationen unterschiedlichster Art sind heute keine Seltenheit mehr. Neue Märkte sollen dadurch erschlossen werden, aber auch Überlegungen zu Rationalisierungen in Entwicklung, Produktion und Vertrieb stehen im Vordergrund.

Hersteller ähnlicher Produkte beispielsweise fusionieren, um Entwicklungsarbeit einzusparen. Bei vielen Elektrogeräten etwa wird die funktionale Einheit nur *einmal* entwickelt und dann allenfalls in unterschiedliche Gehäuseformen eingebaut (oder gar nur mit unterschiedlichen Herstellernamen versehen).

Kooperationen mit Unternehmen in Billiglohnländern sollen helfen, Produktionskosten zu senken. Die Entwicklung findet oftmals noch im „Mutterland“ statt, Experten unterstützen die Produktion typischerweise vor Ort, so dass entsprechend den Maßgaben des Herstellers gefertigt werden kann.

Besonders interkontinentale Unternehmenszusammenschlüsse können auch helfen, ferne Märkte für eigene Produkte zu öffnen. Vorteilhaft erweist sich zum Beispiel, dass das Vertriebsnetz des Fusions-Partners für die Vermarktung des eigenen Produktes verwendet werden kann.

Für die verteilte Produktentwicklung sprechen mehrere Gründe. Naheliegend ist die Vergabe von Entwicklungsaufträgen an externe Entwicklungs-Büros, die sich auf entsprechende Aufgaben spezialisiert haben. Eigenentwicklungen sind oftmals teurer als die Bezahlung externer Spezialisten. Auch die Verteilung der Entwicklungsaufgaben unter Kooperationspartnern entsprechend deren Fähigkeiten und Möglichkeiten ist häufig anzutreffen. Auch hier wird die Spezialisierung als Mittel zur Rationalisierung eingesetzt.

In anderen Fällen wird die verteilte Entwicklung dazu genutzt, länderspezifische Versionen eines Produktes zu entwickeln. Typischerweise sind die Mitarbeiter vor

Ort mit den gültigen gesetzlichen Vorschriften vertraut, auch auf die regional und lokal doch zum Teil recht unterschiedlichen Kundenwünsche kann geeignet reagiert und eingegangen werden.

Großprojekte in der Luft- und Raumfahrt, zum Beispiel die Entwicklung eines neuen Flugzeugs, besitzen oft ihre eigenen Gesetze. Die Entwicklung (und Fertigung) obliegt einem Konsortium von mehreren Partnern, die alle einen möglichst großen Anteil am neuen Produkt beisteuern und damit möglichst große Gewinne erzielen möchten. Um eine „gerechte“ Verteilung zu erreichen, werden in solchen Projekten häufig die angestrebten Verkaufszahlen des neuen Produktes in den verschiedenen Ländern bei der Vergabe der Aufträge herangezogen: Wer viele Instanzen des Produktes zu kaufen beabsichtigt, bekommt tendenziell einen größeren Anteil (*Workshare*) an der Entwicklung und Fertigung zugesprochen.

Besonders in militärischen Projekten findet dieses Vorgehen Anwendung. Hier spielen typischerweise zusätzlich politische Entscheidungen eine Rolle, die hier nicht näher betrachtet werden sollen.

An einer Kollaboration beteiligte Unternehmen lassen sich grob in *Partner* und *Zulieferer* einteilen. Unter Partnern versteht man in der Regel wenige, meist große Unternehmen, die gleichberechtigt (bezüglich das Gesamtprojekt betreffender Entscheidungen) ein Projekt durchführen. Dabei kann ein Partner die Gesamtkoordination übernehmen und Schnittstelle zum Kunden sein.

Zulieferer sind typischerweise Unternehmen, die auf Auftragsbasis arbeiten und kein oder nur sehr eingeschränktes Mitspracherecht bezüglich des Gesamtprojektes besitzen. An Großprojekten können einige hundert bis zu wenigen tausend Zulieferern beteiligt sein, die aber nicht notwendigerweise alle in der Produkt*entwicklung* sondern eher in der Produktion tätig sind.

In ersten „Gehversuchen“ in lokalen Umgebungen konnten PDM-Systeme ihren Nutzen unter Beweis stellen. Dabei wurden besonders die Anwender aus dem Bereich der Produktentwicklung unterstützt, so dass die Entwicklungsprozesse zum Teil deutlich optimiert werden konnten.

Für den Einsatz in weltweit verteilten Entwicklungsumgebungen wurden daraufhin ähnliche „Erfolgsmeldungen“ erwartet. Von der anfänglichen Euphorie auf Seiten der Anwender ist jedoch heute nichts mehr vorhanden, vielmehr hat sich eine gewisse ablehnende Haltung gegenüber PDM-Systemen aufgebaut. Der Grund hierfür liegt in den inakzeptablen Antwortzeiten, welche die Systeme besonders für Aktionen produzieren, die über die Produktstruktur navigieren. Aktionen, die in einem lokalen Umfeld nur ein paar Sekunden dauern, benötigen im weltweit

verteilten Kontext mehrere Minuten, obwohl nur relativ wenige Daten zu übertragen sind.

Der Grund für diese eklatanten Verzögerungen zeigt sich bei einem Blick auf die Architektur der PDM-Systeme. Zur Speicherung der Informationen werden typischerweise relationale Datenbankmanagementsysteme eingesetzt. Die Fähigkeiten, die diese Systeme bieten, werden jedoch nicht ausgereizt, vielmehr werden sie wie ein „dummes Dateisystem“ eingesetzt. Navigationen in der Produktstruktur, z. B. das Absteigen von einem Bauteil zu dessen Unterteilen, werden dabei in eine Sequenz von isolierten, primitiven Datenbankanfragen umgesetzt. Für alle diese Anfragen fallen nun Kommunikationskosten an, insbesondere die im Weitverkehrsnetz langen Latenzzeiten (300–500ms sind durchaus üblich) sind „teuer“. Diese Verzögerungen summieren sich auf und führen zu den entsprechend langen Antwortzeiten.

Neben den vielen Kommunikationen ist auch zu beobachten, dass von dem Datenbankmanagementsystem Daten an das PDM-System übertragen werden, auf die der anfragende Benutzer kein Zugriffsrecht besitzt. Besonders in weltweit verteilten Entwicklungsumgebungen, in denen mehrere hundert Personen in die Entwicklung involviert sind und noch mehr Anwender in nachgelagerten Schritten des Produktlebenszyklus auf die dabei erzeugten Produktdaten zugreifen, sind die Unternehmen sehr darauf bedacht, den Mitarbeitern nur stark eingeschränkte und auf ihre Bedürfnisse ausgerichtete Zugriffsrechte zu gewähren.

In der Praxis ist dies den Anwendern offensichtlich kaum bewusst: Sie stellen Anfragen an das System, auf die sich oft hunderte von Objekten qualifizieren. In der Regel wird jedoch nur ein kleiner Bruchteil davon letztlich auf dem Bildschirm angezeigt. Da der Anfrager nur relativ wenige Ergebnisobjekte sieht, mag es für ihn so aussehen, als ob seine Anfrage entsprechend selektiv gestaltet wäre – in Wahrheit jedoch wurde das Ergebnis, unsichtbar für den Benutzer, auf Grund seiner Zugriffsrechte so stark eingeschränkt.

Diese Einschränkung jedoch erfolgt bei heutigen PDM-Systemen zu einem sehr späten Zeitpunkt. Die angefragten Daten werden – unabhängig von den Rechten des anfragenden Benutzers – zunächst von der Datenbank ermittelt und an das PDM-System weitergereicht. Erst dort werden die Objekte, auf die der Benutzer nicht zugreifen darf, aus dem Ergebnis aussortiert. Die Objekte, die die Prüfung der Zugriffsrechte bestanden haben, werden anschließend visualisiert. Dieses Verfahren ist teuer, falls Datenbankmanagementsystem und PDM-System über ein Weitverkehrsnetz miteinander verbunden sind – in weltweit verteilten Entwicklungsumgebungen ist dies selbstverständlich keine Seltenheit!

Aus diesen „Verhaltensweisen“ der PDM-Systeme kann geschlossen werden, dass sie nicht für den Einsatz in weltweit verteilten Umgebungen konzipiert wurden.

Lokale Umgebungen werden ausreichend unterstützt, d. h. die Anwender bemerken keine Einschränkungen auf Grund systeminhärenter Schwächen. Prinzipiell funktionieren die Systeme auch in verteilten Szenarien, die aktuell anfallenden Antwortzeiten jedoch verhindern einen produktiven, gewinnbringenden Einsatz.

1.3 Herausforderung

Die Herausforderung besteht nun darin, ein PDM-System zu konzipieren, welches insbesondere bei weltweit verteiltem Einsatz akzeptable Antwortzeiten garantiert. Das Hauptaugenmerk liegt dabei auf der effizienten Unterstützung der in den Produktdaten navigierenden Benutzeraktionen, die in heutigen Systemen die größten Performance-Probleme bereiten.

Auf den ersten Blick scheinen sich die beschriebenen Probleme sehr leicht lösen zu lassen: Da die Netzwerkkomponente offensichtlich das limitierende Element darstellt, bietet sich an, schnellere Netze einzusetzen, etwa ATM, FDDI oder dergleichen [SHK+97, Tan89]. Doch bereits ein Blick in die Vergangenheit zeigt, dass die Verbesserung der Netzwerke keine dauerhafte Lösung des Problems sein kann: Zusätzlich bereitgestellte Kapazitäten führen kurzfristig oftmals zu Verbesserungen, sie werden jedoch innerhalb kürzester Zeit durch neu hinzukommende Applikationen und Benutzer, die ebenfalls auf die bestehenden Ressourcen zugreifen, wieder absorbiert – die Antwortzeiten steigen wiederum auf inakzeptable Werte. Des Weiteren sind Netze basierend auf der neuesten Technologie nicht in allen Regionen verfügbar oder nicht zu akzeptablen Konditionen erhältlich.

Gesucht wird folglich eine PDM-Systemarchitektur, mit welcher sich die Performance-Probleme lösen lassen, ohne in die bestehende Infrastruktur und die (semantisch bedingte) Datenverteilung einzugreifen. Das bedeutet, dass bei gleichbleibender Funktionalität durch eine intelligente Anfrage- und Bearbeitungsstrategie die Anzahl der Kommunikationen (insbesondere sind hier die Round-Trips zum Datenbankmanagementsystem im Fokus) sowie das dabei transportierte Datenvolumen minimiert werden müssen.

Die Reduktion des übertragenen Datenvolumens kann erreicht werden, indem nur Daten transportiert werden, auf welche der anfragende Benutzer Zugriffsrechte besitzt. Die heute eingesetzte Strategie, Zugriffsrechte erst unmittelbar vor der Visualisierung der Objekte zu prüfen, muss durch eine *frühzeitige Regelauswertung* ersetzt werden, die es ermöglicht, nicht-zugreifbare Daten schon *vor* der Übertragung über das Weitverkehrsnetz auszufiltern.

Die heutige, primitve sequentielle Navigation in den Produktdaten, in welcher das Datenbankmanagementsystem als einfaches Dateisystem verwendet wird, muss

durch einen kompakten und effizienten Navigationsmechanismus ersetzt werden. Insbesondere bei rekursiv ausgeführten Navigationsschritten auf entfernten Daten müssen die vielen isolierten Datenbankanfragen zu einer – vergleichsweise komplexen – Anfrage zusammengefasst werden, um Kommunikationszyklen zu sparen. Die resultierende Anfrage sollte dabei so konzipiert sein, dass die Funktionalität des Datenbankmanagementsystems auch möglichst effektiv genutzt werden kann. D. h. Funktionalität, die das DBMS bereitstellt, sollte auch dort genutzt und nicht etwa im PDM-System nachimplementiert werden.

Es ist zu erkennen, dass eine einfache, primitive Lösung nicht in Sicht ist: Schnellere Netze, basierend auf neuester Technologie, bringen langfristig nicht den gewünschten Erfolg, und die beschriebenen Herausforderungen lassen schon jetzt annehmen, dass die benötigten Verbesserungen nicht durch geringfügige Anpassungen heute verfügbarer PDM-Systeme erzielt werden können.

1.4 Ziel und Aufbau der Arbeit

Ziel dieser Arbeit ist die Entwicklung einer Architektur für verteilte PDM-Systeme sowie einer Auswertungsstrategie für rekursive Navigationsoperationen, die unter Berücksichtigung frühzeitiger Zugriffsregelauswertung minimale Antwortzeiten garantiert. Bei der Bearbeitung des Gesamtproblems ergeben sich im Wesentlichen folgende Fragestellungen, die in der Arbeit behandelt werden:

- Welches Datenbank-Paradigma ist zur Speicherung von Produktdaten in weltweit verteilten Entwicklungsumgebungen geeignet? Sind relationale oder objektorientierte Datenbankmanagementsysteme die richtige Wahl?
- Welche Arten von Zugriffsregeln werden in PDM-Systemen angewendet? Wie lassen sich diese Regeln darstellen und welche Darstellung ist für eine optimale, frühzeitige Auswertungsstrategie geeignet?
- Welche Zugriffsstrategie hinsichtlich rekursiver Navigationsoperationen kann auf verteilten Produktdaten effizient angewendet werden? Bietet SQL:1999 genügend Funktionalität, um Rekursion abzudecken?
- Wie lassen sich Kommunikationen bei rekursiven Navigationsoperationen zwischen verschiedenen Standorten einsparen? Welche Zusatzinformationen über die Verteilung der Daten ist eventuell erforderlich?
- Welches Potential verbirgt sich hinter der angedachten Optimierung? Welche Einsparungen können mit frühzeitiger Regelauswertung und optimiertem Kommunikationsverhalten theoretisch und praktisch erzielt werden?

- Welche Bausteine enthält eine Architektur für PDM-Systeme, die mit diesen Optimierungen gebaut werden sollen?

Die vorliegende Arbeit baut auf diesen Fragestellungen auf. Den Abschluss des Motivationsteils bildet Kapitel 2. Es enthält eine Einführung in die Grundlagen und Problematik der PDM-Systeme. Es wird vermittelt, welche Daten in diesen Systemen verwaltet werden und welche Operationen darauf angewendet werden können. Anschließend werden die Probleme heutiger Systeme dargestellt und über ein mathematisches Modell quantitativ erfasst. Es folgt eine Diskussion inadäquater Lösungsansätze. Dabei wird gezeigt, dass durch den Einsatz neuerer Netzwerktechnologien sowie durch den Wechsel des Datenbankparadigmas die gezeigten Probleme nicht gelöst werden können. Da den folgenden Kapiteln die Verwendung relationaler Datenbankmanagementsysteme zu Grunde liegt, wird eine Repräsentation von Produktdaten in einem derartigen System vorgeschlagen. Das Kapitel endet mit einigen graphentheoretischen Definitionen zur mathematischen Beschreibung von Produktdaten, die als Grundlage für die weitere Arbeit benötigt werden.

Teil II (Kapitel 3 – Kapitel 7) bildet den Hauptteil dieser Arbeit. Hier werden zunächst die Bausteine für eine geeignete Architektur verteilter PDM-Systeme entwickelt. Abschließend wird über eine simulative Analyse die Effizienz dieser Architektur nachgewiesen.

Kapitel 3 befasst sich mit den Zugriffsregeln in PDM-Systemen. Zunächst werden Zugriffsregeln in verschiedene Kategorien eingeteilt und anhand diverser Merkmale diskutiert. Es folgt die Diskussion verschiedener Auswertungsstrategien mit dem Ziel, eine möglichst frühzeitige Evaluierung der Regeln zu ermöglichen. Anschließend wird die Darstellung der Regeln mit SQL-Mitteln, beispielsweise in WHERE-Klauseln, sowie deren Integration in Datenbankanfragen beschrieben.

In Kapitel 4 werden rekursive Anfragen an Produktdaten behandelt. Dabei wird ein Szenario mit zentraler Datenhaltung sowie verteilter Datenverarbeitung zu Grunde gelegt. Es wird gezeigt, wie die Produktstrukturdaten mittels rekursivem SQL angefragt und damit rekursive Aktionen des PDM-Systems auf rekursive Anfragen des Datenbankmanagementsystems abgebildet werden können. Anschließend werden die Ergebnisse aus Kapitel 3 hinsichtlich der Darstellung und Auswertung von Zugriffsregeln in die rekursiven Anfragen integriert und somit die Grundlage für effiziente rekursive Navigation in Produktdaten gelegt.

Da zentrale Datenhaltung in *weltweit verteilten* Entwicklungsumgebungen zweifelsohne nur die Ausnahme ist, wird in Kapitel 5 auf verteilte Datenhaltung eingegangen. Hier stellt sich das Problem, dass die verteilte Ausführung rekursiver Datenbank-Anfragen nicht unmittelbar effizient möglich ist. Zur Lösung des Pro-

blems werden Produktstrukturen zunächst entsprechend ihrer Verteilung partitioniert und verschiedene Strategien für den Zusammenbau dieser Partitionen diskutiert. Alle bislang bekannten Verfahren sind hinsichtlich des erforderlichen Kommunikationsaufkommens jedoch nur als suboptimal einzustufen: Den Verfahren steht keinerlei „Wissen“ über die Verteilung der Daten zur Verfügung, so dass Mehrfachkontaktierungen einzelner Standorte in der Regel nicht ausgeschlossen werden können.

In Kapitel 6 wird deshalb ein Ansatz vorgestellt, mit welchem ein optimales Kommunikationsverhalten möglich ist. Zunächst wird der Object-Link-and-Location-Katalog, kurz OLL-Katalog, beschrieben, der zusätzliche Informationen über die Datenverteilung enthält. Nach der Darstellung der prinzipiellen Verwendung eines solchen Kataloges folgt die Beschreibung mehrerer Algorithmen, die das Erstellen und Ändern des OLL-Katalogs ermöglichen. Die Komplexitätsbetrachtungen der Algorithmen zeigen, dass für praktisch relevante Produktstrukturen mit akzeptablem Aufwand gerechnet werden kann. Anschließend wird eine mögliche Umsetzung des Ansatzes in relationalen Datenbankmanagementsystemen vorgestellt und die Integration in die Architektur vorgenommen. Ein Beweis über die Vollständigkeit des OLL-Katalogs beschließt dieses Kapitel.

In Kapitel 7 wird der neue Ansatz mittels Simulation hinsichtlich der Effizienz analysiert. Das Kapitel beginnt mit einer allgemeinen Einführung in die Thematik der Simulation und führt die ereignisorientierte Simulation als Grundlage für die hier betrachteten Analysen ein. Es folgt die Beschreibung eines Simulators für PDM-Systemarchitekturen, der im Rahmen dieser Arbeit entstanden ist. Anhand mehrerer Simulationsbeispiele wird die Effizienz des OLL-Katalogs in Kombination mit frühzeitiger Regelauswertung gegenüber den heute üblichen Verfahren nachgewiesen.

Teil III enthält die Zusammenfassung der Arbeit und vergleicht sie mit bereits bestehenden Ansätzen. In Kapitel 8 werden verschiedene Verfahren zu Zugriffssteuerung in anderen, ähnlichen Systemen beschrieben und mit dem in der Arbeit eingesetzten Ansatz verglichen. Des weiteren werden auch andere Verfahren zur Zugriffsoptimierung beleuchtet, beispielsweise Replikation oder Prefetching. Auch bekannte Verfahren zum Zusammenbau komplexer Objekte, wie sie in objektorientierten Ansätzen zu finden sind, werden diskutiert. Kapitel 9 fasst die wesentlichen Ergebnisse dieser Arbeit zusammen.

Im Teil IV (Anhang) findet sich der strukturelle Aufbau von Bedingungen, die im Zusammenhang mit den Zugriffsregeln Verwendung finden, sowie ein Übersetzer in äquivalente SQL-Konstrukte. Ebenso sind hier die Algorithmen zur inkrementellen Anpassung des OLL-Katalogs bei einer Änderung der Produktdaten enthalten.

Kapitel 2

Grundlagen und Problematik der PDM-Systeme

2.1 Die Produktstruktur

Die Produktstruktur ist die zentrale Datenstruktur der PDM-Systeme. In ihr fließen alle Informationen, die ein Produkt oder auch eine Produktfamilie beschreiben, zusammen.

2.1.1 Eigenschaften einer Produktstruktur

Produktstrukturen stellen Zusammenhänge zwischen den Bauteilen eines Produktes, oftmals als *Produktaufbruch* oder auch *Zusammenbauhierarchie* bezeichnet, sowie deren beschreibende Informationen dar. Ein Auszug aus einem stark vereinfachten Produktaufbruch einer Autotür ist in Abbildung 2.1 dargestellt, Abbildung 2.2 zeigt einige produktbeschreibende Informationen am Beispiel des Fensterhebers.

Die Tür eines Autos besteht neben dem Karosserie-Teil und den Tür-Verkleidungen auch aus einem Fenster-Modul. Dieses wiederum setzt sich zusammen aus der Scheibe selbst, dem Dichtungssatz sowie dem Fensterheber, der auch aus mehreren Unterteilen besteht.

Die 'Blattknoten' dieses Graphs, die nicht mehr weiter in Substrukturen zerlegt werden können, entsprechen den *Einzelteilen*, die übergeordneten Knoten sind *Zusammenbauteile* (oder auch *Zusammenbauten*), *Module*, *Systeme* und *Produkte*. Wurzelknoten einer Produktstruktur ist in der Regel ein Produkt, z. B. ein Auto, ein Elektrogerät, aber auch Möbel, Büroutensilien usw. sind denkbare Produkte,

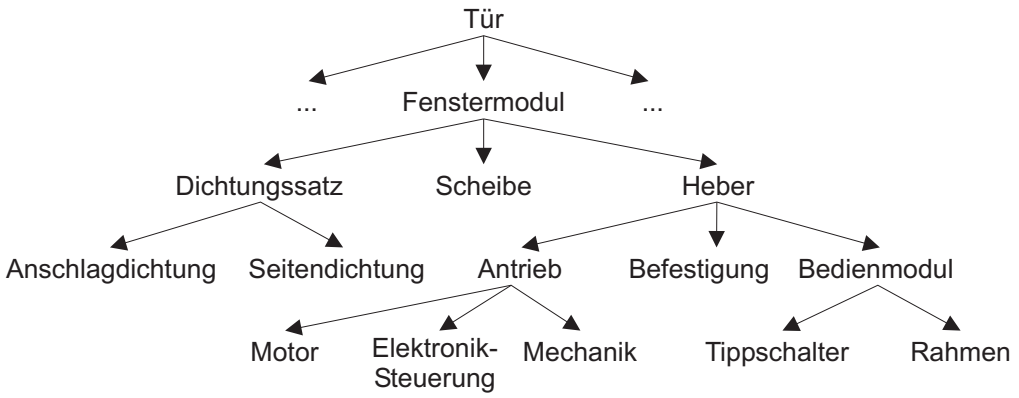


Abbildung 2.1: Zusammenbauhierarchie am Beispiel einer Autotür (stark vereinfacht)

die als Produktstruktur dargestellt werden können. Die Beziehung der Komponenten vom Produkt über die Systeme, Module und Zusammenbauten bis hin zu den Einzelteilen wird oftmals als *uses*-Beziehung bezeichnet, die umgekehrte Lesart heißt *Verwendungsnachweis*.

Einzelteile oder auch Zusammenbauten können in mehrere übergeordnete Zusammenbauten eingehen. Man spricht hierbei von *Mehrfachverwendung*. Produktstrukturen sind folglich nicht zwangsläufig Bäume, sondern allgemeiner azyklische gerichtete Graphen.

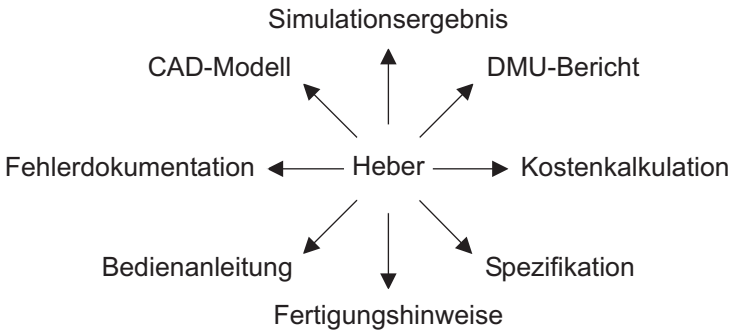


Abbildung 2.2: Produktbeschreibende Daten des Fensterhebers (Auszug)

Produktbeschreibende Informationen können sowohl mit Einzelteilen als auch mit Zusammenbauten etc. verknüpft werden. Das PDM-System speichert dabei nur Meta-Information wie Dateiname, Besitzer, Datum der letzten Änderung usw. über die Nutzdaten, die selbst im Dateisystem abgelegt werden. Abbildung 2.3 zeigt einige typische Daten, die in dieser Form verwendet werden [EWP+99].

- Ablauf- und Flussdiagramme
- Analyse- und Berechnungsmodelle
- Anforderungen und Spezifikationen
- Arbeitspläne
- Archivdaten
- Baubarkeitsbedingungen
- Berechnungsergebnisse
- Betriebsanleitungen
- CAD-Modelle
- Design- und Stylingdaten
- Einzelteil- und Zusammenbauzeichnungen (3D-Modelle)
- Ergebnisse von Simulationen
- Freigabedaten
- Kinematikdaten
- Klassifizierungen
- Konfigurationsinformationen
- Kosteninformationen
- Marktanalysen
- Montageanleitungen
- Multimediale Dokumente
- NC-Programme
- Produktkataloge
- Produktspezifikationen
- Prozessmodelle
- Prozesspläne
- Qualitätsdaten
- Rasterdaten (TIFF)
- Schnittstellendaten (STEP, IGES)
- Skizzen
- Struktur-Informationen
- Stücklisten
- Technische Zeichnungen
- Testergebnisse
- Textdokumente
- Verkaufszahlen
- Werbebroschüren
- Werkzeugkonstruktionen
- Zuliefererinformationen

Abbildung 2.3: Produktdaten

Aus dem Beispiel der Autotür wird bereits deutlich, dass Produktstrukturen prinzipiell beliebig tief werden können. Besonders bei komplexeren Produkten ist es nicht möglich, vor Beginn der Definition der Produktstruktur eine exakte Tiefe vorauszusagen. Vielmehr ergibt sich diese im Laufe der Verfeinerungen während der frühen Phasen im Entwicklungsprozess.

Nach der Definition des Produktaufbruches werden die Komponenten konstruiert. Besonders umfangreiche Änderungen werden dabei nicht auf bestehenden Daten ausgeführt, vielmehr wird hierfür eine neue *Version* des zu ändernden Teils

erzeugt. Auch Weiterentwicklungen älterer Komponenten stellen neue Versionen dar. Eine adäquate Darstellung verschiedener Versionen wird durch das *Master-Version-Konzept* erreicht: Die (unveränderlichen) Stammdaten eines Teils werden im Master abgelegt, die änderbaren Daten bilden die Version. Stammdatensätze binden ihre zugehörigen Versionen mit der so genannten *has_revision*-Beziehung an sich.

Versionen werden typischerweise mit *Gültigkeiten* (auch als *Effectivities* bezeichnet) versehen. Diese Information legt fest, in welchem Zeitraum (Datum- oder auch Losnummern-basiert) eine Version verbaut wurde. So kann beispielsweise der Tippschalter für den elektrischen Fensterheber einer Überarbeitung unterzogen werden, wobei eine neue Version erzeugt wird. Mit der Freigabe dieser neuen Version wird der alte Schalter „ungültig gesteuert“, d. h. der Gültigkeitszeitraum endet, und ab sofort wird nur noch die neue Version in Neuwagen eingebaut.

Gültigkeitssteuerung findet jedoch nicht nur zwischen Mastern und Versionen statt, sondern auch die *uses*-Beziehung kann mit Gültigkeitsinformationen versehen werden. Wird beispielsweise der Zulieferer für den Motor des Fensterhebers gewechselt, so wird die Gültigkeit der Beziehung vom ’Antrieb’ zum alten Motor beendet, der neue Motor in die Produktstruktur aufgenommen und aktuell auf gültig gesetzt. Abbildung 2.4 zeigt ein Beispiel für Gültigkeitssteuerung.[1]

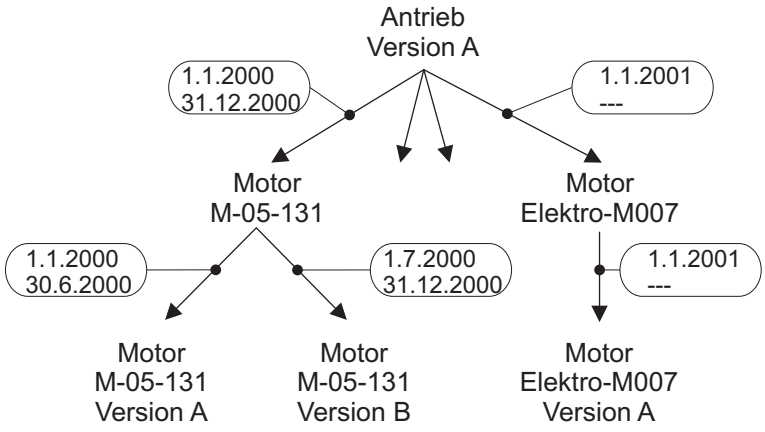


Abbildung 2.4: Effectivities zur Steuerung von Struktur- und Versionsgültigkeiten in Verbindung mit dem Master-Version-Konzept

Der Motor ’M-05-131’ wurde vom 01.01.2000 bis zum 31.12.2000 in verschiedenen Versionen in den Antrieb eingebaut. Seit dem 01.01.2001 wird als Ersatz

[1] Der Fokus dieser Arbeit liegt nicht auf optimaler Darstellung von Versionen. Hinweise auf verwandte Arbeiten zu Versionierung und Historie in Datenbanken finden sich im Abschnitt 8.6.

der Motor 'Elektro-M007' verwendet. Da dieses Bauteil bis auf „Widerruf“ in die Produktstruktur eingehen soll, wurde vorerst kein End-Zeitpunkt im Gültigkeitsobjekt eingetragen.

Besonders komplexere Produkte, beispielsweise PKWs, können vom Kunden in gewissen Grenzen konfiguriert werden. Daraus können möglicherweise mehrere tausend verschiedene Produkte resultieren, die sich jedoch nur in wenigen Details unterscheiden. Es wäre nun nicht sinnvoll, jede mögliche Konfiguration in einer separaten Produktstruktur zu speichern. Stattdessen werden alle möglichen Konfigurationen zu einer einzigen Produktstruktur zusammengefasst, wobei die optionalen Anteile mit *Strukturoptionen* versehen werden.

Soll beispielsweise die Möglichkeit bestehen, einen PKW mit Sonnenschutzverglasung auszustatten, so muss in Abbildung 2.1 eine entsprechende Scheibe hinzugefügt und die zugehörige *uses*-Beziehung mit einer Strukturoption (z. B. „SSV“ als Abkürzung für 'Sonnenschutzverglasung') versehen werden. Um anzuzeigen, dass die standardmäßig eingebaute Scheibe *nicht* Bestandteil eines PKWs mit Sonnenschutzverglasung ist, muss die zugehörige *uses*-Beziehung mit der Option „¬SSV“ markiert werden (vgl. Abbildung 2.5).

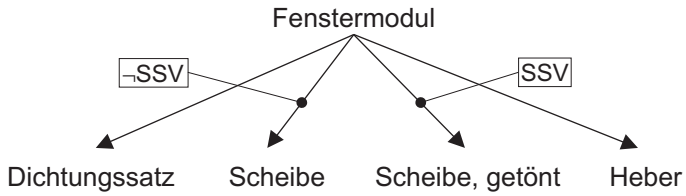


Abbildung 2.5: Produktstruktur (Ausschnitt) mit Strukturoption

Zur Struktur eines konfigurierten Produktes gehören alle Objekte, deren *uses*-Beziehungsobjekt entweder *keine* Strukturoption zugeordnet ist (im Beispiel der Dichtungssatz und der Heber), oder aber es ist mindestens eine Strukturoption zugeordnet, die auch zur Konfiguration des Produktes gehört.

2.1.2 Operationen auf der Produktstruktur

Auf den Objekten einer Produktstruktur werden eine Reihe von Operationen ausgeführt. Wir wollen hier einige davon vorstellen und anschließend hinsichtlich ihrer Ausführungshäufigkeit einordnen.

Selbstverständlich müssen alle Arten von Objekten, d. h. struktur-relevante Objekte wie Einzelteile, Zusammenbauten etc. und auch die produktbeschreibenden

Daten, erzeugt werden können. Löschvorgänge sind typischerweise auf Grund von Dokumentations- und Gewährleistungspflichten auf wenige produktbeschreibende Daten (z. B. Skizzen) eingeschränkt. Auch Änderungen können nur begrenzt (z. B. auf Bezeichnungen) durchgeführt werden, größere Änderungen, die Form, Funktion oder Eignung beeinflussen, erzeugen eine neue Version des ursprünglichen Objekts. Dabei müssen dann auch Gültigkeiten für die alte und neue Version festgelegt werden.

Zwischen den Objekten sind auch deren Beziehungen zueinander anzulegen. Struktur-relevante Objekte werden mit der *uses*- und *has_revision*-Beziehung verknüpft, die beschreibenden Daten werden z. B. mit den Relationen *attaches*, *is_specified_by*, *is_described_by* usw. den struktur-relevanten Objekten zugeordnet.

Möchte ein Benutzer auf den Objekten arbeiten, so muss er sie mittels *CheckOut* gegen Änderungszugriffe anderer Anwender sperren. Nach erfolgter Bearbeitung werden die Objektsperren mit *CheckIn* wieder aufgehoben.

Die Query-Operation (nicht zu verwechseln mit einer Datenbank-Query!) dient der *mengenorientierten Suche* von Objekten aller Arten. Queries werden immer dann abgesetzt, wenn der anfragende Benutzer zwar Eigenschaften des gesuchten Objektes kennt, nicht aber den strukturellen Bezug in der Produktstruktur (oder diesen Bezug auch nicht benötigt).

Strukturorientierte Anfragen werden auch als *Expansions-* oder *Navigations-Vorgänge* bezeichnet. Wir unterscheiden zwei Arten von Expansionen:

- Single-Level-Expand
- Multi-Level-Expand

Der *Single-Level-Expand* traversiert *einstufig* eine Beziehung zwischen zwei Objekten, z. B. vom Einzelteil hin zu dessen Spezifikation, oder vom Zusammenbau hin zu den unmittelbar eingebauten Unterteilen. Single-Level-Expands sind also prinzipiell nicht auf die Navigation über die *uses*-Beziehung beschränkt, in der vorliegenden Arbeit wird jedoch der Fokus auf die Traversierung speziell dieser Beziehung gerichtet. In Abbildung 2.1 ermittelt der Single-Level-Expand angewendet auf das Objekt mit der Bezeichnung 'Fenstermodul' dessen Unterteile 'Dichtungssatz', 'Scheibe' und 'Heber'. Wendet man auf diese Objekte wiederum den Single-Level-Expand an und auf dessen Ergebnisse wieder usw., so kann die Produktstruktur quasi 'Level-by-Level' angefragt werden.

Einfacher lässt sich der komplette Aufbau der Struktur mit dem *Multi-Level-Expand* durchführen. Diese Operation übernimmt die rekursive Ausführung des

Single-Level-Expands und ermittelt sämtliche Unterteile des zu expandierenden Objektes. Besonders für *Digital-Mockup*-Operationen ist dieser automatisierte mehrstufige Expand wichtig: Beim Digital Mockup lassen sich mit IT-Mitteln auf der Basis von CAD-Modellen Tests durchführen, ob komplexere Baugruppen zusammenpassen oder in einem oder mehreren Teilen Konstruktionsfehler bezüglich Passgenauigkeit vorliegen. Um die für eine derartige Prüfung benötigten CAD-Modelle zu ermitteln, müssen zunächst sämtliche Unterbauteile des zu testenden Objektes gefunden werden – exakt die Aufgabe des Multi-Level-Expands!

Die Navigation vom Unterteil zum übergeordneten Zusammenbau (vgl. Verwendungsnachweis im Abschnitt 2.1.1) findet typischerweise nur einstufig statt und entspricht somit quasi dem umgekehrten Single-Level-Expand.

Nun ist eine Einordnung der beschriebenen Operationen hinsichtlich ihrer 'Wichtigkeit' interessant, d. h. welche Operationen für Performance-Optimierungen betrachtet werden sollen. Da sich die 'Wichtigkeit' schlecht beurteilen lässt, bewerten wir stattdessen die Häufigkeit, mit welcher die Operationen von den Anwendern benutzt werden.

Da die PDM-Systeme heute noch in den Kinderschuhen stecken und Masseneinsätze noch selten sind, liegen noch sehr wenige Erfahrungswerte hinsichtlich der Benutzerprofile vor. Aus den Beobachtungen diverser Einsatzszenarien und aus Analysen von realistischen Entwicklungsprozessen lässt sich dennoch eine grobe Einteilung nach Tabelle 2.1 vornehmen.

Operation	selten	häufig	sehr häufig
Create Object:	✓		
Create Relation:	✓		
Create Version:	✓		
Query:		✓	
Single-Level-Expand:			✓
Multi-Level-Expand:		✓	
Digital MockUp:		✓	
Verwendungsnachweis:		✓	

Tabelle 2.1: Häufigkeit der Operationen

Aus dieser Einteilung geht klar hervor, dass die mengen- und strukturorientierten Anfragen die häufigsten Operationen sind. Wie in Kapitel 2.3 gezeigt wird, sind auch diese Operationen besonders performance-kritisch und erhalten bei den Optimierungsaspekten dieser Arbeit die höchste Priorität.

2.1.3 Zugriffssteuerung auf Produktstrukturen

In weltweit agierenden Unternehmen des produzierenden Gewerbes können mehrere tausend Benutzer auf ein PDM-System zugreifen. Für die Bearbeitung der unterschiedlichen Aufgaben der Anwender wird typischerweise jedoch jeweils nur ein Ausschnitt aus dem Gesamtangebot an Produktdaten benötigt. Zumeist ist es auch nicht wünschenswert, dass jeder Benutzer Zugriff auf sämtliche gespeicherten Daten hat. Besonders in verteilten Umgebungen, in welchen externe Partner und Zulieferer integriert sind, müssen Daten vor unbefugtem Zugriff geschützt werden. Zulieferer sollen in der Regel nur sehr eingeschränkt auf Daten des Auftraggebers zugreifen dürfen, der Auftraggeber selbst möchte aber zumindest lesenden Zugriff auf alle Daten des Zulieferers bekommen. Um die vielen Facetten der Zugriffssteuerung zu berücksichtigen, wird ein feingranular abgestuftes Zugriffsberechtigungssystem benötigt.

Zugriffsrechte werden in PDM-Systemen typischerweise regelbasiert beschrieben und ausgewertet. Eine derartige Regel besagt beispielsweise, dass ein Benutzer namens 'Michael Schmidt' alle Zusammenbauten sehen darf. Eine andere Regel etwa erlaubt allen Benutzern, Query-Aktionen auf CAD-Modellen auszuführen usw. Diese Regeln können auch in Abhängigkeit von Bedingungen Zugriffe erlauben: Der Benutzer 'Schmidt' kann damit zum Beispiel nur auf Einzelteile zugreifen, die ihm selbst gehören, d. h. das Attribut 'Owner' muss den Wert 'Schmidt' enthalten (bzw. die ID des Benutzers 'Schmidt'). Auf diese Art lassen sich sehr detailliert Zugriffsberechtigungen auf den unterschiedlichsten Objekten definieren.

Eine andere Art der Zugriffssteuerung erfolgt über die Gültigkeiten, die den Beziehungen zwischen Zusammenbauten und Unterteilen bzw. zwischen den Mastern und ihren Versionen zugeordnet sind. Bei der Expansion einer Produktstruktur kann der anfragende Anwender einen Zeitpunkt definieren, so dass die expandierte Struktur den zu diesem Zeitpunkt gültigen Zustand wiedergibt. Ohne Angabe eines solchen Zeitpunkts wird in der Regel davon ausgegangen, dass der Anwender den momentan gültigen Zustand expandieren möchte. Man spricht in diesem Zusammenhang auch von der *last-revision-only*-Annahme.

In Abschnitt 2.1.1 wurde bereits auf die Konfigurierbarkeit besonders von komplexen Produkten eingegangen. Die Auswahl von Strukturoptionen (z. B. Sonderausstattungspakete etc.) ermöglicht ebenfalls eine gewisse Steuerung der Zugriffe bei Expansionsvorgängen. Unterteile gehen dabei nur in die Ergebnis-Struktur ein, falls der Benutzer mindestens eine zugeordnete Strukturoption auch in seine Auswahl integriert hat.

Die Frage, ob zwei Strukturoptionen miteinander kombinierbar sind, wird durch sogenannte *Baubarkeits-Bedingungen* überprüft. Derartige Bedingungen verhin-

dern beispielsweise, dass ein Cabrio mit einem Schiebedach ausgestattet wird, oder eine Limousine mit einer Dachreling kombiniert wird. In heutigen PDM-Systemen ist die Überprüfung von Baubarkeits-Bedingungen *keine* Standardfunktionalität und muss daher oftmals vom Betreiber selbst bereitgestellt werden.

2.2 Architektur heutiger PDM-Systeme

PDM-Systeme stellen Client-Server-Anwendungen dar. Die Clients dienen zumeist hauptsächlich der Visualisierung der Daten, der Server bietet nahezu die komplette PDM-Funktionalität an. In der Regel übernimmt ein relationales Datenbankmanagementsystem die Speicherung der Daten.

Ein Überblick über die wichtigsten Komponenten hinsichtlich der Anfrage-Generierung und -Ausführung ist in Abbildung 2.6 dargestellt.

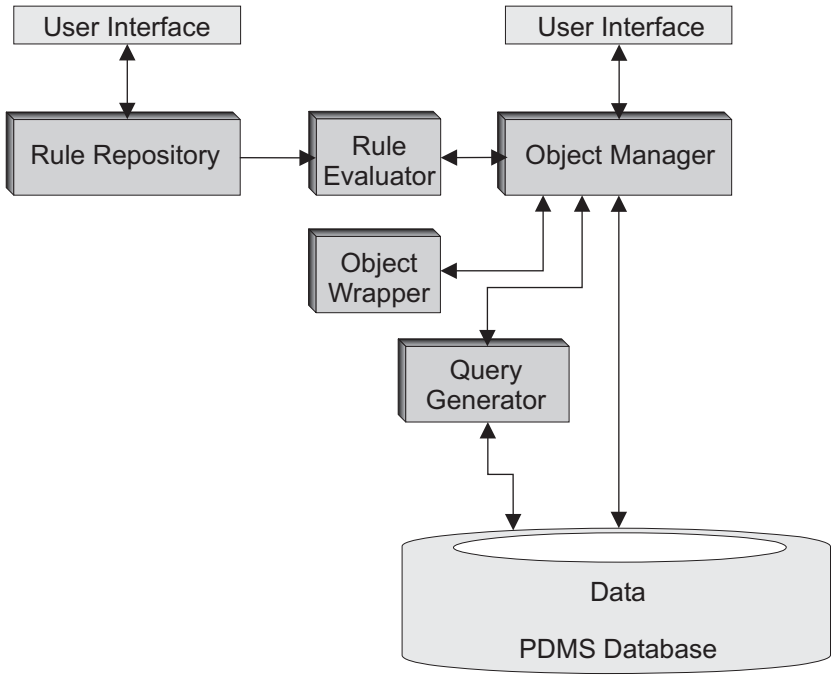


Abbildung 2.6: Architektur heutiger PDM-Systeme

Auf der linken Seite der Abbildung ist das sogenannte 'Rule Repository' zu sehen, in welchem die Zugriffsregeln abgelegt sind. Über die zugehörige Benutzerschnittstelle (User Interface) können diese Regeln (typischerweise durch einen Administrator) verwaltet werden.

Endanwender arbeiten mit der Benutzerschnittstelle des Object Managers. Über dieses Interface können unter anderem die Aktionen, die in Abschnitt 2.1.2 beschrieben sind, angestoßen werden. Um das Zusammenspiel der einzelnen Komponenten zu verstehen, betrachten wir nun eine Aktion, die eine Datenbankanfrage auslöst (z. B. die Query-Aktion):

Über die Benutzerschnittstelle erhält der Object-Manager den Auftrag, die vom Anwender gewählte Aktion auszuführen. Um die Daten aus der Datenbank zu selektieren, wird zunächst der Query-Generator angestoßen, der eine Anfrage entsprechend den angegebenen Selektionskriterien des Benutzers erstellt. Bei Suchanfragen beispielsweise kann eine SQL-WHERE-Klausel nach den Suchkriterien des Anwenders erstellt bzw. erweitert werden. Anschließend wird diese Anfrage an die Datenbank übermittelt. Das Ergebnis der Anfrage – ein oder mehrere Datenbank-Tupel – wird an den Object-Manager zurückgegeben.

Über den sogenannten *Object Wrapper* werden nun diese Tupel in PDM-System-interne Objekte abgebildet. Dabei können Attribute umbenannt, Werte transformiert oder auch berechnet werden. Auf den umgewandelten Objekten wird schließlich der *Rule Evaluator* gestartet, welcher die Zugriffsrechte des Benutzers im Kontext der Aktion unter Zuhilfenahme des Rule Repositories überprüft. Objekte, welche den Test bestehen, werden an die Benutzerschnittstelle zurückgegeben und dort visualisiert, alle anderen Objekte werden verworfen.

Für besondere Aufgaben besitzt der Object-Manager auch direkt die Möglichkeit, mit der Datenbank zu kommunizieren. Dies ist z. B. für nicht-dynamische Änderungsoperationen sinnvoll, die „hart codiert“ werden können.

In den folgenden Betrachtungen wird davon ausgegangen, dass – wie heute üblich – ein relationales Datenbankmanagementsystem zur Speicherung der Produkt-Metainformationen Verwendung findet. Im Abschnitt 2.4.2 wird gezeigt, dass dies jedoch nicht der Grund für die Performance-Probleme ist.

2.3 Performance-Probleme heutiger PDMS

Wie bereits motiviert wurde, leiden PDM-Systeme heutiger Bauart oftmals unter erheblichen Performance-Problemen. Die Benutzer sind zum Teil nicht mehr in der Lage, ihre Arbeit, die durch das PDM-System hätte besser unterstützt werden sollen, zu erledigen, da wichtige Funktionalitäten zu lange Antwortzeiten erfordern.

Die Probleme entstehen typischerweise erst beim Betrieb eines PDM-Systems in weltweit verteilten Umgebungen. Aktionen, die im lokalen Kontext noch Ant-

wortzeiten im Bereich einiger Sekunden erfordern, dauern im „internationalen“ Kontext durchaus mehrere Minuten!

Wir wollen in diesem Abschnitt die Ursachen der Performance-Probleme analysieren und quantitativ erfassen. Auch wird diskutiert, warum relativ einfach anzuwendende Optimierungen nicht greifen werden und deshalb andere Mechanismen entwickelt und angewendet werden müssen.

2.3.1 Ursachen der Performance-Probleme

Aus der Architektur heutiger PDM-Systeme (vgl. Abschnitt 2.2) lässt sich schnell erkennen, dass eine mögliche Ursache der Performance-Probleme im Zeitpunkt der Regelauswertung liegt: Daten werden angefragt, übertragen und erst dann auf Zugreifbarkeit getestet! Der Test findet also zum spätest möglichen Zeitpunkt vor der Visualisierung statt! Startet ein Benutzer eine sehr unspezifizierte Anfrage, die eine große Anzahl an Ergebnis-Tupeln liefert, so muss – je nach Zugriffsrechten des Anwenders – möglicherweise ein großer Prozentsatz dieses Ergebnisses nach der Übertragung quasi „weggeworfen“ werden. Abgesehen von dem Aufwand, den der Object Wrapper zur Umwandlung der Tupel in PDM-Objekte generiert, führt diese Vorgehensweise in Weitverkehrsnetzen, welche typischerweise nicht allzu große Bandbreiten aufweisen, zu verlängerten Übertragungs- und Antwortzeiten.

Es stellt sich die Frage, ob die Zugriffsregeln nicht schon früher, d. h. *vor* der Übertragung über das WAN, im Idealfall also schon durch das Datenbanksystem selbst, getestet werden können. Dadurch wäre gewährleistet, dass nur Daten übertragen werden, auf die der anfragende Benutzer tatsächlich auch Zugriffsrechte besitzt. Das heute praktizierte *data shipping* würde dabei durch *function shipping* ersetzt. Kapitel 3 beschäftigt sich mit den möglichen Alternativen hierzu.

Aus der Analyse von Kommunikations- und Interaktionsprotokollen, die in Testumgebungen mit existierenden PDM-Systemen erzeugt wurden, geht hervor, dass ein weiterer Grund für die mangelhafte Performance in der Art und Weise der Datenbanksystem-Verwendung liegt. Dies wird besonders deutlich bei Operationen, die durch die Produktstruktur navigieren (vgl. Abschnitt 2.1.2):

Ein Benutzer beginnt die Navigation in der Struktur (Single-Level-Expand) an dem gewünschten Zusammenbau. Die Expansion dieses Knotens bewirkt das „Aufklappen“ der Struktur um die Knoten aus dem nächsttiefer liegenden Level. Dazu wird eine Serie von SQL-Anfragen an die Datenbank abgesetzt: Zunächst werden alle Objekte der *uses*-Beziehung angefragt, die von dem betrachteten Zusammenbau ausgehen. Anschließend werden alle Master-Objekte angefragt, auf

welche diese uses-Objekte zeigen. Von diesen Master-Objekten werden die Objekte der *has_revision*-Beziehung ermittelt und dann erst können die relevanten Versionen aus der Datenbank angefragt werden. Je nach Implementierung können für die Berücksichtigung der gewählten Strukturoptionen noch zusätzliche Anfragen benötigt werden.

Das Datenbanksystem fungiert hier mehr oder weniger als Dateisystem ohne „Intelligenz“. Selbst die Basis-Fähigkeiten heutiger Datenbankmanagementsysteme, z. B. Join-Operationen, werden dabei praktisch kaum verwendet, neuere Ansätze wie Rekursion in SQL-Anfragen sucht man vergeblich.

Führt man das Beispiel des Single-Level-Expands weiter und betrachtet den Multi-Level-Expand, so wird das gesamte Ausmaß dieser äußerst primitiven Datenbank-Verwendung sichtbar: Die Expansion über alle Hierarchie-Stufen der Produktstruktur wird typischerweise auf das wiederholte einstufige Aufklappen zurückgeführt. Jeder Ergebnisknoten eines derartigen Single-Level-Expands wird dabei selbst wieder zum Ausgangspunkt eines erneuten Single-Level-Expands, bis die Struktur sämtlicher Zusammenbauten expandiert ist.

In jedem Schritt werden dabei die bereits beschriebenen SQL-Anfragen erzeugt und abgesetzt. Eine enorme Anzahl von einzelnen, isolierten und von einander unabhängigen Anfragen ist die Folge. In produktiven Umgebungen werden für die Bearbeitung eines Multi-Level-Expands somit rasch einige hundert bis tausend Anfragen benötigt!

Lokale Umgebungen mögen diesen Aufwand noch verkraften und zu dennoch akzeptablen Antwortzeiten führen. In Installationen jedoch, die eine räumliche Trennung von Datenbanksystem und PDM-System erfordern, und deren Kommunikation auf der Verwendung von Weitverkehrsnetzen basiert, werden auf Grund der langen Latenzzeiten Antwortzeiten erreicht, die von den Benutzern nicht mehr toleriert werden können.

Es stellt sich nun die Frage, ob nicht durch geschicktes Zusammenfassen von aufeinander folgenden Anfragen Zeit eingespart werden könnte. In Kapitel 4 werden wir darauf eingehen und auf der Basis von *rekursivem SQL*, einem relativ neuen Feature von Datenbanksystemen, eine Lösung präsentieren.

Die in diesem Abschnitt vorgestellten Vorgehensweisen der späten Regelauswertung und Navigation über das Netz bei rekursiven Aktionen stammen aus der Zeit, als noch nicht abzusehen war, ob in naher Zukunft objektorientierte anstelle der relationalen Datenbankmanagementsysteme zum Einsatz kommen werden. Ziel der PDM-System-Designer war ganz offensichtlich, eine möglichst vollkommene Trennung von PDM- und Datenbanksystem zu erzielen, so dass ein relativ problemloser Austausch der Datenbank-Komponente möglich sein sollte.

Inzwischen hat sich der Trend jedoch bestätigt, nur noch auf den relationalen (oder objekt-relationalen) Datenbanksystemen aufzusetzen, so dass kein Grund mehr besteht, nicht die komplette Objektbearbeitung, d. h. gerade auch die Datenbankzugriffe und damit die Art und Weise der Datenbankverwendung, in die Performance-Optimierung mit einzubeziehen.

2.3.2 Quantitative Analyse der Probleme

2.3.2.1 Mathematisches Modell

Um quantitative Aussagen über die in Abschnitt 2.3.1 identifizierten Probleme machen zu können, bilden wir die Kommunikationsschicht der PDM-Systemumgebung auf ein mathematisches Modell ab. Mit diesem Modell lassen sich Antwortzeiten aus der Sicht der Datenbankzugriffe in verschiedenen Szenarien errechnen (im Sinne von „vorhersagen“) und miteinander vergleichen. Tabelle 2.2 listet einige informelle Definitionen auf, die wir im Folgenden dazu verwenden.

Symbol	Beschreibung
dtr	Datentransferrate im WAN
T_{Lat}	Latenzzeit im WAN
$size_p$	Paketgröße im WAN
$\varnothing size_n$	durchschnittliche Größe eines Objektes in der Produktstruktur
$n_t(t)$	Anzahl übertragener Objekte eines Teilgraphs t
τ	Tiefe der Struktur (beginnend mit 0)
σ	Prozentsatz sichtbarer Master-Objekte (pro Level)
ν	Verzweigungsgrad der Zusammenbau-Versionen
μ	Verzweigungsgrad der Zusammenbau- und Einzelteil-Master
q	Anzahl benötigter Datenbank-Anfragen
c	Anzahl benötigter WAN-Kommunikationen
vol	aus der Benutzeraktion resultierendes Datenvolumen
T	Antwortzeit

Tabelle 2.2: Definitionen für die Berechnung von Antwortzeiten

Wir betrachten für die Modellbildung Produktstruktur-Graphen, die nach dem Master-Version-Ansatz organisiert sind. Jeder Master habe μ Versionen, jede Version eines Zusammenbaus bestehe aus weiteren ν Unterteilen. Die Struktur bildet einen vollständigen Baum, d. h. alle Einzelteilversionen besitzen die gleiche Tiefe τ. Die Navigationen (Single-Level- und Multi-Level-Expands) sind jeweils nur an der aktuell gültigen Version interessiert (*last revision only*), Query-Aktionen

sollen alle aktuell gültigen Versionen in dem Baum liefern. Abbildung 2.7 zeigt einen Ausschnitt aus einer derartigen Produktstruktur mit $\tau = 1$.

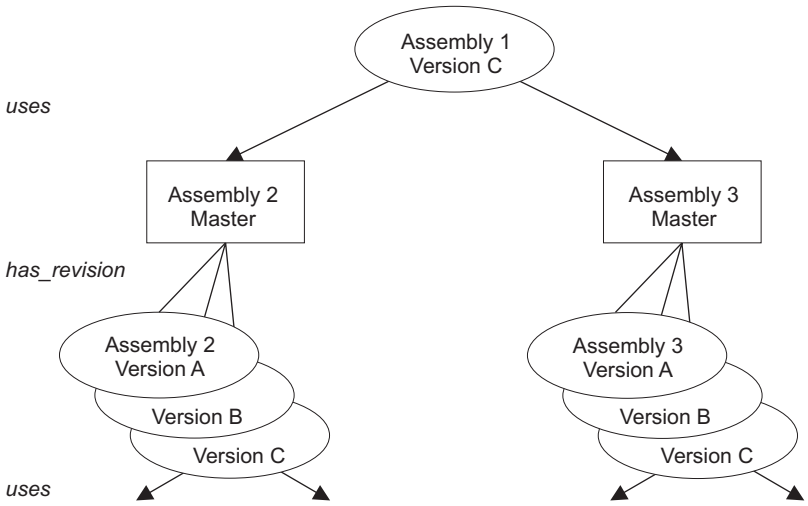


Abbildung 2.7: Prinzip der Produktstruktur für die Modellbildung

Für einen Single-Level-Expand beispielsweise des Knotens 'Assembly 1 Version C' gilt damit: Die ν Objekte der *uses*-Beziehung werden mit *einer* Anfrage ermittelt. Die jeweils zugehörigen Master-Objekte werden (unabhängig vom Verzweigungsgrad ν) mit *einer* weiteren Anfrage ermittelt. Auf Grund der Zugriffsregeln sind jedoch nur $\sigma * \nu$ viele Master für den Anfrager zugreifbar. Pro sichtbarem Master wird nun eine Anfrage gestellt nach den μ Objekten der *has_revision*-Beziehung. Da jeweils nur *eine* Version weiterverfolgt werden soll, wird jeweils die letzte gültige Version in einer separaten Anfrage ermittelt. Insgesamt erhalten wir für einen Single-Level-Expand also $2 + 2 * (\nu\sigma)$ viele Anfragen.

Im Kontext eines Multi-Level-Expands werden die bisher gefundenen Versionen weiter expandiert. Auf Grund des Verzweigungsgrades ν sowie der Selektivität σ werden im Level $l+1$ folglich $\nu\sigma$ so viele Anfragen benötigt wie im Level l. Durch Summieren der Anfragen über alle Level erhalten wir für die Anzahl der Datenbank-Anfragen q_s des Multi-Level-Expands Gleichung 2.1 (der Index s steht für den „simplen“ navigierenden Zugriff):

$$\begin{aligned} q_s &= (2 + 2\nu\sigma) + \nu\sigma(2 + 2\nu\sigma) + (\nu\sigma)^2(2 + 2\nu\sigma) + \ldots \\ &\quad +(\nu\sigma)^{\tau-1}(2 + 2\nu\sigma) \\ &= (2 + 2\nu\sigma)\sum_{i=0}^{\tau-1}(\nu\sigma)^i \end{aligned} \tag{2.1}$$

Die Anzahl der Anfragen für Single-Level-Expands ergibt sich aus Gleichung 2.1 mit $\tau = 1$, für Query-Aktionen gilt $q_s = 1$.

Da jede Anfrage auch ein Ergebnis produziert, werden doppelt so viele Kommunikationen c_s wie Anfragen benötigt, d. h.

$$c_s = 2 * q_s \tag{2.2}$$

Das zu übertragende Datenvolumen vol_s setzt sich zusammen aus den übermittelten Anfrage-Daten sowie den zugehörigen Antworten. Wir gehen dabei davon aus, dass Anfragen in einem einzigen Paket übertragen werden können. Dann lässt sich das resultierende Datenvolumen abschätzen mit

$$vol_s = q_s * size_p \; + \; n_t(t) * \varnothing size_n \tag{2.3}$$

Durch Kombination der Gleichungen 2.2 und 2.3 erhalten wir die gesamte Antwortzeit T_s:

$$T_s = c_s * T_{Lat} + vol_s / dtr \tag{2.4}$$

Zur Berechnung der Antwortzeiten fehlt nun lediglich noch die Definition des Ausdrucks $n_t(t)$, d. h. die Anzahl der übertragenen Objekte. Dazu betrachten wir wieder zunächst den Single-Level-Expand auf der Produktstruktur in Abbildung 2.7. Die Anfrage an die *uses*-Beziehung liefert ν Objekte, ebenso die Anfrage an die Master-Tabellen. Auch hier kommt der Parameter σ zum Tragen: Von den ν Master-Objekten sind nur $\sigma * \nu$ für den Anfrager zugreifbar. Folglich werden bei der Anfrage an die *has_revision*-Beziehung auch nur $(\sigma * \nu) * \mu$ Objekte ermittelt.[2] Da zu jedem Master nur jeweils *eine* Version betrachtet werden soll, sind im nächsten Schritt entsprechend $\sigma * \nu$ Versionsobjekte zu übertragen. Pro Single-Level-Expand werden demnach $2 * \nu + (\nu\sigma)\mu + \nu\sigma = \left(\frac{2}{\sigma} + \mu + 1\right) * \nu\sigma$ Objekte übertragen. Für den Multi-Level-Expand wiederholen sich diese Überlegungen pro ermittelter Version, die Anzahl der Objekte erhöht sich damit pro Level auf das $\nu\sigma$-fache des vorangehenden Levels.

Damit ergibt sich für $n_t(t)$ die Gleichung 2.5:

$$n_t(t) = \begin{cases} (2 + 2\mu) \sum_{i=1}^{\tau} \nu^i \mu^{i-1} & \text{für Query-Aktionen} \\ \left(\frac{2}{\sigma} + \mu + 1\right) * \sum_{i=1}^{\tau} (\nu\sigma)^i & \text{für Expansionen} \end{cases} \tag{2.5}$$

Für Single-Level-Expands ergibt sich die Anzahl der übertragenen Objekte wieder mit $\tau = 1$.

Beispiele für Berechnungen mit diesem Modell finden sich in Abschnitt 2.3.2.2.

[2] Hier ist noch nicht entschieden, welche „Kante“ zu der letzten gültigen Version führt. Deshalb werden alle μ Kanten ausgehend von den $\nu\sigma$ vielen zugreifbaren Mastern übertragen!

2.3.2.2 Beispielberechnungen

In Tabelle 2.3 sind beispielhaft einige Ergebnisse nach dem in Abschnitt 2.3.2.1 eingeführten mathematischen Modell aufgelistet. Grundlage der Berechnungen sind jeweils *vollständige Bäume* mit den jeweils angegebenen Parametern. In der Spalte „Query“ finden sich die Ergebnisse für mengenorientierte Anfragen an alle Knoten des Baumes. „SLE“ steht für Single-Level-Expand, „MLE“ bezeichnet den Multi-Level-Expand, der wie die Query-Aktion auf dem kompletten Baum ausgeführt wird.

$size_p = 4$kB	$\tau = 3, \nu = 9, \sigma = 2/3$			$\tau = 7, \nu = 3, \sigma = 2/3$			$\tau = 6, \nu = 5, \sigma = 3/5$		
$\varnothing\, size_n = 0.5$kB	Query	SLE	MLE	Query	SLE	MLE	Query	SLE	MLE
T_{Lat} = 0.6	1.2	16.8	722.4	1.2	7.2	914.4	1.2	9.6	3494.4
dtr = 512	25.7	1.1	47.7	102.5	0.5	57.6	610.4	0.6	227.5
$T_s = \Sigma$	26.9	17.9	770.1	103.7	7.7	972.0	611.6	10.2	3721.9
T_{Lat} = 0.3	0.6	8.4	361.2	0.6	3.6	457.2	0.6	4.8	1747.2
dtr = 1024	12.8	0.6	23.9	51.3	0.2	28.8	305.2	0.3	113.8
$T_s = \Sigma$	13.4	9.0	385.1	51.9	3.8	486.0	305.8	5.1	1861.0
T_{Lat} = 0.15	0.3	4.2	180.6	0.3	1.8	228.6	0.3	2.4	873.6
dtr = 10240	1.3	0.1	2.4	5.1	<0.1	2.9	30.5	<0.1	11.4
$T_s = \Sigma$	1.6	4.3	183.0	5.4	1.8	231.5	30.8	2.4	885.0

Datentransferrate *dtr* in kBits/Sekunde, Latenzzeit T_{lat} und Antwortzeit T_s in Sekunden; $\mu = 1$;
Antwortzeiten sind in Latenzzeit und Transferzeit aufgeteilt

Tabelle 2.3: Antwortzeiten mehrerer Szenarien in heutigen Umgebungen

2.4 Inadäquate Lösungsansätze

Auf den ersten Blick mag der Eindruck entstehen, dass die im Abschnit 2.3.1 beschriebenen Probleme leicht lösbar sind. Da das Weitverkehrsnetz einerseits und das Datenbankmanagementsystem, bzw. die Art und Weise dessen Verwendung, andererseits als Ursachen identifiziert wurden, könnten eine Verbesserung des Netzes und vielleicht auch der Wechsel von relationalen Datenbankmanagementsystemen zu anderen, beispielsweise objektorientierten, Systemen logische Optimierungswege sein. In diesem Abschnitt werden diese beiden Ansätze untersucht und bewertet.

2.4.1 Netzwerkoptimierung

Verbesserungen der Infrastruktur scheinen ein lohnendes Vorhaben zu sein. Die Entwicklung der letzten Jahre hat stetig wachsende Bandbreiten möglich gemacht,

dennoch klagen Anwender noch immer über „zu langsame Netze“. Die Frage stellt sich also, ob neuere Technologien zu einer Performance-Verbesserung für PDM-Systeme beitragen können.

In [SHK+97] findet sich ein Vergleich verschiedener Netztechnologien hinsichtlich ihrer Eignung für a) verschiedene Anwendungen (z. B. Video und Audio) sowie b) unterschiedliche Reichweiten (LAN oder WAN). Nach dieser Einordnung kommen für weltweit verteilte PDM-Systeme lediglich *ATM* und *Frame Relay*, sowie bedingt *ISDN*, *FDDI* und *DQDB* in Frage. Wir werden außerdem noch UMTS betrachten, das in Zukunft bei der multimedialen Kommunikation eine große Rolle spielen wird. Ebenso werden wir kurz das 10Gigabit-Ethernet ansprechen, welches sich zur Zeit in der Entwicklung befindet.

2.4.1.1 ATM

ATM (Asynchroner TransferModus) [RTG97a, RTG97b, SHK+97] war ursprünglich für Einsatzgebiete wie Videokonferenzen, Dateitransfers, Backupsysteme, Video on Demand, Remote Visualization und Supercomputer Networking (Kopplung von Höchstleistungsrechnern) gedacht. Dabei handelt es sich durchwegs um Szenarien, in welchen große Datenmengen zwischen den Kommunikationspartnern ausgetauscht werden.

ATM arbeitet verbindungsorientiert, d. h. zu Beginn einer Kommunikation wird ein Verbindungspfad vom Sender zum Empfänger definiert, über welchen sämtliche Pakete (sogenannte ATM-Zellen) transportiert werden. Bei diesem Verbindungsaufbau kann eine Verbindungsqualität gefordert werden (Quality of Service), die nach einmaliger Zusicherung während der gesamten Lebensdauer dieser Verbindung eingehalten wird.

Der Verbindungsaufbau ist sehr zeitaufwändig. Aktuelle ATM-Switches, über welche der Verbindungspfad definiert wird, benötigen dabei pro Gerät zwischen 25 und etwa 1000 Millisekunden Schaltzeit. Typischerweise sind mehrere Switches in den Verbindungsaufbau involviert, so dass dieser Vorgang durchaus mehrere Sekunden in Anspruch nehmen kann. Da die anschließende Datenübertragung jedoch äußerst performant abgewickelt wird (25 bis 155Mbit/s bis zum Endgerät sind durchaus realistisch, für Massendatentransfer sogar bis zu 2.5 Gbit/s), fällt diese anfängliche Verzögerung bei den anvisierten Einsatzgebieten nicht ins Gewicht.

Heutige PDM-Systeme jedoch sind aus der Sicht des Kommunikationsverhaltens mit diesen Anwendungen nicht vergleichbar. Hier werden *viele Kommunikationen* durchgeführt, wobei jeweils relativ *kleine Datenmengen* übertragen werden.

Würde man nun für jede dieser Kommunikationen eine eigene Verbindung aufbauen, so wäre das Antwortzeitverhalten noch schlechter als mit heute eingesetzten Netztechnologien. Alternativ könnte man für eine Serie von Anfragen *eine* Verbindung aufbauen, doch wird dadurch die möglichst große Ausnutzung des Übertragungspotentials von ATM untergraben, da die geforderte – und zugesicherte – Bandbreite zum großen Teil ungenutzt bleibt. Beschränkt man sich auf eine Verbindung ohne zugesicherte Qualität, so läuft man Gefahr, dass in Zeiten starken Datenverkehrs auf Grund von „Verkehrsstau“ ATM-Zellen verloren gehen und damit einhergehend erneut Verzögerungen entstehen.

Fazit: ATM kann die in dieser Arbeit aufgezeigten Schwächen heutiger PDM-Systeme nicht kompensieren. Wohl können die Systeme bei der Übertragung von Massendaten, wie CAD-Modelle, von ATM profitieren, die Navigation in den Produktstrukturen jedoch lässt sich damit nicht optimieren.

2.4.1.2 Frame Relay

Frame Relay ist eine verbindungsorientierte Datenkommunikationstechnologie, die insbesondere in den USA, aber auch zunehmend in Deutschland Verbreitung findet. Es bietet sich besonders in Grafik- und CAD-Anwendungen als kostengünstige Alternative zur ATM-Technologie an.

Ähnlich wie ATM kann Frame Relay die Vorzüge nur bei der Übertragung von großen Datenmengen ausspielen (die bereitgestellte Bandbreite bewegt sich zwischen 56 Kbit/s und 45 Mbit/s). Da die Latenzzeiten jedoch nicht entscheidend gesenkt werden können, sind die Probleme heutiger PDM-Systeme auch über diese Technologie nicht zu beheben.

2.4.1.3 10GBit Ethernet

Das *10 Gigabit Ethernet* wurde konzipiert für den Einsatz in LAN, MAN und WAN. Es hält sich an das Ethernet Frame-Format sowie die Frame-Größe (nach IEEE 802.3, vgl. [DDW02, Oli02, Tol00]) und stellt quasi ein „überdimensionales LAN“ dar.

Hauptsächliches Anwendungsgebiet wird der Backbone-Bereich sein, über welchen mehrere Gigabit-Segmente gekoppelt werden. Aussagen zu erwarteten Latenzzeiten sind heute kaum erhältlich, es ist jedoch anzunehmen, dass sie im WAN immer noch deutlich über den Latenzzeiten von LANs liegen werden und die Performance-Probleme der PDM-Systeme somit erhalten bleiben.

2.4.1.4 FDDI und DQDB

FDDI (Fiber Distributed Data Interface) und DQDB (Distributed Queue Dual Bus) sind zwei Technologien für den MAN-Bereich (Metropolitan Area Network, Ausdehnung bis etwa 100 Kilometer). Beide Technologien basieren auf zwei gegenläufigen Ringen bzw. Bussen und sind hauptsächlich im Backbone-Bereich zu finden. FDDI unterstützt Übertragungsraten von etwa 100 Mbit/s, DQDB bis zu 140 Mbit/s.

Da beide Verfahren nicht für den WAN-Einsatz konzipiert sind, fallen sie als mögliche Lösung für das Performance-Problem aus.

2.4.1.5 UMTS

UMTS (Universal Mobile Telecommunication System) [For98a, For98b, For00, Les02] soll die sich ständig erweiternde Welt der Mobilkommunikation revolutionieren. Es ist gedacht für die Bereitstellung von Informations-Services für den Massenmarkt, Multimedia-Anwendungen (Video-Telefon, Internetzugriff etc.), Mobile Commerce, Whiteboarding und vieles mehr.

Die Verbindung der Endgeräte erfolgt für kurze Übertragungswege über terrestrische Funkverbindungen, im internationalen Kontext werden Satellitenverbindungen eingesetzt. Auch für die Kommunikation weltweit verteilter PDM-Systeme mit UMTS wäre der Einsatz von Satelliten nötig. Allein die Signallaufzeiten jedoch vom Sender über den Satellit (oder auch mehrere Satelliten) zum Empfänger, die rein rechnerisch bereits mehr als eine halbe Sekunde betragen können (vgl. Abbildung 2.8), verhindern eine vernünftige Performance für PDM-Systeme heutiger Bauart.

2.4.1.6 Fazit

Die beschriebenen Performance-Probleme lassen sich keinesfalls nur durch den Einsatz schnellerer und besserer Netzwerktechnologien beheben. Selbst schnelle terrestrische Leitungen sind auf Grund der hohen Latenzzeiten und/oder Verzögerungen beim Verbindungsaufbau nicht in der Lage, die Performance-Verluste einzuschränken. Es ist zudem nicht zu erwarten, dass gerade Länder, deren IT-Infrastruktur nicht den heutigen Anforderungen entspricht, kurz- bis mittelfristig entsprechende Leitungen zur Verfügung stellen können – hier sind die Anwender auf Satellitenverbindungen mit den entsprechend langen Signallaufzeiten angewiesen. Eine radikale Verbesserung des Kommunikationsverhaltens der PDM-Systeme ist daher unabdingbar.

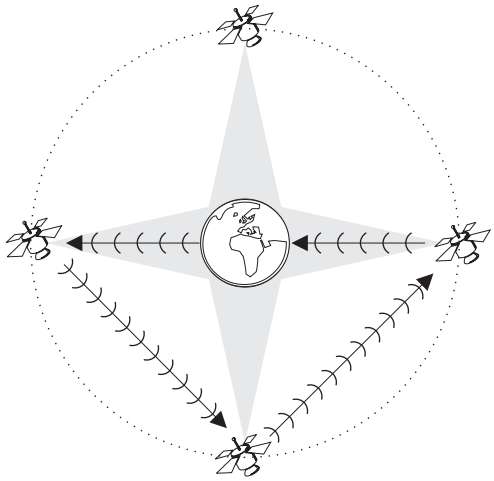


Abbildung 2.8: Prinzip InmarSat mit vier geostationären Satelliten – weltweite Kommunikation über maximal drei Satelliten. Das Signal legt ca. $2 * 10^5$ km zurück und benötigt dafür etwa 2/3s zuzüglich der Verarbeitungszeit in den Satelliten.

2.4.2 Wechsel des Datenbankparadigmas

In Abschnitt 2.3.1 wurde bereits darauf hingewiesen, dass zumindest einige PDM-Systemhersteller beim Design ihrer Systeme wohl darauf spekulierten, nach einer gewissen Anlaufphase für die Datenhaltung die ursprünglich genutzten relationalen Datenbankmanagementsysteme gegen objektorientierte Systeme auszutauschen. Ob mit diesem Paradigmenwechsel die aufgezeigten Performance-Probleme gelöst werden könnten, wollen wir hier diskutieren.

2.4.2.1 Objektbegriff

Objekte, die eine Substruktur aufweisen, werden häufig als *komplexe Objekte* bezeichnet. Oftmals wird dabei davon ausgegangen, dass die eingeschachtelten Strukturen quasi untrennbar mit dem übergeordneten Objekt verbunden sind. Auf der Instanz-Ebene ist diese Annahme sicher korrekt: Eine konkrete Ausprägung eines Fensterhebers beispielsweise wird in genau eine Instanz einer Autotür eines konkreten Modells eingebaut. Der Fensterheber kann folglich als eingeschachteltes Bauteil einer Autotür verstanden werden.

Auf der PDM-System-Ebene jedoch findet diese Einschachtelung keine Entsprechung: Der gleiche Fensterheber – genauer ausgedrückt: ein Fensterheber der gleichen Bauart – kann auch in anderen Türen sogar anderer Fahrzeugmodelle

eingebaut werden. Prinzipiell gilt diese Aussage für alle Komponenten eines Produktes! Es ist demnach nicht sinnvoll, die *uses*-Beziehung als Einschachtelung zu betrachten.

Die Produktstruktur besteht folglich aus miteinander verknüpften, eigenständigen Objekten und stellt damit *kein* komplexes Objekt im herkömmlichen Sinn dar.

2.4.2.2 Page-Server versus Objekt-Server

Page Server sind von Haus aus funktional sehr primitiv ausgelegt, d. h. sie können nur Seiten speichern und bei Bedarf bereitstellen, Objekte kennen sie hingegen nicht. Folglich ist es auch nicht möglich, bereits auf einem derartigen Datenbankserver Zugriffsrechte auf Objekten auszuwerten. Die komplette Funktionalität bezüglich des Objekt-Managements – und damit auch die Regelauswertung – liegt beim Client. Das angestrebte *function shipping* ist damit nicht realisierbar.

Objekt-Server sind mit deutlich mehr Funktionalität ausgestattet. Das komplette Objekt-Management liegt beim Server, der auf Anfrage konkrete Objekte zurückliefert. Relationale DBMS, objekt-relationale DBMS und ein Teil der objektorientierten DBMS sind Objekt-Server, wobei die Objekte unterschiedliche Qualität (primitive Tupel oder strukturierte Objekte) haben können. Objekt-Server arbeiten nach dem *function shipping*-Prinzip und sind deshalb für die Bedürfnisse der PDM-Systeme besser geeignet als Page-Server.

2.4.2.3 OODBMS oder (O)RDBMS?

Objektorientierte Datenbankmanagementsysteme (kurz: OODBMS, vergleiche [ABD+89, CBB+97, MW00]) galten noch vor ein paar Jahren als zukunftsträchtige Technologie und besonders geeignet zur Unterstützung technisch orientierter Anwendungen [AMKP85, KM94, Nit89]. Inzwischen hat die Euphorie nachgelassen, vielen OODBMS blieb der Durchbruch am Markt verwehrt, und sie führen heute mehr oder weniger nur noch ein Nischendasein.[3]

Als Vorteile dieser Systeme für technische Applikationen, z. B. CAD-Anwendungen, zählen vor allem die folgenden beiden Aspekte:

a) OODBMS speichern komplexe Objekte, ohne sie in Elementarobjekte zu zerlegen, d. h. die Objektstruktur der Applikation wird direkt in der Datenbank abgebildet. Das Problem des *impedance mismatch* relationaler DBMS tritt hier folglich nicht auf. Objekte können dabei aus Subobjekten bestehen, die selbst keine eigene Identität besitzen.

[3] Ein umfassender Vergleich der verschiedenen Datenbank-Paradigmen findet sich in [SC97].

b) Beziehungen können explizit modelliert werden (Referenztyp), d. h. Relationen werden nicht auf der Basis von wert-gleichen Attributen hergestellt, sondern auf der Ebene von eindeutigen Objekt-IDs. Strukturen, die durch derartige Referenzen aufgebaut werden, lassen sich effizient traversieren (Direktzugriff auf Subobjekte).

Betrachten wir nun diese beiden Aspekte in Bezug auf PDM-Systeme:

zu a): Wie bereits im Abschnitt 2.4.2.1 dargelegt, existieren im PDM-Umfeld keine komplexen Objekte in diesem Sinne. Die Objekte im PDM-System sind vielmehr rekursiv verknüpfte „Elementarobjekte“. Mit der Speicherung komplexer Objekte lässt sich folglich keine Verbesserung für PDM-Systeme erzielen.

zu b): Beziehungen in OODBMS sind Zeiger (Referenzen) auf Subobjekte, im Falle von bidirektionalen Beziehungen auch die inversen Zeiger von Subobjekten auf Superobjekte. Zeiger enthalten direkt die Speicheradresse des referenzierten Objektes im Vergleich zu wertbasierten Referenzen, wie sie beispielsweise Fremdschlüssel in relationalen Datenbankmanagementsystemen darstellen. Somit entfällt hier die Übersetzung der Referenz in die zugehörige Speicheradresse (vgl. Abbildung 2.9).

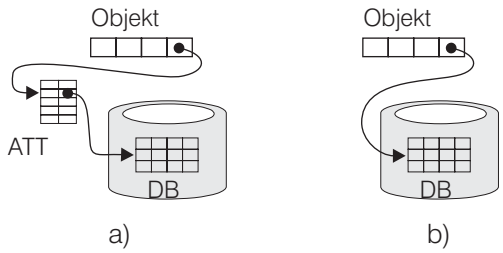


Abbildung 2.9: Wertbasierte Beziehung a) mit Übersetzung durch eine Adresstransformationstabelle (ATT) im Vergleich zur Referenz b)

Neben diesem positiven Gesichtspunkt sind jedoch auch folgende Aspekte zu berücksichtigen: Die Zeiger können nicht an Bedingungen geknüpft werden. Es kann folglich nicht ausgedrückt werden, dass eine Referenz nur zu gegebenen Zeitintervallen bzw. unter gegebenen Strukturoptionen gültig ist. Gültigkeiten und Strukturoptionen sind also nicht unmittelbar in das Konzept integrierbar. Würde man ein PDM-System auf der Basis eines OODBMS aufsetzen, so müssten die Beziehungen (z. B. die *uses*- oder *has_revision*-Beziehung) analog zur relationalen Darstellung als eigenständige Objekte modelliert werden, die mit Bedingungen verknüpft werden

können. Diese Objekte könnten dann Referenzen (anstelle von heute verwendeten Fremdschlüsseln) auf die in Beziehung stehenden Objekte (Zusammenbauten und Versionen) enthalten.

In verteilten Umgebungen birgt die Verwendung von Referenzen keine gravierenden Vorteile hinsichtlich der Performance von Navigationen in verteilten Strukturen: Ob eine Speicheradresse oder eine wertbasierte Referenz über das Netz übertragen werden muss, spielt für die Gesamt-Antwortzeit des Systems keine Rolle. Entscheidend ist, dass das Navigations-Verfahren heutiger PDM-Systeme unabhängig vom gewählten DBMS immer gleich viele Anfragen an entfernte Server benötigt.

Neben den nur als gering einzuschätzenden Vorteilen, die OODBMSe für PDM-Systeme bieten können, existieren auch einige Nachteile dieser Systeme. So wird z. B. in [SC97] auf die mangelnde Performance heutiger Implementierungen hinsichtlich mengenorientierter Anfragen hingewiesen. Diese sind in PDM-Systemen jedoch ebenfalls effizient zu unterstützen. Die im Weiteren bemängelte Robustheit von am Markt erhältlichen OODBMS in Bezug auf Backup und Recovery stellt für produktiv eingesetzte PDM-Systeme ein untragbares Risiko dar. Ein Ausfall des PDM-Systems könnte je nach Einsatzbereich ein Unternehmen von der Entwicklung über die Produktion bis hin zum Vertrieb „lahmlegen“. Inkonsistenzen in den Produktdaten könnten zu Verzögerungen führen und, abhängig vom Zeitpunkt der Aufdeckung, hohe Kosten verursachen.

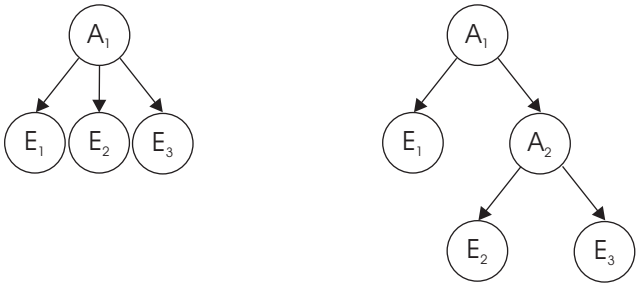
Relationale Datenbankmanagementsysteme (RDBMS) und deren Erweiterungen zu objekt-relationalen Systemen (ORDBMS) stellen heute den Standard bei der Speicherung von Daten dar. Die langjährige Markterfahrung bedeutet deutliche Vorteile hinsichtlich der Stabilität der Systeme sowie der Erfahrung bezüglich Optimierung bei Zugriffen und Speicherung der Daten.

Die Funktionalität der (objekt)relationalen DBMS ist im SQL:1999-Standard [ANS99a, ANS99b, EM99] definiert. Neben den bekannten „flachen“ Datentypen (Integer, String, Boolean etc.) werden nun auch Arrays angeboten, mit welchen ansatzweise NF^2-ähnliche Strukturen erzeugt werden können (vgl. [DKA+86]).

Die Abbildung von Produktstrukturen auf (objekt-)relationale Datenbanksysteme kann nahezu als Abbildung jedes Elementarobjekttyps (Zusammenbau, Einzelteil, Spezifikation etc.) auf eine separate Tabelle geschehen. Auch Beziehungen (z. B. *uses*- und *has_revision*-Beziehung) werden auf jeweils eine Tabelle abgebildet. Für die Speicherung zugewiesener Gültigkeiten und Strukturoptionen bietet sich die Verwendung *mengenwertiger Attribute*[4] an.

[4]Nach derzeitiger Planung werden mengenwertige Attribute erst im Standard SQL:2003 enthalten sein [Tür03], in diesem Fall können sie aber auch durch Arrays ersetzt werden.

Diese auf den ersten Blick möglicherweise primitiv wirkende Speicherung der flachen *Elementarobjekte* hat im Hinblick auf den Lebenszyklus eines Produkts einen entscheidenden Vorteil: Die „Sicht“ auf die Struktur des Produktes ändert sich im Laufe des Produktlebenszyklus. Ist für die Entwickler und Konstrukteure die Design-Sicht wichtig, d. h. der funktionale Aufbruch des Produktes in Systeme und Subsysteme, so benötigen die Mitarbeiter in der Fertigung eine möglicherweise davon abweichende Sicht (vgl. Abbildung 2.10): Hier steht nicht der funktionale Zusammenhang im Vordergrund, statt dessen sind Informationen z. B. darüber erforderlich, in welcher Reihenfolge welche Teile zusammengebaut werden müssen! Durch datentechnische Einschachtelung von Objekten würde die der Einschachtelung zugrunde liegende Sicht bevorzugt, andere Sichten wären unter Umständen nur sehr ineffizient benutzbar. Durch die separate Speicherung der Elementarobjekte hingegen können beliebig viele Beziehungstypen (für die unterschiedlichsten Sichten) definiert werden, die alle gleichermaßen unterstützt werden können.



a) b)

Abbildung 2.10: Unterschiedliche Sichten auf die gleichen Objekte: a) In der *design*-View besteht der Zusammenbau A_1 unmittelbar aus den drei Einzelteilen E_1, E_2, E_3, wohingegen b) in der *as-built*-View zunächst E_2 und E_3 zusammengebaut werden.

Bei Systemen, die mengenwertige Attribute unterstützen, lässt sich die Zuordnung von Gültigkeiten und Strukturoptionen vereinfachen: Jedes Beziehungsobjekt kann hier in einem mengenwertigen Attribut die zugeordneten Gültigkeiten aufnehmen, und in einem zweiten mengenwertigen Attribut die Menge der zugeordneten Strukturoptionen referenzieren. Damit können umständliche $n : m$-Relationen zwischen Beziehungsobjekten und Strukturoptionen vermieden werden.

Objektrelationale DBMS bieten nun auch Referenz-Typen an. Analog zu den Referenzen in objektorientierten DBMS können jedoch auch hier keine Bedingungen verknüpft werden.

Im Gegensatz zu den OODBMS sind (O)RDBMS auf komplexe Queries gut abgestimmt und stellen entsprechend ausgereifte Optimierer zur Verfügung. Wirkt die Anfragesprache OQL [ASL89, Clu98] bei den OODBMS eher aufgesetzt, ist die Anfragesprache SQL der (O)RDBMS ein integraler Bestandteil der Systeme.

Der Einsatz eines objektrelationalen Datenbankmanagementsystems, wie es z. B. von IBM im Produkt DB2 Universal Database angeboten wird, ist unter den gegebenen Voraussetzung durchaus sinnvoll.

Alle weiterführenden Konzepte, die auf der Einschachtelung von Subobjekten basieren, bringen keine weiteren Vorteile mit sich: Die Produktstruktur *soll* einerseits aus den bereits genannten Gründen nicht als ein komplexes Objekt gespeichert werden, andererseits *kann* diese Einschachtelung zumindest bei statischen Typsystemen nicht durchgeführt werden, da die Tiefe der Struktur (und damit die Tiefe des korrespondierenden geschachtelten Typs) nicht von vornherein festgelegt werden kann!

2.4.3 Fazit

Die Performance-Probleme heutiger PDM-Systeme, die ihre Ursachen in dem Kommunikationsverhalten und der Art und Weise der Datenbankverwendung haben, lassen sich nicht nur durch eine verbesserte Infrastruktur oder den Austausch des Datenbankmanagementsystems beheben.

Schnellere Netze können – wo vorhanden oder erhältlich – kurzfristig und auch nur zeitlich begrenzt geringfügige Verbesserungen des Antwortzeitverhaltens bewirken, langfristig jedoch, und besonders auch in Gebieten, die auf Funkverbindungen via Satellit angewiesen sind, kann das Problem nicht beseitigt werden.

Die Lösung des Problems kann folglich nur in einer Änderung des Kommunikationsverhaltens bei gleichzeitig fortschrittlicher Nutzung des Datenbankmanagementsystems liegen. Es gilt also, das Kommunikationsaufkommen so gering wie möglich zu halten, wobei Aufgaben, die heute noch das PDM-System ausführt, an das Datenbanksystem übergeben werden.

2.5 Repräsentation von Produktstrukturen

In den weiteren Kapiteln dieser Arbeit werden wir die Produktstruktur sowohl in der Datenbankdarstellung als auch in der Graphdarstellung betrachten. Abschnitt 2.5.1 zeigt einen kleinen Ausschnitt aus einem Datenbankschema für PDM-Systeme auf Basis eines (objekt)relationalen DBMS, in Abschnitt 2.5.2

werden die graphentheoretischen Grundlagen eingeführt sowie eine neue Klasse von Graphen definiert, mit denen Produktstrukturen dargestellt werden können.

2.5.1 Datenbank-Schema

Es würde den Rahmen dieser Arbeit sprengen, wollte man hier alle Objekte eines PDM-Systems zusammen mit deren Abbildung auf das Datenbanksystem formal und vollständig definieren. Standard-Metaphase in der Version 3.2 [MP] beispielsweise wird mit knapp 250 verschiedenen Objekttypen ausgeliefert, in großen Unternehmen können durch Customizing leicht noch einmal so viele Typen hinzukommen.

Für das prinzipielle Verständnis genügt aber ein kleiner Ausschnitt, mit dem die Struktur entsprechend des Master-Version-Konzepts zusammen mit Gültigkeiten und Strukturoptionen dargestellt werden kann. Dazu zählen die Objekte und Beziehungen, die in Abbildung 2.7 dargestellt sind, zuzüglich der Strukturoptionen. Abbildung 2.11 zeigt ein UML-Modell[5] dieser Daten, Abbildung 2.12 zeigt ansatzweise die Schemata der zugehörigen Datenbanktabellen.

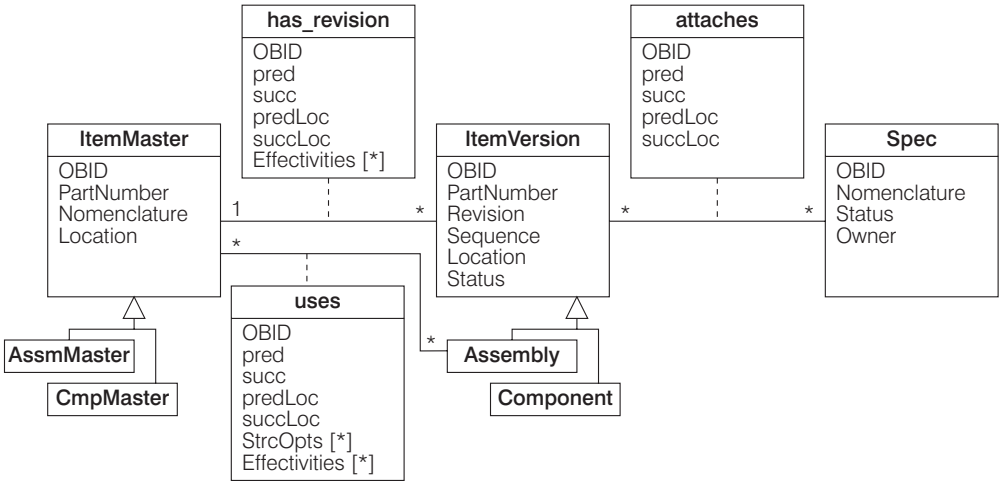


Abbildung 2.11: Datenmodell der Produktstruktur (Auszug als UML-Modell)

Die Bedeutung der Attribute im Einzelnen:

Alle Objekte werden mit einer eindeutigen ID, der *OBID*, versehen. Unter *PartNumber* wird die Teilenummer verstanden, *Nomenclature* stellt den Bezeichner

[5] Näheres zu UML siehe [HK03, OMG01].

AssmMstr (Zusammenbau-Stammdaten)

OBID	PartNumber	Nomenclature	Location	...

CmpMstr (Einzelteil-Stammdaten)

OBID	PartNumber	Nomenclature	Location	...

Assembly (Zusammenbau)

OBID	PartNumber	Revision	Sequence	Location	Status	...

Component (Einzelteil)

OBID	PartNumber	Revision	Sequence	Location	Status	...

Spec (Spezifikation)

OBID	Nomenclature	Status	Owner	...

StructureOpt (Strukturoption)

OBID	StrcOptName	...

uses

OBID	pred	succ	predLoc	succLoc	StrcOpts	Effectivities	...

has_revision

OBID	pred	succ	predLoc	succLoc	Effectivities	...

attaches

OBID	pred	succ	predLoc	succLoc	...

Abbildung 2.12: Relationenschema (Auszug)

dar. Im Attribut *Location* wird der Speicherort (Standort bzw. Server) gehalten, an welchem sich das Objekt befindet.

Revision und *Sequence* bezeichnen die Version (typischerweise mit A, B usw.) eines Objektes, sowie innerhalb einer Version fortlaufend nummeriert unterschiedliche Zwischenstände.[6]

Die Attribute *pred* und *succ* in den Beziehungsrelationen bezeichnen den direkten Vorgänger bzw. Nachfolger. Im Falle der *uses*-Beziehung referenziert *pred* ein Assembly, *succ* zeigt auf einen Assembly- oder Component-Master. *predLoc* und *succLoc* enthalten jeweils den Speicherort der Objekte, auf die *pred* bzw. *succ* verweisen. Damit sind Referenzen über Standortgrenzen hinweg möglich.

[6]Oftmals bedarf es mehrerer Änderungsiterationen, bis aus einer Version A tatsächlich eine neue Version B wird. Diese Zwischenstände werden typischerweise über die Sequenz-Nummer unterschieden.

Zur Abbildung der Referenzen auf Strukturoptionen verwenden wir das mengenwertige(!) Attribut *StrcOpts*. Hier werden die Objekt-IDs aller Strukturoptionen gespeichert, die dem Beziehungsobjekt zugeordnet sind. Analog wird für die Speicherung der Gültigkeiten eine Menge von Intervallen in dem Attribut *Effectivities* abgelegt.

2.5.2 Die Produktstruktur als Graph

2.5.2.1 Graphentheoretische Grundlagen

Ein *gerichteter Graph* $G = (V, E)$ besteht aus einer Menge V von *Knoten* und einer Menge $E \subseteq V \times V$ von *Kanten* (auch: *Pfeilen*). Ein Paar (u, v) heißt *Kante von* u *nach* v. Wir nennen u den *Anfangs-* und v den *Endknoten* der Kante (u, v). Wir beschränken uns auf endliche Mengen von Knoten und Kanten, also *endliche Graphen* [OW96, Sed95].

2.5.2.2 Produktstruktur-Graphen

Auf den ersten Blick mag eine Produktstruktur ähnlich aussehen wie ein gerichteter, azyklischer Graph (DAG, Directed Acyclic Graph, vgl. [CLR96]): Jedes (Teil-)Projekt, jeder Zusammenbau und jedes Einzelteil (sowohl Master als auch Version) ist ein Knoten in einem derartigen Graph, und alle gerichteten Beziehungen zwischen diesen Knoten – z. B. die *uses*-Beziehung – stellen die entsprechenden Kanten dar.[7]

Die Ausdrucksmächtigkeit eines DAGs reicht allerdings nicht aus, um die Flexibilität einer Produktstruktur, wie wir sie in Abschnitt 2.1 eingeführt haben, auszudrücken. Es fehlt die Möglichkeit, unterschiedliche Produkt-Konfigurationen, also Gültigkeiten und Strukturoptionen, zu berücksichtigen: Entweder müsste jede Produktinstanz in einem separaten DAG abgelegt werden, was bei extrem konfigurierbaren Produkten etwa in der Automobil- oder Flugzeug-Industrie auf Grund der gigantischen Datenmenge unmöglich ist, oder die Strukturen aller möglichen Instanzen werden zu einem einzigen Graph zusammengefasst, woraus sich jedoch eine konkrete Produktinstanz nicht mehr rekonstruieren lässt.

Um dieses Problem zu lösen, benötigen wir einen Test auf dem DAG, der aussagt, ob eine Kante (u, v) des DAGs zu einer konkreten Produktinstanz gehört oder nicht. Dieser Test beruht auf der Evaluierung von Bedingungen, die wir im Folgenden einführen werden.

[7] Wenn wir die Objekte einer Produktstruktur betrachten und dabei den Zusammenhang zum Graph suchen, so sprechen wir auch von *Knoten-Objekten* und *Kanten-Objekten*.

Sei $\mathcal{O}_\chi$ die Menge der dem Anwender zur Verfügung stehenden Optionen $\mathcal{O}_\chi = \{\chi_1, \chi_2, \ldots, \chi_n\}$ zur Konfiguration von Produktinstanzen, und $\mathcal{O}_\Psi$ die Menge der Gültigkeiten $\mathcal{O}_\Psi = \{\Psi_1, \Psi_2, \ldots, \Psi_n\}$. Ein Benutzer des PDM-Systems kann keine, einige oder auch alle Optionen wählen um anzuzeigen, welche optionalen Strukturteile der Basisstruktur hinzugefügt werden sollen. Zudem kann er einen Zeitpunkt wählen, welcher die Gültigkeit der gesuchten Struktur festlegt. Wir beschreiben diesen Sachverhalt mit dem folgenden Formalismus, der auf der Aussagenlogik basiert (vgl. [Fit90, LE78]):

Wir betrachten $\mathcal{O} = \mathcal{O}_\chi \cup \mathcal{O}_\Psi$ als eine Menge von atomaren Formeln. Indem der Benutzer einige Optionen sowie einen Zeitpunkt wählt, definiert er eine *Zuordnung* $\bar{\mathcal{A}} : \mathcal{O} \to \{0, 1\}$, d. h. den gewählten Optionen wird der Wert 1 zugeordnet, alle nicht gewählten werden auf 0 gesetzt, ebenso werden die Gültigkeiten, in deren Start-Ende-Intervall der gewählte Zeitpunkt fällt, auf 1 gesetzt, alle anderen auf 0.

Basierend auf $\mathcal{O}$ führen wir nun die Menge $\mathcal{C}$ logischer Ausdrücke ein, die wir im Folgenden als *Bedingungen* bezeichnen. Sie sind ausschließlich über der Menge der atomaren Formeln $\mathcal{O}$ definiert, d. h. die Elemente aus $\mathcal{O}$ können mit den Operatoren AND ($\wedge$), OR ($\vee$) und NOT ($\neg$) verknüpft werden. Der Vollständigkeit halber sei eine Bedingung *true* ($\top$) immer Element von $\mathcal{O}$.

Jetzt erweitern wir noch $\bar{\mathcal{A}}$ zu $\mathcal{A} : \mathcal{C} \to \{0, 1\}$, um anzuzeigen, ob eine Bedingung $c \in \mathcal{C}$ wahr (d. h. 1) ist oder nicht. Damit sind die Grundlagen geschaffen, um den gerichteten, azyklischen Bedingungsgraph DACG (Directed Acyclic Condition Graph) einzuführen:

Definition 1: Directed Acyclic Condition Graph (DACG)

> Ein *Directed Acyclic Condition Graph* G^c ist ein erweiterter DAG mit den folgenden Eigenschaften:
>
> $G^c = (V, E, \mathcal{C})$ mit $E \subseteq V \times V \times \mathcal{C}$, V ist die Menge der Knoten, $\mathcal{C}$ ist eine Menge von Bedingungen, und E ist die Menge der Kanten, die mit Bedingungen aus $\mathcal{C}$ versehen sind.

Die Semantik ist intuitiv klar: Eine Kante von Knoten u zu Knoten v ist nur dann gültig (d. h. ein Teil der Produktstruktur), falls die zugeordnete Bedingung c unter der gegebenen Belegung $\bar{\mathcal{A}}$ wahr ist (d. h. $\mathcal{A}(c) = 1$).[8] Wir schreiben für solche Kanten in der Regel $u \xrightarrow{c} v$, falls jedoch $c = \top$, so verzichten wir auf die Angabe der Bedingung und schreiben einfach $u \rightharpoonup v$.

[8] In der Automobil-Industrie kann eine solche Bedingung c z. B. garantieren, dass ein Zusammenbau v nur dann Teil von u ist, falls das zu bauende Fahrzeug mit einem Schiebedach ausgestattet werden soll.

Auf dem DACG können wir nun die Vorgänger- und Nachfolger-Relationen definieren:

Definition 2: Vorgänger und Nachfolger eines Knotens

Die Menge der *direkten Vorgänger* eines Knotens u wird durch die Funktion $pred : V \rightarrow 2^V$ definiert:

$pred(u) = \{v \in V | \exists v \xrightarrow{c} u \in E\}$

$pred^*_{G^c}$ beschreibt den transitiven Abschluss bezüglich G^c.

Die Menge der *direkten Nachfolger* eines Knotens u wird durch die Funktion $succ : V \rightarrow 2^V$ definiert:

$succ(u) = \{v \in V | \exists u \xrightarrow{c} v \in E\}$

$succ^*_{G^c}$ beschreibt den transitiven Abschluss bezüglich G^c.

Da wir in Kapitel 6 auf die Bildung der transitiven Hülle zurückgreifen werden, wollen wir sie bereits an dieser Stelle einführen:

Definition 3: Transitive Hülle eines DACGs

Die transitive Hülle G^{c*} eines DACGs $G^c = (V, E, \mathcal{C})$ ist definiert wie folgt:
$G^{c*} = (V, E^*, \mathcal{C}^*)$ mit $E^* = E \cup \{(u \xrightarrow{c} v) | u, v \in V \wedge$
$\exists d = \langle u \xrightarrow{c_1} n_1 \xrightarrow{c_2} \ldots \xrightarrow{c_k} n_k \xrightarrow{c_{k+1}} v \rangle$ in $G^c, (k > 0)$
und $c = c_1 \wedge c_2 \wedge \ldots \wedge c_{k+1}\}$ und $\mathcal{C}^* = \mathcal{C} \cup \{c | u \xrightarrow{c} v \in E^*\}$.

Analog zu gewöhnlichen DAGs verbindet die transitive Hülle eines DACGs Knoten, die Start- und Endpunkt eines Pfades im ursprünglichen Graphen sind. Im Falle eines DACGs wird solch eine zusätzliche Kante mit einer neuen Bedingung versehen, die sich aus der AND-Verknüpfung aller Bedingungen entlang des Pfades im Ursprungsgraphen ergibt.[9]

Ein Unterschied zur transitiven Hülle gewöhnlicher gerichteter Graphen ist hier allerdings anzumerken: Existieren in einem DAG mehrere Pfade von einem Knoten u zu einem anderen Knoten v, so enthält die transitive Hülle des DAGs *genau einen* Eintrag (u, v). Im Falle eines DACGs jedoch wird *für jede Bedingung*, unter welcher v von u aus erreichbar ist, *eine separate Kante* in der transitiven Hülle eingeführt!

[9] Formal lässt sich diese Verknüpfung über eine sogenannte Pfad-Algebra darstellen, vgl. [ADJ90, IRW93].

Teil II

Architekturen und Zugriffsstrategien für weltweit verteilte PDMS

Kapitel 3

Auswertung von Zugriffsregeln

3.1 Einführung

Informationen aller Art über die Produkte eines Unternehmens stellen ein bedeutsames Kapital dar. Daher ist es nur verständlich, dass für die Bearbeitung dieser Daten durch mitunter viele hundert Benutzer eine recht feingranulare Zugriffssteuerung entsprechend der unterschiedlichen Vollmachten und Befugnisse der Anwender möglich sein muss. Besonders in Umgebungen, in welchen mehrere Partner und Zulieferer zusammenarbeiten sollen, wird auf entsprechende Schutzmechanismen großer Wert gelegt.

In [RPM96] und [Sch96a] wird *Sicherheit* in drei Kategorien eingeteilt:

1. *Vertraulichkeit*, d. h. Schutz vor unbefugtem Informationsgewinn
2. *Integrität*, d. h. Schutz vor unbefugter Modifikation von Informationen
3. *Verfügbarkeit*, d. h. Schutz vor unbefugtem Vorenthalten von Informationen und Betriebsmitteln

Vertraulichkeit und Integrität können durch Zugriffsregeln, wie sie nachfolgend beschrieben werden, erreicht werden, sofern kriminelle Vorgehensweisen wie das Abhören von Leitungen etc. außer Acht gelassen werden.[10] Der Aspekt der Verfügbarkeit (z. B. durch stabile Netzverbindungen, Maßnahmen zur Verhinderung von „Einbrüchen“ in ein System etc.) wird hier nicht betrachtet.

[10] Maßnahmen gegen derartige „Überfälle“ sind z. B. Verschlüsselungen und One-Way-Hashfunktionen, vgl. [Woh00].

Zugriffsregeln werden in PDM-Systemen zu Vergabe unterschiedlichster Berechtigungen eingesetzt. Um einen Eindruck darüber zu gewinnen, mögen die folgenden, typischen Beispiele dienen:

In der Automobilindustrie etwa soll ein Konstrukteur eines Zulieferers, welcher beispielsweise einen Scheinwerfer entwickelt, nur auf die umliegenden Komponenten wie Kotflügel, Motorhaube, Stoßstange oder Kühlergrill lesend zugreifen können. Daten entfernterer Komponenten, beispielsweise die des Motors, haben den Mitarbeiter dieses Zulieferers nicht zu interessieren und dürfen daher von ihm auch nicht visualisiert werden können. Umgekehrt genügt dem Auftraggeber möglicherweise der lesende Zugriff auf die Hüll-Geometrie des Scheinwerfers, um Einbau-Untersuchungen durchführen zu können. Die internen Details des Scheinwerfers – die ja wiederum spezifisches Know-How des Zulieferers darstellen, welches dieser nur ungern preisgeben wird – können dem Auftraggeber eventuell verborgen bleiben.

Anwender aus dem kaufmännischen Bereich etwa, die keinen technischen Hintergrund besitzen, sollen keine Bauteile und auch keine strukturellen Abhängigkeiten zwischen Bauteilen erzeugen können. Stattdessen arbeiten diese bevorzugt mit Dokumenten, die den Zusammenbauten und Einzelteilen angehängt sind. Dafür benötigen sie die entsprechenden Lese- und Änderungsrechte für derartige Dokumente.

Eine etwas andere Art der Zugriffssteuerung ergibt sich aus der Konfigurationssteuerung eines Produktes. Können Produkte in verschiedenen Versionen hergestellt werden, so werden alle diese Versionen in einer einzigen Produktstruktur verwaltet, aus welcher – durch die Angabe entsprechender Strukturoptionen und Gültigkeiten – konkrete Produkt-Instanzen ausgewählt werden können. So kann beispielsweise eine Strukturoption mit der Bezeichnung „Schiebedach“ vorliegen, die einer Baugruppe zugeordnet wird, welche die Struktur des Schiebedaches repräsentiert. Wünscht ein Anwender die Struktur eines Autos ohne Schiebedach zu sehen, d. h. er wählt diese Sonderausstattung oder Option ab, so wird aus der Gesamtstruktur die Struktur des Schiebedaches ausgeblendet. – Ähnlich zu den bereits motivierten Zugriffsregeln findet also auch hierbei eine gewisse Selektion aus allen verfügbaren Informationen statt.

In heutigen PDM-Systemen werden alle diese Typen der Zugriffssteuerung vom PDM-Server oder -Client überwacht und ausgewertet. Wie bereits im Abschnitt 2.3.1 beschrieben, ist diese Auswertungsstrategie eine der Ursachen der Performance-Probleme. Im Folgenden werden die verschiedenen Facetten der Zugriffssteuerung genauer betrachtet und Möglichkeiten aufzeigt, um die Auswertung von Zugriffsregeln möglichst frühzeitig durchführen zu können und damit akzeptable Antwortzeiten für die Anwender zu erzielen.

3.2 Zugriffssteuerung in PDM-Systemen

3.2.1 Gewährung von Zugriffsrechten

Zugriffsregeln in PDM-Systemen sollten den Benutzern Rechte *gewähren*. Das bedeutet, ein Benutzer, für welchen keine Zugriffsregel (vgl. Abschnitte 3.2.2.1 und 3.2.2.2) existiert, hat per Definition *keine Rechte*. Erst durch Hinzufügen entsprechender Regeln (durch den System-Administrator) erhält dieser Benutzer Rechte, um die ihm übertragenen Aufgaben mit dem System durchführen zu können. Dabei gilt, dass ein einmal gewährtes Recht nicht durch Hinzufügen anderer Zugriffsregeln entzogen werden kann. Stattdessen muss die das Recht gewährende Regel aus dem System entfernt werden, um das Zugriffsrecht wieder zu entziehen.

Im Prinzip wäre auch der umgekehrte Ansatz denkbar: Hier hätte ein Benutzer zunächst alle Rechte, d. h. er könnte auf allen Objekten jede beliebige Operation ausführen, und durch entsprechende Regeln würden diese Rechte dann *eingeschränkt*. Neben dem zu erwartenden erhöhten Administrationsaufwand kann dieser Ansatz jedoch dazu führen, dass – auf Grund fehlender Regeln – Benutzer Aktionen ausführen können, die jenseits ihrer Befugnisse liegen. Der Missbrauch derartiger, fälschlicherweise eingeräumten Rechte könnte zudem nur schwerlich bemerkt und nachgewiesen werden.

Aus diesem Grund ist der *gewährende* Ansatz eindeutig vorzuziehen. Stellt dabei ein Benutzer das Fehlen von zur Ausführung seiner Aufgaben benötigten Rechten fest, so kann der Administrator jederzeit die entsprechenden Regeln definieren und somit bedarfsgesteuert Benutzer-Aktionen „freischalten“.

3.2.2 Zugriffsregeln in PDM-Systemen

Zugriffsregeln in PDM-Systemen können durch vier Merkmale beschrieben werden: Sie legen fest,

- welcher *Benutzer*
- auf welchem *Objekttyp*
- welche *Aktion*
- unter welcher *Bedingung*

ausführen darf.[11]

[11] In Groupware-Produkten finden sich ähnliche Beschreibungsmechanismen, ein guter Überblick findet sich in [SWW00].

Der *Benutzer* kann dabei ein konkreter PDM-Benutzer sein (z. B. ein Mitarbeiter namens „Michael Schmidt“), es kann aber auch eine Benutzergruppe angegeben werden (z. B. „Konstrukteure“). Im ersten Fall gilt die Regel genau für diesen spezifizierten PDM-Anwender, im zweiten Fall erhalten alle Anwender, die in der angegebenen Benutzergruppe enthalten sind, das durch die Regel definierte Recht.

Objekttyp (auch als *Klasse* bezeichnet) kann jeder im PDM-System definierte Objekttyp sein, darunter Zusammenbauten, Einzelteile, aber auch Dokumente, CAD-Modelle (siehe auch Abbildung 2.3).

Unter *Aktion* sind die Benutzeraktionen zu verstehen, wie sie auch in Abschnitt 2.1.2 beschrieben sind. Dazu zählen also Erzeugen, Ändern und Anzeigen von Objekten, Suchen nach Objekten, Ein- und Aus-Checken, Expandieren der Struktur und einiges mehr.

Die Angabe einer *Bedingung* ermöglicht es, ein Zugriffsrecht auf ein Objekt z. B. von Attributwerten des Objektes selbst abhängig zu machen. So lässt sich beispielsweise definieren, dass ein Benutzer auf ein Objekt nur dann zugreifen darf, falls dessen Objekt-ID mit der Zeichenkette ’P101-’ beginnt.

Im Folgenden werden die Unterschiede zwischen *Objektzugriffsregeln* und *Klassenzugriffsregeln* aufgezeigt und festgelegt, welche Regeln für die weiteren Betrachtungen von Interesse sind.

3.2.2.1 Objektzugriffsregeln

Objektzugriffsregeln ermöglichen, wie der Name schon andeutet, den Zugriff auf konkrete Objekte, d. h. auf *Instanzen* eines bestimmten Objekttyps.[12] Derartige Regeln werden benötigt, um Aktionen wie Ändern und Anzeigen von Instanzen, Expandieren etc. zu erlauben. Dazu beziehen sich die *Bedingungen* dieser Regeln auf ein oder meherere Attributwerte der Instanz, auf welcher die Aktion ausgeführt werden soll.

Beispiele für Objektzugriffsregeln:

(1)
Benutzer:	Michael Schmidt
Klasse:	Zusammenbau
Aktion:	Display
Bedingung:	Status = ’released’

Der PDM-Benutzer Michael Schmidt darf Zusammenbauten nur anzeigen, falls das jeweilige Status-Attribut den Wert ’released’ enthält, d. h. der Zusammenbau ist freigegeben.

[12] Man beachte, dass die Regel selbst dennoch auf einem *Objekttyp* definiert wird! Andere Ansätze, z. B. in [Kel90], definieren Zugriffsregeln nur auf Instanz-Ebene.

(2) *Benutzer:* Designer
Klasse: Spec
Aktion: Change
Bedingung: Status = 'working' AND Owner=$User

Regel (2) verwendet eine etwas komplexere Bedingung: Sie erlaubt allen Benutzern, die zur Gruppe 'Designer' gehören, Spec-Objekte (das können z. B. mit einer Textverarbeitung geschriebene Spezifikationen sein) zu ändern, solange das Objekt noch nicht freigegeben ist (Status 'working', d. h. in Bearbeitung) und der Benutzer selbst auch Besitzer (Owner) des Objektes ist. Dabei bezeichnet $User den Benutzer, der zur Laufzeit die Änderungsaktion auf dem Objekt durchführen möchte. □

Eine detaillierte Betrachtung der möglichen Bedingungen erfolgt im Abschnitt 3.2.3. Es bleibt an dieser Stelle jedoch bereits festzuhalten, dass für die Auswertung von Objektzugriffsregeln auf die Attributwerte der entsprechenden Instanzen zugegriffen werden muss! Um zu garantieren, dass die Auswertung auf dem jeweils aktuellen Zustand der Instanz erfolgt, wird das Objekt – selbst falls es auf Grund einer zuvor ausgeführten Aktion bereits am PDM-Server vorliegt und beispielsweise beim Benutzer angezeigt wird – nochmals von der PDM-Datenbank angefragt.

Welche Strategie zur Auswertung am effizientesten zu sein verspricht, d. h. ob die Regeln vom PDM-Server oder eventuell schon von der Datenbank ausgewertet werden, in welcher die Objekte liegen, wird im Abschnitt 3.4 diskutiert.

3.2.2.2 Klassenzugriffsregeln

Klassenzugriffsregeln ermöglichen Aktionen, die nicht an bestimmte Instanzen einer Klasse gebunden sind. Dazu zählen insbesondere das Erzeugen neuer Instanzen sowie die Suche nach bereits vorhandenen Instanzen.

Beispiele für Klassenzugriffsregeln:

(1) *Benutzer:* Michael Schmidt
Klasse: Zusammenbau
Aktion: Create
Bedingung: TRUE

Hier wird dem Benutzer Michael Schmidt erlaubt, neue Objekte des Typs Zusammenbau zu erzeugen. Man bemerke, dass die Bedingung immer wahr ist und sich nicht auf Attributwerte einer Objektinstanz stützt.

In der Praxis nicht besonders relevant, aber dennoch denkbar wäre, dass der Benutzer nur Objekte mit bestimmten Eigenschaften erzeugen darf. Die Erfüllung

dieser Eigenschaften könnte dann in der Bedingung geprüft werden. Für solche Fälle werden in der Praxis jedoch andere Mechanismen eingesetzt: Müssen beispielsweise die Teile-Nummern einem vorgegebenen Schema entsprechen, so sorgt ein Teile-Nummern-Generator für die Einhaltung dieser Maßgabe.

(2) *Benutzer:* All_Users
Klasse: User
Aktion: Query
Bedingung: TRUE

Diese Regel besagt, dass alle Benutzer der Gruppe 'All_Users' eine Anfrage (Query) auf der Klasse 'User' absetzen dürfen, d. h. die Anwender dürfen nach anderen Benutzern suchen. Ob die Instanzen, die das Ergebnis einer solchen Anfrage bilden, vom anfragenden Benutzer auch angezeigt werden dürfen, entscheiden dann aber wiederum Objektzugriffsregeln für die Aktion 'Display' auf den User-Objekten! □

Aktionen, die den Benutzern über Klassenzugriffsregeln ermöglicht werden, werden typischerweise über Menüpunkte in der graphischen Oberfläche des PDM-Clients angestoßen. Deshalb liegt es nahe, Klassenzugriffsregeln am PDM-Client auszuwerten und Menüpunkte von Aktionen, die nicht freigeschaltet wurden, entsprechend auszublenden. Damit kann schon im Voraus verhindert werden, dass ein Benutzer derartige unerlaubte Aktionen durchführt. Diese Klasse von Regeln spielt daher für Performance-Optimierungen keine Rolle, sie ist hier nur der Vollständigkeit halber aufgeführt. Im Folgenden werden diese Regeln nicht näher betrachtet.

3.2.3 Klassifikation der Bedingungen in Objektzugriffsregeln

3.2.3.1 Row Conditions

Die Bedingungen in den Beispielen aus Abschnitt 3.2.2 fallen in die Kategorie der *row conditions*. Sie beziehen sich ausschließlich auf Attribute der zu betrachtenden Objekt-Instanz. Einzige Ausnahme hiervon bilden die beiden Basis-Bedingungen TRUE und FALSE, die keinen Bezug zur Instanz besitzen, sowie die Variable $User, welche den PDM-Anwender beschreibt, der die Aktion auf dem Objekt durchführen möchte.

Zur Auswertung von row conditions genügt also eine „Zeile“ (row) aus der Datenbank-Tabelle, welche die Instanzen speichert. Darauf können alle gängigen Bedingungen definiert werden, die beispielsweise auch in IF-Abfragen in Programmiersprachen bekannt sind. Row Conditions stellen die mit Abstand am häu-

figsten verwendeten Bedingungen dar und können zur Zugriffsregulierung für alle Benutzeraktionen eingesetzt werden.

Ein erweiterbares „Grundgerüst“ für den strukturellen Aufbau von row conditions ist im Anhang A.1 angegeben.

3.2.3.2 ∃structure Conditions

In manchen Fällen kann es sinnvoll sein, den Zugriff auf eine Objektinstanz davon abhängig zu machen, ob diese Instanz in Beziehung zu einem anderen Objekt steht. Beispielsweise könnte einem Benutzer nur Zugriff (Display-Recht) auf Einzelteile gewährt werden, für die bereits eine Spezifikation vorliegt.

Derartige Bedingungen können in zwei Varianten existieren:

1. ∃ <relation> <related-obj>
2. ∃ <relation> <related-obj> : rowcond

In der ersten Variante ist lediglich zu testen, ob ein <related-obj> in der Beziehung <relation> zur betrachteten Objektinstanz existiert. Bei der zweiten Variante wird zusätzlich geprüft, ob das <related-obj> auch noch die angegebene row condition erfüllt.

Hier ein Beispiel für eine Objektzugriffsregel mit einer ∃structure condition:

Benutzer:	Michael Schmidt
Klasse:	Einzelteil
Aktion:	Display
Bedingung:	∃ attaches spec : status = ’released’

Nicht alle PDM-Systeme unterstützen Regeln mit ∃structure conditions. Typischerweise müssen solche Bedingungen durch „Customizing“ des PDM-Systems quasi hart verdrahtet werden. Soweit es im Folgenden sinnvoll erscheint, werden auch Bedingungen dieser Klasse betrachtet.

3.2.3.3 ∀rows Conditions

Eine ∀rows condition ist eine Bedingung, die nicht nur an der Objektinstanz zu prüfen ist, an welcher die Regel definiert ist, sondern auch an allen Objektinstanzen, die entsprechend der Produktstruktur in das betrachtete Objekt eingehen.

Eingesetzt werden können ∀rows conditions z. B. bei Multi-Level-Expansionen. Ein Benutzer kann beispielsweise nur dann berechtigt sein, einen Multi-Level-Expand auf einem Zusammenbauteil auszuführen, falls alle in diesen Zusammenbau eingebauten Teile den Status 'released' besitzen, also bereits freigegeben sind. Denkbar ist auch, dass die Freigabe eines Zusammenbauteils voraussetzt, dass alle eingebauten Teile selbst freigegeben sind.

∀rows conditions haben folgende Form:

∀ rows : <rowcond>

wobei <rowcond> eine Bedingung entsprechend Abschnitt 3.2.3.1 darstellt.

Ein Beispiel für eine Objektzugriffsregel mit einer ∀rows condition:

Benutzer:	Michael Schmidt
Klasse:	Zusammenbau
Aktion:	Multi-Level-Expand
Bedingung:	∀ rows : status='released'

Bemerkung: Beim Einsatz dieser Bedingungen ist sicherzustellen, dass entweder alle Objekte in der zu betrachtenden Teil-Produktstruktur auf die Bedingung getestet werden können (im Beispiel müssen alle Objekte das Attribut 'status' besitzen), oder dass die Regelauswertung für Fälle, in denen die Bedingung nicht geprüft werden kann, entsprechende Maßnahmen vorsieht, um ein definiertes Ergebnis zu erzielen.

∀rows conditions werden in heutigen PDM-Systemen nicht unterstüzt und sind in der Praxis auch nicht besonders häufig anzutreffen. Im Folgenden wird daher nur bei Bedarf hin und wieder auf Bedingungen dieser Kategorie eingegangen.

3.2.3.4 Tree-Aggregate Conditions

Ähnlich zu den ∀rows conditions bezieht auch diese Klasse von Bedingungen einen Teilgraph der Produktstruktur in die Auswertung ein. Hier sind jedoch nicht an jedem involvierten Objekt gewisse Bedingungen zu prüfen, sondern die Gesamtheit aller Instanzen muss einer Bedingung genügen. Die *Aggregatfunktionen*, die hierfür zum Einsatz kommen können, sind Bildung der Summe (SUM) oder des Durchschnitts (AVG) der Werte eines Attributes, sowie das Zählen (COUNT) der Objektinstanzen oder verschiedenen Werte eines Attributes .

Tree-aggregate conditions können in folgender Form beschrieben werden:

agg_func(attrib-name) relop signed-expr

Hierbei sind für relop ein Vergleichsoperator ($=, <>, <, >, <=, >=$) einzusetzen, und für signed-expr die Produktionen aus dem Anhang A.1 zu verwenden. (Im Falle der Aggregat-Funktion COUNT kann der Parameter 'attrib-name' auch entfallen.)

Ein Beispiel für eine Regel, welche einem Benutzer den Multi-Level-Expand auf Zusammenbauten erlaubt, die aus nicht mehr als 15 weiteren Zusammenbauten und Einzelteilen bestehen, kann demnach wie folgt beschrieben werden:

Benutzer:	Michael Schmidt
Klasse:	Zusammenbau
Aktion:	Multi-Level-Expand
Bedingung:	COUNT() <= 15

Auch diese Klasse von Bedingungen ist in der Praxis nicht von allzu großer Bedeutung. Der Vollständigkeit halber werden sie aber auch im Folgenden hin und wieder in die Betrachtungen mit einbezogen.

3.2.3.5 Auswahl für weitere Betrachtungen

Wie bereits in Abschnit 1.4 aufgezeigt, ist ein Ziel der Arbeit, Performance-Engpässe heutiger PDM-Systeme aufzuzeigen und zu beseitigen. Da als besonders kritische Aktionen die Navigationen in der Produktstruktur ermittelt wurden, liegt der Fokus bei den weiteren Betrachtungen im Wesentlichen auf Bedingungen, die in Struktur-Expansionen eine bedeutende Rolle spielen.

Die vorangegangenen Abschnitte haben vier verschiedene Kategorien von Bedingungen in Objektzugriffsregeln vorgestellt und es wurde bereits angedeutet, dass besonders die row conditions (vgl. Abschnitt 3.2.3.1) von Bedeutung sind. Im Abschnitt 3.5 liegt der Fokus daher besonders auf dieser Kategorie von Bedingungen. $\exists$structure, $\forall$rows und tree aggregate conditions werden nur am Rande betrachtet.

Dies schließt nicht aus, dass außer den Struktur-Expansionen auch andere Aktionen von frühzeitiger Regelauswertung profitieren können. Der zu erwartende Performance-Gewinn ist jedoch eher gering einzuschätzen, und somit erhält die Optimierung dieser Aktionen nicht oberste Priorität.

3.3 Konfigurationssteuerung

Unter der Konfiguration eines Produktes versteht man verschiedene Ausprägungen (vgl. Abschnitt 2.1.1), in welchen ein Produkt hergestellt werden kann. Alle Konfigurationen werden dabei in einer einzigen Produktstruktur zusammengefasst, wobei der Anwender durch Anwahl verschiedener Strukturoptionen und

Gültigkeiten festlegen kann, auf welcher Konfiguration er seine Aktionen auszuführen gedenkt.

Nicht alle Aktionen sind von Konfigurationen betroffen: Die Suche (Query) nach Objekten ist von Konfigurationseinstellungen unabhängig, ebenso das Ein- und Aus-Checken von Einzel- und Zusammenbauteilen.

Interessant wird die Auswahl der Konfiguration bei struktur-orientierten Aktionen, wie beim Multi-Level-Expand. Wird ein Zusammenbau expandiert, so wird für jedes enthaltene Unterteil getestet, ob dieses in der ausgewählten Konfiguration überhaupt Verwendung findet.

Eine adäquate Modellierung von Strukturoptionen und Gültigkeiten ist in Abbildung 3.1 skizziert. Die Abbildung zeigt, dass einem Kanten-Objekt (z.B. einem Objekt aus der *uses*-Beziehung, vgl. Fußnote 7 auf Seite 40) mehrere Strukturoptionen und Gültigkeiten zugewiesen werden können.

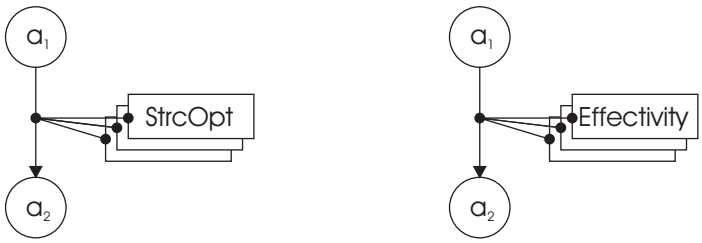


Abbildung 3.1: Konfiguration mittels Strukturoptionen und Gültigkeiten

Offensichtlich lässt sich der Test, ob ein Objekt in einer ausgewählten Konfiguration Verwendung findet, auf das Testen einer row condition zurückführen: Falls das mengenwertige Attribut 'StrcOpts' des Kanten-Objekts zwischen a_1 und a_2 (vgl. Abbildung 3.1) mindestens eine Strukturoption enthält, die auch der PDM-Anwender in seine Konfiguration eingeschlossen hat, so gehört a_2 in den Zusammenbau a_1. Das gleiche gilt, falls der Benutzer *keine* Strukturoptionen gewählt hat[13], oder der Kante keine Strukturoptionen zugewiesen sind. Sind in 'StrcOpts' jedoch nur Strukturoptionen enthalten, die der Anwender nicht gewählt hat, so gehört a_2 nicht zur gewählten Konfiguration.

Hier ein Beispiel für eine Regel, die mittels einer row condition die Konfigurationssteuerung für Strukturoptionen ermöglicht:

[13]PDM-Systeme nehmen hier typischerweise implizit an, dass *alle* möglichen Konfigurationen angezeigt werden sollen!

Benutzer:	All_Users
Klasse:	uses
Aktion:	navigate
Bedingung:	StrcOpts IS EMPTY OR $UsrConfig IS EMPTY OR StrcOpts INTERSECTS $UsrConfig

Die Regel soll für alle Benutzer gelten und auf die Klasse 'uses' wirken, welche die Struktur-Beziehungen zwischen Zusammenbauten und Einzelteilen enthalten soll. Die „virtuelle Aktion" 'navigate', die nicht explizit vom Benutzer ausgeführt werden kann, wird implizit beim Navigieren in der Produktstruktur durchgeführt.

Bei der Auswertung einer solchen Regel ist eine Ausnahme zu beachten: Sind einem Kanten-Objekt *keine* Strukturoptionen zugeordnet, so gehört diese Kante – und damit auch die Komponente, auf die die Kante verweist – zu *jeder* Konfiguration!

Die Aussagen, die hier für die Strukturoptionen gemacht wurden, gelten sinngemäß auch für die Zuordnung von Gültigkeiten (Effectivities). Es ist lediglich zu berücksichtigen, dass Gültigkeiten einen *Zeitraum* definieren (von – bis), Benutzer aber einen *Zeitpunkt* festlegen, für den z. B. die Produktstruktur expandiert werden soll. Die Bedingung für den Test auf gültige Beziehungen lautet daher wie folgt:

```
$UsrEff BETWEEN ANY Effectivities.Start AND Effectivities.End
```

Der Ausdruck 'BETWEEN ANY' bedeutet, dass der vom Anwender vorgegebene Zeitpunkt $UsrEff zwischen 'Start' und 'End' mindestens einer Gültigkeit aus dem mengenwertigen Attribut 'Effectivities' liegen muss.

3.4 Strategien zur Zugriffsregelauswertung

3.4.1 Szenario

In den folgenden Abschnitten wird diskutiert, welche Alternativen hinsichtlich der Auswertung für Zugriffsrechte, wie sie in Abschnitt 3.2.2 eingeführt wurden, existieren und wie sie zu bewerten sind. Ausgangslage ist dabei ein Szenario, wie es in Abbildung 3.2 dargestellt ist.

Ein Benutzer führt über seinen graphischen PDM-Client eine Aktion aus, die Objekte aus der entfernten Datenbank anfordert (z. B. Expansion der Produktstruktur oder Suche nach Bauteilen etc.). Die Aktion wird an den PDM-Server geleitet ①, welcher eine der Aktion entsprechende Anfrage an die Datenbank stellt ②. Von

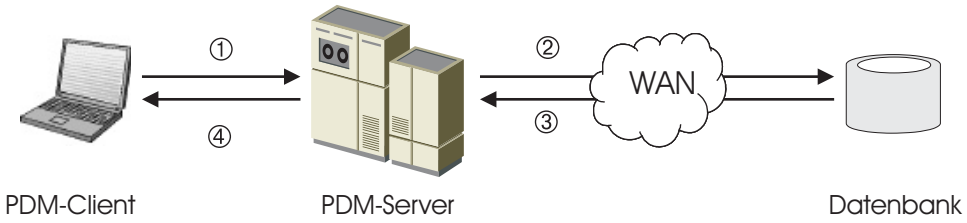


Abbildung 3.2: Zugriff auf entfernte Datenbank von einem PDM-Client aus

dort wird das Anfrage-Ergebnis über den PDM-Server ③ an den PDM-Client zurückgegeben ④.

Es gilt nun zu klären, welche Zugriffsregeln für eine derartige Aktion zu berücksichtigen sind, und an welcher Stelle in diesem Ablauf deren Auswertung erfolgen sollte, um die Anfrage möglichst performant durchführen zu können.

3.4.2 Bestimmung auszuwertender Zugriffsregeln

Bei der Ausführung einer Benutzeraktion sind alle Zugriffsregeln zu berücksichtigen, die auf den ausführenden Benutzer, die auszuführende Aktion sowie den betroffenen Objekt-Typ zutreffen. Für die folgenden Betrachtungen seien die vier „Attribute“ Benutzer (oder Gruppe), Objekt-Typ, Aktion und Bedingung, die eine Zugriffsregel definieren, als 4-Tupel in einer „Zugriffsregel-Tabelle“ abgelegt.

Regeln, die einen konkreten Benutzer spezifizieren, können sehr leicht aufgefunden werden: Der Zugriff auf die Tabelle erfolgt über Benutzer, Objekt-Typ und Aktion, die zugehörigen Bedingungen müssen dann von den betroffenen Instanzen erfüllt werden.

Um auch Regeln zu finden, die als Benutzer eine Benutzergruppe spezifizieren, muss die Gruppenzugehörigkeit des ausführenden Benutzers geprüft werden. Abbildung 3.3 zeigt exemplarisch eine Gruppenhierarchie.

Es ist nicht zwingend nötig, dass genau ein oberstes Element in der Hierarchie existiert. Prinzipiell kann auch jede Gruppe für sich allein stehen, die Hierarchie also aufgelöst sein. In umfangreichen Systemumgebungen allerdings ist eine hierarchische Organisation auf Grund des geringeren Administrationsaufwandes empfehlenswert.

Benutzer können mehreren Gruppen zugeordnet sein, auch können Gruppen wiederum „Mitglied“ in mehreren anderen Gruppen sein. Zyklische Beziehungen zwischen den Gruppen sind jedoch ausgeschlossen. Aus der Abbildung 3.3 ist

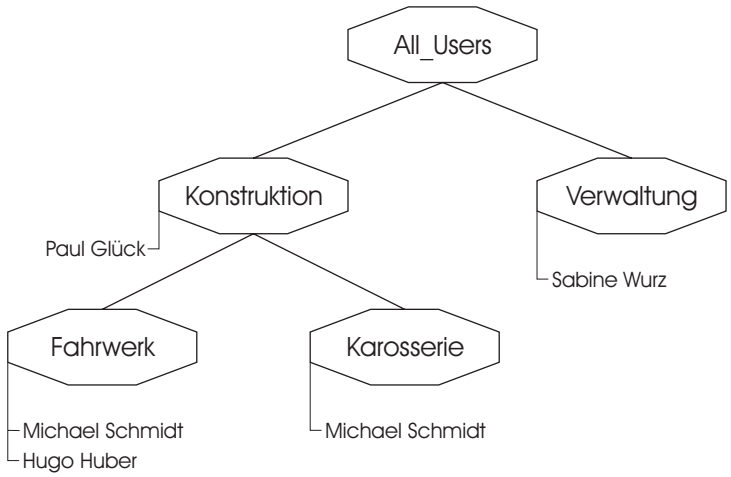


Abbildung 3.3: Benutzergruppen-Hierarchie

beispielsweise ersichtlich, dass der Anwender „Michael Schmidt“ zu den Gruppen ’Fahrwerk’, ’Karosserie’, ’Konstruktion’ und ’All_Users’ gehört.

Bezeichnet ’is_in_group’ die Relation zwischen Benutzern und Gruppen sowie zwischen Gruppen selbst („von unten nach oben“ gelesen), so gilt es bei der Bestimmung der für eine Aktion auszuwertenden Zugriffsregeln diese Relation *rekursiv* zu traversieren, beginnend bei dem ausführenden Benutzer. Da die Benutzer/Gruppen-Zuordnung sowie die Gruppen-Hierarchie in der Datenbank abgelegt sind, bietet sich zur Auflösung der Gruppenzugehörigkeit die Verwendung von *rekursivem SQL* an. Unterstützt die verwendete Datenbank dies nicht, so muss die Rekursion „von Hand“ durchgeführt werden. Näheres zu rekursivem SQL findet sich in Abschnitt 4.3.1 oder in [ANS99a, MS02].

Sind alle Gruppen ermittelt, denen der anfragende Benutzer zugeordnet ist, so können aus der Zugriffsregel-Tabelle nun auch die Regeln ermittelt werden, die sich auf Benutzergruppen beziehen und für den ausführenden Benutzer hinsichtlich Aktion und Objekt-Typ in Betracht kommen. Anschließend kann die Regelauswertung nach einer der nachfolgend beschriebenen Strategien durchgeführt werden.

3.4.3 Regelauswertung am Standort des Anfragers

Die beiden folgenden Auswertungsstrategien evaluieren die Zugriffsregeln *nach* der Übertragung der angefragten Daten von der Datenbank zum PDM-Server.

3.4.3.1 Regelauswertung am PDM-Client

Wie bereits in Abschnitt 3.2.2.2 angemerkt, lassen sich *Klassenzugriffsregeln* am Client überprüfen und präventiv entsprechende Menüpunkte ausblenden. Damit können unerlaubte Aktionen bereits vor deren Initiierung verhindert werden.

Für die Auswertung der *Objektzugriffsregeln* eignet sich der Client hingegen nicht. Der Client ist das letzte Glied in der Anfrage/Ergebnis-Kette, d. h. sämtliche Objekte, welche die Datenbank im Ergebnis zurückliefert, durchlaufen den kompletten Weg vom Datenbank-Server bis hin zum Client, wo sie gegebenenfalls nicht visualisiert werden dürfen und deshalb aus dem Ergebnis entfernt werden müssen. Da der Client hinsichtlich der Regelauswertung auch nicht mehr Informationen besitzt als der PDM-Server, besteht kein Grund, die Auswertung derart verzögert durchzuführen.

3.4.3.2 Regelauswertung am PDM-Server

Bei heute erhältlichen PDM-Systemen übernimmt der PDM-Server die Auswertung von Objektzugriffsregeln. Hier liegen alle Informationen vor, die zur Regelauswertung benötigt werden. Selbst falls sich die Objekte, die das PDM-System verarbeitet, von den Objekten bzw. Tupeln in der Datenbank unterscheiden – z. B. durch Hinzufügen transienter Attribute oder Decoding codierter Attribute durch den Object Wrapper, vgl. Abbildung 2.6 –, kann der PDM-Server unter Verwendung des Rule Evaluators die Regeln auswerten.

Diese Strategie bringt jedoch in verteilten Umgebungen die bereits angesprochenen Performance-Probleme mit sich, da die Daten vor der Regelauswertung über das WAN übertragen werden müssen. Eine noch frühere Evaluierung der Regeln ist daher anzustreben.

Anmerkung:
Eine andere Art von Regeln, die bisher nicht betrachtet wurden und im Weiteren auch für die vorliegende Arbeit nicht von Bedeutung sind, sind sogenannte *Baubarkeitsregeln* oder *-bedingungen*. Diese legen fest, welche Kombinationen von verschiedenen Strukturoptionen gültig sind. Beispielsweise kann eine Baubarkeitsregel ausschließen, dass ein als Cabriolet konfigurierter PKW mit einem Schiebedach ausgestattet wird. Derartige Regeln sind von konkreten Objekt-Instanzen unabhängig und können deshalb schon während der Definition einer Konfiguration am PDM-Server überprüft werden. Falls beim Design eines PDM-Systems kein absolut minimaler Client gefordert wird, können diese Tests auch in den Client integriert werden.

3.4.4 Regelauswertung am Standort der Daten

Ziel der beiden im Folgenden beschriebenen Strategien ist die Vermeidung von unnötiger Datenübertragung über das WAN. Dazu werden die Zugriffsregeln bereits vor der Datenübermittlung ausgewertet.

3.4.4.1 Regelauswertung nahe der PDM-Datenbank

In diesem Verfahren werden die im Ergebnis der Datenbankanfrage enthaltenen Daten von einem lokal erreichbaren Regelauswerter auf Zugriffsrechte überprüft.

In einem homogenen Umfeld, in welchem an allen Standorten das gleiche PDM-System vorhanden ist, kann hierzu der Regelauswerter der lokal verfügbaren System-Instanz verwendet werden. Ebenso wird der lokale Object Wrapper eingesetzt, der die Datenbank-Tupel auf die gleiche Weise in PDM-System-Objekte umwandelt wie am Standort des Anfragers, so dass Zugriffsrechte äquivalent ausgewertet werden können.

Sind für diese Umwandlung z. B. Sitzungs-Informationen des Anwenders nötig, so müssen diese Informationen zusammen mit der Anfrage an den Standort der PDM-Datenbank übergeben werden. Gleiches gilt auch für Informationen, die zur Auswertung der Zugriffsregeln benötigt werden, so z. B. der Name oder die ID des anfragenden Benutzers sowie – für Expansionsvorgänge – die gewählten Konfigurationseinstellungen.

Im Vergleich zu den im Abschnitt 3.4.3 angesprochenen Strategien erzielt dieses Verfahren schon beachtliche Einsparungen: Es werden lediglich die für den Benutzer sichtbaren Objekte über das WAN transportiert, und der dafür nötige Aufwand hält sich in Grenzen: User- und Sitzungs-Informationen sind typischerweise nur wenige hundert Bytes groß, können also oftmals in einem einzigen Paket zusammen mit der Anfrage über das Netz übertragen werden.

Einziger Nachteil dieses Verfahrens ist, dass zunächst die Datenbank zur Ermittlung des Ergebnisses sämtliche Tupel anfasst, und unmittelbar darauf das Ergebnis vom Regelauswerter noch eimal komplett traversiert und auf Zugriffsberechtigungen überprüft werden muss. Das im folgenden Abschnitt (3.4.4.2) vorgestellte Verfahren wird dieses „Manko“ beheben, indem diese beiden Schritte zu einem einzigen zusammengefasst werden.

3.4.4.2 Regelauswertung durch das Datenbankmanagementsystem

Relationale Datenbankmanagementsysteme besitzen reichhaltige Möglichkeiten, um Selektionen auf Datenbeständen durchzuführen. Deshalb liegt es nahe, diese

Möglichkeiten zu nutzen, um Zugriffsrechte schon beim Ermitteln eines Anfrageergebnisses auszuwerten. Die verschiedenen Alternativen, die sich hierfür anbieten, werden in Abschnitt 3.5 beschrieben.

Es ist unmittelbar klar, dass die Anfragen, welche eine integrierte Regelauswertung vornehmen, komplexer sein werden als die ursprünglichen, primitiven Anfragen. Des Weiteren muss die Auswertung von Regeln, die ursprünglich auf PDM-System-Objekten definiert waren, auf Datenbankobjekten stattfinden. Es muss also eine Abbildung der Regel-Bedingungen und der in diesen Bedingungen referenzierten Attribute auf Funktionen und Attribute der Datenbank vorgenommen werden.

Die möglichen Problemfälle, deren Lösung von der jeweiligen Darstellung der Bedingungen abhängt (vgl. Abschnitt 3.5), sind im Einzelnen:

- Die Regel-Bedingungen können Aufrufe von Funktionen enthalten, die dem Datenbankmanagementsystem nicht bekannt sind.
- Attribute der PDM-System-Objekte können anders benannt sein als die Attribute der zugrunde liegenden Datenbanktabellen.
- Attributwerte können beim Umwandeln der Tupel in PDM-System-Objekte geändert werden.
- PDM-System-Objekte können mehr Attribute besitzen, als die zugrunde liegenden Datenbanktabellen. Die Werte dieser *transienten* Attribute werden im Object Wrapper berechnet und liegen folglich an der Datenbank nicht vor.

Werden diese Herausforderungen hinreichend gelöst, so ist die Regelauswertung durch die Datenbank der ideale Weg, um das durch Benutzeraktionen auftretende Datenvolumen möglichst frühzeitig einzugrenzen und damit zu akzeptableren Antwortzeiten beizutragen.

3.5 Darstellung und Integration in SQL-Anfragen

3.5.1 Datenbankspezifische Zugriffskontrolle

Datenbankmanagementsysteme bieten Zugriffskontrollmechanismen an, mit welchen sich verschiedene Benutzer gegenseitig Rechte auf Daten einräumen können. Es gilt zu klären, ob diese Mechanismen ausreichen, um die PDM-systemspezifischen Regeln abzubilden.

Die Vergabe von Rechten erfolgt mittels GRANT-Anweisung [MS93]. Hiermit können jedoch nur „elementare“ Rechte wie Lesen, Schreiben oder Einfügen auf *Datenbanktabellen* vergeben werden. Eine Steuerung auf Instanz-Ebene, wie sie für PDM-Systeme benötigt wird, ist ohne Umwege (Views o. ä., siehe auch Abschnitt 3.5.5) nicht möglich.

Auch aus der Sicht der PDM-Systemarchitektur ist die Verwendung datenbankspezifischer Zugriffskontrollmechanismen nicht unbedenklich. Jeder PDM-Benutzer muss auf einen eigenen Datenbank-Benutzer abgebildet werden. Für das PDM-System bedeutet dies aber, dass für jeden aktiven Anwender eine separate Datenbankverbindung offengehalten werden muss – der Overhead für die Datenbank-Kommunikation steigt damit enorm an. Aus Effizienzgründen ist eine PDM-Systemarchitektur zu bevorzugen, welche nur wenige Datenbank-Verbindungen offen hält, die sich die Anwender teilen.

Die Verwendung von datenbankspezifischen Mechanismen zur Zugriffskontrolle scheidet folglich aus.

3.5.2 Zugriffskontrolllisten

3.5.2.1 Prinzipieller Einsatz in PDM-Systemen

In vielen Informationssystemen werden Zugriffskontrolllisten (Access Control Lists, kurz ACL) zur Steuerung der Zugriffsberechtigung eingesetzt. In UNIX-Betriebssystemen etwa definieren sie, welche Benutzer[14] Lese-, Schreib- oder Ausführungsrecht auf den Dateien besitzen.

Prinzipiell bestehen mehrere Möglichkeiten, um ACLs auch in PDM-Systemen einzusetzen:

Entsprechend Beispiel (1) in Abschnitt 3.2.2.1 darf der Benutzer Michael Schmidt alle Zusammenbauten sehen (Aktion ’Display’), deren Status auf ’released’ gesetzt ist. Nun lässt sich die Menge aller Zusammenbauten bestimmen, die der Benutzer sehen darf, indem einmalig für jeden Zusammenbau der Status geprüft wird und die Objekt-IDs der freigegebenen Zusammenbauten in einer ACL festgehalten werden. Abbildung 3.4 zeigt eine solche ACL für Zusammenbauten, wobei mehrere Benutzer und Aktionen berücksichtigt werden.

Erweitert man nun das skizzierte Verfahren auf alle verfügbaren Objekttypen, so erhält man pro Objekttyp eine ACL nach dem Muster aus Abbildung 3.4. Da

[14] Die Granularität der Steuerung ist hier beschränkt auf a) den Eigentümer der Datei, b) dessen Gruppe, oder c) die Gesamtheit aller registrierter Benutzer

Zusammenbauten	Michael Schmidt	Karl Maier	Hugo Huber	...
display	4711,3132,0815	0815	3132,0815	...
checkout	3132,0815	0815	3132	...
...	...	...	...	...

Abbildung 3.4: Access Control List für Zusammenbauten; Zugriff über Aktion und Benutzer

Objekt-IDs sinnvollerweise eindeutig über das gesamte System sind – und damit eindeutig über alle Objekttypen hinweg – bietet sich die Zusammenfassung aller Objekttyp-spezifischen ACLs zu einer einzigen ACL unmittelbar an.

Zu beachten ist, dass Regeln, die auf Benutzergruppen definiert sind, für jeden Benutzer in dieser Gruppe gesondert ausgewertet und pro Benutzer in die ACL aufgenommen werden müssen.

Eine alternative ACL ist in Abbildung 3.5 dargestellt. Hier werden für jeden Benutzer und jede Objekt-Instanz die Aktionen festgehalten, die auf der Instanz durchgeführt werden dürfen.

	Michael Schmidt	Karl Maier	Hugo Huber	...
4711	display	-	-	...
3132	display, checkout	-	display, checkout	...
0815	display, checkout	display, checkout	display	...
...	...	...	...	...

Abbildung 3.5: Access Control List, Zugriff über Objekt-Instanz und Benutzer

Eine dritte Alternative speichert die Menge der Benutzer, die eine Aktion auf einem Objekt ausführen dürfen (vgl. Abbildung 3.6, aus Platzgründen sind die Namen gekürzt).

	4711	3132	0815	...
display	Schmidt	Schmidt, Huber	Schmidt, Maier, Huber	...
checkout	-	Schmidt, Huber	Schmidt, Maier	...
...	...	...	...	...

Abbildung 3.6: Access Control List, Zugriff über Objekt-Instanz und Aktion

Allen drei ACL-Varianten gemein ist, dass eine unmittelbare Abbildung auf SQL-Tabellen nicht möglich ist: Der zur Zeit gültige Standard SQL:1999 [ANS99a,

EM99] unterstützt keine listenwertigen Attribute[15], weshalb die Abbildung auf relationale Tabellen einem „Flachklopfen“ der Listenstruktur gleichkommt. Das Ergebnis der Abbildung aller vorgestellen drei Varianten von ACLs ist folglich identisch. Abbildung 3.7 zeigt die Definition dieser Relation in SQL sowie die Ausprägung aus dem Beispiel.

```
CREATE TABLE acl (
user   VARCHAR(..),
action VARCHAR(..),
obid   VARCHAR(..))
```

user	action	obid
Michael Schmidt	display	0815
Michael Schmidt	display	3132
Michael Schmidt	display	4711
Michael Schmidt	checkout	0815
Michael Schmidt	checkout	3132
Karl Maier	display	0815
Karl Maier	checkout	0815
Hugo Huber	display	0815
Hugo Huber	display	3132
Hugo Huber	checkout	3132

Abbildung 3.7: ACL als relationale Tabelle; Definition und Ausprägung

Um diese ACL nun in Benutzeraktionen des PDM-Systems nutzen zu können, müssen Anfragen an die Datenbank entsprechend erweitert werden. Möchte beispielsweise der Benutzer Michael Schmidt im Rahmen einer ’Display’-Aktion auf Zusammenbauten zugreifen, so ist folgende Anfrage-Transformation nötig:

original: SELECT * FROM assembly

modifiziert: SELECT * FROM assembly WHERE obid IN
(SELECT obid FROM acl WHERE user=’Michael Schmidt’
AND action=’display’)

Hinweis: Es gibt mehrere Alternativen, mittels SQL das gleiche Ergebnis zu erzielen. Beispielsweise könnte auch ein JOIN zwischen den Relationen ’assembly’ und ’acl’ über die Objekt-ID berechnet werden, wobei zusätzlich die Projektion auf Attribute der Relation ’assembly’ durchgeführt werden muss. Für die folgenden Betrachtungen spielt die Wahl der SQL-Formulierung jedoch keine Rolle.

3.5.2.2 Aktualisierung der ACL

Änderungen an Objekten, Benutzern und Regeln erfordern Anpassungen der ACL, wobei der dazu treibende Aufwand vom jeweiligen Änderungs-Szenario abhängt.

[15] Die verfügbaren Arrays sind wegen der festen Obergrenze nicht ausreichend.

Das Hinzufügen eines neuen Benutzers stellt keine Herausforderung dar, da für ihn per Definition keine Rechte vorhanden und somit keine Änderungen der ACL nötig sind (vgl. Abschnitt 3.2.1).

Das Löschen eines Benutzers zieht lediglich das Löschen sämtlicher ACL-Einträge nach sich, welche diesen Benutzer referenzieren.

Beim Hinzufügen einer neuen Regel für einen Benutzer (oder eine Gruppe) müssen sämtliche Instanzen des von der Regel referenzierten Objekttyps überprüft und gegebenenfalls neue Einträge in die ACL geschrieben werden.

Das Entfernen einer Regel gestaltet sich deutlich komplexer. Es stellt sich die Frage, welche Einträge aus der ACL zu entfernen sind. Die ACL aus Abbildung 3.7 lässt keine Rückschlüsse zu, welche Regel welchen Eintrag verursacht hat. Somit besteht die einzige Möglichkeit darin, sämtliche Einträge zu löschen, die sich auf den gleichen Benutzer (Vorsicht bei Gruppen!) und die gleiche Aktion beziehen, wie die zu löschende Regel. Da hiermit möglicherweise zu viele Einträge gelöscht wurden, müssen anschliessend mit den noch vorhandenen Regeln zu den selben Benutzern und Aktionen die Einträge der ACL neu berechnet werden. In der Praxis ist dies auf Grund des zu hohen Aufwandes nicht praktikabel.

Alternativ hierzu könnte man die ACL um ein Attribut erweitern, welches die Regel referenziert, die den jeweiligen ACL-Eintrag verursacht hat. Damit kann beim Löschen einer Regel direkt auf die betroffenen ACL-Einträge zugegriffen werden. Nachteilig ist jedoch, dass ACL-Einträge, die von mehreren Regeln erzeugt werden, dann auch entsprechend mehrfach in der ACL enthalten sind.

Die Änderung von Werten der Objekte stellt ein weiteres Problem dar. Eine derartige Änderung kann dazu führen, dass manchen Benutzern die Zugriffsberechtigung auf das Objekt verloren geht und gleichzeitig anderen Benutzern auf dieses Objekt Rechte gewährt werden. Deshalb müssen zum einen alle bereits existierenden Einträge der ACL, die ein Recht auf diesem Objekt gewähren, daraufhin überprüft werden, ob diese Rechte nach der Objekt-Änderung noch immer gültig sind, zum andern müssen alle Regeln, die sich auf den Typ des geänderten Objektes beziehen, evaluiert werden, um gegebenenfalls neue ACL-Einträge generieren zu können.

3.5.2.3 Fazit

Zugriffskontrolllisten stellen „materialisierte“ Regeln dar. Die Abhängigkeit der Zugriffsrechte vom Zustand der Objekte führt dazu, dass häufige Anpassungen der ACL notwendig sind, auch wenn sich die zugrunde liegenden Regeln *nicht* geändert haben. Eine performante Nutzung kann deshalb nicht erzielt werden. Diese Aussage gilt unabhängig von der möglichen Datenbankdarstellung der ACL, d. h.

auch wenn statt der „flachgeklopften“ Darstellung aus Abbildung 3.7 mengenwertige Attribute unterstützt würden, sind Anpassungen von ACLs bei Attributänderungen extrem teuer.

Des Weiteren enthalten ACLs Einträge für jede zulässige Kombination aus Benutzer, Aktion und Objekt-Instanz. In der Praxis werden die Anwender jedoch nur mit einer Teilmenge aller Objekte arbeiten, auf die sie prinzipiell Zugriffsrechte besitzen. Das Vorhalten sämtlicher Rechte ist folglich nicht besonders ökonomisch.

ACLs können aber auch einen Pluspunkt verbuchen. Da die Erzeugung der ACLs über die Auswertung der Regeln durch das PDM-System und nicht durch das Datenbankmanagementsystem erfolgt, können hierfür sämtliche Mechanismen heutiger PDM-Systeme eingesetzt werden. Im Einzelnen bedeutet dies, dass transiente Attribute, im Vergleich zu Datenbanktupeln geänderte Attributwerte oder Attributnamen, sowie Funktionsaufrufe innerhalb der Bedingungen für diese Art der Regelauswertung kein Problem darstellen.

Insgesamt jedoch muss die Leistungsfähigkeit von ACLs negativ bewertet werden, als Darstellungsform der Zugriffsregeln für PDM-Systeme scheiden sie daher aus.

3.5.3 Parameter-Tabellen

3.5.3.1 Prinzipieller Einsatz in PDM-Systemen

Betrachtet man Zugriffsregeln einmal etwas genauer, so kann man feststellen, dass sich manche Regeln quasi wiederholen und sich nur geringfügig voneinander unterscheiden. So wird einem Benutzer beispielsweise erlaubt, die Aktion ’Display’ auf Zusammenbauten auszuführen, falls der Status des Objektes auf ’released’ gesetzt ist. Ein anderer Benutzer hingegen darf dieselbe Aktion ausführen, falls der Status auf ’working’ steht usw. Es liegt also nahe, Gruppen von Bedingungen zu bilden, wobei sich Bedingungen einer Gruppe auf den gleichen Objekttyp und die gleichen Attribute beziehen, und sich nur durch die Werte in gewissen „Parametern“ unterscheiden. Abbildung 3.8 zeigt zwei Regeln, deren Bedingungen zur selben Gruppe gehören.

Benutzer:	Michael Schmidt	*Benutzer:*	Hugo Huber
Klasse:	Zusammenbau	*Klasse:*	Zusammenbau
Aktion:	Display	*Aktion:*	CheckIn
Bedingung:	Status = ’released’	*Bedingung:*	Status = ’working’

Abbildung 3.8: Zwei Regeln mit Bedingungen der gleichen Gruppe

Die Regeln unterscheiden sich zwar im Benutzer und in der Aktion, nicht aber im Objekttyp und im Aufbau der Bedingungen.

Die Belegung der Parameter lässt sich für jeden Benutzer, auf den die Regel zutrifft, zusammen mit der jeweiligen Aktion in einer *Parameter-Tabelle* festhalten. Für das obige Beispiel mit der Bedingung Status = '...' ergibt sich die Tabelle aus Abbildung 3.9.

```
CREATE TABLE check_status (
user   VARCHAR(..),
action VARCHAR(..),
status VARCHAR(..))
```

user	action	status
Michael Schmidt	Display	released
Hugo Huber	CheckIn	working
...	...	...

Abbildung 3.9: Parameter-Tabelle für eine Gruppe von Bedingungen an Zusammenbauten; Definition und Ausprägung

Zu jeder Gruppe – und damit auch zur korrespondierenden Tabelle – wird eine Vorschrift benötigt, die besagt, wie die Parameter beim Testen der Zugriffsbedingung einzusetzen sind. Sinnvollerweise wird dies eine SQL-WHERE-Klausel sein, die lediglich in eine originale Anfrage eingebaut werden muss. Für die Gruppe aus Abbildung 3.9 lautet die Bedingung:

```
assembly.status = check_status.status
AND check_status.user=$User
AND check_status.action=$Action
```

Zur Laufzeit, d. h. bei der Modifikation der originalen Anfrage, muss $User durch den anfragenden Benutzer ersetzt und der Kontext der Anfrage in $Action eingesetzt werden.

Diesem Muster entsprechend muss für eine Anfrage – analog zu Abschnitt 3.5.2.1 – nach Zusammenbauten im Kontext einer 'Display'-Aktion für den Benutzer Michael Schmidt die folgende Anfrage-Modifikation durchgeführt werden:

original:
```
SELECT * FROM assembly
```

modifiziert:
```
SELECT assembly.* FROM assembly JOIN check_status
WHERE assembly.status = check_status.status
AND check_status.user='Michael Schmidt'
AND check_status.action='Display'
```

Anmerkung: Auf Grund des Fazits in Abschnitt 3.5.3.2 wird an dieser Stelle bewusst auf die Behandlung transienter Attribute, Funktionen, Attribut-Umbenennungen und -Wertumwandlungen verzichtet. Eine Berücksichtigung dieser Spezialfälle bei der Erzeugung der Verwendungs-Vorschrift kann analog zu dem im Abschnitt 3.5.4.2 beschriebenen Verfahren erfolgen.

3.5.3.2 Fazit

Auf den ersten Blick mag dieser Ansatz recht flexibel erscheinen. Eine neue Regel, deren Bedingung in eine bereits vorhandene Gruppe fällt, ist in der jeweiligen Tabelle leicht hinzuzufügen, und auch das Löschen von Regeln kann relativ einfach durch Entfernen der Einträge aus der zugehörigen Tabelle durchgeführt werden.

Problematischer allerdings gestaltet sich das Hinzufügen einer Regel, für die noch keine Gruppe existiert. Das Erzeugen einer neuen Gruppe bedingt auch das Erstellen einer neuen Parameter-Tabelle mit der zugehörigen Verwendungs-Vorschrift. Der Query-Modifikator muss über die hinzugekommene Parameter-Tabelle informiert werden und diese bei der Anfrage-Modifikation berücksichtigen.[16]

Für jeden Objekttyp können folglich viele Parameter-Tabellen benötigt werden, denn die Möglichkeiten, Bedingungen auf den Attributen zu formulieren, sind sehr umfangreich.

Damit verhält sich dieses Verfahren deutlich komplexer als es vielleicht zunächst erscheinen mag. Es ist auch zu erkennen, dass für bisher einfache Anfragen nun Anfragen mit mehreren Joins generiert werden müssen, wobei die Anzahl der benötigten Joins von der Anzahl der Parameter-Tabellen und damit den Gruppen abhängt.

In Anbetracht dieser Probleme muss vom Einsatz der Parameter-Tabellen zur Steuerung der Zugriffsberechtigungen in PDM-Systemen abgeraten werden.

3.5.4 WHERE-Klauseln

3.5.4.1 Prinzipieller Einsatz in PDM-Systemen

Die in den Abschnitten 3.2.3.1 und 3.2.3.2 eingeführten *row conditions* und $\exists$*structure conditions* bilden in ihrer Ausdrucksmächtigkeit offensichtlich eine

[16] Die Verwendungs-Vorschriften der verschiedenen Parameter-Tabellen sind dabei mittels OR-Operator zu verknüpfen.

Teilmenge aller gültigen Bedingungen in SQL-WHERE-Klauseln. Es liegt daher nahe, diese Bedingungen direkt in äquivalente SQL-konforme Bedingungen umzuwandeln und diese beim Zugriff auf die Tabellen im SELECT-FROM-WHERE-Statement integriert zu überprüfen.

Als Beispiel diene wieder der Benutzer Michael Schmidt, der auf alle Zusammenbauten mit dem Status 'released' die Aktion 'Display' ausführen darf (vgl. Beispiel (1) im Abschnitt 3.2.2.1). Die Bedingung lautet hier einfach:

```
Status = 'released'
```

Fragt der Anwender nun alle Zusammenbauten von der Datenbank an, so muss das ursprüngliche Select-Statement um eine entsprechende WHERE-Klausel erweitert werden:

original:
```
SELECT * FROM assembly
```

modifiziert:
```
SELECT * FROM assembly
WHERE Status = 'released'
```

Schränkt der Anwender die Suche bereits durch Suchkriterien ein, z. B. indem er einen konkreten Owner angibt, so findet lediglich eine Erweiterung der bestehenden WHERE-Klausel statt:

original:
```
SELECT * FROM assembly
WHERE owner = 'Michael Schmidt'
```

modifiziert:
```
SELECT * FROM assembly
WHERE owner = 'Michael Schmidt'
AND Status = 'released'
```

Die Details über die nötige Transformation der Bedingungen, insbesondere das Handling transienter Attribute und diverser Umbenennungen, sowie die durchzuführende Anfrage-Modifikation können Abschnitt 3.5.4.2 entnommen werden.

3.5.4.2 Bedingungs-Transformation und Anfrage-Modifikation

In der Regel ist es nicht möglich, eine Regel-Bedingung einfach an die bestehende WHERE-Klausel einer Anfrage hinzuzufügen (oder eine neue WHERE-Klausel zu definieren). Gründe hierfür sind, wie bereits in Abschnitt 3.4.4.2 aufgezeigt, Attribut- und Objekt-Umbenennungen, Wertänderungen, Funktionsaufrufe sowie

transiente Attribute, die der Object Wrapper bei der Umwandlung der Datenbank-Tupel in PDM-System-Objekte durchführt bzw. erzeugt und folglich der Datenbank nicht bekannt sind. Für diese Fälle sind demnach Transformationen nötig, um eine korrekte Evaluierung der Regeln durch die Datenbank zu ermöglichen.

In dem Beispiel in Abschnitt 3.5.4.1 wirkt sich positiv aus, dass für das Objekt-Attribut 'Status' eine Entsprechung gleichen Namens in der Datenbanktabelle 'assembly' existiert, und SQL case-insensitive arbeitet, d. h. der SQL-Parser unterscheidet nicht zwischen Groß- und Kleinschreibung der Schlüsselworte, Tabellen- und Attributnamen. Nur deshalb konnte das Objekt-Attribut 'Status' problemlos auf das – wie auch immer geschriebene – status-Attribut der Datenbank-Tabelle 'assembly' abgebildet werden (vgl. auch das Datenbank-Schema im Abschnitt 2.5.1).

Umgekehrt bedeutet dies natürlich, dass ein in einer Bedingung verwendetes Objekt-Attribut, welches eine andere Bezeichnung besitzt als das zugrunde liegende Datenbanktabellen-Attribut, auf dieses „zurück" abgebildet werden muss. Andernfalls kann das Datenbankmanagementsystem die Anfrage auf Grund des unbekannten Attributs nicht bearbeiten.

Ähnlich verhält es sich auch mit Attribut-Werten, die vom Object Wrapper transformiert werden. Angenommen, es gäbe für das Attribut 'Status' nur die beiden möglichen Werte 'released' und 'working'. Beim Datenbankdesign könnte man nun entscheiden, aus Speicherplatzgründen nicht die kompletten Zeichenketten abzulegen, sondern beispielsweise nur den ersten Buchstaben, also 'r' statt 'released', und 'w' statt 'working'. Das PDM-System jedoch soll wegen der besseren Lesbarkeit für den Anwender mit der Langform arbeiten, d. h. der Object Wrapper wandelt diese Abkürzungen wieder in die zugehörige Langform um. Bedingungen auf diesem Attribut würden vom Administrator in der Langform definiert, ohne Berücksichtigung dieser Werte-Transformation jedoch könnte das Datenbankmanagementsystem zwar die Anfrage korrekt verarbeiten, das Ergebnis wäre aber fälschlicherweise leer.

Für die Lösung dieses Problems existieren prinzipiell zwei Möglichkeiten:

1. Ersetzung des Objekt-Attribut-Wertes bei der Generierung der korrespondierenden WHERE-Klausel durch das Datenbanktupel-Äquivalent
2. *On-the-fly*-Umwandlung des Datenbank-Wertes in den zugehörigen Objekt-Attribut-Wert beim Zugriff

In dem angeführten Beispiel wäre ein Verfahren nach Möglichkeit 1 denkbar. Eine Ersetzungs-Tabelle etwa speichert die Abbildung zwischen Langform und Abkürzung, bei der Query-Modifikation muss lediglich diese Tabelle interpretiert und dementsprechend die Zeichenkette 'released' durch 'r' ersetzt werden.

Für komplexere Attribut-Berechnungen jedoch funktioniert dieser Ansatz nicht. Hier können nicht sämtliche Funktionsergebnisse mit ihren Ausgangswerten abgelegt werden: Für jede hinreichend große Menge von Ausgangswerten scheitert der Versuch, diese Ausgangswerte zusammen mit den zugehörigen Funktionsergebnissen abzulegen, an der nötigen Effizienz – falls eine derartige Speicherung überhaupt möglich ist.

Stattdessen ist es sinnvoll, die Funktion zur Berechnung des Wertes des Objekt-Attributes an der Datenbank zu hinterlegen, und diese bei Bedarf mit den entsprechenden Ausgangswerten (gleichbedeutend mit Funktions-Argumenten) aufzurufen. Im Beispiel bedeutet dies, dass auf dem Status-Attribut eine Funktion 'AbbrevToLong(...)' aufgerufen werden muss; der Vergleichswert 'released' bleibt dabei ungeändert:

```
... AND AbbrevToLong(status) = 'released'
```

Analog müssen auch Funktionen zur Berechnung transienter Attribute an der Datenbank bereitgestellt werden: Setzt sich ein solches Attribut aus einem oder mehreren Datenbank-Attributen zusammen, so lässt es sich mit den entsprechenden Funktionen bereits durch das Datenbanksystem berechnen.

Gesamt gesehen müssen folglich alle Änderungsoperationen, die der Object Wrapper bei der Umwandlung von Datenbanktupeln in Objekte des PDM-Systems durchführt, auch an der Datenbank durchgeführt werden. Werden dabei Funktionen des PDM-Systems aufgerufen, so müssen auch diese datenbankseitig verfügbar gemacht werden.

Im Folgenden wird nun die Transformation der *row conditions* und $\exists$*structure conditions*, den beiden wichtigsten Kategorien von Bedingungen, beschrieben.

Die bereits beschriebenen Abbildungen von Attribut-Namen und -Werten werden in einer *Ersetzungstabelle* gespeichert. Zusätzlich zu diesen einmalig zu definierenden Einträgen können noch weitere Einträge dynamisch hinzugefügt werden, die den Umgebungsvariablen konkrete Werte zuweisen, z. B. kann bei der Anmeldung des Benutzers Michael Schmidt der Eintrag '$User, Michael Schmidt' generiert werden. Damit definiert die Ersetzungstabelle die gesamte *Umgebung* für den Bedingungstransformator. Eine (fiktive) Ersetzungstabelle ist in Abbildung 3.10 dargestellt.

Im Folgenden seien die Funktionen, die zur Berechnung der Attributwerte oder bei der Evaluierung von Regeln benötigt werden, bereits an der Datenbank verfügbar. Da die (einmalig durchzuführende) Bereitstellung dieser Funktionen vom jeweils eingesetzten Datenbankmanagement-System abhängig ist, sei an dieser Stelle nur

PDM-System	**Datenbank**
Zusammenbau	assembly
Spezifikation	spec
Zusammenbau.Status	AbbrevToLong(assembly.status)
document.name	document.name ‖ document.extension
$User	Michael Schmidt

Abbildung 3.10: Ersetzungstabelle

auf die Beschreibungen des CREATE-FUNCTION-Statements in [ANS99a] oder [EM99] verwiesen.

Der Transformator besteht aus mehreren Übersetzungs-Funktionen, die Zeichenketten verarbeiten und auch ausgeben. Die Beschreibung all dieser Funktionen würde den Rahmen dieser Arbeit sprengen, so dass hier nur auf einige wichtige Übersetzungs-Funktionen eingegangen wird. Der komplette Transformator ist im Anhang A.2 angegeben und setzt das Datenbankschema aus Abbildung 2.12 voraus.

Gestartet wird die Transformation mit der Funktion *condition-trans*(c, ϱ, τ), die eine Bedingung c für einen Objekt-Typ τ unter Zuhilfenahme der Umgebung ϱ in eine SQL-konforme Bedingung übersetzt.[17] Dabei wird c entlang der Struktur, welche die Grammatik aus Anhang A.1 definiert, aufgespaltet und die dabei entstehenden Substrukturen (Expressions, Terme etc.) rekursiv übersetzt.

Am Beispiel von Attribut-Namen lässt sich das Prinzip der Übersetzung verdeutlichen. Die Transformation erfolgt mittels *factor-trans*:

factor-trans$($*attrib-name*$, \varrho, \tau)$:= $\varrho(\tau$.*attrib-name*$)$

Der Name dieser Funktion deutet bereits darauf hin, dass ein „Faktor“ innerhalb eines Terms oder einer Expression übersetzt wird. Der Aufruf dieser Übersetzungsfunktion für das Status-Attribut des Objekt-Typs ’Zusammenbau’ aus obigem Beispiel lautet also:

factor-trans$($*Status*$, \varrho,$ *Zusammenbau*$)$

wobei ϱ die Ersetzungstabelle aus Abbildung 3.10 ist. Damit ergibt sich die rechte Seite der Funktionsdefinition zu

$\varrho($*Zusammenbau.Status*$)$

[17] Dabei gelte für alle Werte x, die nicht in der Ersetzungstabelle definiert sind: $\varrho(x) = x$, d. h. für diese Werte findet keine Ersetzung statt.

und ein Nachschlagen in der Umgebung ϱ ergibt letztlich als Übersetzungsergebnis:

AbbrevToLong(*assembly.status*)

Nach diesem Muster verläuft auch die Übersetzung von Funktionen, Umgebungsvariablen und Konstanten, die in den *row conditions* verwendet werden.

Deutlich komplexer ist das Übersetzungsmuster für $\exists$*structure conditions*. Die Transformationsfunktion *scond-trans* nimmt eine derartige Bedingung und übersetzt sie zusammen mit der Umgebung und dem zugehörigen Objekt-Typ direkt in eine äquivalente SQL-Klausel. Für die Variante 2 der $\exists$*structure conditions* (vgl. Abschnitt 3.2.3.2) lautet dieses Übersetzungsmuster (vom Transformator ausgegebene Zeichenketten sind dabei fettgedruckt):

scond-trans($\exists$ *rel-type obj-type : rowcond*, ϱ, τ) :=
EXISTS (SELECT * FROM ϱ(*rel-type*) **JOIN** ϱ(*obj-type*)
ON ϱ(*rel-type*)**.succ** = ϱ(*obj-type*)**.obid**
WHERE ϱ(*rel-type*)**.pred** = $\varrho(\tau)$**.obid**
AND *rcond-trans*(*rowcond*, ϱ, *obj-type*)**)**

Das eingebettete SELECT-Statement ermittelt alle Objekte, die sich aus dem Join zwischen 'rel-type' und 'obj-type' ergeben, wobei die Tupel aus 'obj-type' die angegebene row condition erfüllen müssen. Über die EXISTS-Klausel wird geprüft, ob derartige Objekte existieren oder nicht.

Das folgende Beispiel (vgl. Abschnitt 3.2.3.2) verdeutlicht die Benutzung dieses Übersetzungsschemas: Der Benutzer Michael Schmidt darf alle Zusammenbauten anzeigen, für die eine Spezifikation via 'attaches'-Beziehung existiert, sofern das 'status'-Attribut der Spezifikation den Wert 'released' enthält. Die Ersetzung der Variablen in der Funktionsdefinition mit den Werten aus diesem Beispiel ergibt:

scond-trans($\exists$ *attaches Spezifikation:status='released'*, ϱ, *Zusammenbau*) :=
EXISTS (SELECT * FROM ϱ(*attaches*) **JOIN** ϱ(*Spezifikation*)
ON ϱ(*attaches*)**.succ** = ϱ(*Spezifikation*)**.obid**
WHERE ϱ(*attaches*)**.pred** = ϱ(*Zusammenbau*)**.obid**
AND *rcond-trans*(**status** = **'released'**, ϱ, *Spezifikation*)**)**

Nun fehlt noch die Ersetzung der Objekt-Namen mittels ϱ. Für 'attaches' existiert in der Ersetzungstabelle kein Eintrag, folglich findet keine Ersetzung statt. 'Spezifikation' muss durch 'spec', und 'Zusammenbau' wieder durch 'assembly' ersetzt werden. Man bemerke, dass die angegebene *row condition* bezüglich des Typs

'Spezifikation'(!) übersetzt wird, wohingegen die gesamte ∃*structure condition* bezüglich 'Zusammenbau' übersetzt wird.

Das Ergebnis der Übersetzung lautet somit:

EXISTS (SELECT * FROM *attaches* JOIN *spec*
ON *attaches.succ* = *spec.obid*
WHERE *attaches.pred* = *assembly.obid*
AND *spec.status* = *'released'*)

Diese Klausel stellt nun eine gültige SQL-Bedingung dar und kann in einem SELECT-Statement, welches in der FROM-Klausel die Tabelle 'assembly' verwendet, in die WHERE-Klausel eingefügt werden.

Die Anfrage-Modifikation zur Erweiterung einer Datenbank-Anfrage Q um die SQL-Bedingung(en) erfolgt in 5 Schritten:

1. Ermittlung aller auszuwertenden Bedingungen c_i
2. Übersetzung aller c_i
3. Führe ggf. Umbenennungen von Tabellen-Namen in c_i analog zu denen im FROM-Teil von Q durch
4. Bilde Oder-Verknüpfung $c = \vee_{j=1}^{i}(c_j)$
5. (a) Falls keine WHERE-Klausel vorhanden, erweitere Q um „WHERE c“
   (b) Sonst klammere existierende Bedingung und erweitere Q um die Bedingung „AND (c)“

Die auszuwertenden Bedingungen werden entsprechend des ausführenden Benutzers, der Aktion sowie des davon betroffenen Objekt-Typs ermittelt (vgl. auch Abschnitt 3.4.2).

Die Übersetzung in Schritt 2 erfolgt mit den bereits eingeführten Übersetzungsfunktionen. Sollte Q im FROM-Teil Tabellen umbenannt haben, so müssen in den Bedingungen ebenfalls diese Namen Verwendung finden (Schritt 3).

Da alle Regeln gewährend sind, genügt *eine* zu TRUE evaluierende Bedingung, d. h. die passenden Bedingungen müssen Oder-verknüpft werden (Schritt 4). Da im Schritt 5(b) der AND-Operator für die Erweiterung (c) stärker bindet als ein eventuell vorhandener OR-Operator in der originalen WHERE-Klausel von Q, muss vorsichtshalber die bisherige Bedingung geklammert werden.

Prinzipiell lassen sich auch ∀*rows conditions* und *tree aggregate conditions* auf SQL-WHERE-Klauseln abbilden. Voraussetzung für die Überprüfung von Regeln dieser Klassen jedoch ist, dass das gesamte (Zwischen-)Ergebnis, auf das sich eine solche Regel bezieht, an der Datenbank „greifbar“ vorliegt, z. B. in einer temporären oder virtuellen Tabelle, auf welcher dann diese speziellen Bedingungen ausgewertet werden. Mit SQL-Mitteln kann dies z. B. erreicht werden, indem sämtliche Teilergebnisse einer Aktion nach einer Schema-Angleichung mittels UNION-Operator vereinigt werden, oder durch Verwendung von rekursivem SQL (siehe auch Kapitel 4).

Sowohl die ∀*rows conditions* als auch die *tree aggregate conditions* setzen das *Alles-oder-Nichts*-Prinzip um: Entweder wird das Zwischenergebnis komplett zurückgegeben (d. h. die Bedingung wurde zu TRUE evaluiert), oder es wird NULL zurückgeliefert (d. h. die Bedingung war FALSE).

Bezeichne *temptable* für die folgenden Betrachtungen das Zwischenergebnis, auf welches die Regeln anzuwenden sind. Dann lässt sich dieses Prinzip für ∀*rows conditions* wie folgt übersetzen:

allrcond-trans(∀*rows : rowcond*, ϱ, τ) :=
SELECT * FROM *temptable* **WHERE NOT EXISTS (**
SELECT * FROM *temptable* **WHERE NOT**
rcond-trans(*rowcond*, ϱ, *temptable*) **)**

Es werden alle Tupel des Zwischenergebnisses zurückgegeben, falls kein Tupel in diesem Zwischenergebnis existiert, für welches die row condition nicht gilt. Da das eingeschachtelte SELECT-Statement, welches auf die row condition testet, vom äußeren unabhängig ist, ist klar, dass dieses Statement für alle äußeren Tupel das gleiche Ergebnis liefert. Somit gilt die Bedingung entweder für alle Tupel des Zwischenergebnisses, oder für keines.

Anmerkung: Dem aufmerksamen Leser wird nicht entgangen sein, dass die Übersetzungsfunktion *rcond-trans* mit *temptable* als Parameter für den Objekt-Typ τ aufgerufen wird. Der Grund hierfür liegt darin, dass die Bedingung für *alle* Objekt-Typen gelten muss, die im Zwischenergebnis vorkommen. Des Weiteren ist zu beachten, dass bereits bei der Erzeugung des Zwischenergebnisses mögliche Attributumbenennungen etc. zu berücksichtigen sind. Da diese Regeln in der Praxis jedoch keinen großen Stellenwert besitzen, sind hier keine weiteren Details angegeben.

Für die tree aggregate conditions lässt sich zur Umsetzung des Alles-oder-Nichts-Prinzips folgendes Übersetzungsschema definieren:

treecond-trans(*agg_func*(*attrib-name*) *relop signed-expr*, ϱ, τ) :=
SELECT * FROM *temptable* **WHERE (**
SELECT *func-trans*(*agg_func*(*attrib-name*), ϱ, ε)
FROM *temptable*) *relop signedexpr-trans*(*signed-expr*, ϱ, *temptable*)

Dabei ist *relop* einer der bekannten Vergleichsoperatoren ($=, <, <=, \ldots$).

Auch hier gilt, dass das innere SELECT-Statement vom äußeren unabhängig ist und somit für alle Tupel des äußeren Statements das gleiche Ergebnis liefert. Folglich werden wiederum alle oder keines der Tupel aus dem Zwischenergebnis zurückgegeben.

Analog zur Übersetzung der $\forall$*rows conditions* werden auch hier die Übersetzungsfunktionen aus Anhang A.2 mit $\tau = temptable$ gerufen, da sie sich auf mehrere Objekt-Typen im Zwischenergebnis beziehen können.

3.5.4.3 Optimierung durch Template-Bildung

Das bisher beschriebene Verfahren führt sämtliche Transformationen zur Laufzeit aus, d. h. wenn der Benutzer eine Aktion anstößt, müssen zunächst sämtliche passenden Bedingungen übersetzt und die Anfrage entsprechend modifiziert werden. Führt der Benutzer dieselbe Aktion mehrmals auf dem gleichen Objekttyp aus, so wiederholt sich der Vorgang entsprechend.

Hier bietet es sich an, die *statischen Anteile* der Bedingungen vorab *einmal* in ein SQL-konformes Template zu übersetzen und zusammen mit dem in der zugehörigen Regel definierten Benutzer (bzw. der Gruppe), dem Objekt-Typ und der Aktion in einer Tabelle zu speichern. Zur Laufzeit sind nur noch die dynamischen Anteile (wie die Belegung der $User-Variablen) zu ergänzen. Da Bedingungen mit dynamischen Anteilen relativ selten sind, lässt sich hiermit der Transformations-Aufwand deutlich senken.

3.5.4.4 Fazit

Die Übersetzung von Zugriffsbedingungen in äquivalente SQL-WHERE-Klauseln ist ein vielversprechender Ansatz, um die Bedingungen direkt beim Zugiff auf die Datenbank-Tupel zu testen. Die Template-Bildung ermöglicht die Einsparung eines großen Teils des Transformations-Aufwandes zur Laufzeit, so dass auch eine performante Modifikation zumindest von einfachen Anfragen, die sich auf nur eine Datenbanktabelle stützen, zu erwarten ist.

Nachteilig ist jedoch, dass je nach Umfang und Anzahl zu einer Aktion passender Regeln das resultierende SQL-Statement sehr umfangreich werden kann, besonders bei *rekursiven* Anfragen, die bereits ohne Modifikation sehr komplex sind (siehe auch Kapitel 4). Deshalb zeigt Abschnitt 3.5.5 eine elegantere Variante, die sich dann auch für die integrierte Regelauswertung in komplexen Anfragen eignet.

3.5.5 Einsatz von Table-Functions

Die drei bisher betrachteten Verfahren ACL, PT und WHERE-Klausel-Modifikation bieten nur unzureichenden Schutz vor „Daten-Piraterie“: Sie vertrauen darauf, dass sich die Applikation (in unserem Anwendungsfall das PDM-System) an die „Spielregeln“ hält und die jeweils notwendigen Anfrage-Transformationen durchführt. In diesem Fall führt zwar – wie ja angestrebt war – das Datenbanksystem die Regelauswertung durch, unterbleibt die Anfragetransformation jedoch, so stehen den Benutzern alle verfügbaren Daten und alle darauf ausführbaren Operationen offen!

Wünschenswert ist daher, dass das Datenbanksystem selbst für die Evaluierung der Zugriffsrechte sorgt und damit eine Untergrabung des gesamten Zugriffskontrollmechanismus per se verhindert.

Die Lösung für dieses Problem liegt auf der Hand: Jedem Benutzer wird eine *Sicht* auf die Basis-Tabellen zur Verfügung gestellt, die alle Instanzen ausblendet, die für ihn hinsichtlich seiner initiierten Aktivität nicht verfügbar sind. Zumindest für die *row conditions* und $\exists$*structure conditions*, die sich ja nur auf Basis-Tabellen (und nicht auf Teilgraphen) beziehen, erscheint dies möglich. Analoges gilt für die Konfigurationssteuerung: Für jede definierte Strukturoption wird eine Sicht erzeugt, welche exakt die Objekte freigibt, die bei der Wahl dieser Strukturoption in der Produktstruktur enthalten sind.

Auf den ersten Blick scheint das *Sichtenkonzept* relationaler Datenbanksysteme hierfür geeignete Mechanismen anzubieten (Details siehe [EM99, MS02]). Problematisch jedoch ist die aus diesem Ansatz resultierende Anzahl von *Views*, da jeder Benutzer für jeden Objekt-Typ und jede auszuführende Aktion eine derartige View benötigt – dies gilt auch dann, wenn der Benutzer auf *keine* Instanz eines Objekt-Typs Zugriffsrechte besitzt! PDM-Systeme im praktischen Einsatz besitzen durchaus 300–400 verschiedene Objekt-Typen[18], bei mehreren hundert Anwendern werden folglich schnell tausende von Views benötigt.

Abhilfe könnten hier *parametrisierte Views* schaffen, welche die Zugriffsrechte aller Benutzer und Aktionen für einen Objekt-Typ vereinen und durch Parameter-

[18] Die Anzahl kann deutlich darunter aber auch darüber liegen, je nachdem, in welchen Phasen eines Produktlebenszyklus das PDM-System eingesetzt wird.

Übergabe eines konkreten Benutzers und der initiierten Aktivität die hierfür auszuwertenden Rechte überprüfen. Am folgenden Beispiel wird die Vorgehensweise klarer:

Der Benutzer 'Michael Schmidt' soll Zusammenbauten nur anzeigen können, die im Status-Attribut den Wert 'released' besitzen. Daneben soll der Benutzer 'Hugo Huber' Zusammenbauten einchecken können, deren Status-Attribut den Wert 'working' enthalten. Abbildung 3.11 zeigt die Definition der parametrisierten View (unter Verwendung einer fiktiven Syntax) sowie die Verwendung bei der Selektion aller sichtbaren Zusammenbauten für 'Michael Schmidt' und der Selektion aller eincheckbaren Zusammenbauten für 'Hugo Huber'.

```
CREATE VIEW ASSEMBLYVIEW(user VARCHAR(50), action VARCHAR(15)) AS (
SELECT *
FROM assembly AS a
WHERE (user='Michael Schmidt' AND action='display'
       AND a.status='released')
OR    (user='Hugo Huber' AND action='checkin'
       AND a.status='working'))

SELECT * FROM ASSEMBLYVIEW('Michael Schmidt','display')

SELECT * FROM ASSEMBLYVIEW('Hugo Huber','checkin')
```

Abbildung 3.11: Parametrisierte View und ihre Verwendung

Parametrisierte Views sind nicht Bestandteil des SQL:1999-Standards. Das relationale Datenbankmanagementsystem IBM DB2 UDB V7.2 (vgl. [IBM01]) bietet jedoch mit den sogenannten *Table Functions* ein alternatives Konzept an, mit welchem – von kleinen syntaktischen Unterschieden abgesehen – ein vergleichbarer Effekt erzielt werden kann. Auch Oracle bietet seit der Version 9 dieses Konzept an (vgl. [Ora01a, Ora01b, Ora01c]), welches entsprechend den heutigen Planungen auch im Standard SQL:2003 enthalten sein wird.

3.5.5.1 Einführung in Table Functions

Table Functions liefern als Ergebnis einer Berechnung eine Tabelle zurück, die in der FROM-Klausel einer SQL-Anfrage verwendet werden kann. Um einen ersten Eindruck über Table Functions zu erhalten, genügt die Betrachtung eines Beispiels für deren Definition und Verwendung. Nähere Informationen dazu können [IBM01] entnommen werden.

```
CREATE FUNCTION t_assembly (user varchar(24), action varchar(15))
RETURNS TABLE (obid VARCHAR(24), PartNumber VARCHAR(24),
               status VARCHAR(10), ...)
LANGUAGE SQL RETURN
SELECT a.obid, a.PartNumber, a.status, ...
FROM assembly AS a
WHERE (user='Michael Schmidt' AND action='display'
       AND a.status='released')
OR    (user='Hugo Huber' AND action='checkin'
       AND a.status='working')
```

Abbildung 3.12: Definition einer *Table Function*

In Abbildung 3.12 wird die Funktion 't_assembly' definiert. Sie erhält die beiden Parameter 'user' und 'action'. Die Angabe RETURNS TABLE identifiziert 't_assembly' als Table Function. Die Spalten der zurückgegebenen Tabelle werden analog zur Anweisung 'CREATE TABLE assembly' deklariert.

Benutzerdefinierte Funktionen können prinzipiell auch in externen Programmiersprachen wie C oder JAVA codiert werden. Für unsere Zwecke hier genügen die Sprachmittel von SQL, weshalb die Funktionsdefinition die Klausel 'LANGUAGE SQL' enthält. Auf das Schlüsselwort 'RETURN' folgt der Funktions-Rumpf, in unserem Anwendungsfall ein SELECT-Statement ähnlich dem in der (fiktiven) parametrisierten View (vgl. Abbildung 3.11).

Die Verwendung der Table Functions erfolgt im SELECT-Statement in der FROM-Klausel. Eine Anfrage, die der Anwender Michael Schmidt im Kontext der Aktion 'Display' auf Zusammenbauten ausführt, lautet wie folgt:

```
SELECT *
FROM TABLE(t_assembly('Michael Schmidt','display')) AS assembly
```

Das Schlüsselwort TABLE is zwingend vorgeschrieben, ebenso eine Benennung mittels 'AS ...', wobei der Bezeichner beliebig gewählt werden darf. Insbesondere kann also auch der Name der Basis-Tabelle, die der Table Function zugrunde liegt, verwendet werden. Das kann besonders dann sinnvoll sein, falls in einer existierenden SQL-Anfrage die bisher verwendete Basis-Tabelle durch die Table Function ersetzt wird, und in der WHERE-Klausel Referenzen auf Attribute der Basis-Tabelle in der Form *tabelle.attribut* enthalten sind – eine Anpassung der WHERE-Klausel ist somit nicht nötig.

Resümee: Aus der Sicht der für PDM-Systeme erforderlichen Funktionalität sind Table Functions bis auf kleinere syntaktische Unterschiede äquivalent zu den (fiktiven) parametrisierten Views.

3.5.5.2 Generierung der Table Functions

Für die folgenden Ausführungen wird die Konfigurationssteuerung (vgl. Abschnitt 3.3) zunächst außer Acht gelassen. Wir betrachten nur Regeln, die Zugriffe im ursprünglichen Sinn auf Objekte steuern.

Die Table Functions für derartige Regeln entsprechen einem fix vorgegebenen Schema, wie es Abbildung 3.12 zeigt: Es werden die beiden Parameter 'user' und 'action' benötigt, die erzeugte Tabelle entspricht in ihrem Schema dem der zugrunde liegenden Basistabelle, und die WHERE-Klausel lässt sich mittels Übersetzungsfunktionen aus den zutreffenden Regel-Bedingungen herleiten.

Table Functions können folglich relativ leicht automatisch generiert werden, d. h. falls sich das Regelwerk ändert, können die betroffenen Funktionen ohne Programmier-Aufwand nachgezogen werden. Der Algorithmus 3.5.1 beschreibt einen *Table Function Generator*, der ein korrektes CREATE-FUNCTION-Statement für die Überprüfung der Zugriffsrechte auf einem PDM-Objekt-Typ (Zusammenbauten, Einzelteile, Dokumente etc.) erzeugt.

Algorithmus 3.5.1: (Table Function Generator)

TFGenerator(in ObjektTyp τ, in Umgebung ϱ, out String TF)

Datenbanktabelle $\leftarrow \varrho(\tau)$
DTS $\leftarrow$ *load_schema*(*Datenbanktabelle*)
ZR $\leftarrow$ *sort* ZRT[*ObjektTyp*] *on Benutzer (oder Gruppe) und Aktion*
for each *rule* $\in$ *ZR* **do**
    rule.condition $\leftarrow$ *condition-trans*(*rule.condition,* ϱ, τ)
enddo
TF $\leftarrow$ "CREATE FUNCTION t_" $\|$ *Datenbanktabelle*
TF $\leftarrow$ *TF* $\|$ "(user VARCHAR(..), action VARCHAR(..))"
TF $\leftarrow$ *TF* $\|$ "RETURNS TABLE (" $\|$ $DTS[1].name \,\|\, DTS[1].type$
for $(i = 2)$ **to** $|DTS|$ **do**
    TF $\leftarrow$ *TF* $\|$ ", " $\|$ $DTS[i].name \,\|\, DTS[i].type$
enddo
TF $\leftarrow$ *TF* $\|$ ") LANGUAGE SQL RETURN SELECT " $\|$ $DTS[1].name$
for $(i = 2)$ **to** $|DTS|$ **do**
    TF $\leftarrow$ *TF* $\|$ ", " $\|$ $DTS[i].name$
enddo
TF $\leftarrow$ *TF* $\|$ "FROM " $\|$ *Datenbanktabelle* $\|$ "WHERE"
for each maximal set $\{r_1, r_2, ..., r_n : r_j \in ZR \wedge$
$r_1.user = r_2.user = ... = r_n.user \wedge$
$r_1.action = r_2.action = ... = r_n.action\}$ **do**
    if (r_1.*user is a group*) **then**
        TF $\leftarrow$ *TF* $\|$ "(is_in(user,' " $\|$ $r_1.user$ $\|$ " ')"

```
23    else
24       TF ← TF ‖ "(user=' " ‖ r1.user ‖ " ' "
25    endif
26    TF ← TF ‖ " AND action=' " ‖ r1.action " ' AND ("
27    TF ← TF ‖ "(" ‖ r1.condition ‖ ")"
28    for (i = 2) to n do
29       TF ← TF ‖ "OR (" ‖ ri.condition ‖ ")"
30    enddo
31    TF ← TF ‖ "))"
32    if (!done) then TF ← TF ‖ " OR " endif
33 enddo
34 return
```

Der Algorithmus verfährt wie folgt: Zunächst wird zu dem übergebenen Objekt-Typ die zugehörige Datenbanktabelle bestimmt, das Schema der Basistabelle ermittelt und die Attribut-Namen und -Typen in DTS (DatenbankTabellenSchema) abgelegt (vgl. Zeilen 1 und 2).

In Zeile 3 werden alle Zugriffsregeln (ZR), die den Objekt-Typ betreffen, bestimmt und für die spätere Zusammenfassung nach Benutzer (oder Gruppen) und Aktionen sortiert. Daraufhin werden die darin enthaltenen Bedingungen mit den Transformationsfunktionen (vgl. Anhang A.2) übersetzt.

Der nun folgende Teil des Algorithmus dient nun dazu, aus den bisher erzeugten Informationen ein korrektes CREATE-FUNCTION-Statement zu erzeugen. Das Schema der zu erzeugenden Tabelle (vgl. Zeilen 9 bis 12) entspricht dabei dem der zugrunde liegenden Basistabelle (bis auf eventuell vorhandene Integritätsbedingungen, die nicht abgebildet werden können und müssen). Dementsprechend werden auch alle Attribute der Basistabelle in der vorgegebenen Reihenfolge selektiert (Zeilen 13 bis 16).

Die WHERE-Klausel wird durch eine OR-Verknüpfung (Zeile 32) aller Bedingungen gebildet, wobei Bedingungen, die auf dem gleichen Benutzer und der gleichen Aktion definiert sind, zusammengefasst werden (Zeilen 18 bis 31). Unter der ***maximalen Menge*** $\{r_1, r_2, ..., r_n\}$ ist zu verstehen, dass ***kein*** weiteres Element r_x existiert mit $r_1.name = r_x.name$ und $r_1.action = r_x.action$, welches nicht in der Menge enthalten ist.

Nach erfolgter Generierung kann das Statement dem Datenbanksystem übergeben werden. Sollte es sich dabei nicht um die erstmalige Erzeugung der Table Function handeln, so muss zuvor noch der Befehl DROP FUNCTION abgesetzt werden (siehe auch [EM99, MS02]), um die überholte Version der Funktion aus dem Datenbanksystem zu löschen.

Aus den bisherigen Betrachtungen war die Konfigurationssteuerung ausgenommen, da sich die Konfigurations-Information ausschließlich auf die Beziehungen zwischen Zusammenbauten und ihren Bauteilen bezieht und *nicht* von dem anfragenden Anwender und der auszuführenden Aktion abhängig ist, sondern von dessen gewählten Strukturoptionen und der gesetzten Gültigkeit.

Die Regeln zur Steuerung von Strukturoptionen und Gültigkeiten (vgl. Abschnitt 3.3) sind typischerweise fix vorgegeben und keinen Änderungen unterworfen. Das bedeutet nicht, dass sich Konfigurationen nicht ändern können, sondern dass das zugrunde liegende Konzept der Konfigurationssteuerung erhalten bleibt! Aus diesem Grund genügt es, die benötigte Table Function auch einmalig (quasi von Hand) zu erzeugen. Abbildung 3.13 zeigt eine adäquate Funktionsdefinition für die *uses*-Beziehung.

```
CREATE FUNCTION t_uses
       (options {optid VARCHAR(24)},effcty VARCHAR(10))
RETURNS
TABLE (obid VARCHAR(24), pred VARCHAR(24), succ VARCHAR(24), ...)
LANGUAGE SQL RETURN
SELECT u.obid, u.pred, u.succ, ...
FROM u IN uses
WHERE (u.StrcOpts IS EMPTY OR options IS EMPTY
       OR (u.StrcOpts INTERSECT options) IS NOT EMPTY)
AND
(EXISTS(SELECT * FROM eff IN u.Effectivities
WHERE effcty BETWEEN eff.Start AND eff.End))
```

Abbildung 3.13: Konfigurationssteuerung über eine *Table Function*

In der WHERE-Klausel des SELECT-Statements im Funktions-Rumpf werden die Übersetzungen der Strukturoptions- und Gültigkeitsbedingung eingesetzt. Ein Objekt der *uses*-Relation ist somit nur dann in der vom Benutzer definierten Konfiguration enthalten, falls es mit den ausgewählten Strukturoptionen *und* der angegebenen Gültigkeit harmoniert.

Eine Funktion für die *has_revision*-Beziehung kann auf ähnliche Art definiert werden. In Abbildung 3.13 ist im Wesentlichen lediglich die Prüfung der Strukturoptionen sowie der Parameter `options` zu entfernen.

Anmerkung: Die Notation in Abbildung 3.13 lehnt sich der besseren Verständlichkeit halber an [PT85] und [PT86] an. Im Standard SQL:2003 (vgl. [Tür03]) werden voraussichtlich entsprechende Funktionen für Schnittmenge (INTERSECT) und Kardinalitätstest (CARDINALITY(..) != 0 anstelle von IS EMPTY) auf mengenwertigen Attributen definiert sein.

3.5.5.3 Hilfsfunktionen für komplexere Anfragen

Komplexere Anfragen, wie sie in Abschnitt 4 eingeführt werden, müssen möglicherweise eine Kombination von Zugriffsrechten auswerten. Bei der Expansion einer Produktstruktur können beispielsweise in einem ersten Schritt alle sichtbaren Unterteile angefragt werden, in einem zweiten Schritt sollen davon alle wiederum expandiert werden, für die der Benutzer das Expansionsrecht besitzt. Man benötigt folglich einen Prüfmechanismus, um festzustellen, ob ein vorliegendes Objekt expandiert werden darf.

Hier bietet sich an, für jeden Objekt-Typ eine Hilfsfunktion folgender Form zu definieren:

BOOLEAN type_*permits*(*action* VARCHAR(..), *obid* VARCHAR(..), *user* VARCHAR(..))

wobei 'type' durch den Objekt-Typ ersetzt wird (z. B. *assembly_permits*(...)). Falls der Benutzer *user* auf dem Objekt mit der ID *obid* die Aktion *action* ausführen darf, wird TRUE zurückgegeben, andernfalls FALSE.

Eine Implementierung dieser Hilfsfunktionen, die sich auf die Table Functions aus Abschnitt 3.5.5.2 stützen, ist in Abbildung 3.14 dargestellt.

```
CREATE FUNCTION assembly_permits (action VARCHAR(..),
                obid VARCHAR(..), user VARCHAR(..))
RETURNS BOOLEAN
LANGUAGE SQL RETURN
CASE WHEN obid IN (
  SELECT a.obid FROM TABLE(t_assembly(user, action)) AS a) THEN TRUE
ELSE FALSE
END
```

Abbildung 3.14: Hilfsfunktion zur Regelauswertung auf Zusammenbauten

Selbstverständlich lassen sich diese Hilfsfunktionen ebenfalls generieren. Da dies keine allzu schwierige Aufgabe ist, bleibt die Lösung dem Leser überlassen.

3.5.5.4 Fazit

Table Functions bieten genügend Funktionalität, um die datenbankseitige Auswertung von Zugriffsberechtigungen durchzuführen. Sie sind effizient generierbar, d. h. der Aufwand bei Änderungen des Regelwerkes hält sich in überschaubaren Grenzen.

Vorteilhaft – wenn auch für PDM-Systemumgebungen nicht besonders wichtig – ist die vollständige Kapselung der Regeln und damit eine Trennung von Anfrage und Regelauswertung. Die Sicherheit lässt sich sogar erhöhen, indem nicht nur ein Parameter für den anfragenden Benutzer, sondern auch ein Passwort übergeben wird, so dass zusätzlich eine Authentifizierung stattfinden kann.

Sofern in der weiteren Arbeit von frühzeitiger Regelauswertung die Rede ist, wird – falls nicht explizit anders erwähnt – eine Darstellung in Table Functions angenommen.

Weiterführende Informationen zu Table Functions können [RPK+99] entnommen werden.

3.5.6 Architektur der Anfragekomponente

Die Einbindung der verschiedenen Komponenten, die für die frühzeitige Regelauswertung benötigt werden und deren Funktionsweisen in diesem Kapitel beschrieben wurden, ist in Abbildung 3.15 dargestellt.

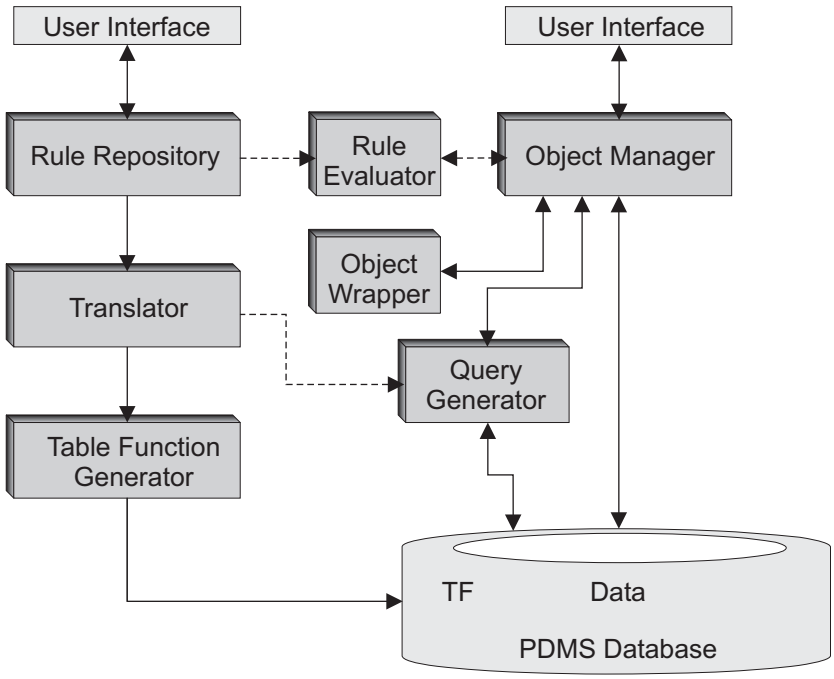


Abbildung 3.15: Architektur der Anfragekomponente mit frühzeitiger Regelauswertung; TF steht für die Table Functions, die wie die PDM-Daten in der Datenbank verwaltet werden.

Kapitel 4

Datenbankseitige Unterstützung rekursiver Aktionen

4.1 Szenario: Anfragebearbeitung bei zentraler Datenhaltung und verteilter Verarbeitung

Die in heutigen PDM-Systemen eingesetzte Strategie zur Anfragebearbeitung (vgl. Kapitel 2) kann bei zentraler Datenverarbeitung (und damit auch zentraler Datenhaltung) zu durchaus akzeptablen Antwortzeiten führen. Deshalb wird im Folgenden der Fokus insbesondere auf *verteilter Datenverarbeitung* liegen.

Wir betrachten ein Unternehmen, das ein zentrales Rechenzentrum (oder eine zentrale EDV-Abteilung) betreibt, welches für die Speicherung und Verwaltung sämtlicher Produktdaten verantwortlich ist. Das Unternehmen unterhält mehrere Entwicklungsabteilungen an verschiedenen Standorten, die alle im Umkreis von maximal einigen zehn Kilometern um das Rechenzentrum angesiedelt sind.

An jedem Standort wird den Anwendern die volle PDM-Funktionalität lokal zur Verfügung gestellt, die Produktdaten jedoch werden aus dem zentralen Rechenzentrum bezogen. Mehrere hundert PDM-System-Benutzer – verteilt auf die verschiedenen Standorte – sind dabei durchaus denkbar. Auch externe, kleinere Entwicklungsbüros können z. B. für spezielle Projektlösungen in einen derartigen „Entwicklungsverbund“ einbezogen werden. Abbildung 4.1 zeigt ein solches Szenario.

Die Kommunikation zwischen den Unternehmensbereichen und dem Rechenzentrum stützt sich zum Beispiel auf ein Metropolitan Area Network (MAN, vgl. [SHK+97]). MANs besitzen im Vergleich zu Wide Area Networks (WAN) kürzere Latenzzeiten (ca. 50 - 100 ms) und potentiell höheren Datenduchsatz (bis ca.

140Mbit/s), und bieten damit für verteilte Entwicklungsumgebungen etwas bessere Voraussetzungen als WANs.

In *weltweit* verteilten Produktentwicklungs-Umgebungen ist eine zentrale Datenhaltung auf Grund der hohen Kommunikationskosten nur in seltenen Ausnahmefällen anzutreffen.

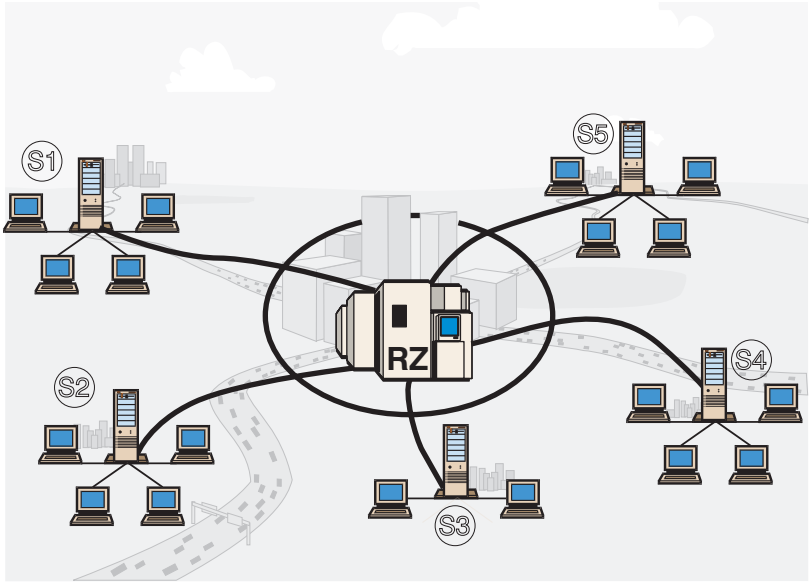


Abbildung 4.1: Zentrale Datenhaltung im Rechenzentrum (RZ) und verteilte Verarbeitung an mehreren Standorten (S1 – S5) unter Verwendung eines Metropolitan Area Networks

4.2 Problematik und Lösungsansatz

Metropolitan Area Networks sind in der Regel deutlich langsamer als Local Area Networks. Zwar sind die theoretisch möglichen Kennzahlen hinsichtlich Latenzzeit und Datentransferrate inzwischen auf einem recht beeindruckenden Niveau, wie wir bereits motiviert haben können jedoch auch schnelle Netze – nicht zuletzt auf Grund der wachsenden Anzahl von Netzbenutzern, die sich die zur Verfügung gestellte Bandbreite teilen müssen – nicht dauerhaft für ausreichende Performance sorgen.

In unserem Szenario aus Abschnitt 4.1 bedingt *jeder* Zugriff auf die Datenbank des PDM-Systems eine Kommunikation über das MAN. Für jeden dieser Zugriffe

fällt also die Latenzzeit sowohl für die Anfrage als auch für das Anfrageergebnis an, zuzüglich der für die Datenübertragung benötigten Zeit. Wie bereits in Abschnitt 2.3 motiviert wurde, werden hier Antwortzeiten bereits für einfache Aktionen wie etwa eine Query erforderlich, die von den Benutzern nicht mehr akzeptiert werden. Das Problem wird noch deutlicher bei komplexeren Aktionen, die rekursiv größere Teilbäume der Produktstruktur traversieren, also beispielsweise bei den Multi-Level-Expansionen (siehe Abschnitt 2.1.2).

Es treten nun also die Systemschwächen zu Tage, die in Abschnitt 2.3.1 beschrieben sind: Die rekursiven Aktionen verursachen sehr viele Round-Trips zur Datenbank, wobei in den meisten Fällen auch Daten übertragen werden, auf die der anfragende Benutzer keinen Zugriff hat. Beides trägt dazu bei, die Antwortzeiten in die Höhe zu treiben: Durch die vielen Round-Trips summieren sich die Latenzzeiten, auch die unnötigerweise übertragenen Daten benötigen Übertragungszeit und belasten zudem noch das Netz. Im ungünstigsten Fall werden Daten, auf die der Anfrager (zumindest für die ausgeführte Aktion) keine Zugriffsrechte besitzt, auch noch in einer separaten Kommunikation übertragen!

Die angestrebte Lösung wird also zunächst versuchen, mit möglichst wenigen Round Trips über das MAN zur Datenbank auszukommen. Hierfür bieten sich prinzipiell die folgenden Möglichkeiten an:

1. Verlagerung der Rekursion vom Server des PDM-Systems auf (oder zumindest nahe an) den Datenbankserver. Dadurch kann zwar nicht die Anzahl der Datenbank-Aufrufe reduziert werden, jedoch finden diese Aufrufe nur noch lokal, d. h. im Wesentlichen ohne nennenswerte Latenzzeit, statt.

2. Ersetzen der applikationsseitigen Rekursion durch datenbankseitige Rekursion mittels rekursivem SQL [ANS99a, EM99]. Hierbei können mehrere einzeln ausgeführte SQL-Statements durch ein einziges rekursives Statement ersetzt werden, so dass nur noch eine einzige Kommunikation (und damit nur noch ein Round-Trip) zum Datenbankserver benötigt wird.

Die erstgenannte Möglichkeit offenbart mehrere Schwächen: Trotz der durchgeführten Optimierung können keine Anfrage-/Antwort-Zyklen eingespart werden. Das Datenbankmanagementsystem kann dabei nur die einzelnen Queries separat und unabhängig voneinander optimieren – eine Optimierung des gesamten Rekursionsablaufes kann nicht erfolgen. Daher wird die Wahl der Implementierungsmethode für die Rekursion durch den Programmierer (naive Methode, semi-naive Methode etc., siehe [CCH93]) die Antwortzeit der Benutzeraktion ganz entscheidend mitbestimmen.

Demgegenüber stehen die Vorteile des Einsatzes rekursiven SQLs: Ein einziges rekursives Statement ersetzt eine Sequenz einzelner Statements, die bei der ap-

plikationsseitigen Rekursion erforderlich sind. Folglich wird auch nur noch eine einzige Kommunikation mit der Datenbank benötigt. Da der gesamte rekursive Ablauf in einer einzigen Anfrage enthalten ist, kann hier das Datenbankmanagementsystem die geeignete Methode zur Berechnung des rekursiven Ergebnisses selbst bestimmen und damit optimale Performance erzielen.

Neben der Einsparung von Round-Trips über das MAN gilt es auch, die Übertragung nicht-sichtbarer Datensätze weitestgehend zu vermeiden, d. h. möglichst nur die Daten aus der Datenbank anzufordern, für die der Benutzer auch Zugriffsrechte besitzt. Entsprechend Kapitel 3 müssen folglich Zugriffsregeln möglichst frühzeitig, also bereits am Datenbankserver, ausgewertet werden.

In den folgenden Abschnitten wird der Lösungsansatz vorgestellt, der die Verwendung von rekursivem SQL mit frühzeitiger Zugriffsregelauswertung kombiniert und damit in dem aufgezeigten Szenario akzeptable Antwortzeiten möglich macht.

4.3 Anfrage-Bearbeitung mittels rekursivem SQL

4.3.1 Überblick über rekursives SQL

Rekursives SQL ist im SQL:1999-Standard [ANS99a] enthalten und ist in dem relationalen Datenbanksystem DB2 UDB von IBM verfügbar. An dieser Stelle soll ein kurzer Überblick über die Fähigkeiten dieses Features genügen.

Ein rekursives SQL-Statement gliedert sich in drei Teile (siehe Abbildung 4.2). Teil eins definiert das Schema der Tabelle, die das Ergebnis der Rekursion enthalten soll. Die Berechnung dieses Ergebnisses wird im Teil zwei beschrieben. Nach der Ermittlung des Rekursions-Beginns (Schritt 2a) wird hierbei entsprechend der Rekursionsvorschrift in jedem Rekursionsschritt ein Join zwischen den der Rekursion zugrunde liegenden Basis-Tabellen und dem bereits vorhandenen Teilergebnis berechnet (Schritt 2b), wobei neu ermittelte Datensätze dem Rekursionsergebnis hinzugefügt werden. Zu beachten ist, dass diese Datensätze dem in Teil eins definierten Schema entsprechen müssen. Im Teil drei schließlich werden die Datensätze und Attribute aus dem Rekursionsergebnis extrahiert, die dem Benutzer als Ergebnis der Anfrage präsentiert werden sollen.

In frühen Implementierungen von rekursiven SQL-Ausdrücken konnte die Rekursion nur über *eine* Basistabelle berechnet werden. Heutige Implementierungen, z. B. in IBM DB2 UDB Version 7.2, erlauben Rekursionsvorschriften über mehrere Basistabellen hinweg. Damit wurden die Voraussetzungen für die Nutzung

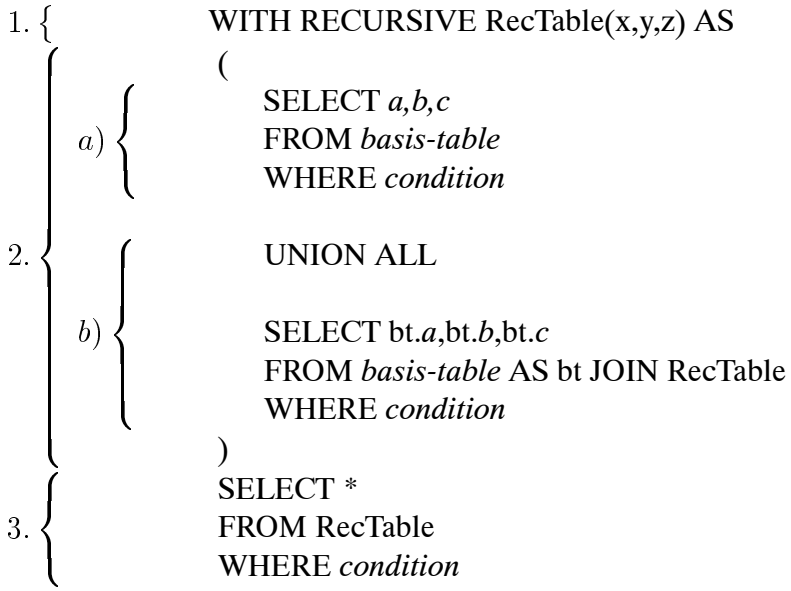


Abbildung 4.2: Prinzipieller Aufbau eines rekursiven SQL-Statements

rekursiven SQLs auch im Umfeld der PDM-Systeme geschaffen (siehe Abschnitt 4.3.2).

4.3.2 Abbildung rekursiver PDM-Benutzeraktionen auf rekursives SQL

Beschreibung der Rekursionsvorschrift

Während der Ausführung rekursiver Benutzeraktionen, welche Teilgraphen der Produktstruktur durchlaufen, werden die strukturbeschreibenden Datenbanktabellen immer nach dem gleichen Muster angefragt (Abbildung 4.3):

Der Start einer rekursiven Aktion erfolgt in der Regel an einer konkreten Version eines Zusammenbaus. Ausgehend von diesem Zusammenbau werden sämtliche *uses*-Objekte erfragt, die auf Stammdaten (ComponentMaster bzw. AssemblyMaster) untergeordneter, d. h. eingebauter Objekte zeigen. Nach Abfrage der ermittelten Stammdaten wird auf die konkreten Versionen (Component bzw. Assembly) über die Verkettungsrelation *has_revision* zugegriffen.

Aus Abbildung 4.3 ist ersichtlich, dass die Rekursion endet, sobald ein Rekursionsschritt nur noch Einzelteile (Components) liefert, die nicht weiter zerlegt werden können, oder – dies ist in der Regel nur während der Entwicklungsphase möglich – falls für einen Zusammenbau (Assembly) keine Substruktur definiert

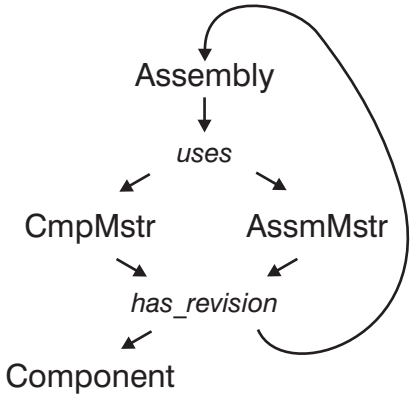


Abbildung 4.3: Anfragereihenfolge der Datenbanktabellen bei rekursiven Benutzeraktionen

wurde, d. h. es existieren keine zugeordneten *uses*-Objekte und damit auch keine Subobjekte für diesen Zusammenbau.[19]

Abbildung auf rekursives SQL

Bei der Übertragung der in Abbildung 4.3 dargestellten Rekursion auf rekursives SQL sind folgende Punkte zu berücksichtigen:

1. Die Ausführung einer rekursiven Benutzeraktion muss bei Unterstützung durch rekursives SQL zum exakt gleichen Ergebnis führen wie bei der heute üblichen applikationsgetriebenen Rekursionsberechnung. Folglich ist die anfragende Applikation durch das rekursive SQL-Statement mit allen Informationen zu versorgen, die für die Konstruktion des Endergebnisses benötigt werden. Dies bedeutet, dass *alle* Objekte, die zur Rekursionsberechnung herangezogen werden, im Ergebnis der rekursiven SQL-Anfrage enthalten sein müssen. Es genügt dabei nicht, nur die Knoten-Objekte der Produktstruktur (Stammdaten und deren Versionen) zu übertragen, sondern es müssen auch die Kanten-Objekte (*uses*- und *has_revision*-Tupel) zurückgegeben werden.

2. Das Ergebnis von SQL-Anfragen – und damit auch das einer rekursiven SQL-Anfrage – ist eine *(homogene) Menge* von Objekten. Dies bedeutet,

[19] Umgekehrt gilt: Falls mindestens ein *uses*-Objekt existiert, dann existiert auch mindestens ein Einzel- oder Zusammenbauteil, welches in das betrachtete Assembly eingeht! Dies liegt daran, dass Stammdaten (Master) nicht ohne Version(en) existieren können.

dass alle Ergebnistupel das gleiche Schema besitzen müssen. Die Tupel jedoch, die aus den einzelnen Basis-Tabellen ermittelt werden, können völlig unterschiedlich aufgebaut sein: die *uses*-Relation enthält andere Attribute als die AssmMstr-Relation, hingegen sind CmpMstr und AssmMstr typischerweise identisch (vgl. Datenbankschema in Abbildung 2.12)! Folglich müssen die verschiedenen Schemata der Basistabellen zu einem homogenen Ergebnisschema integriert werden.

3. Wenn nun alle Tupel des Ergebnisses den gleichen Typ besitzen, dann ist die aufrufende Applikation möglicherweise nicht mehr in der Lage, den Ursprungstyp eines Tupels zu bestimmen. Dies tritt beispielsweise immer dann auf, wenn mindestens zwei Basistabellen das gleiche Schema besitzen (z. B. CmpMstr und AssmMstr). Es ist daher sinnvoll, dem Ergebnisschema ein Diskriminator-Attribut hinzuzufügen, welches den Ursprungstyp der Tupel beschreibt (beispielsweise „CmpMstr“ oder „AssmMstr“).

Für die Übertragung der Rekursion auf rekursives SQL empfiehlt sich – unter Beachtung der genannten Punkte – folgendes Vorgehen:

Zunächst ist das **Schema der Ergebnisrelation** zu definieren. Das primitivste Verfahren hierfür ist die Vereinigung der Schemata aller Basisrelationen, die im Rekursionspfad vorkommen. Bei Namensgleichheit von Attributen aus verschiedenen Basistabellen kann durch Voranstellen des Basistabellennamens Eindeutigkeit erzielt werden. Die Tupel des Anfrageergebnisses besitzen nur verwertbare Information in den Attributen ihrer Basistabellen, alle anderen Attribute werden auf NULL gesetzt.

Trotz der vielen zu erwartenden NULL-Werte mag dieses Schema den Anforderungen im Allgemeinen genügen. Im Falle der PDM-Anwendung jedoch sind wesentliche Vereinfachungen möglich: Viele Attribute aus verschiedenen Basistabellen besitzen gleiche Namen, Typen und Semantik (z. B. das Attribut „Nomenclature“ der Tabellen AssmMstr, CmpMstr). Es bietet sich daher an, diese Attribute nicht mehrfach im Ergebnisschema zu repräsentieren, sondern durch ein einziges zu ersetzen. Das Anfrage-Ergebnis lässt sich dadurch ohne Informationsverlust kompakter darstellen.

Im nächsten Schritt ist nun der **Rekursions-Beginn** festzulegen. Das Start-Tupel, welches durch den Benutzer festgelegt wird (beispielsweise durch Übergabe einer Objekt-ID), muss als erstes Tupel in das Ergebnis selektiert werden (vgl. Abbildung 4.2, Schritt 2a).

Die für die **Rekursions-Schritte** benötigten SQL-Subanfragen ergeben sich aus dem Rekursionsschema (vgl. Abbildung 4.3). Mit der bereits berechneten Ergebnismenge wird für jede Basistabelle ein Join – entsprechend der Semantik der

Basistabelle – über die jeweiligen Attribute berechnet. Die zugehörige Select-Klausel wandelt die neu hinzukommenden Tupel aus dem Schema der Basisrelation um in das Schema der Ergebnisrelation. Für das in Abbildung 4.3 gezeigte Rekursionsschema werden somit sieben derartige Subanfragen benötigt. Alle diese SQL-Statements werden (innerhalb des rekursiven Teils, vgl. Abbildung 4.2, Schritt 2b) mittels UNION ALL verknüpft. Die Reihenfolge, in welcher diese Statements aufgelistet werden, spielt dabei keine Rolle.

Die Definition der Rekursionsberechnung in SQL ist damit beendet, es müssen lediglich noch die gefundenen Ergebnistupel an die anfragende Applikation zurückgegeben werden. Für eine effiziente Bearbeitung des Ergebnisses ist es sinnvoll, während der Rekursionsberechnung an jedem Tupel die Information zu vermerken, in welcher Rekursionstiefe es gefunden wurde. Wird das Gesamtergebnis nach der Tiefe sortiert, so kann mit relativ geringem Aufwand die originale Struktur rekonstruiert werden.

Für das Rekursionsschema in Abbildung 4.3 ergibt sich zusammen mit den Basistabellen aus Abschnitt 2.5.1 entsprechend der hier vorgestellten Vorgehensweise das SQL-Statement aus Abbildung 4.4.

Bislang haben wir auf die Berücksichtigung von Benutzer-Zugriffsberechtigungen bewusst verzichtet. Das bedeutet, dass das erzeugte rekursive SQL-Statement *alle* Objekte der Produktstruktur entlang des Rekursionspfades zurückliefert, also auch Objekte, auf die der anfragende Benutzer keine Zugriffsberechtigung besitzt. Im folgenden Abschnitt wenden wir uns dieser Problematik zu und zeigen eine geeignete Lösung auf.

4.4 Integration der Regelauswertung

Das in Abschnitt 4.3 aufgezeigte Verfahren zur Nutzung rekursiven SQLs stellt nur einen Teilerfolg hinsichtlich der Performance-Optimierungen dar. Es werden zwar nur noch *eine Anfrage* und damit auch nur *ein Ergebnis* über das MAN transportiert, das Ergebnis jedoch stellt möglicherweise noch eine Obermenge der Daten dar, die dem anfragenden Benutzer angezeigt werden dürfen. Im Sinne der Performance-Optimierung werden wir nun auch die zu übertragende Datenmenge auf ein Minimum reduzieren.

Eine Einschränkung der bislang ermittelten Ergebnismenge wird auf Grund der eingeschränkten Zugriffsrechte des anfragenden Benutzers sowie der gewählten Konfigurationsvorgaben erforderlich. In den Abschnitten 3.2 und 3.3 haben wir bereits gezeigt, welche Gestalt Zugriffsrechte und Konfigurationsvorgaben besit-

```
WITH ProdStr(LVL,TYPE,OBID,PN,NC,LOC,REV,SEQ,ASY,MSTR,ASDB,MDB,VER,VDB) AS
(
    SELECT 0, 'ASY', OBID, PartNumber, NULL, Location, Revision,
    Sequence, NULL(6)
    FROM Assembly
    WHERE PartNumber=<PN> AND Revision=<Rev> AND Sequence=<Seq>
      UNION ALL
    SELECT ProdStr.LVL+1, 'USE', uses.OBID, NULL(5), uses.pred,
    uses.succ, uses.predLoc, uses.succLoc, NULL(2)
    FROM uses, ProdStr
    WHERE uses.pred=ProdStr.OBID
      UNION ALL
    SELECT ProdStr.LVL+1, 'ASM', am.OBID, am.PartNumber,
    am.Nomenclature, am.Location, NULL(8)
    FROM AssmMstr AS am, ProdStr
    WHERE am.OBID=ProdStr.MSTR
      UNION ALL
    SELECT ProdStr.LVL+1, 'CPM', cm.OBID, cm.PartNumber,
    cm.Nomenclature, cm.Location, NULL(8)
    FROM CmpMstr AS cm, ProdStr
    WHERE cm.OBID=ProdStr.MSTR
      UNION ALL
    SELECT ProdStr.LVL+1, 'HRV', hrv.OBID, NULL(6), hrv.pred, NULL,
    hrv.predLoc, hrv.succ, hrv.succLoc
    FROM has_revision AS hrv, ProdStr
    WHERE hrv.pred=ProdStr.OBID
      UNION ALL
    SELECT ProdStr.LVL+1, 'ASY', ay.OBID, ay.PartNumber, NULL,
    ay.Location, ay.Revision, ay.Sequence, NULL(6)
    FROM Assembly AS ay, ProdStr
    WHERE ay.OBID=ProdStr.Version
      UNION ALL
    SELECT ProdStr.LVL+1, 'CMP', cp.OBID, cp.PartNumber, NULL,
    cp.Location, cp.Revision, cp.Sequence, NULL(6)
    FROM Component AS cp, ProdStr
    WHERE cp.OBID=ProdStr.Version
)
SELECT DISTINCT * FROM ProdStr ORDER BY LVL
```

Abbildung 4.4: Rekursives SQL-Statement zur vollständigen Expansion der Produktstruktur eines Zusammenbaus. <PN>, <Rev> und <Seq> müssen im konkreten Aufruf durch PartNumber, Revision und Sequence des zu expandierenden Startknotens ersetzt werden. NULL(n) steht abkürzend für: die nächsten n Attribute sind NULL-wertig.

zen und wie sie mit SQL-Mitteln unmittelbar auf der Datenbank ausgewertet werden können.

Wir gehen im Folgenden davon aus, dass die *row conditions* und ∃*structure conditions* in entsprechenden Table Functions berücksichtigt sind, und die ∀*rows conditions* und *tree aggregate conditions* in adäquate WHERE-Klauseln übersetzt werden können (vgl. auch Abschnitt 3.5.4).

Des Weiteren sei Q_{act} ein rekursives SQL-Statement wie in Abbildung 4.4 für eine bestimmte Benutzeraktivität act, beispielsweise bezeichnet Q_{mle} die Anfrage für einen Multi-Level-Expand.

Für den anfragenden Anwender setzen wir wiederum die Variable $User ein.

4.4.1 Regelauswertung während der Rekursion

Bereits während des rekursiven Aufbaus der Ergebnisstruktur können an den Knoten-Objekten die Zugriffsberechtigungen und an den Kanten-Objekten die Konfigurationsvorgaben überprüft werden. Letzteres lässt sich einfach erledigen, indem statt der Basis-Tabellen 'uses' und 'has_revision' die Table Functions 't_uses' und 't_has_revision' eingesetzt und dabei die vom Benutzer ausgewählten Optionen übergeben werden.

Nicht ganz so eindeutig mag zunächst das Vorgehen für die Knoten-Objekte sein: Klar ist, dass auch deren Basis-Tabellen durch die entsprechenden Table Functions ersetzt werden müssen, aber welche Aktivität wird dabei übergeben? Im Falle eines Multi-Level-Expands beispielsweise, muss die hier eingesetzte Aktivität 'expand' lauten oder vielleicht doch nur 'display'?

Diese Frage lässt sich nur mit Anwendungswissen bezüglich der jeweiligen Aktivität act beantworten: Bei einem Multi-Level-Expand sollen die Ergebnisobjekte letztendlich *angezeigt* werden. Es muss also geprüft werden, ob der Benutzer entsprechende Display-Rechte auf den Objekten besitzt. Außerdem müssen während des rekursiven Ergebnisaufbaus zusätzlich alle Zusammenbauten, die rekursiv expandiert werden sollen, darufhin überprüft werden, ob der Benutzer auch die dazu erforderlichen Rechte besitzt. Es ist durchaus denkbar, dass es einem Benutzer zwar erlaubt ist, einen Zusammenbau zu sehen ('display'), nicht aber ihn weiter zu detaillieren ('expand'). Die Anfrage Q_{mle} muss in diesem Fall den fraglichen Zusammenbau noch in das Ergebnis aufnehmen, die Rekursion stoppt dort jedoch.

Aus diesem Beispiel leiten wir eine allgemeine Vorgehensweise ab:

> Die Table Functions für Knoten-Objekte werden mit der Aktivität angestoßen, die letztlich auf ihnen ausgeführt werden soll. Sind im re-

kursiven Abstieg weitere Bedingungen an „Vater“-Objekte geknüpft, so werden diese mit den Hilfsfunktionen überprüft.

Betrachten wir das zweite SELECT-Statement innerhalb der Rekursion in Abbildung 4.4. Hier findet die Expansion aller bereits im (Zwischen-)Ergebnis enthaltenen Zusammenbauten statt, indem die temporäre Tabelle 'ProdStr' mit der 'uses'-Relation verknüpft wird. Um zu überprüfen, ob die Zusammenbauten aus 'ProdStr' vom Benutzer expandiert werden dürfen, erweitern wir die bestehende WHERE-Klausel um die Bedingung

AND *assembly_permits('expand', ProdStr.obid, $User)*

Dass dieser Zusammenbau vom Benutzer auch angezeigt werden darf, wurde dabei schon eine Rekursionstiefe zuvor durch den Einsatz der Table Function *t_assembly*(...) im sechsten SELECT-Statement sichergestellt.

Es gilt hier noch ein kleines Problem zu lösen: Das Ergebnis einer wie beschrieben modifizierten Anfrage Q'_{act} kann Kanten-Objekte zu Knoten-Objekten enthalten, die nicht im Ergebnis enthalten sind.

Als Lösung bietet es sich an, beim „Aufsammeln“ der Kanten-Objekte zu testen, ob der Anwender die gewünschte Aktion auf dem Zielobjekt (Nachfolger) ausführend darf. Dazu verwenden wir wiederum die Hilfsfunktionen.

Betrachten wir noch einmal das zweite SELECT-Statement in Abbildung 4.4. Das Zielobjekt der uses-Beziehung ist entweder ein Assembly Master oder ein Component Master. Folglich ergänzen wir die WHERE-Klausel:

AND (*assmmstr_permits('display', uses.right, $User)*
OR *compmstr_permits('display', uses.right, $User)*)

Bemerkung: Dem aufmerksamen Leser wird nicht entgangen sein, dass die resultierende Anfrage eine Vielzahl von Table Function-Aufrufen enthält. Ein guter Optimierer jedoch wird erkennen, dass häufig die gleiche Table Function gerufen wird und deshalb die Auswertung jeweils nur einmal durchzuführen ist!

4.4.2 Regelauswertung nach der Rekursion

Die Berücksichtigung von eventuell vorhandenen ∀*rows conditions* und *tree aggregate conditions* steht bislang noch aus. Um dem „Alles-oder-Nichts“-Prinzip dieser Bedingungen Genüge zu leisten, erfolgt deren Überprüfung im Anschluss

an die Berechnung des rekursiven Zwischenergebnisses, d. h. im Teil 3 des rekursiven SQL-Statements (vgl. Abbildung 4.2).[20] Damit ergibt sich für die Anfrage-Modifikation folgendes Vorgehen:

1. Ermittle alle relevanten[21] ∀rows conditions $c_1 \ldots c_k$ und tree aggregate conditions $c_{k+1} \ldots c_n$ und übersetze sie entsprechend den Schemata aus Abschnit 3.5.4.2.
2. Bilde die Disjunktion aller ermittelten Bedingungen $d = \bigvee_{i=1}^{n} c_i$.
3. Erweitere alle SELECT-Statements im Teil 3 der rekursiven Anfrage: bestehende WHERE-Klauseln werden mit d mittels AND verknüpft, im Falle nicht vorhandener WHERE-Klauseln wird eine neue Klausel bestehend aus d erzeugt.

Für den Fall, dass – wie in unserem Beispiel in Abbildung 4.4 – in der Definition des Ergebnisschemas (Teil 1 der rekursiven Anfrage) Umbenennungen von Attributen stattgefunden haben, muss noch ein Zwischenschritt zwischen 2. und 3. eingeführt werden: Alle Attribute in den ermittelten Bedingungen sind entsprechend des Ergebnisschemas umzubenennen! Ohne diese Anpassung würden Bedingungen möglicherweise auf Attribute zugreifen, die im Ergebnisschema nicht enthalten (weil umbenannt) sind.

Abbildung 4.5 zeigt schematisch eine um Regelauswertung ergänzte Anfrage für Multi-Level-Expands. Die Änderungen gegenüber Abbildung 4.4 sind auf der linken Seite durch einen senkrechten Strich gekennzeichnet.

4.4.3 Optimierung durch Template-Bildung

Die in den Abschnitten 4.4.1 und 4.4.2 vorgestellte Anfrage-Modifikation für rekursive Anfragen Q_{act} wird man nicht bei jeder Initiierung der Aktion act durchführen. Stattdessen bietet es sich an, jedes Q_{act} in ein entsprechendes Template zu übersetzen, welches zur Laufzeit mit minimalem Aufwand lediglich durch die Ersetzung des Start-Objektes (Teilenummer etc.) und der Benutzer-Variablen $User, $UsrKonfig und $UsrEff in ein lauffähiges SQL-Statement transformiert wird.

[20] ∀*rows conditions* könnten ja prinzipiell schon innerhalb der Rekursion überprüft werden um bei Nichterfüllung abzubrechen. Der SQL-Standard sieht jedoch keine Möglichkeit vor, eine Rekursion abzubrechen und einen vordefinierten Wert (hier: NULL) anstatt des bis zu diesem Zeitpunkt errechneten Ergebnisses zurückzugeben.

[21] „relevant“ in diesem Zusammenhang bedeutet, dass die Bedingung auf den anfragenden Benutzer, den betroffenen Objekt-Typ und die darauf auszuführende Aktion passt.

```
WITH  ProdStr(LVL,TYPE,OBID,PN,NC,LOC,
      REV,SEQ,ASY,MSTR,ASDB,MDB,VER,VDB) AS
(
      SELECT 0, 'ASY', OBID, PartNumber, NULL, Location, Revision,
      Sequence, NULL(6)
|     FROM TABLE(t_Assembly($User, 'display')) AS Assembly
      WHERE PartNumber=<PN> AND Revision=<Rev> AND Sequence=<Seq>
|     AND Assembly_permits('expand', Assembly.obid, $User)

      UNION ALL

      SELECT ProdStr.LVL+1, 'USE', uses.OBID, NULL(5), uses.pred,
      uses.succ, uses.predLoc, uses.succLoc, NULL(2)
|     FROM TABLE(t_uses($UsrKonfig, $UsrEff)) AS uses, ProdStr
      WHERE uses.pred=ProdStr.OBID
|     AND (AssmMstr_permits('display', uses.succ, $User)
|     OR CmpMstr_permits('display', uses.succ, $User))
|     AND Assembly_permits('expand', uses.pred, $User)

      UNION ALL

      SELECT ProdStr.LVL+1, 'ASM', am.OBID, am.PartNumber,
      am.Nomenclature, am.Location, NULL(8)
|     FROM TABLE(t_AssmMstr($User, 'display')) AS am, ProdStr
      WHERE am.OBID=ProdStr.MSTR

      UNION ALL

      ...
)
SELECT DISTINCT *
FROM ProdStr
| WHERE (⋁_ν ∀row condition_ν) OR (⋁_μ tree aggregate condition_μ)
ORDER BY LVL
```

Abbildung 4.5: Rekursives SQL-Statement (Ausschnitt) zur vollständigen Expansion der Produktstruktur eines Zusammenbaus entsprechend den Zugriffsrechten und den Konfigurationsvorgaben eines Benutzers.

Besonders elegant lässt sich diese Laufzeit-Transformation durchführen, indem das Template wiederum in eine Table Function verpackt wird, wobei die zu ersetzenden Variablen als Parameter übergeben werden. Für die Anfrage Q_{mle} beispielsweise lässt sich die Anfrage aus Abbildung 4.5 in eine Table Function *mle*(*obid, user, config, eff*) verpacken, so dass der Aufruf eines Multi-Level-Expands für den Zusammenbau mit der Objekt-ID 4711 beispielsweise lautet:

```
SELECT *
FROM TABLE(mle(4711, 'Hugo Huber', 'Schiebedach', '15.08.2000')) AS mle
```

Das PDM-System hat also zur Laufzeit keinerlei Transformationen mehr durchzuführen, sämtliche Übersetzungen erfolgen einmalig und werden am Datenbanksystem in geeigneten Funktionen bereitgestellt.

4.5 Performance-Gewinn durch rekursives SQL

Die Datenbankanfragen und Table Functions, die in den vorangegangenen Abschnitten durch entsprechende Modifikationen der primitiven rekursiven Anfragen erzeugt wurden, sind im Vergleich zu den von heutigen PDM-Systemen abgesetzten Anfragen deutlich komplexer. Es ist daher mit einem höheren Aufwand auf Seiten des Datenbanksystems zu rechnen. Im Vergleich zu den Kosten jedoch, die durch hohe Netzlast auf Grund naiver Datenbankbenutzung entstehen, können wir bei den folgenden Betrachtungen diesen lokal zu treibenden Aufwand vernachlässigen.

Für die Beurteilung des Performance-Gewinns des in diesem Kapitel beschriebenen Ansatzes erweitern wir den in Abschnitt 2.3.2.1 eingeführten Formalismus um $n_v(t)$, das die Anzahl der zugreifbaren Objekte in der Struktur wiedergibt. Expandiert man einen Zusammenbau einstufig, so erhält man $4 * \nu\sigma$ viele zugreifbare Objekte: $\nu\sigma$ viele Objekte der *uses*-Beziehung, genauso viele Master, *has-revision*-Objekte und weitere Zusammenbauten bzw. Einzelteile. Für mehrstufige Expansionen muss für die Anzahl der zugreifbaren Objekte im Level $l + 1$ die Anzahl des Levels l lediglich mit $\nu\sigma$ multipliziert werden, also gilt:

$$n_v(t) = 4 * \sum_{i=1}^{\tau} (\nu\sigma)^i$$

Da das Anfrageergebnis des in diesem Kapitel gezeigten Ansatzes nur Objekte zurückgibt, auf die der Anfrager zugreifen darf, ersetzten wir in Gleichung 2.3 den Ausdruck $n_t(t)$ durch $n_v(t)$ und lösen Gleichung 2.4. Tabelle 4.5 zeigt die zu erwartenden Antwortzeiten von rekursiven Benutzeraktionen (Multi-Level-Expansionen) in den Beispielszenarien aus Abschnitt 2.3.2.2, wobei der Einsatz

$size_{packet}$ = 4kB $\varnothing size_{node}$ = 512Byte	$\tau = 3$, $\nu = 9$, $\sigma = 2/3$ MLE	$\tau = 7$, $\nu = 3$, $\sigma = 2/3$ MLE	$\tau = 6$, $\nu = 5$, $\sigma = 3/5$ MLE
T_{Lat} = 0.6	1.2	1.2	1.2
dtr = 512	8.1	8.0	34.2
$T_s = \Sigma$	9.3	9.2	35.4
Einsparung in %	98.8	99.1	99.0
T_{Lat} = 0.3	0.6	0.6	0.6
dtr = 1024	4.1	4.0	17.1
$T_s = \Sigma$	4.7	4.6	17.7
Einsparung in %	98.8	99.1	99.0
T_{Lat} = 0.15	0.3	0.3	0.3
dtr = 10240	0.4	0.4	1.7
$T_s = \Sigma$	0.7	0.7	2.0
Einsparung in %	99.6	99.7	99.8

Tabelle 4.1: Antwortzeiten für Multi-Level-Expansionen mit rekursiven SQL-Statements und frühzeitiger Regelauswertung

von rekursivem SQL in Verbindung mit frühzeitiger Zugriffsregelauswertung zugrunde gelegt wurde. Auch die jeweilige Einsparung gegenüber dem naiven Ansatz (vgl. Tabelle 2.3) ist mit angegeben.

Fazit: In allen Szenarien sind für Multi-Level-Expansionen hinsichtlich der durch Kommunikation verursachten Antwortzeiten Verbesserungen um mehr als 95% zu erwarten. Selbstverständlich profitieren auch Single-Level-Expansionen von den hier beschriebenen Optimierungen, da auch sie nur noch eine einzige Kommunikation zur Datenbank benötigen. Für verteilt arbeitende Entwicklungsbereiche regional angesiedelter Unternehmen mit einem zentralem Rechenzentrum stellt somit die hier vorgestellte Zugriffsstrategie für rekursive Benutzeraktionen eine wesentliche Produktivitätssteigerung aller PDM-Benutzer in Aussicht. Dass diese Einsparungen nicht nur rein rechnerisch möglich sind, zeigen wir in Kapitel 7 an komplexeren Szenarien mittels Simulation.

Kapitel 5

Verteilte Ausführung rekursiver Aktionen

5.1 Szenario: Anfragebearbeitung bei vollständig azyklisch verteilter Datenhaltung und Verarbeitung

Wir betrachten ein weltweit agierendes Unternehmen, welches beispielsweise länderspezifische Versionen seiner Produktpalette an mehreren Standorten in den jeweiligen Ländern entwickeln und teilweise fertigen lässt. Die weltweit verteilt angesiedelten Entwicklungsabteilungen sind jeweils für ihre Datenhaltung selbst verantwortlich und verwalten alle Daten, die am jeweiligen Standort erzeugt und verändert werden. Zusätzlich werden noch mehrere Zulieferer angebunden, die ebenfalls ihre eigene Datenverwaltung betreiben.

Mehrere tausend Benutzer mit den unterschiedlichsten Aufgaben, Anforderungen und Berechtigungen sind in solchen Szenarien keine Seltenheit. In der Regel wird vorausgesetzt, dass sich alle Partner auf den Einsatz *eines* PDM-Systems einigen - andernfalls können Daten, falls überhaupt, nur äußerst umständlich und kostspielig ausgetauscht werden.[22]

In einem derartigen Entwicklungsverbund ist die Datenverteilung entsprechend den zugestandenen Kompetenzen vorgegeben – die Replikation von Daten an mehrere Standorte ist hier zumeist aus unternehmenspolitischen Gründen nicht

[22] Wir übernehmen diese „idealisierte Welt“ für die folgenden Betrachtungen. Selbstverständlich können aber auch Entwicklungsverbunde, deren Teilnehmer unterschiedliche PDM-Systeme einsetzen, von den hier vorgeschlagenen Verbesserungen profitieren.

erwünscht oder sogar untersagt, in vielen Fällen auch technisch auf Grund des gigantischen Datenvolumens nicht realisierbar und sinnvoll.

In diesem Kapitel wollen wir von einer strengen Hierarchie *Auftraggeber–Auftragnehmer* ausgehen, d. h. wenn eine Entwicklungsabteilung E_1 eine andere Abteilung (oder einen Zulieferer) E_2 beauftragt, einen Teil des Gesamtproduktes zu entwickeln, so kann E_2 zur Lösung dieser Aufgabe wohl noch weitere Entwicklungsabteilungen E_3, E_4, ... hinzuziehen,[23] nicht aber E_1. Dadurch wird eine azyklische Verteilung der Produktstruktur gewährleistet (vergleiche Abbildung 5.1: A ist Auftrag*geber* von B, C und D, aber nicht wieder deren Auftrag*nehmer*).

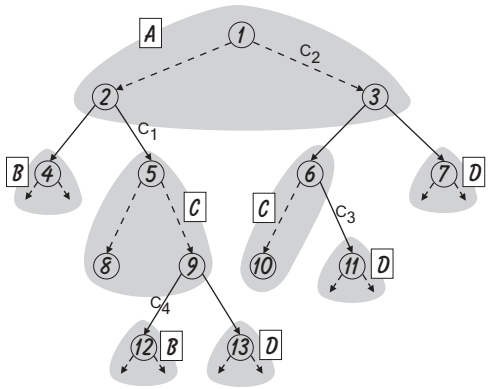


Abbildung 5.1: Eine primitive, auf die Standorte A, B, C und D azyklisch verteilte Produktstruktur (als verteilter DACG)

Die Kommunikation zwischen den Partnern wird über das Wide Area Network (WAN) abgewickelt. Hier sind Latenzzeiten bis zu 500 Millisekunden keine Seltenheit. Die Datentransferrate liegt typischerweise bei einigen hundert Kilobit pro Sekunde, in Ausnahmefällen sind auch Übertragungsraten im niedrigen Megabit-Bereich möglich. Abbildung 5.2 zeigt ein Szenario der hier beschriebenen Art.

5.2 Problematik und Lösungsansatz

Im Gegensatz zu dem im Abschnitt 4.1 aufgezeigten Szenario können hier viele Zugriffe auf die Datenbank lokal erfolgen, d. h. solange die Benutzer nur Daten von ihrem Standort benötigen, sind keine Performance-Engpässe zu erwarten. Sobald jedoch standortübergreifende Datenbankzugriffe erforderlich sind, treten auf Grund der Datenverteilung neue Schwierigkeiten und Fragestellungen auf:

[23] Man spricht hier auch von „Unterbeauftragung“.

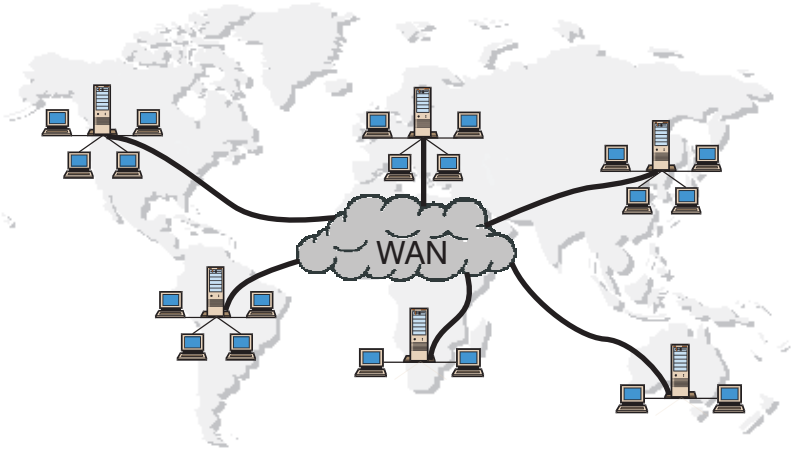


Abbildung 5.2: Weltweit verteilte Produktentwicklung

Die Datenbanktabellen, welche die Produktstruktur-Objekte und deren Beziehungen untereinander speichern, liegen nun nicht mehr an *einem* Datenbankserver, sondern sie sind auf *mehrere Partitionen* an verschiedenen Servern verteilt. Dies hat zur Folge, dass die gesamte Produktstruktur (und auch verteilt gespeicherte Sub-Strukturen) nicht mehr mit einem einzigen rekursiven SQL-Statement effizient zusammengesetzt werden kann: Zum Zeitpunkt der Anfrage-Generierung ist nicht bekannt, welche Partitionen im Laufe der Abarbeitung angefasst werden müssen, um einen Zusammenbau vollständig zu rekonstruieren. Die hierfür benötigte Information kann nur durch die Interpretation der Informationen in den Beziehungsobjekten ermittelt werden, ist also erst zur Laufzeit verfügbar. SQL bietet jedoch keine Möglichkeit, die Datenquellen während der Anfragebearbeitung, also dynamisch, auf Grund solcher „laufzeit-interpretierten“ Daten zu wechseln. Folglich kann ein rekursives SQL-Statement entweder a) nur eine (lokale) Partition zur Auswertung heranziehen, oder b) die Vereinigung aller möglichen Partitionen (evtl. in einer View versteckt). Im ersten Fall stoppt die Rekursion nach Ermittlung aller lokal verfügbaren Objekte und kann somit nicht die komplette Struktur ermitteln. Im zweiten Fall werden möglicherweise viel zu viele Daten, die nicht zum Ergebnis gehören, an den auswertenden Server übertragen.

Die Lösung besteht offensichtlich darin, rekursive Benutzeraktionen auf mehrere rekursive SQL-Anfragen abzubilden, die an den involvierten Servern abgearbeitet werden müssen. Diese „Aufteilung“ der Benutzeraktion erfordert Anwendungswissen: Die Objekte der lokal ermittelten Produktstruktur müssen interpretiert werden und bei Bedarf – d. h. falls Beziehungsobjekte auf Objekte in entfernten Datenbanken zeigen – weitere rekursiv definierte Anfragen an entfernte Datenbankserver gestellt werden.

Prinzipiell sind für die Ausführung verteilter rekursiver Aktionen mehrere Strategien denkbar. So kann etwa ein Datenbankserver als „zentraler Auftraggeber“ ausgezeichnet werden, der sukzessive entfernte Server beauftragt, eine fehlende Teilstruktur – soweit dort *lokal* verfügbar – zu ermitteln. Alternativ dazu könnten auch alle angesprochenen Datenbankserver selbst wieder Unteraufträge an andere Server weiterreichen (kaskadierende Serveraufrufe). Zusätzlich kann auch die Parallelisierung der Bearbeitung von Unteraufträgen in Betracht gezogen werden. Wir werden in Abschnitt 5.4 diskutieren, welche Strategie unter welchen Bedingungen jeweils geeignet erscheint, um dem Ziel der Gesamtkostenminimierung möglichst nahe zu kommen.

5.3 Partitionierung der Produktstruktur-Graphen

Wir betrachten die Produktstruktur wieder als DACG $G^c = (V, E, \mathcal{C})$. Entsprechend des Szenarios aus Abschnitt 5.1 muss dieser DACG verteilt gespeichert vorliegen, wobei prinzipiell jeder Knoten auf einem beliebigen Server liegen kann. Sei S die Menge aller am Entwicklungsverbund beteiligten PDM-Server oder PDM-Standorte. Dann bildet die Funktion $f : V \rightarrow S$ jeden Knoten auf die Menge der involvierten Server ab, d. h. f weist jedem Elementarobjekt der Produktstruktur einen Speicherort (Server) zu.

Die einzelnen Server $s \in S$ speichern *Partitionen* P_{s_i} von G^c.

Definition 4: (Partitionierung eines Graphs)

> Sei $G^c = (V, E, \mathcal{C})$ ein DACG und $f : V \rightarrow S$ eine Funktion zur Markierung von Knoten. Für jedes $s \in S$ existiert genau ein Teilgraph P_s von G^c mit:
>
> $P_s = (V_s, E_s, \mathcal{C})$ mit
> $V_s = \{v \in V | f(v) = s\}$ und $E_s = \{u \xrightarrow{c} v \in E | u \in V_s \vee v \in V_s\}$
>
> Die Zusammenhangskomponenten[24] jedes Teilgraphs P_s nennen wir *Partitionen* $P_{s_i} = (V_{s_i}, E_{s_i}, \mathcal{C})$ von G^c.

Man beachte, dass im Falle $|S| > 1 \quad P_{s_i}$ kein Graph im Sinne der Definition 1 in Abschnitt 2.5.2.2 ist: Die Kantenmenge E_{s_i} enthält im Gegensatz zu „gewöhnlichen“ Graphen auch Kanten, deren Anfangs- oder Endknoten nicht in V_{s_i} liegen.

[24] Die Zusammenhangskomponenten eines DACGs entsprechen denen des korrespondierenden ungerichteten Graphs [CLR96].

Diese Kanten sind sogenannte *Serverübergänge*, d. h. Anfangs- und End-Knoten solcher Kanten liegen an unterschiedlichen Speicherorten (vgl. Definition 5).

Definition 5: (Serverübergang, Eingangs- und Ausgangsknoten)

Sei $G^c = (V, E, \mathcal{C})$ ein DACG. Die Menge $E^{\bowtie}(G^c)$ der *Serverübergänge von* G^c ist definiert wie folgt:

$$E^{\bowtie}(G^c) = \{u \xrightarrow{c} v \in E \mid f(u) \neq f(v)\}$$

Analog gilt für die Serverübergänge einer Partition $P_{s_i} = (V_{s_i}, E_{s_i}, \mathcal{C})$ von G^c:

$$E^{\bowtie}(P_{s_i}) = \{u \xrightarrow{c} v \in E_{s_i} \mid f(u) \neq f(v)\}$$

Die Menge $V^{\top}(P_{s_i})$ der *Partitions-Eingangsknoten* (kurz: Eingangsknoten) sowie die Menge $V^{\perp}(P_{s_i})$ der *Partitions-Ausgangsknoten* (kurz: Ausgangsknoten) sind definiert wie folgt:

$$V^{\top}(P_{s_i}) = \{v \in V_{s_i} \mid f(v) = s \wedge \exists\, u \xrightarrow{c} v \in E^{\bowtie}(P_{s_i})\}$$
$$V^{\perp}(P_{s_i}) = \{u \in V_{s_i} \mid f(u) = s \wedge \exists\, u \xrightarrow{c} v \in E^{\bowtie}(P_{s_i})\}$$

Die Eingangs- und Ausgangsknoten eines DACGs sind jeweils über die Vereinigung der Eingangs- und Ausgangsknoten der Partitionen definiert:

$$V^{\top}(G^c) = \bigcup_{s_i} V^{\top}(P_{s_i}) \qquad \text{und} \qquad V^{\perp}(G^c) = \bigcup_{s_i} V^{\perp}(P_{s_i})$$

5.4 Strategien zur Anfrage-Bearbeitung bei verteilter Rekursion

Es stellt sich nun die Frage, mit welcher Strategie die an den unterschiedlichen Speicherorten abgelegten Partitionen möglichst effizient abgerufen werden können. Das Ziel ist wiederum, den Kommunikationsaufwand so gering wie möglich zu halten.

[MGS+94] teilt die Auswertungsstrategien für verteilte *Assembly-Operationen* (entspricht den Multi-Level-Expansionen) in drei Kategorien ein:

Kategorie I

Strategien der Kategorie I erfordern, dass alle Daten unbearbeitet an eine zentrale (i.d.R. die anfragende) Stelle transportiert und dort – analog zum nicht-verteilten Fall – verarbeitet werden. Der Vorteil dieser Strategien liegt in der einfachen Implementierbarkeit.

Kategorie II

In die Kategorie II fallen Strategien, die zunächst den Zusammenbau der Teilstrukturen vor Ort durchführen und die Teilergebnisse an den Anfrager übertragen. Relativ einfach erweist sich die Kontrolle dieser Verfahren, da *ein* ausgezeichneter Master den kompletten Expansionsvorgang steuert. Kommunikation findet nur zwischen dem Master und den entfernten Servern statt, die entfernten Server selbst tauschen untereinander keine Daten aus.

Kategorie III

Strategien der Kategorie III ignorieren die Partitions-Grenzen und bauen mittels kaskadierender Aufrufe anderer Server komplette Teilbäume zusammen, die sie dann ihrem Aufrufer wieder zurückgeben. Für den Anfrager ist bei diesen Strategien der geringste Aufwand zu erwarten, insgesamt (über alle Server) kann jedoch – abhängig von der Qualität der Netze sowie der Verteilung der Daten – erhöhter Aufwand im Vergleich zu Verfahren der Kategorie II nötig werden.

Im Folgenden werden wir die zwei Vorgehensweisen aus Kategorie II und III (zentraler Auftraggeber und kaskadierende Remote-Aufrufe) sowie einen Mix daraus betrachten und bewerten.

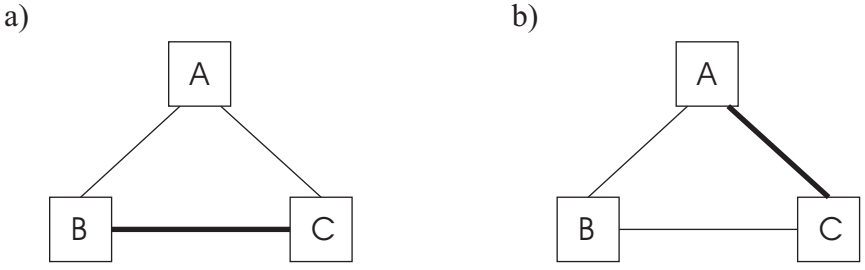


Abbildung 5.3: Entwicklungsverbund mit drei Standorten A, B und C sowie unterschiedlichen Netzqualitäten. Schnelle Netzverbindungen sind durch dickere Linien dargestellt, langsame durch dünne Linien.

In Abbildung 5.3 sind zwei Beispiele für einen kleinen Entwicklungsverbund mit drei Partnern A, B und C skizziert. Die Fälle a) und b) zeigen unterschiedliche Netzqualität zwischen den Standorten der Partner. Konform zu unserem Szenario aus Abschnitt 5.1 nehmen wir nun an, dass A Auftraggeber von B, und B wiederum Auftraggeber von C ist.

5.4.1 Verteilte Rekursion mit Anfrage-Master

Bei diesem Verfahren übernimmt ein Standort die Koordination der verteilten Rekursionsausführung. Dieser Master fragt sukzessive bei anderen Servern nach fehlenden Teilstrukturen an und erstellt somit nach und nach das Gesamtergebnis.

Prinzipiell kann jeder Server, der an dem Gesamtszenario beteiligt ist, die Rolle des Masters ausüben. Da jedoch vor der Ausführung der Rekursion nicht bekannt ist, welche Server für die Ermittlung des Ergebnisses tatsächlich angesprochen werden müssen, werden in der Praxis nur zwei Server für die Master-Rolle in Frage kommen:

1. Der Server, an welchem die Benutzeraktion gestartet wird, denn dort muss das Ergebnis letztendlich vorliegen.
2. Der Server, welcher das zu expandierende Objekt speichert, denn dieser Server ist an der Rekursionsberechnung beteiligt.

Für die weiteren Betrachtungen ist es unerheblich, welche Variante gewählt wird.

Die prinzipielle Anfragebearbeitung mit einem ausgezeichneten Anfrage-Master ist im Algorithmus 5.4.1 dargestellt.

Algorithmus 5.4.1: Expansion mit Anfrage-Master

ExpandMstr(x)

```
1  (V, E) ← Expand(x) at server storing x
2  while (∃ u -c-> v ∈ E | v ∉ V) do
3      for each (u -c-> v ∈ E | v ∉ V) in parallel do
4          (V_v, E_v) ← Expand(v) at server storing v
5          V ← V ∪ V_v
6          E ← E ∪ E_v
7      end do
8  end do
9  return (V, E)
```

Der Anfrage-Master ermittelt zunächst den Teilgraph in der lokal verfügbaren Partition (Zeile 1) bzw. am Server, der das zu expandierende Objekt speichert. Alle Serverübergänge in diesem Zwischenergebnis führen zu weiteren Aufrufen der Expand-Routine (Zeilen 3-7), welche nur die am jeweiligen Standort lokal verfügbare Partition traversiert. Solange weitere, noch nicht bearbeitete Serverübergänge vorhanden sind, werden die entfernten Aufrufe fortgesetzt. Am Ende enthält (V, E) den angefragten Graph.

In unserem Beispiel sei der Anfrage-Master der Standort A. Nach der lokalen Expansion ruft dieser Master erst Standort B, und nach Erhalt des zugehörigen Ergebnisses erfolgt noch der Auftrag an Standort C. Dessen Anfrage-Ergebnis enthält keine weiteren Serverübergänge, das Ergebnis ist somit komplett.

Wir wenden diese Vorgehensweise nun in den in Abbildung 5.3 dargestellen Netzen an. Im Falle a) muss über die beiden langsamen Leitungen von A nach B und von A nach C kommuniziert werden. Die qualitativ hochwertigere Verbindung zwischen den Standorten B und C kann nicht genutzt werden. Je nach Qualität der Verbindungen kann es vorteilhaft sein, das Ergebnis von C nach B und von dort gemeinsam mit dem Ergebnis von B nach A zu transportieren – dies würde einen Round Trip über die langsame Verbindung von A nach C einsparen. Im Fall b) wird sich die schnelle Verbindung zwischen A und C unmittelbar positiv auf die Gesamtantwortzeit auswirken.

Schon an diesem einfachen Beispiel wird deutlich, dass mit einer Anfrage-Bearbeitung nach dem Prinzip des Anfrage-Masters nicht in jedem Fall optimale Antwortzeiten garantiert werden können. Als weiteres Problem kommt hinzu, dass das Anfordern von verschiedenen Partitionen an ein und demselben Standort durch mehrmaliges Kommunizieren nicht ausgeschlossen werden kann. Man betrachte hierzu einen einfachen Fall: A ist Auftraggeber von B und D, die beide Auftraggeber von C sind. Dies führt dazu, dass am Standort C mindestens zwei Partitionen vorliegen, die bei einer standortübergreifenden Expansion durch mehrere isolierte Anfragen abgearbeitet werden müssen.

Fazit: Es ist offensichtlich, dass mit dieser Methode unser Ziel der möglichst minimalen Kommunikation zwischen den beteiligten Standorten nicht erreicht werden kann.

5.4.2 Verteilte Rekursion mit kaskadierenden Aufrufen

Alternativ zum Verfahren mit ausgezeichnetem Anfrage-Master können entfernte Aufrufe auch sukzessive von den Standorten aus gestartet werden, an denen die Serverübergänge auftreten. Algorithmus 5.4.2 zeigt das prinzipielle Vorgehen.

Algorithmus 5.4.2: Expansion eines verteilten Graphen mit kaskadierenden Remote-Aufrufen

ExpandCasc(x)
1 $(V, E) \leftarrow$ *Expand*(x) *at server storing* x
2 **for each** $(u \xrightarrow{c} v \in E | v \notin V)$ **in parallel do**
3 $(V_v, E_v) \leftarrow$ *ExpandCasc*(v) *at server storing* v
4 $V \leftarrow V \cup V_v$
5 $E \leftarrow E \cup E_v$
6 **end do**
7 **return** (V, E)

In diesem Verfahren spiegelt sich die rekursive Struktur der Expansion deutlich wider. An jedem Standort wird die angefragte Partition lokal expandiert (Zeile 1), falls dabei Serverübergänge erreicht werden, werden rekursiv Anfragen für die entfernt gespeicherten Partitionen gestartet. Die Ergebnisse dieser Anfragen werden mit dem lokalen Ergebnis zusammengeführt und an den Aufrufer zurückgegeben.

In unserem einfachen Beispiel (vgl. Abbildung 5.3) startet die Expansion am Standort A, von dort wird der Standort B gerufen, der seinerseits eine Anfrage an Standort C stellt. Die Ergebnisse werden auf umgekehrtem Weg zurückgeliefert.

Im Fall a) der Abbildung 5.3 wird bei diesem Verfahren nun die schnelle Verbindung zwischen den Standorten B und C genutzt. Sind die Verbindungen zwischen A und B sowie A und C in etwa identisch, so wird jedoch praktisch kein signifikanter Vorteil gegenüber dem Verfahren mit Anfrage-Master zu verzeichnen sein: Das Teilergebnis des Standortes C muss auf jeden Fall über eine relativ schlechte Verbindung zum Standort A übermittelt werden – entweder indirekt über den Standort B, oder direkt. Ist hingegen die Verbindung zwischen A und B deutlich besser als zwischen A und C, so ist durchaus eine zählbare Verbesserung zu erwarten.

Liegt Fall b) der Abbildung 5.3 vor, so wird die schnelle Leitung zwischen den Standorten A und C durch das hier betrachtete Verfahren nicht genutzt. Noch schlechter als im Fall a) muss nun das Teilergebnis von Standort C gleich zweimal über langsame Leitungen zum originären Anfrager (über B zu A) transportiert werden. Folglich bleiben die Antwortzeiten sicherlich über dem angestrebten Minimum.

Analog zu den Ausführungen in Abschnitt 5.4.1 kann auch durch kaskadierende Aufrufe das mehrmalige Ansprechen eines Servers zur Traversierung verschiedener dort gespeicherter Partitionen nicht verhindert werden. Besonders ineffizient wird das Kommunikationsverhalten, falls auf Grund von Mehrfachverwendungen Partitionen mehr als einmal referenziert und deshalb mehrfach angefragt werden.

Fazit: Auch kaskadierende Aufrufe entfernter Standorte führen nicht automatisch zu möglichst minimaler Kommunikation und damit einhergehend zu den angestrebten optimalen Antwortzeiten.

5.4.3 Verteilte Rekursion mit eingeschränkt kaskadierenden Aufrufen

Nachdem die beiden vorgestellten (Standard-)Vorgehensweisen nicht zu befriedigenden Ergebnissen führen, wollen wir hier noch eine Kombination der beiden Strategien betrachten.

Zur Erinnerung, die Schwachpunkte sowohl des Verfahrens mit Aufruf-Master als auch des kaskadierenden Aufrufes waren

1. Nichtbenutzung schneller Verbindungen, und
2. mehrfache Kommunikation mit einem Standort zur Traversierung mehrerer dort gespeicherter Partitionen.

Wir wollen uns nun zunächst mit der Benutzung schneller Verbindungen beschäftigen. In den Abschnitten 5.4.1 und 5.4.2 haben wir an den Beispielen gezeigt, dass in verschiedenen Situationen entweder das Verfahren mit Aufruf-Master oder aber der kaskadierende Aufruf Vorteile hinsichtlich des zu erwartenden Antwortzeitverhaltens besitzt. Nun stellt sich die Frage, ob entscheidbar ist, welches Verfahren zu einem gegebenen Szenario das günstigere ist, d. h. an jedem Standort muss individuell entschieden werden, ob ein Serverübergang selbst aufgelöst wird (d. h. kaskadierend entfernte Server gerufen werden), oder ob diese Auflösung von einem aufrufenden Standort aus erfolgen soll.

Um diese Entscheidung zu fällen, muss zunächst jeder Standort die Kosten kennen, die anfallen, wenn ein Standort S_1 mit einem anderen Standort S_2 kommuniziert. Diese Kosten können beispielsweise in einer (quadratischen) Matrix an jedem Standort vorgehalten werden. Zusätzlich muss ein Standort bei der Bearbeitung einer Anfrage im Kontext eines Expands wissen, welche anderen Standorte beim „Rücktransport“ der Teilergebnisse besucht werden (es handelt sich hierbei um die Standorte, die beim rekursiven Abstieg in andere Partitionen den kaskadierenden Ansatz verwendet haben) und folglich die Auflösung der noch ausstehenden Serverübergänge vornehmen könnten.

An einem Beispiel (vgl. Abbildung 5.3) wollen wir die Verwendung dieser Informationen erläutern: Standort A, bei dem die Anfrage gestartet wurde, habe bereits Standort B mit der Traversierung einer dort gespeicherten Partition beauftragt.

Dort wird nun ein Serverübergang zum Standort C ermittelt. Am Standort B kann nun entschieden werden, ob kaskadierend zum Standort C gewechselt wird, oder ob Standort A selbst den Auftrag an C stellen soll: Ist die Kommunikation von C via B nach A erwartungsgemäß günstiger als die Kommunikation von C direkt nach A, so verwende den kaskadierenden Ansatz, andernfalls überlasse den Aufruf dem Standort A.

Hier ist eine kleine, aber entscheidende Einschränkung zu vermerken: Zum einen sind die Kosten, die für die Kommunikation zwischen zwei Servern S_1 und S_2 anfallen, ständigen Schwankungen unterworfen: Tageszeitlich bedingt kann eine Leitung mehr oder weniger stark ausgelastet sein und damit mehr oder weniger hohe Kosten (im Sinne von Antwortzeiten) verursachen. Zum anderen hängen die Kosten auch von der zu transportierenden Datenmenge ab – und diese ist ja vor der Ausführung der entfernten Aktion noch nicht bekannt! Hier kann folglich allenfalls ein Schätzwert zu Hilfe genommen werden.

Das zweite angesprochene Problem bleibt leider auch bei der Kombination der beiden Verfahren erwartungsgemäß erhalten: Mehrfach-Aufrufe desselben entfernten Servers zur Anfrage verschiedener Partitionen sind nicht auszuschließen. Durch Zusammenfassen mehrerer gleichzeitig abzusetzender Aufträge können zwar sicherlich einige wenige Kommunikationen eingespart werden, im Allgemeinen jedoch werden für derartige Aufträge isolierte Anfragen erforderlich sein, da über die Verteilung des Graphs keine Informationen bekannt sind.

Fazit: Der Aufwand für die Bereitstellung und Aktualisierung der Informationen, die eine Kombination des Anfrage-Masters mit den kaskadierenden Aufrufen ermöglichen, ist hoch im Vergleich zu dem zu erwartenden verhältnismäßig geringen Benefit.

5.5 Integration der Regelauswertung

Die einzelnen Partitionen P_{s_i} eines DACGs können prinzipiell mit den gleichen Mitteln traversiert werden wie ein kompletter DACG, d. h. der Einsatz von rekursivem SQL, wie er in Kapitel 4 eingeführt wurde, ist auch hier sinnvoll.

Die frühzeitige Regelauswertung stößt im verteilten Anwendungsfall jedoch an ihre Grenzen: Row conditions können wie im lokalen Fall mittels Table Functions überprüft werden. ∀rows conditions, tree aggregate conditions und möglicherweise auch ∃structure conditions jedoch werfen Probleme auf, die wir im Folgenden erläutern und bewerten werden.

5.5.1 ∀rows conditions in verteilten Umgebungen

∀rows conditions werden im Abschnitt 4.4 *nach* der Rekursionsberechnung ausgewertet. Im verteilten Anwendungsfall bedeutet dies, dass bei einer verteilten Anfrage zunächst alle Teilergebnisse von den entfernten Partnern zum Anfrager übermittelt werden, und dann die ∀rows conditions überprüft werden können.[25]

Diese Strategie steht in einem gewissen Widerspruch zu dem Ziel, Regeln möglichst frühzeitig auszuwerten, um unnötige Datenübertragungen zu vermeiden: Liefert die Überprüfung der Regeln ein negatives Ergebnis, so wurden sämtliche Teilergebnisse vergeblich übertragen. Es stellt sich also die Frage, ob ∀rows conditions nicht dezentral, d. h. an jedem involvierten Standort lokal ausgewertet werden können, wobei am Standort des Anfrager letztlich die teilweise durchgeführten Tests nur noch zusammengefasst werden. Problematisch hierbei ist, dass das Ergebnis einer Anfrage, die auf Grund einer benutzerdefinierten Suchbedingung keine Treffer liefert, nicht unterschieden werden kann von einer Anfrage, deren Ergebnis auf Grund einer ∀rows condition keinen Treffer enthält! Im ersten Fall sind Ergebnisse von anderen Standorten gültig, im zweiten Fall sind diese ebenfalls zu invalidieren!

Im Falle des zentralen Anfrage-Masters bietet sich ein Ausweg an: Man dezentralisiert nicht die Evaluierung der Bedingung, sondern man führt den Test auf jedem eingehenden Teilergebnis beim Anfrage-Master aus! Sobald der Test ein negatives Ergebnis liefert, bricht dieser die gesamte Aktion einfach ab.

Da ∀rows conditions in der Praxis nicht sonderlich häufig auftreten, messen wir diesem Problem auch keine größere Bedeutung zu.

5.5.2 Tree aggregate conditions in verteilten Umgebungen

Die Auswertung von tree aggregate conditions gestaltet sich ähnlich problematisch. Prinzipiell lassen sich Aggregat-Funktionen in der Regel zwar parallelisieren, eine Bedingung kann jedoch im Allgemeinen nicht an Teilergebnissen getestet werden.

In einer Umgebung mit zentralem Anfrage-Master kann auch hier analog zu Abschnitt 5.5.1 ein „Ausweg“ gefunden werden: Der Anfrage-Master prüft bei jedem eingehenden Teilergebnis, ob die Bedingung auf dem bisherigen Zwischenergebnis erfüllt ist. Falls ja, so kann der nächste Teilauftrag an einen entfernten Standort gesendet werden. Falls nein, so muss überprüft werden, ob die Bedingung durch Hinzunahme noch fehlender Teilergebnisse möglicherweise erfüllt werden kann!

[25] Man beachte, dass dieser Test nicht notwendigerweise von einem Datenbanksystem durchgeführt werden muss. Hier kann auch der PDM-System-eigene Regelauswerter eingesetzt werden.

Beispiele: (1) Eine Bedingung besage, dass der Durchschnitt (AVG) der Werte eines Attributes eine Grenze k nicht überschreiten darf. Nach Erhalt des ersten Teilergebnisses liegt der Durchschnitt über k. Die Aktion darf aber zu diesem Zeitpunkt nicht abgebrochen werden, da noch nicht angeforderte Teilergebnisse den Durchschnitt unter k senken können. (2) Eine Bedingung besage, dass nicht mehr als l Objekte im Ergebnis sein dürfen (COUNT). Nach Erhalt des dritten Teilergebnisses sind bereits mehr als l Objekte eingegangen. Da die Bedingung nicht mehr erfüllt werden kann, wird die Aktion abgebrochen.

Auch die tree aggregate conditions sind nicht besonders praxisrelevant, wir stufen dieses Problem als sehr geringfügig ein.

5.5.3 ∃structure conditions in verteilten Umgebungen

Im Abschnitt 3.2.3.2 haben wir zwei Versionen von ∃structure conditions eingeführt. Variante (1) prüft für ein Objekt o_1 nur das Vorhandensein eines in Beziehung stehenden Objektes o_2, Variante (2) hingegen prüft zusätzlich eine row condition auf o_2.

Die Kanten-Objekte, die zwei Knoten-Objekte auf verschiedenen Partitionen verbinden, werden in beiden beteiligten Partitionen gespeichert (vgl. Definition 4). Verweisen derartige Kanten-Objekte nur auf Knoten-Objekte *eines* Objekt-Typs, so können Bedingungen der Variante (1) damit lokal überprüft werden, selbst falls sich das Objekt o_2 an einem anderen Standort befindet. Andernfalls kann nicht sichergestellt werden, dass das referenzierte Objekt tatsächlich den in der Zugriffsregel geforderten Typ besitzt!

Für Bedingungen der Variante (2) ist eine lokale Prüfung der row condition auf o_2 ausgeschlossen. Eine Kommunikation mit dem entfernten Standort ist deshalb nötig.

Für PDM-Systeme können wir eine entscheidende Einschränkung hinsichtlich der Verwendung von ∃structure conditions vornehmen. Derartige Bedingungen werden in Zugriffsregeln ausschließlich dazu verwendet, um die Existenz *produktbeschreibender Dokumente*, z. B. Spezifikationen, CAD-Modelle, Berechnungen etc., sowie möglicherweise deren Eigenschaften zu überprüfen. Alle diese Dokumente befinden sich jedoch im Zuständigkeitsbereich des Partners, welcher auch für den zugehörigen Produktstrukturknoten (Zusammenbau oder Einzelteil etc.) verantwortlich ist. Das bedeutet, dass in PDM-Systemen die ∃structure conditions sowohl in Variante (1) als auch in Variante (2) *lokal* überprüft werden können!

Beim Einsatz von ∃structure conditions in verteilten PDM-Systemen sind also keine Probleme zu erwarten.

5.6 Abschließende Bewertung

Bei den in Abschnitt 5.4 beschriebenen Verfahren entsteht das Problem, dass besonders bei in der Realität relevanten Szenarien für die Bearbeitung einer rekursiven Aktion ein oder mehrere Standorte mehrfach kontaktiert werden müssen, um sämtliche benötigten Teilergebnisse dort abzuholen. Dadurch kann das Ziel einer möglichst minimalen Antwortzeit nicht erreicht werden.

Die Ursache für dieses Manko liegt darin, dass *keinerlei Information über die Verteilung* der rekursiven Struktur bekannt ist. Dies führt dazu, dass erst während der Traversierung der diversen Partitionen festgestellt wird, welche weiteren Partitionen und damit welche Standorte noch angefragt werden müssen. In Kapitel 6 wird dieses Problem durch Ermittlung und Nutzung entsprechender Zusatzinformationen angegangen.

Kapitel 6

Der Object-Link-and-Location-Katalog — Effizienzsteigerung durch Verteilungsinformation

6.1 Szenario: Anfragebearbeitung bei zyklisch verteilter Datenhaltung

Analog zu Kapitel 5 wird auch hier ein Unternehmen betrachtet, dessen Entwicklungs- und Fertigungsabteilungen weltweit verteilt sind. Die Datenhaltung erfolgt dezentral an den verschiedenen Standorten, auch Zulieferer speichern die von ihnen erzeugten Daten selbst.

Im Gegensatz zu Kapitel 5 sind nun auch zyklische Abhängigkeiten der Partner und Zulieferer zugelassen, d. h. die strenge Hierarchie Auftraggeber – Auftragnehmer entfällt: Eine Entwicklungsabteilung E_1, die eine andere Entwicklungsabteilung E_2 beauftragt, kann selbst von E_2 wiederum beauftragt werden.

Derartige zyklische Beauftragungen sind in der Praxis recht häufig anzutreffen. Oftmals wird ein Zulieferer beauftragt, einen Teil der Produktstruktur eigenverantwortlich konstruktiv umzusetzen, wobei er unter anderem vorgegebene Einzelteile oder kleinere Zusammenbauten des Auftraggebers einzusetzen hat. Oftmals erzwingt auch ein vertraglich vereinbarter Partner-Workshare aus politischen Gründen[26] eine zyklische Verteilung der Produktstruktur (vgl. Abbildung 6.1).

[26] z. B. in militärischen Projekten soll der „Alleingang“ eines Partners durch zu starke Bündelung des Know-Hows in einem Bereich verhindert werden.

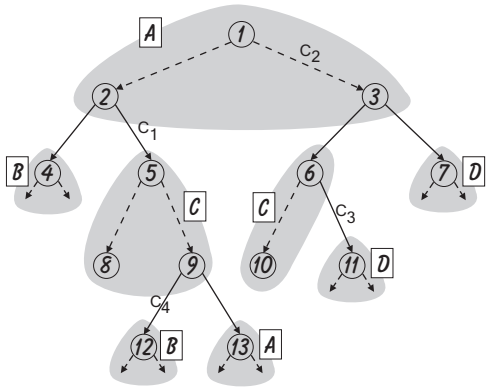


Abbildung 6.1: Ein primitiver, zyklisch verteilter DACG: Standort A speichert eine Partition (mit Knoten 13), die zyklisch von einer anderen Partition (mit Knoten 1) am gleichen Standort abhängt.

6.2 Problematik und Lösungsansatz

Die im Abschnitt 5.2 erläuterten Probleme hinsichtlich der Verteilung der Rekursionsausführung sind naturgemäß auch in diesem Szenario vorhanden, ebenso das Problem, dass ein entfernter Server mehrfach kontaktiert werden muss, um alle erforderlichen Partitionen zu übertragen. Erschwerend kommt nun hinzu, dass vor allem der in Abschnitt 5.4.2 beschriebene Ansatz mit kaskadierenden Aufrufen auf Grund der möglichen Zyklen extrem ineffizient werden kann: Besteht eine zyklische Abhängigkeit der Form $S_1, S_2, \ldots, S_{i-1}, S_i, S_1$, so würden Teilergebnisse „im Kreis herum“ übertragen werden: von S_1 über $S_i, S_{i-1}, \ldots, S_2$ zurück nach S_1. Es ist unschwer zu erkennen, dass hier Optimierungsbedarf besteht.

Die prinzipielle Lösung besteht darin, an jedem Standort *zusätzliche Information* über die Verteilung der Daten bereitzustellen, um lokal alle in der Benutzeraktion involvierten Partitionen (und damit alle beteiligten Server) mit den dort anzufragenden Toplevel-Knoten zu ermitteln, ohne auf Zwischenergebnisse entfernter Standorte angewiesen zu sein. Dabei müssen Zugriffsrechte und Konfigurationseinstellungen etc. berücksichtigt werden. Fasst man dann alle Anfragen, die an den gleichen entfernten Server gestellt werden sollen, zu einer einzigen Anfrage zusammen, so kann garantiert werden, dass jeder entfernte Standort im Entwicklungsverbund maximal einmal angesprochen wird.

Die Iteration über die Sequenz *Anfrage stellen – Zwischenergebnis interpretieren – evtl. weitere Anfrage stellten – etc.* entfällt folglich. Stattdessen können alle Anfragen, die an entfernte Server gestellt werden, *parallel* ausgeführt werden – eine weitere Verkürzung der Antwortzeiten ist die Folge.

Der einfachste Weg, die angedeutete Zusatzinformation zu ermitteln, wäre die Berechnung und Speicherung der transitiven Hülle des DACGs, oder in der Terminologie der Produktstrukturen ausgedrückt, die transitive Hülle der *uses*- und *has_revision*-Beziehungen. Diese Methode ist jedoch auf Grund der großen Datenmenge und mangelnder Flexibilität (man denke nur an eine Erweiterung der Produktstruktur) in der Praxis untauglich. Außerdem sind in der transitiven Hülle auch Informationen enthalten, die nicht dem gewünschten Zweck dienen, z. B. Kanten zwischen indirekt erreichbaren Knoten, die der gleichen Partition angehören. Es genügt daher, aus der transitiven Hülle die Kanten herausnehmen, die zur Lösung des Problems beitragen, und diese in einem sogenannten Object-Link-and-Location Katalog, kurz OLL-Katalog, zu speichern.

6.3 Definition OLL-Katalog

Die Herausforderung besteht nun darin, den OLL-Katalog so zu konstruieren, dass er möglichst einfach zu handhaben ist, möglichst wenig Einträge besitzt und dennoch vollständig ist.

Die einfache Handhabbarkeit gilt im Sinne von Effizienz sowohl hinsichtlich Initialisierung und Aktualisierung, ganz besonders jedoch im Hinblick auf die Unterstützung der rekursiven Anfragen. Die Größe des Katalogs ist kein Hauptkriterium, sollte sich jedoch auf Grund der ohnehin enormen Datenmengen möglichst in Grenzen halten, d. h. unnötige Einträge (z. B. doppelt auftretende Einträge) sollten möglichst vermieden werden. Die Forderung nach Vollständigkeit erklärt sich daraus, dass jeder Standort bei der Abarbeitung einer rekursiven Benutzeraktion *garantiert maximal einmal* kontaktiert werden soll, und hat folglich absoluten Vorrang vor allen anderen Anforderungen. Basierend auf diesen Überlegungen definieren wir den OLL-Katalog wie folgt:

Definition 6: OLL-Katalog

> Sei $G^c = (V, E, \mathcal{C})$ ein DACG und $f : V \to S$ eine Funktion zur Markierung von Knoten. Des Weiteren sei $G^{c*} = (V, E^*, \mathcal{C}^*)$ die transitive Hülle von G^c. Dann ist der *OLL-Katalog* E' von G^c definiert als:
>
> $E' = \{u \xrightarrow{c} v \in E^* \setminus E | \exists\, d = \langle u \xrightarrow{c_1} n_1 \xrightarrow{c_2} \ldots \xrightarrow{c_k} n_k \xrightarrow{c_{k+1}} v\rangle \text{ in } G^c,$
> $(k > 0), \text{mit } f(n_i) \neq f(u) \wedge f(n_i) \neq f(v), 1 \leq i \leq k$
> $\text{und } c = \bigwedge_{i=1}^{k+1} c_i\}.$

Mit anderen Worten: Der OLL-Katalog enhält eine „virtuelle“ Kante zwischen den Knoten u und v, falls im DACG u und v keine direkten Nachbarn sind und

v von u erreichbar ist, indem ausschließlich Knoten besucht werden, deren Markierungen ungleich derer der Knoten u und v sind. Wenn die Funktion f (analog zu Abschnitt 5.3) wiederum jedem Knoten den Speicherort zuweist, dann liegen folglich alle Knoten auf dem Pfad von u nach v an anderen Speicherorten als u und v selbst.

Abbildung 6.2 zeigt den DACG aus Abbildung 6.1, erweitert um die Kanten des zugehörigen OLL-Katalogs: $2 \overset{c_1 \wedge c_4}{\rightarrow} 12$, $2 \overset{c_1}{\rightarrow} 13$ und $3 \overset{c_3}{\rightarrow} 11$.

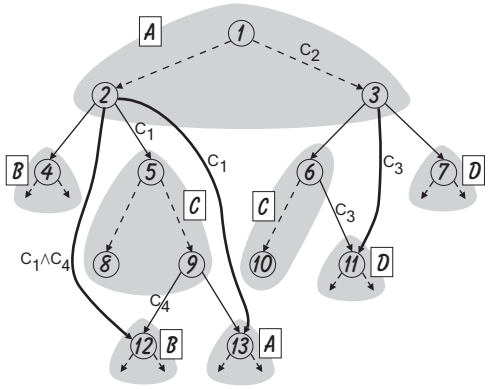


Abbildung 6.2: Der DACG G^c aus Abbildung 6.1 zusammen mit OLL-Katalogeinträgen

Der Beweis für die Vollständigkeit des aus dieser Definition resultierenden OLL-Katalogs findet sich in Abschnitt 6.9.

Um die verteile Rekursionsberechnung mit dem OLL-Katalog zu beschleunigen, muss nicht an jedem Standort der komplette Katalog vorhanden sein. Es ist völlig ausreichend, wenn jeweils die Kanten des OLL-Katalogs vorgehalten werden, deren Quell- oder Zielknoten im lokalen Teilgraph enthalten sind. Die jeweiligen OLL-(Teil-)Kataloge der beteiligten Standorte werden demnach wie folgt definiert:

Definition 7: Verteilter OLL-Katalog

Sei G^c ein DACG verteilt auf die Standorte $s \in S$.
Sei ferner $P_s = (V_s, E_s, \mathcal{C})$ der Teilgraph bei s.
Dann ist der OLL-Katalog E'_s am Standort s definiert als

$$E'_s = \{u \overset{c}{\rightarrow} v \in E' | u \in V_s \vee v \in V_s\}$$

6.4 Prinzipielle Verwendung des OLL-Katalogs

Einträge im OLL-Katalog entsprechen in ihrer Form den Kanten des DACGs und können deshalb auch auf ähnliche Art verarbeitet werden. In den beiden folgenden Abschnitten wird zunächst an dem Beispiel aus Abbildung 6.2 gezeigt, wie der OLL-Katalog in rekursiven Anfragen genutzt werden kann. Anschließend werden die Prinzipien verallgemeinert.

6.4.1 Verwendung des OLL-Katalogs am Beispiel

Im Beispiel in Abbildung 6.2 ist die Produktstruktur auf vier verschiedene Server (gleichbedeutend mit Standorten) verteilt. Am Server A liegt unter anderem der Knoten 1, der das Toplevel-Element der gesamten Struktur darstellt. Dieser Knoten soll expandiert werden unter der Annahme, dass c_1=c_2=c_3=c_4=$\top$, d. h. alle Partitionen des DACGs müssen traversiert werden. Die folgenden Schritte beschreiben dieses Vorgehen:

Schritt 1: Zunächst expandiert der Server A die Struktur lokal, beginnend bei Knoten 1: Die Kanten $1 \rightharpoonup 2$ und $1 \rightharpoonup 3$ werden in das Ergebnis übernommen und interpretiert. Da die Knoten 2 und 3 lokal vorliegen, können sie ebenfalls unmittelbar abgerufen werden. Damit sind alle von Knoten 1 ausgehenden Kanten bearbeitet (vgl. Abbildung 6.3a).

Schritt 2: Nun wird der Knoten 2 expandiert. Die zugehörigen Kanten aus dem DACG werden aus der lokalen Partition ermittelt, jedoch zeigen diese Kanten auf die Knoten 4 und 5 an den Servern B bzw. C. Anstatt diese Knoten nun sofort von den entfernten Standorten anzufordern, sammelt A diese für die spätere Bearbeitung in einer Menge nicht-aufgelöster Referenzen (vgl. Abbildung 6.3b).

Schritt 3: An dieser Stelle kommt der OLL-Katalog ins Spiel: Er enthält zwei Einträge, die vom Knoten 2 aus auf die Knoten 12 und 13 an den Servern B und A zeigen. Die Bedingungen, die mit diesen Einträgen verknüpft sind, evaluieren beide zu *wahr*, somit sind beide OLL-Katalogeinträge im Weiteren zu berücksichtigen.

Kanten im OLL-Katalog, deren Zielknoten in einer lokal gespeicherten Partition liegen, werden sofort bearbeitet, d. h. die Expansion wird am Zielknoten fortgesetzt. Damit wird sichergestellt, dass alle lokal gespeicherten Partitionen, die von dem betrachteten Startknoten (im Beispiel Knoten 1) indirekt erreichbar sind, in einem einzigen rekursiven Abstieg durchlaufen und expandiert werden. Zielknoten, die in Partitionen an anderen Standorten liegen, werden zur Menge der nicht-aufgelösten Referenzen hinzugefügt. Server A also fährt mit der Expansion des

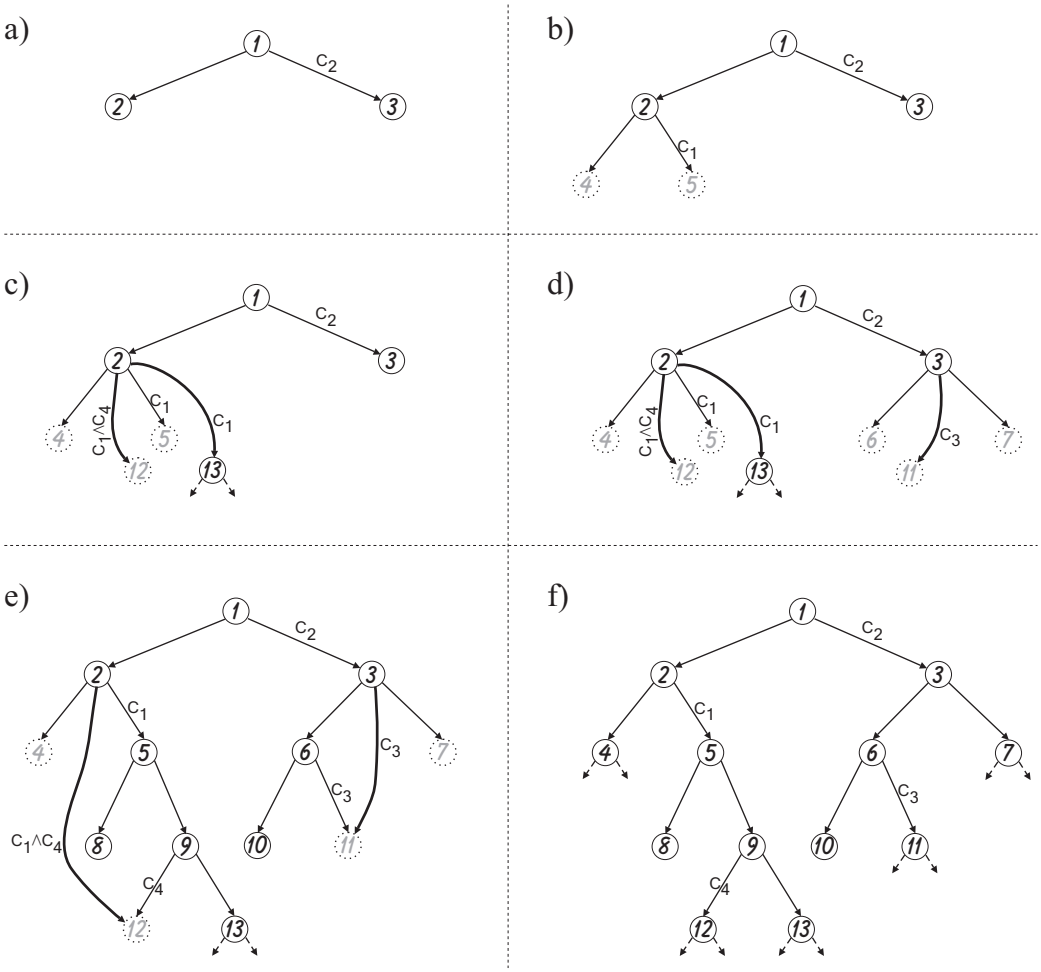


Abbildung 6.3: Sukzessiver Ergebnis-Aufbau einer Expansion des Knotens 1 am Server A in mehreren Schritten

Knotens 13 fort in der gleichen Weise, wie für Knoten 2 beschrieben, und Knoten 12 wird in die Menge der nicht-aufgelösten Referenzen aufgenommen (vgl. Abbildung 6.3c).

Schritt 4: Nachdem nun Server A die Expansion von Knoten 13 beendet hat, wird mit Knoten 3 fortgefahren. Alle Kanten sowohl im DACG als auch im OLL-Katalog mit Startknoten 3 zeigen auf Knoten in entfernten Partitionen. Entsprechend werden alle diese Zielknoten zu der Menge der nicht-aufgelösten Referenzen hinzugefügt (vgl. Abbildung 6.3d).

Zu diesem Zeitpunkt hat der Server A sämtliche lokal verfügbaren Informationen ausgewertet. Damit endet die lokale Rekursion am Server A. Die Menge der nicht-

aufgelösten Referenzen enthält zu diesem Zeitpunkt die Knoten 4, 5, 6, 7, 11 und 12.

Nun können die entfernten Server mit der lokalen Expansion der fehlenden Anteile der Struktur beauftragt werden: Server B muss die Teilgraphen beginnend bei den Knoten 4 und 12 liefern, Server C die Teilgraphen mit Startknoten 5 und 6, und Server D liefert die Teilgraphen mit den Startknoten 7 und 11. Man bemerke, dass jeder entfernte Server nur noch einmal kontaktiert werden muss, um die lokale Expansion dort anzustoßen!

Schritt 5: Die entfernten Server arbeiten nun alle nach dem gleichen Muster, es genügt deshalb nur die Betrachtung der lokalen Expansion am Server C. Dieser Server muss nun die Knoten 5 und 6 expandieren. Ohne Beschränkung der Allgemeinheit soll mit Knoten 5 begonnen werden, die Expansion des Knotens 6 erfolgt entweder im Anschluss daran oder parallel.

Die Kanten in der lokalen Partition des DACGs mit Startknoten 5 verweisen auf die Knoten 8 und 9. Knoten 8 ist ein Blattknoten und daher nicht weiter expandierbar. Der Knoten 9 jedoch referenziert die Knoten 12 und 13 an den Servern B und A. Knoten 13 wurde schon von Server A bearbeitet, und der Knoten 12 wird parallel am Server B expandiert. Somit sind – und das ist schließlich das Ziel des Verfahrens – keine weiteren Aufrufe an andere entfernte Server erforderlich, aus der Sicht des Servers C stoppt die Expansion des Knotens 5 hier. Alle traversierten Knoten und Kanten, inklusive der Serverübergänge $9 \rightharpoonup 12$ und $9 \rightharpoonup 13$, werden zusammen mit den Teilergebnissen der eventuell vorhandenen weiteren lokalen Expansionen (hier: Knoten 6) an den Aufrufer zurückgegeben (vgl. Abbildung 6.3e).

Nachdem alle Teilergebnisse von den beauftragten entfernten Servern am Server A eingegangen sind, liegt dort die komplette Struktur vor (vgl. Abbildung 6.3f).

Bemerkung: Da für den Endbenutzer die OLL-Katalogeinträge irrelevant sind und diese folglich auch nicht visualisiert werden müssen, können sie nach der Traversierung aus dem Endergebnis ohne Informationsverlust wieder entfernt werden.

Nähere Details zu dem Verfahren finden sich in den Algorithmen in Abschnitt 6.5.

6.4.2 Verallgemeinerung

Nicht alle DACGs sind ähnlich primitiv wie der in Abbildung 6.1 dargestellte Graph. Repräsentiert ein DACG eine Produktstruktur, so werden in der Regel deutlich komplexere Graphen vorliegen. Einige ausgewählte Pfade werden den OLL-Katalog zunächst etwas veranschaulichen.

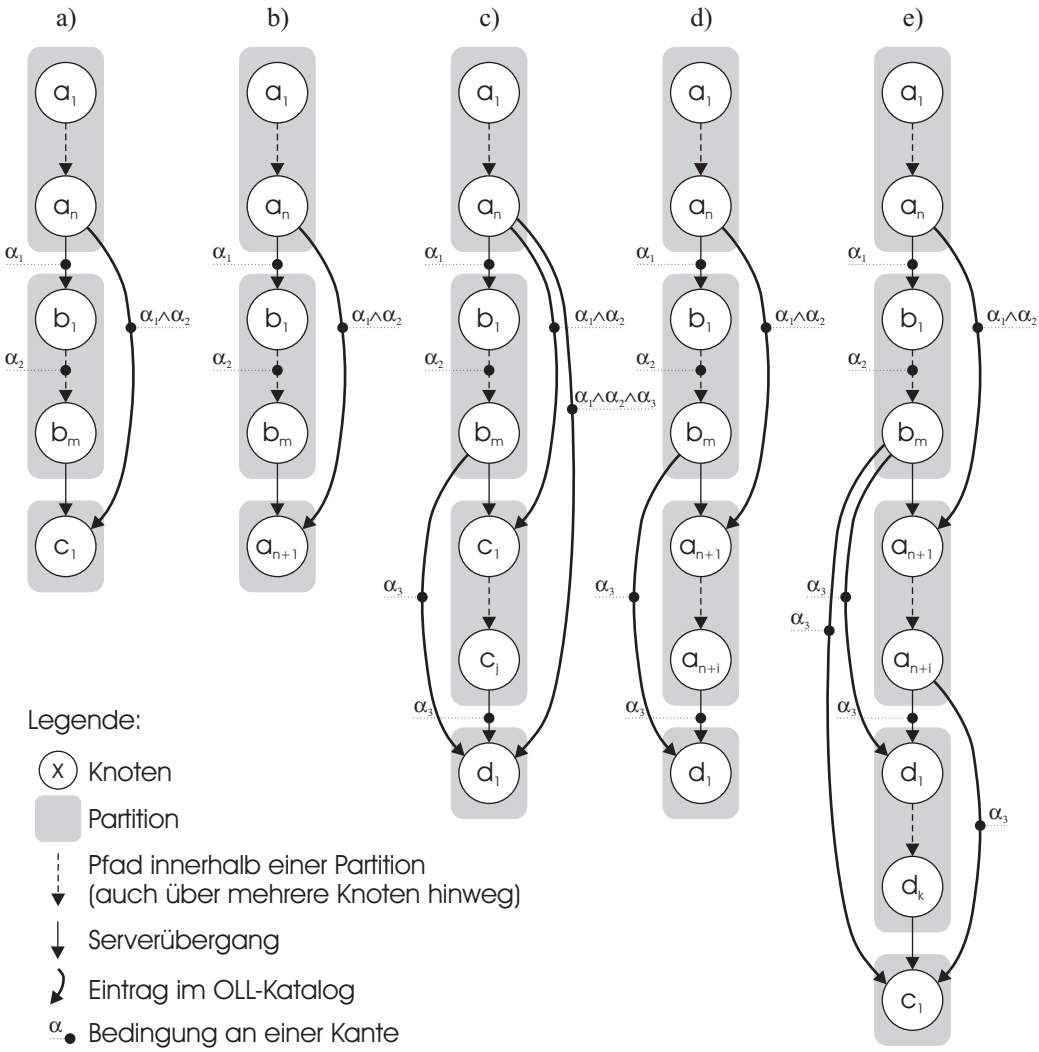


Abbildung 6.4: Charakteristische Pfade mit OLL-Katalogeinträgen

In Abbildung 6.4 sind in den Teilbildern a) bis e) fünf typische Pfade dargestellt, wie sie in einem DACG auftreten können. Im Folgenden seien alle Knoten a_i am Server A gespeichert, alle Knoten b_i am Server B usw.

Der einfachste Fall ist im **Teilbild a)** dargestellt: Am Server A existiert ein Pfad $d_A = \langle a_1 \rightharpoonup \ldots \rightharpoonup a_n \rangle$. Der Knoten a_n hat einen direkten Nachfolger b_1 am Server B, welcher unter der Bedingung α_1 erreicht wird. Dort existiert ein lokaler Pfad $d_B = \langle b_1 \rightharpoonup \ldots \overset{\alpha_2}{\rightharpoonup} \ldots \rightharpoonup b_m \rangle$ sowie ein Serverübergang zum Knoten c_1 am Server C.

Für den kompletten Pfad $d = \langle a_1 \rightharpoonup \ldots \rightharpoonup c_1 \rangle$ existiert ein Teilpfad von a_n nach

c_1, welcher den Kriterien für einen Eintrag in den OLL-Katalog genügt: a_n hat einen direkten Nachfolger in einer entfernten Partition, c_1 hat einen direkten Vorgänger in einer entfernten Partition, und sämtliche Knoten auf dem Pfad zwischen a_n und c_1 liegen in einer Partition auf dem Server B, also nicht auf A und C. Um vom Knoten a_n ausgehend den Knoten c_1 zu erreichen, müssen die Bedingungen α_1 und α_2 erfüllt sein. Folglich ist $a_n \overset{\alpha_1 \wedge \alpha_2}{\rightharpoonup} c_1$ ein Eintrag des OLL-Katalogs. Entsprechend der Definition 7 gehört dieser Eintrag sowohl zum OLL-Teilkatalog E'_A als auch zu E'_B.

Ein ähnlicher Fall ist in **Teilbild b)** dargestellt. Der Pfad verläuft dort lediglich nicht vom Server B zum Server C, sondern zyklisch zurück zu A. Damit lautet der Eintrag des OLL-Katalogs $a_n \overset{\alpha_1 \wedge \alpha_2}{\rightharpoonup} a_{n+1}$. Da sowohl a_n als auch a_{n+1} auf dem Server A liegen, gehört dieser Eintrag nur zum OLL-Teilkatalog E'_A.

Teilbild c) greift den Pfad aus a) nocheinmal auf und erweitert diesen um einen weiteren Serverübergang $c_j \overset{\alpha_3}{\rightharpoonup} d_1$ zum Server D. Neben dem bereits bekannten Teilpfad von a_n zu c_1 existieren nun zwei weitere Teilpfade, welche zu Einträgen im OLL-Katalog führen: $\langle a_n \overset{\alpha_1}{\rightharpoonup} \ldots \overset{\alpha_2}{\rightharpoonup} \ldots \overset{\alpha_3}{\rightharpoonup} d_1 \rangle$ ergibt Eintrag $a_n \overset{\alpha_1 \wedge \alpha_2 \wedge \alpha_3}{\rightharpoonup} d_1$ in E'_A und E'_D, und $\langle b_m \rightharpoonup \ldots \overset{\alpha_3}{\rightharpoonup} d_1 \rangle$ ergibt $b_m \overset{\alpha_3}{\rightharpoonup} d_1$ in den beiden OLL-Teilkatalogen E'_B und E'_D.

Im **Teilbild d)** wird der Pfad aus b) um einen Serverübergang $a_{n+i} \overset{\alpha_3}{\rightharpoonup} d_1$ erweitert. Durch den Zyklus auf dem Server A entfällt im Vergleich zu Teilbild c) der OLL-Katalogeintrag $a_n \overset{\alpha_1 \wedge \alpha_2 \wedge \alpha_3}{\rightharpoonup} d_1$, da der Pfad zwischen diesen beiden Knoten über eine Partition auf dem Server A mit den Knoten $a_{n+1}, \ldots, a_{n+i}$ läuft. Aber auch ohne diesen Eintrag kann am Server A durch Auswertung lokal verfügbarer Informationen festgestellt werden, dass es einen Pfad von a_1 zu d_1 geben muss: von a_1 nach a_n in einer lokalen Partition, über den OLL-Katalogeintrag $a_n \overset{\alpha_1 \wedge \alpha_2}{\rightharpoonup} a_{n+1}$ gelangt man zur nächsten lokalen Partition, in der auch der Knoten a_{n+i} liegt, welcher der Startknoten des Serverübergangs nach d_1 ist. Folglich wäre ein entsprechender OLL-Katalogeintrag von a_n zu d_1 überflüssig.

Den komplexesten Fall zeigt **Teilbild e)** auf. Hier wurde der Pfad aus d) nocheinmal erweitert um einen Serverübergang $d_k \rightharpoonup c_1$. Vom Knoten a_{n+i}, dem letzten Knoten einer Partition in der zyklischen Abhängigkeitskette am Server A, geht wiederum ein OLL-Katalogeintrag aus hin zu c_1, nicht aber von a_n aus. Der Knoten a_n hat folglich keine Relevanz mehr hinsichtlich neuer OLL-Katalogeinträge, deren Zielknoten auch von a_{n+i} aus erreichbar sind.

Der Knoten b_m am Server B ist von dem Zyklus auf A unabhängig (auch wenn b_m im Pfad innerhalb des Zyklus vorkommt), der Teilpfad $\langle b_m \rightharpoonup \ldots \overset{\alpha_3}{\rightharpoonup} \ldots \rightharpoonup c_1 \rangle$ entspricht den Kriterien der Definition des OLL-Katalogs und führt folglich zum Eintrag $b_m \overset{\alpha_3}{\rightharpoonup} c_1$ in E'_B und E'_C.

6.5 Algorithmen für Aufbau, Pflege und Nutzung von OLL-Katalogen

Die in diesem Abschnitt präsentierten Algorithmen zur Initialisierung, Erweiterung und Nutzung von OLL-Katalogen arbeiten auf einem DACG $G^c = (V, E, \mathcal{C})$. Zur Beschreibung der Algorithmen werden die folgenden Notationen verwendet:

S	ist eine Menge von Servern (Standorten), die eine oder mehrere Partitionen von G^c speichern.
$f(x)$	ist eine Funktion $f : V \rightarrow S$, die jedem Knoten des Graphs einen Speicherort zuweist.
$z = \langle a, b, c \rangle$	beschreibt ein Tripel z bestehend aus den Elementen a, b und c. Auf die Elemente kann mit $z.a$, $z.b$ und $z.c$ zugegriffen werden.
$g^{(f(v))}(x)$	bedeutet, dass die Funktion $g(x)$ an dem Server ausgeführt wird, an dem der Knoten v gespeichert ist.
$s[z.x \leftarrow y]$	beschreibt eine Änderungsoperation auf allen Objekten z in der Menge s. Dabei wird der Wert des Attributs x auf den Wert y gesetzt.
OLL	bezeichnet den OLL-Katalog am aktuellen (ausführenden) Server (das ist der Server, welcher aktuell die Prozedur mit Zugriff auf den OLL-Katalog ausführt).
$\mathcal{A}$	beschreibt die gewählten Strukturoptionen und Gültigkeiten, wie sie im Abschnitt 2.5.2.2 eingeführt wurden.

6.5.1 Initialisierung

Ausgangslage und Zielsetzung

Ausgangssituation im Folgenden sei ein DACG G^c, der in mehreren Partitionen P_{s_i} auf verschiedenen Servern $s \in S$ verteilt gespeichert ist. An allen Standorten soll noch kein OLL-Katalog vorhanden sein.

Ziel der Initialisierung ist es, mittels Graph-Traversierung für jeden Standort s den zugehörigen OLL-Katalog E'_s mit den Eigenschaften aus der Definition 7 zu ermitteln und vor Ort bereitzustellen.

Das Verfahren

Der Initialisierungsvorgang läuft rekursiv ab und ist verteilt über alle Server, die mindestens eine Partition von G^c speichern. Um Katalog-Einträge zu erzeugen, ist die komplette Struktur *depth-first* zu durchlaufen, beginnend bei dem Server,

der das Toplevel-Element der Struktur speichert. Bei der Traversierung eines Serverübergangs $x \xrightarrow{c} y$ wird die Initialisierung an dem Server fortgesetzt, an dem der Knoten y gespeichert ist.

Zur Erinnerung: Die Definition des OLL-Katalogs besagt, dass für jeden Eintrag $u \xrightarrow{c} v$ gilt: u hat einen direkten Nachfolger in einer entfernten Partition, v hat einen direkten Vorgänger in einer entfernten Partition, alle zwischen u und v zu durchlaufenden Partitionen sind entfernt, und v ist kein direkter Nachfolger von u. Bei der Initialisierung sind nun genau die Knotenpaare u und v zu finden, deren „Verbindungspfad“ $\langle u \xrightarrow{c_1} n_1 \xrightarrow{c_2} \ldots \xrightarrow{c_k} n_k \xrightarrow{c_{k+1}} v \rangle$ diese Eigenschaft besitzt.

Dabei tritt folgendes Problem auf: Besucht man während eines Graph-Durchlaufes einen Knoten u, der Startknoten eines OLL-Katalogeintrages sein kann, so hat man zunächst keine Information darüber, ob in tieferen Leveln ein passender Knoten v vorkommt (das ist letztendlich ja auch die Information, die im OLL-Katalog gespeichert werden soll!). Eine einfache Lösung hierfür wäre, für jedes besuchte u den darunter befindlichen Teilbaum zu traversieren und nach passenden Knoten v_i zu suchen. Das bedeutet jedoch insbesondere bei einer Aufsplittung des Graphs auf sehr viele Partitionen, dass Teilbäume mehrfach traversiert werden müssen, um die gesamte Struktur abzuarbeiten.

Da dieses Verfahren äußerst ineffizient ist, verfährt der Initialisierungsalgorithmus quasi umgekehrt: Beim Besuch eines Knotens v, der über einen Serverübergang erreicht wurde, wird getestet, ob der Pfad d beginnend beim Toplevel-Element bis hinunter zu v einen oder mehrere passende Knoten u_i enthält! Dieser Test lässt sich prinzipiell recht einfach realisieren:

Es müssen nicht alle Knoten in d berücksichtigt werden, sondern lediglich jene, die mindestens einen direkten Nachfolger in einer entfernten Partition besitzen (Partitions-Ausgangsknoten, siehe Definition 5). Diese Menge wird für die folgenden Betrachtungen mit $\mathcal{B}$ bezeichnet[27]. Um aus d diejenigen Knoten $u_j \in \mathcal{B}$ herauszufiltern, die Ausgangspunkt einer neuen Kante $u_j \rightharpoonup v$ im OLL-Katalog sind, wird d *rückwärts* traversiert, beginnend beim Knoten v. Hier genügt es nun, den jeweils *zuerst* erreichten Knoten u_s eines jeden Standortes $s \in S$ zu ermitteln – dies stellt sicher, dass alle anderen Knoten im Pfad $\langle u_s \rightharpoonup \ldots \rightharpoonup v \rangle$ auf anderen Standorten liegen müssen. Damit ist ein Pfad mit der gewünschten Eigenschaft gefunden. Dieses Rückwärts-Traversieren wird vorzeitig abgebrochen, falls erneut ein Knoten $u_{f(v)}$ besucht wird, d. h. falls zyklische Verteilung der Daten vorliegt. Die bis dahin gefundenen Knoten u_s (einschließlich $u_{f(v)}$) entsprechen folglich den gesuchten Knoten u_j. Lediglich der direkte Vorgänger von v muss hiervon noch ausgeschlossen werden.

[27] $\mathcal{B}$ als Symbol für „border nodes“, also Knoten, die an der „Grenze“ zu einer anderen Partition liegen und dort Nachfolger besitzen.

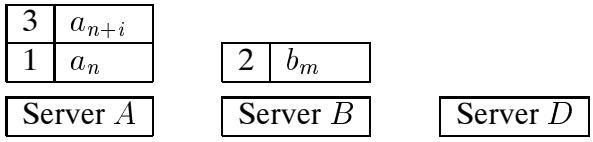


Abbildung 6.5: Stacks der Server A, B und D für Abbildung 6.4d) mit $v = d_1$

Als Beispiel setzen wir dieses Verfahren für den Pfad d) in Abbildung 6.4 ein: Sei $v = d_1$. Dann ist $d = \langle a_1 \rightharpoonup \ldots \rightharpoonup d_1 \rangle$ und $\mathcal{B} = \{a_n, b_m, a_{n+i}\}$. Traversieren wir nun d rückwärts, so ist a_{n+i} am Server A der erste erreichbare Knoten in $\mathcal{B}$. a_{n+i} ist jedoch direkter Vorgänger von v, scheidet daher als Ausgangsknoten eines OLL-Katalogeintrages aus. Gehen wir weiter rückwärts in d, so ist b_m am Server B der nächste Knoten aus $\mathcal{B}$. Da Server B bei der bisherigen Rückwärts-Traversierung noch nicht besucht wurde, ist b_m Ausgangsknoten eines neuen OLL-Katalogeintrages. Als nächstes erreichen wir a_n am Server A, der jedoch schon einmal besucht wurde (durch Knoten a_{n+i}) und deshalb keine Ausgangsknoten für neue Einträge beisteuern kann. Es sind nun alle Knoten in $\mathcal{B}$ betrachtet, die Traversierung von d ist demnach beendet.

Die hier vorgeschlagene Rückwärts-Traversierung des Pfades d lässt sich algorithmisch noch weiter optimieren: Der *erste* Knoten auf einem entfernten Server s bei der Rückwärts-Traversierung entspricht dem *zuletzt* besuchten Knoten auf s bei der ursprünglichen Traversierung. Verwendet man für jeden Standort einen eigenen, global verfügbaren Stack, auf den man diese Partitions-Ausgangsknoten bei einem Besuch ablegt, so sind die Kandidaten für u_j immer nur die obersten Elemente auf den Stacks! Alle tiefer liegenden Elemente sind Knoten, die früher besucht wurden und daher nicht für eine Kante im OLL-Katalog in Frage kommen. Bei Graphen mit Zyklen hinsichtlich des Speicherortes muss lediglich dafür gesorgt werden, dass die ursprüngliche Besuchsreihenfolge dieser Stack-Elemente rekonstruiert werden kann. Hierfür genügt es, die Elemente beim Einlagern stack-übergreifend fortlaufend zu nummerieren. Sortiert man die obersten Stack-Elemente nach dieser Nummer in fallender Folge, so sind alle Elemente u_s, deren Nummer nicht kleiner ist als die des Elements u_k mit $f(u_k) = f(v)$, Startknoten von neuen Kanten im OLL-Katalog (ausgenommen der direkte Vorgänger von v). Abbildung 6.5 zeigt diese Stacks für Pfad d) in Abbildung 6.4 und $v = d_1$.

Für jeden neuen OLL-Katalogeintrag müssen auch die Bedingungen $c_1, \ldots c_{k+1}$ ermittelt werden, die zu erfüllen sind, um v von u_s aus zu erreichen. Dieser Vorgang lässt sich dadurch sehr effizient gestalten, dass während der Traversierung der Kanten die dort vermerkten Bedingungen gleich aufgesammelt und mittels logischem UND-Operator verknüpft werden. Die daraus resultierende Bedingung kann dann unmittelbar der neu zu erstellenden Kante zugewiesen werden. Diese

Kante wird dann in den OLL-Teilkatalog $E'_{f(v)}$ am aktuellen Server sowie – nach dem rekursiven Wiederaufstieg – in $E'_{f(u)}$ eingetragen.

Der Algorithmus

Die Initialisierung des OLL-Katalogs wird an dem Server gestartet, welcher das Top-Level-Element (im Folgenden mit `root` bezeichnet) der kompletten Struktur speichert. Dort wird die Initialisierungs-Routine mit `Init(root,`ε`,`$\emptyset$`, `$+\infty$`)` aufgerufen.

Algorithmus 6.5.1: (Initialisierung des OLL-Katalogs)

Init(in vertex u, in condition precond, in borders B, in int multi_used)

```
1  new_edges ← ∅
2  sorted_B ← [ ]
3  if (B == ∅) then seq ← 1
4  else
5      sorted_B ← sort(B) on B[z.seq_number] desc
6      seq ← max(B[z.seq_number]) + 1
7  endif
8  if (precond == ε) then
9      for all ⟨v, c, q⟩ ∈ sorted_B in descending order do
10         if ((v -c-> u ∉ E) and q < multi_used) then
11             if (v -c-> u ∉ OLL) then
12                 OLL ← OLL ∪ {⟨v -c-> u, 1⟩}
13             else
14                 OLL ← OLL[⟨v -c-> u, k⟩ ← ⟨v -c-> u, k + 1⟩]
15             endif
16             if (f(u) ≠ f(v)) then
17                 new_edges ← new_edges ∪ {v -c-> u}
18             endif
19         endif
20         if (f(u) == f(v)) then
21             break
22         endif
23     enddo
24 endif
25 for all u -c-> p ∈ E do
26     succ.push ⟨u -c-> p, precond⟩
27 enddo
28 while (succ.test ! = NULL) do
29     ⟨u -c-> v, cond⟩ ← succ.pop
```

```
30    if (f(u) == f(v)) then
31        for all v -c̄-> w ∈ E do
32            succ.push ⟨v -c̄-> w, cond ∧ c⟩
33        enddo
34    else
35        B' ← (B[z.cond ← z.cond ∧ cond ∧ c]\
36            {b ∈ B : f(b) == f(u)}) ∪ {⟨u, c, seq⟩}
37        if (marked_visited(u -c-> v) and multi_used = +∞) then
38            succ_edges ← Init^(f(v)) (v, ε, B', seq)
39        else
40            succ_edges ← Init^(f(v)) (v, ε, B', multi_used)
41        endif
42        mark_visited(u -c-> v)
43        for all x -c-> y ∈ succ_edges do
44            if (x == u) then
45                if (x -c-> y ∉ OLL) then
46                    OLL ← OLL ∪ {⟨x -c-> y, 1⟩}
47                else
48                    OLL ← OLL[⟨x -c-> y, k⟩ ← ⟨x -c-> y, k + 1⟩]
49                endif
50            else
51                new_edges ← new_edges ∪ {x -c-> y}
52            endif
53        enddo
54    endif
55 enddo
56 return new_edges
```

Ablaufbeschreibung

Dem Algorithmus liegt das im Abschnitt 6.5.1 beschriebene Verfahren zugrunde. Hier sollen nur die wesentlichen Anteile den einzelnen Abschnitten des Algorithmus zugeordnet werden.

Der Initialisierungs-Algorithmus wird mit dem Top-Level-Element des DACGs, einer leeren Bedingung sowie einer leeren Menge von Partitions-Ausgangsknoten, und einem auf $+\infty$ gesetzten Parameter `multi-used` aufgerufen. Der Parameter `precond` hat in der hier beschriebenen Initialisierungsphase noch keine Bedeutung. Bei der Änderung des OLL-Katalogs auf Grund einer Änderung des zugrunde liegenden DACGs wird dieser Parameter dann benötigt (vgl. Abschnitt 6.5.2).

In den Zeilen 1 bis 7 werden die Initialisierungen der Variablen sowie die Sortierung der Ausgangsknoten vorgenommen. Dabei wird auch die nächste laufende Nummer (seq) bestimmt, die einem eventuell auftretenden neuen Ausgangsknoten zugewiesen werden kann.

Anschließend (Zeilen 8 bis 24) werden OLL-Katalogeinträge auf Basis der sortierten Ausgangsknoten generiert. Ist ein Ausgangsknoten v kein direkter Vorgänger des gerade besuchten Knotens u, so wird der OLL-Katalog aktualisiert. Hier sind einige implementations-spezifische Details zu betrachten:

Ein OLL-Katalog-Eintrag kann mehrere korrespondierende Pfade im DACG repräsentieren. Deren Anzahl wird für effizientere Änderungsoperationen an dieser Stelle mit abgespeichert. Gibt es die Kante $v \xrightarrow{c} u$ noch nicht im Katalog, so wird ein neuer Eintrag erzeugt mit Anzahl gleich 1. Andernfalls wird beim bestehenden Eintrag die Anzahl inkrementiert.

Bei Mehrfachverwendung eines Bauteils muss mehrmaliges Erzeugen derselben OLL-Katalog-Einträge verhindert werden. Dazu wird der Parameter `multi_used` in Zeile 10 mit abgefragt. Liegt keine Mehrfachverwendung vor, so ist der Wert $+\infty$, andernfalls enthält der Parameter die Sequenz-Nummer der ersten Partition im Pfad von der Wurzel hinunter zum betrachteten Knoten u, die eine Mehrfachverwendung aufweist und für die bereits die OLL-Katalog-Einträge generiert wurden. Partitionen, die im Pfad nach dieser mehrfach verwendeten Partition auftreten, dürfen bei der Erzeugung weiterer OLL-Katalog-Einträge nicht mehr beachtet werden. Nur Partitionen, deren Sequenz-Nummer kleiner ist als die der mehrfach verwendeten Partitionen (d. h. sie treten im Pfad früher auf), enthalten möglicherweise Startknoten für neue OLL-Katalog-Einträge.

Liegt v am gleichen Standort wie der besuchte Knoten (Zyklus!), so werden keine weiteren Ausgangsknoten mehr betrachtet und die Bearbeitung abgebrochen; andernfalls wird die neue Kante in der Menge new_edges für eine spätere Bearbeitung zwischengespeichert.

Handelt es sich bei dem besuchten Knoten u um das Top-Level-Element der Struktur, so ist die Liste der bereits besuchten Ausgangsknoten leer (vgl. Rückwärtstraversierung). Folglich sind keine OLL-Katalogeinträge zu erstellen, d. h. es kann unmittelbar mit Zeile 25 im Algorithmus fortgefahren werden.

Die nun folgende Bearbeitung aller Nachfolgerknoten von u stellt eine depth-first-Traversierung der Struktur unterhalb von u dar. Dazu werden zunächst alle Kanten ausgehend von u auf den Stack $succ$ gebracht (Zeilen 25 bis 27) und anschließend nach und nach abgearbeitet (ab Zeile 28). Die Variable $precond$, die in Zeile 26 mit auf den Stack gelegt wird, beschreibt die Bedingung, die erfüllt sein muss, um den Startknoten der gerade aufzulegenden Kante (hier: u) zu erreichen.

Das oberste Element des Stacks wird in Zeile 29 abgeholt: Es handelt sich dabei um eine Kante von u nach v mit der Bedingung c, sowie einer Vorbedingung $cond$ für u. Liegen u und v auf dem gleichen Server, so werden alle von v ausgehenden Kanten mit den entsprechenden Bedingungen auf $succ$ abgelegt (Zeilen 30 bis 33). Damit wird die depth-first-Traversierung der lokalen Partition fortgesetzt.

Stellt $u \stackrel{c}{\rightharpoonup} v$ jedoch einen Serverübergang dar, so sind die in Zeilen 35 bis 54 aufgeführten Schritte zu durchlaufen:

Zunächst wird die Menge der Ausgangsknoten erstellt (Z. 35 und 36), die für den Server $f(v)$ und möglicherweise weitere Server in tieferen Leveln in der Struktur unterhalb von v interessant sind. Dazu werden die Bedingungen aller bisherigen Ausgangsknoten erweitert um die Bedingung c des gerade betrachteten Serverübergangs und der dazu erforderlichen Vorbedingung $cond$ (vgl. Zeile 29). Ein früher besuchter Ausgangsknoten, der auf dem gleichen Server liegt wie der aktuell betrachtete Knoten u, wird dabei ausgeschlossen und durch $\langle u, c, seq \rangle$ ersetzt. Dabei bedeuten u den Knoten selbst, c die Bedingung für den Serverübergang, und seq ist die Sequenz-Nummer des Ausgangsknotens.

Nun kann der rekursive Aufruf der Initialisierungs-Prozedur am entfernten Server erfolgen (Z. 37 bis 41). Hier spielt die Mehrfachverwendung wieder eine Rolle: Wurde der betrachtete Serverübergang bereits traversiert, so erfolgt dieser Aufruf mit der aktuellen Sequenz-Nummer als neuem Wert für den Parameter `multi_used`, falls dieser noch auf dem Ausgangswert $+\infty$ stand. Andernfalls wird der aktuelle Wert dieses Parameters weitergegeben. In beiden Fällen wird der betrachtete Serverübergang anschließend als *besucht* markiert (Z. 42).

Ergebnis des rekursiven Aufrufs der Initialisierungs-Prozedur ist eine Menge von OLL-Katalogeinträgen, die in der Struktur *unterhalb* von v erzeugt wurden und Startknoten besitzen, die in der Struktur *oberhalb* von v liegen. Diese OLL-Katalogeinträge müssen an den Servern der Startknoten während des rekursiven Wiederaufstieges in die lokalen OLL-Teilkataloge übernommen werden.

In den Zeilen 43 bis 53 wird nun jeder neue OLL-Katalogeintrag aus $succ\_edges$ geprüft, ob der Startknoten in der aktuellen Partition liegt. Ist dies der Fall, so wird der lokale OLL-Katalog um diesen Eintrag erweitert (bzw., falls ein entsprechender Eintrag bereits existiert, wiederum die Anzahl erhöht), andernfalls wird dieser Eintrag zur Menge der neuen Einträge (new_edges) hinzugefügt.

Schließlich endet die Initialisierung mit der Rückgabe der neuen OLL-Katalogeinträge an den Aufrufer. Bei der vollständigen Rückkehr aus der Rekursion muss die Menge der neuen Einträge leer sein, d. h. sämtliche erzeugten OLL-Katalogeinträge sind sowohl am Server des Startknotens als auch am Server des Zielknotens eingetragen.

Optimierungsmöglichkeiten

Der Initialisierungsalgorithmus kann sicherlich hinsichtlich Performanz noch verbessert werden. Die Initialisierung selbst ist zwar nicht besonders zeitkritisch, da der OLL-Katalog einer kompletten Struktur nur selten – typischerweise nur ein Mal – initialisiert werden muss. Im Abschnitt 6.5.2 wird für Änderungsoperationen jedoch die Initialisierung auf Teilstrukturen angewendet, deshalb kann die Betrachtung von einigen wenigen Ansatzmöglichkeiten zur Optimierung durchaus sinnvoll sein:

Die Sortierung der Ausgangsknoten in Zeile 5 kann entfallen, falls bei der Erzeugung der Menge B' in Zeilen 35 und 36 auf die bereits sortierte Menge $sorted\_B$ anstatt auf B selbst Bezug genommen wird:

35 $B' \leftarrow \langle u, c, seq \rangle + sorted\_B[z.cond \leftarrow z.cond \wedge cond \wedge c] \quad -$

36 $\quad - \quad \Big(b \in B : f(b) == f(u)\Big)$

Dabei bedeutet die Addition $(+)$ eine Erweiterung der sortierten Liste am Listenkopf, die Subtraktion $(-)$ entfernt das angegebene Listenelement und füllt die dabei möglicherweise entstehende Lücke durch „Aufrücken“ der folgenden Elemente wieder auf. Auf diese Weise bleibt eine absteigend sortierte Liste (bezüglich der Nummerierung der Objekte) erhalten, die Sortierung in der Initialisierungsphase kann entfallen.

Die Mehrfachverwendung besonders von stark strukturierten Baugruppen führt bei der Initialisierung nach Algorithmus 6.5.1 dazu, dass – wegen der depth-first-Traversierung – die Strukturen dieser Baugruppen mehrfach komplett durchlaufen werden. Dabei werden OLL-Katalogeinträge, die nur die Baugruppe selbst betreffen (d. h. Start- und Zielknoten sind in der Baugruppe enthalten), unnötigerweise mehrfach ermittelt.

Abhilfe kann man wie folgt schaffen: Knoten, deren Substrukturen vollständig bearbeitet sind, erhalten eine Markierung. Trifft man beim rekursiven Abstieg auf einen derartig markierten Knoten u, so werden während der Traversierung der zugehörigen Substruktur von u nur die bis dahin aufgesammelten Ausgangsknoten $\tilde{B} = \{b_1, \ldots, b_k\}$ als mögliche Startknoten von neuen OLL-Katalogeinträgen betrachtet. Die Traversierung der Substruktur von u kann abgebrochen werden, falls für jeden Ausgangsknoten $b_i \in \tilde{B}$ in dieser Substruktur ein Zyklus ermittelt wurde.

6.5.2 Inkrementelle Anpassung bei Änderung der Produktstruktur

6.5.2.1 Ausgangslage und Zielsetzung

Während eines Entwicklungsprozesses werden Produktstrukturen häufig geändert: Neue Dokumente werden erstellt, neue Versionen bereits existierender Einzel- und Zusammenbauteile werden entwickelt, und in selteneren Fällen wird ein bereits existierender Produktaufbruch weiter verfeinert, d. h. es werden neue Einzel- und Zusammenbauteile hinzugefügt.

All diese Änderungen haben in der Regel jedoch keine Auswirkung auf die Verteilung der Daten. Objekt-Migrationen sind die absolute Ausnahme, da typischerweise alle Entwicklungspartner und Zulieferer daran interessiert sind, die ihren Workshare betreffenden Daten lokal zu verwalten.

In manchen Fällen jedoch kann eine Produktstruktur zu Beginn eines Projektes nicht komplett definiert werden, so dass während der Produktentwicklung beispielsweise noch weitere Zulieferer in den Entwicklungsverbund integriert werden müssen. Hierbei werden gegebenenfalls Daten neu verteilt bzw. neue Substrukturen erzeugt. Dadurch wird eine Anpassung des OLL-Katalogs erforderlich.

Einige Benutzeraktionen ändern auch die Bedingungen an einzelnen Kanten. So kann beispielsweise die sogenannte *Freigabe* eines neu entwickelten Bauteils dazu führen, dass bisher verbaute Komponenten ab sofort nicht mehr verwendet werden dürfen. Produktkonfigurationen neueren Datums müssen demnach die neue Komponente enthalten. Man spricht in diesem Zusammenhang auch von „ungültig steuern“ einer alten Komponente. Derartige Änderungen müssen sich auch im OLL-Katalog widerspiegeln.

Löschoperationen auf Produktstrukturobjekten sind in operativen PDM-Systemen auf Grund von Dokumentations-, Nachweis- und Gewährleistungspflichten (Produkthaftung) nicht gestattet. Deshalb werden im Weiteren auch keine derartigen Änderungen betrachtet.

Ausgangssituation sei im Folgenden wiederum ein DACG G^c, der in mehreren Partitionen P_{s_i} auf verschiedenen Servern $s \in S$ verteilt gespeichert ist. Die Initialisierung soll an allen Standorten erfolgreich durchgeführt sein, d. h. an allen Standorten sind die lokalen OLL-Kataloge vorhanden.

Ziel des OLL-Katalog-Updates ist die *inkrementelle Anpassung* des OLL-Katalogs nach strukturellen Änderungen des DACGs, d. h. es sollen lokal erforderliche Änderungen durchgeführt werden, ohne dabei den kompletten OLL-Katalog neu aufbauen zu müssen. Selbstredend müssen die geänderten lokalen OLL-Kataloge die Eigenschaften aus Definition 7 erfüllen.

6.5.2.2 Erweiterung von Produktstrukturen

Produktstrukturen werden erweitert durch Hinzufügen einer Kante $u \xrightarrow{c} v$ zu einem DACG. Diese neue Kante muss *korrekt* sein in dem Sinne, dass der durch die Erweiterung entstehende Graph wiederum die Eigenschaften eines DACGs besitzt.

Im Folgenden wird an mehreren Änderungsszenarien gezeigt, welche Schritte durchzuführen sind, um korrekte OLL-Teilkataloge an den einzelnen Standorten zu garantieren.

Änderungsszenario 1: Hinzufügen einer Kante $u \xrightarrow{c} v$, die nicht im OLL-Katalog enthalten ist.

Fall a): Die Knoten u und v liegen an verschiedenen Standorten, d. h. die neue Kante $u \xrightarrow{c} v$ stellt einen Serverübergang dar. Abbildung 6.6 zeigt mögliche Ergebnisse von derartigen Änderungsoperationen.

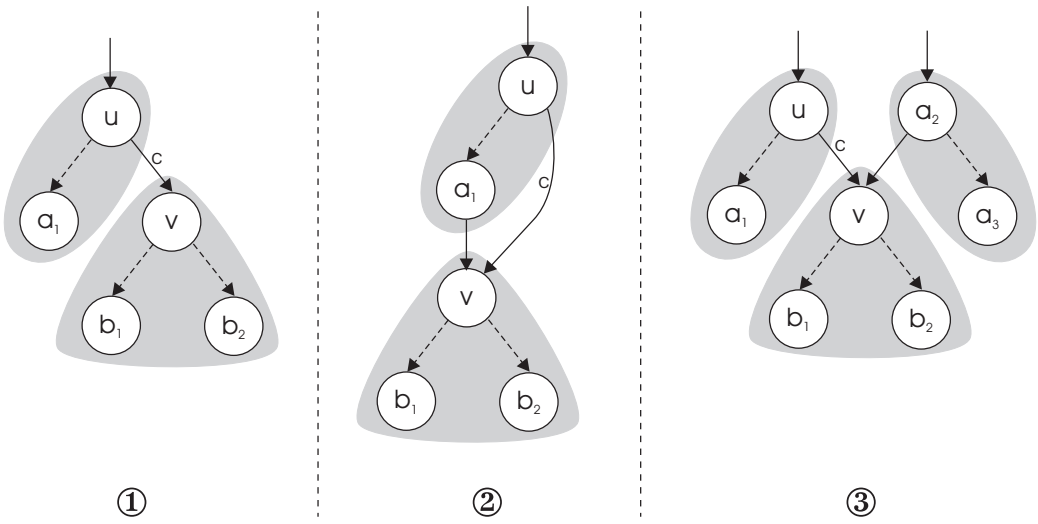


Abbildung 6.6: Drei Szenarien für das Hinzufügen einer Kante $u \xrightarrow{c} v$, die nicht im OLL-Katalog enthalten ist, mit $f(u) \neq f(v)$.

Bei näherer Betrachtung stellt man schnell fest, dass sich das Problem der Anpassung des OLL-Katalogs im Prinzip offensichtlich auf das Initialisierungsproblem zurückführen lässt: Wenn man alle Partitions-Ausgangsknoten aus den Vorgängern des Knotens u zusammen mit den zugehörigen konkatenierten Kanten-Bedingungen zur Verfügung hätte, so könnte man mit dieser Information die Initialisierungsroutine (vgl.

Algorithmus 6.5.1) für v aufrufen! Diese Idee liegt auch dem im Folgenden beschriebenen Verfahren zugrunde.

Zunächst gilt es, Partitions-Ausgangsknoten in allen Pfaden vom Top-Level-Element zu u zu finden. Dazu wird der Graph rekursiv rückwärts durchlaufen, beginnend bei u. Dabei wird von jedem entfernten Server $s \in S$ der *zuerst* erreichte Ausgangsknoten u_s zusammen mit den Bedingungen entlang des Pfades $\langle u_s \rightharpoonup \ldots \rightharpoonup u \rangle$ aufgesammelt.

Zur Erinnerung: Bei der Initialisierungsroutine (Algorithmus 6.5.1) war es nötig, die Reihenfolge des Vorkommens der Ausgangsknoten im entsprechenden Pfad zu kennen. Dazu werden dort die Ausgangsknoten bei deren Besuch entsprechend fortlaufend (und aufsteigend) nummeriert. Diese Reihenfolge muss – um die Initialisierungsroutine korrekt einsetzen zu können – auch in dem Ablauf hier bestimmt und festgehalten werden. Da hier im Gegensatz zu der Initialisierungsroutine der Baum zunächst *rückwärts* traversiert wird, werden die betrachteten Ausgangsknoten fortlaufend *absteigend* nummeriert, beginnend bei Null.

Am Top-Level-Element angekommen, wird die Initialisierungsroutine für den Knoten v gerufen. Von dieser Stelle an läuft die Initialisierung des Knotens v wie im Abschnitt 6.5.1 ab. Falls neue OLL-Katalogeinträge $u' \overset{c'}{\rightharpoonup} v'$ am Standort $f(v')$ generiert werden, so werden diese wieder an den Aufrufer zurückgegeben, um auch beim Standort $f(u')$ in den dortigen lokalen OLL-Katalog eingepflegt zu werden.

Die Anpassungen der lokalen OLL-Kataloge sind erledigt, wenn der rekursive Aufstieg aus der rückwärts gerichteten Traversierung des DACGs am Knoten u endet. Abbildung 6.7 zeigt das prinzipielle Vorgehen.

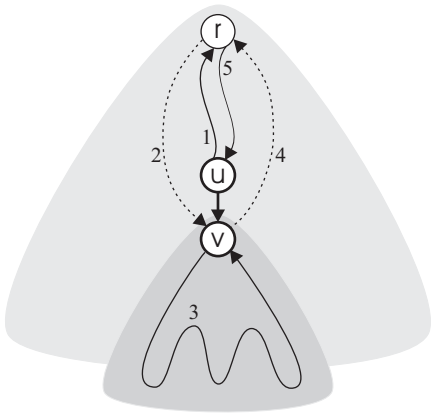


Abbildung 6.7: Traversierung der Produktstruktur in den Änderungsalgorithmen

Fall b): Die Knoten u und v liegen am selben Standort, d. h. durch die neue Kante wird kein Serverübergang hinzugefügt. Abbildung 6.8 zeigt die zu Abbildung 6.6 äquivalenten Ergebnisse für diesen Änderungsfall.

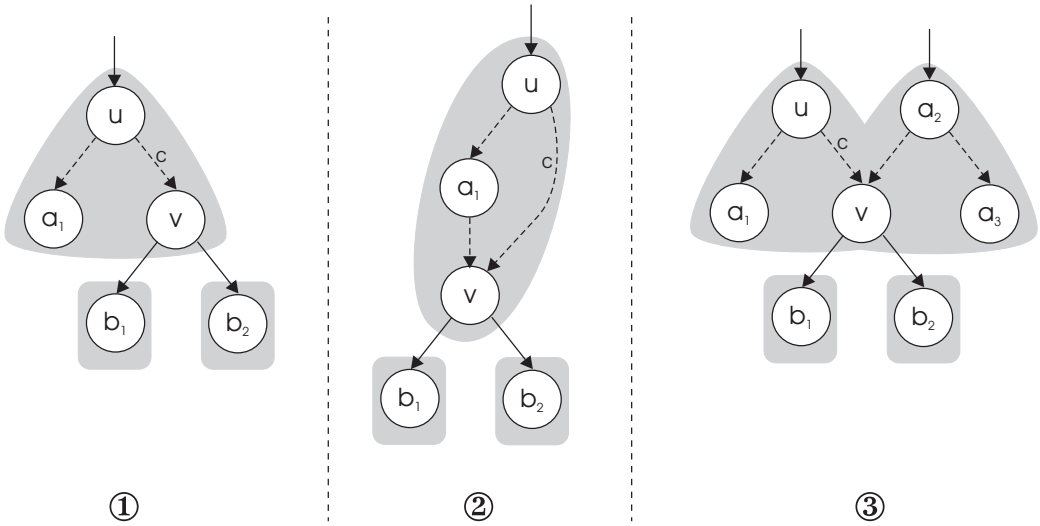


Abbildung 6.8: Drei Szenarien für das Hinzufügen einer Kante $u \xrightarrow{c} v$, die nicht im OLL-Katalog enthalten ist, mit $f(u) = f(v)$.

Im Wesentlichen gilt hier das für Fall a) beschriebene Verfahren. Die Struktur wird wiederum zunächst rückwärts durchlaufen, dabei werden Ausgangsknoten aufgesammelt, und schließlich wird die Init-Routine für den Knoten v aufgerufen. Hier jedoch besteht ein wesentlicher Unterschied zum bisherigen Verfahren.

Zur Rekapitulation: Bei der Initialisierung eines DACGs wird `Init` genau dann rekursiv aufgerufen, falls die Traversierung des DACGs auf einen *Serverübergang* stößt. Bei den Szenarien, die im Fall a) für die DACG-Änderungen betrachtet wurden, wird $Init(v, \ldots)$ ebenfalls bei einem Serverübergang gerufen, nämlich für den Übergang von $f(u)$ nach $f(v)$. Die Init-Routine ermittelt für den zu initialisierenden Knoten v zu Beginn alle erforderlichen neuen OLL-Katalogeinträge auf Grund der übergebenen Menge von Ausgangsknoten (vgl. Zeilen 8 bis 24 des Algorithmus 6.5.1).

Bei den Szenarien im Fall b) stellt die Kante $u \xrightarrow{c} v$ jedoch keinen Serverübergang dar, denn u und v liegen am gleichen Standort. Daraus folgt, dass für eventuell benötigte neue OLL-Katalogeinträge der Knoten v selbst gänzlich ohne Belang ist. Da v auch noch lokale Nachfolger

besitzen kann, muss der Algorithmus quasi mitten in den rekursiven Abstieg in der lokalen Partition hineinspringen, ohne OLL-Katalogeinträge zu generieren, die v als Zielknoten referenzieren würden.

Dieses Ziel wird wie folgt erreicht: Der Initialisierungsroutine wird im Parameter `precond` die konkatenierte Bedingung vom Eingangsknoten der Partition, zu welcher der Knoten v gehört, bis hinunter zu v übergeben. In der Initialisierungsroutine erfolgt der Test, ob dieser Parameter gesetzt ist oder nur ε enthält. Falls – wie hier bei den gezeigten Änderungsoperationen – ein logischer Wert gesetzt ist, so wird die Generierung der OLL-Katalogeinträge (vgl. Zeilen 8 bis 24 des Algorithmus 6.5.1) übersprungen und sofort mit der lokalen Expansion der Partition fortgefahren.

Die Generierung neuer OLL-Katalogeinträge an in der Struktur tiefer liegenden Serverübergängen sowie deren Rückgabe erfolgt wie für den Fall a) beschrieben.

Änderungsszenario 2: Hinzufügen einer Kante $u \xrightarrow{c} v$, die bereits im OLL-Katalog enthalten ist.
Abbildung 6.9 zeigt die beiden möglichen Szenarien für diesen Änderungsfall. Im Teilbild a) gilt $f(u) = f(v)$, in Teilbild b) gilt $f(u) \neq f(v)$.

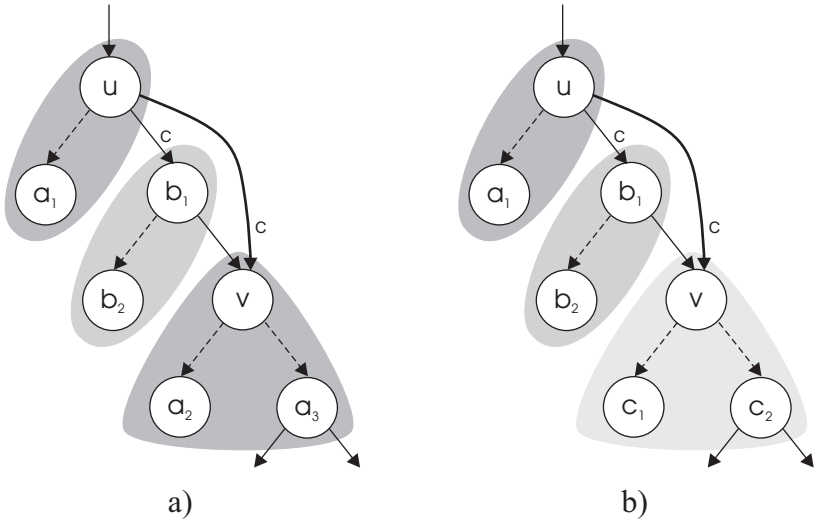


Abbildung 6.9: Zwei Szenarien für das Hinzufügen einer Kante $u \xrightarrow{c} v$, die bereits im OLL-Katalog enthalten ist. Es gilt in a) $f(u) = f(v)$, in b) $f(u) \neq f(v)$.

In beiden Teilbildern gilt: Vor dem Hinzufügen der Kante $u \xrightarrow{c} v$ existiert bereits ein gleichlautender OLL-Katalogeintrag auf Grund des Pfades $\langle u \xrightarrow{c} b_1 \rightharpoonup v \rangle$.[28]

Die Definition des OLL-Katalogs (vgl. Definition 6 im Abschnitt 6.3) besagt, dass im OLL-Katalog keine Kanten des zugrunde liegenden DACGs enthalten sein dürfen. Folglich muss bei einer Erweiterung des DACGs um eine Kante, die auf Grund von vorangegangener Initialisierung und/oder Änderungen bereits im OLL-Katalog enthalten ist, diese dort entfernt werden.

Auf den ersten Blick mag es so aussehen, als ob mit dieser Löschung die Anpassung des OLL-Kataloges bereits komplett durchgeführt wäre. An einem einfachen Beispiel lässt sich jedoch zeigen, dass auch in diesem Fall möglicherweise neue OLL-Katalogeinträge zu erstellen sind.

Gegenstand der Betrachtung sei aus der Abbildung 6.9 ohne Beschränkung der Allgemeinheit das Teilbild a) und hiervon der Ausschnitt aus einem Pfad $d_1 = \langle \ldots \rightharpoonup u \xrightarrow{c} b_1 \rightharpoonup v \rightharpoonup a_3 \rightharpoonup \ldots \rangle$. Die Knoten u, v und a_3 liegen alle am Standort A, der Knoten b_1 liegt entfernt am Standort B. Eine Erweiterung des betrachteten Ausschnitts aus dem Pfad d_1 dergestalt, dass u das Ziel eines Serverübergangs und a_3 Startknoten eines anderen Serverübergangs ist, also $d_2 = \langle x_1 \rightharpoonup u \xrightarrow{c} b_1 \rightharpoonup v \rightharpoonup a_3 \rightharpoonup b_3 \rangle$, ergibt Folgendes:

Vom Knoten x zum Knoten b_3 existiert nun wegen der neuen Kante $u \xrightarrow{c} v$ ein Pfad, der die Kriterien für einen Eintrag von $x \xrightarrow{c} b_3$ in den OLL-Katalog erfüllt! Es genügt also nicht, nur $u \xrightarrow{c} v$ aus dem OLL-Katalog zu entfernen, auch neue Kanten können hinzukommen.

Das Verfahren entspricht dem für Änderungsszenario 1, Fall a), wobei lediglich noch das Entfernen „alter“ Einträge aus dem OLL-Katalog berücksichtigt werden muss.

Der Algorithmus

Algorithmus 6.5.2 zeigt ein Fragment einer Routine zur Erweiterung von Produktstrukturen. Diese Routine fügt die neue Kante in E an den Servern $f(u)$ und $f(v)$ ein, löscht einen eventuell vorhandenen gleichlautenden OLL-Eintrag (ebenfalls an beiden Servern), und ruft schließlich die Extend-Prozedur (Algorithmus 6.5.3) auf. Dabei wird unterschieden, ob die neue Kante einen Serverübergang darstellt oder nicht.

[28] Alternativ wäre auch möglich $\langle u \rightharpoonup b_1 \xrightarrow{c} v \rangle$, dies spielt jedoch für die folgenden Betrachtungen keine Rolle.

Algorithmus 6.5.2: (Erweiterung einer Produktstruktur)

CreateNewEdge(in vertex u, in vertex v, in condition c)

```
1  ...
2  E^f(u) ← E^f(u) ∪ {u -c-> v}
3  if (f(u) ≠ f(v)) then
4      E^f(v) ← E^f(v) ∪ {u -c-> v}
5  endif
6  if (u -c-> v ∈ OLL) then
7      OLL^f(u) ← OLL^f(u) \ {u -c-> v}
8      if (f(u) ≠ f(v)) then
9          OLL^f(v) ← OLL^f(v) \ {u -c-> v}
10     endif
11 endif
12 if (f(u) == f(v)) then
13     dset ← Extend^(f(u))(u, ⟨v, c⟩, ⊤, ∅, +∞)
14 else
15     dset ← Extend^(f(u))(u, ⟨v, ε⟩, c, {⟨u, c, 0⟩}, +∞)
16 endif
17 ...
```

Algorithmus 6.5.3: (Erweiterung des OLL-Katalogs)

Extend(in vertex x, in ⟨vertex u, condition precond⟩,
in condition $\bar{c}$, in borders B, in int multi_used)

```
1  new_edges ← ∅
2  multi_used_indicator ← false
3  if (B == ∅) then
4      seq ← 0
5  else
6      seq ← min(B[z.seq_number]) − 1
7  endif
8  if (∃ w -c-> x ∈ E) then
9      for all w -c-> x ∈ E do
10         pred.push ⟨w -c-> x, c̄⟩
11     enddo
12 else
13     pred_edges ← Init^(f(u))(u, precond, B, multi_used)
14     for all u -c'-> v ∈ pred_edges do
15         if (f(u) == f(x)) then
```

```
16            if (u -c'-> v ∉ OLL) then
17                OLL ← OLL ∪ {⟨u -c'-> v, 1⟩}
18            else
19                OLL ← OLL[⟨u -c'-> v, k⟩ ← ⟨u -c'-> v, k + 1⟩]
20            endif
21        else
22            new_edges ← new_edges ∪ {u -c'-> v}
23        endif
24    enddo
25 endif
26 while (pred.test ! = NULL) do
27    ⟨x -c-> y, cond⟩ ← pred.pop
28    if (f(x) == f(y)) then
29        if (∃ w -c''-> x ∈ E) then
30            for all w -c''-> x ∈ E do
31                pred.push ⟨w -c''-> x, cond ∧ c⟩
32                if (seq == 0) then
33                    precond ← precond ∧ c
34                endif
35            enddo
36        else
37            pred_edges ← Init^(f(u))(u, precond, B, multi_used)
38            for all u -c-> v ∈ pred_edges do
39                if (f(u) == f(x)) then
40                    if (u -c-> v ∉ OLL) then
41                        OLL ← OLL ∪ {⟨u -c-> v, 1⟩}
42                    else
43                        OLL ← OLL[⟨u -c-> v, k⟩ ← ⟨u -c-> v, k + 1⟩]
44                    endif
45                else
46                    new_edges ← new_edges ∪ {u -c-> v}
47                endif
48            enddo
49        endif
50    else
51        multi_used' ← (multi_used_indicator?seq : multi_used)
52        if (¬∃b ∈ B|f(x) == f(b)) then
53            pred_edges ← Extend^(f(x))(x, ⟨u, precond⟩,
54                        cond ∧ c, B ∪ {⟨x, cond ∧ c, seq⟩}, multi_used')
55        else
56            pred_edges ← Extend^(f(x))(x, ⟨u, precond⟩,
```

```
57                          cond ∧ c, B, multi_used')
58          endif
59          multi_used_indicator ← true
60          for all u -c-> v ∈ pred_edges do
61              if (f(u) == f(y)) then
62                  if (u -c-> v ∉ OLL) then
63                      OLL ← OLL ∪ {⟨u -c-> v, 1⟩}
64                  else
65                      OLL ← OLL[⟨u -c-> v, k⟩ ← ⟨u -c-> v, k + 1⟩]
66                  endif
67              else
68                  new_edges ← new_edges ∪ {u -c-> v}
69              endif
70          enddo
71      endif
72  enddo
73  return new_edges
```

Ablaufbeschreibung

Die beiden Algorithmen arbeiten nach den bereits beschriebenen Vorgehensweisen. Im Folgenden sind nur die wesentlichen Anteile den einzelnen Abschnitten der Algorithmen zugeordnet.

Algorithmus 6.5.2 erzeugt zunächst die neue Kante an dem Server, der Startknoten u speichert (Zeile 2). Falls die neue Kante einen Serverübergang darstellt, wird die Kante auch am entfernten Server erzeugt (Zeile 3 bis 5). In den Zeilen 6 bis 11 werden – falls nötig – die OLL-Einträge, die der neuen Kante entsprechen, entfernt. Anschließend wird Algorithmus 6.5.3 aufgerufen. Unterschieden wird wiederum, ob ein Serverübergang vorliegt (Zeile 15) oder nicht (Zeile 13).

Algorithmus 6.5.3 beginnt mit der Initialisierung (Zeilen 1 bis 7), wobei der absteigenden Sortierung der Ausgangsknoten Rechnung getragen wird.

Sofern Vorgänger des übergebenen Startknotens vorhanden sind, werden diese auf dem Stack abgelegt (Zeilen 8 bis 11), andernfalls ist der rekursive, rückwärtsgerichtete Aufstieg bei der Wurzel angekommen und die *Init*-Routine kann gerufen werden (Zeile 13, vgl. Algorithmus 6.5.1). Im Anschluss daran (Zeilen 14 bis 24) werden die von *Init* produzierten OLL-Kanten in den lokalen OLL-Katalog eingearbeitet.

In den Zeilen 26 bis 72 erfolgt die rekursive Bearbeitung (wiederum in *depth-first*-Strategie) aller Vorgänger. Der auf dem Stack zuoberst liegende Eintrag (Zeile 27) wird geholt und überprüft, ob es sich um einen Serverübergang handelt. Falls nein

(Zeilen 28 bis 49), so werden – falls vorhanden – weitere Vorgänger auf den Stack gelegt, oder analog zu den Zeilen 8 bis 25 *Init* gerufen und der OLL-Katalog angepasst. Stellt die betrachtete Kante jedoch einen Serverübergang dar (Zeilen 51 bis 71), so wird die Extend-Routine rekursiv gerufen, beim Wiederaufstieg aus der Rekursion werden die zurückgegebenen OLL-Einträge in den lokalen OLL-Katalog eingearbeitet, nicht verwertbare Einträge werden an den Aufrufer zurückgegeben (Zeilen 60 bis 70).

Beim rekursiven Aufruf der Extend-Routine wird die Menge B der Ausgangsknoten nur dann angepasst, falls sie noch kein Objekt des aktuellen Standortes enthält (vgl. Zeilen 52 bis 57). Da die Traversierung im Baum *rückwärts* gerichtet ist, ist nur der erste erreichte Ausgangsknoten eines jeden Standorts von Interesse, alle anderen sind nicht zu betrachten.

Die Extend-Routine für die Kante $u \xrightarrow{c} v$ endet, wenn die Initialisierung von v abgeschlossen und alle dabei erzeugten OLL-Katalogeinträge bei u und allen Vorgängern eingearbeitet sind.

6.5.2.3 Änderungen von Kantenbedingungen

Für die Änderung von Kantenbedingungen gibt es nur wenige relevante Szenarien. So kann beispielsweise ein neues Ausstattungspaket definiert werden, oder bislang als Sonderausstattung angebotene Module in eine erweiterte Serienausstattung aufgenommen werden. Typischerweise werden dabei zusätzliche Kantenbedingungen *hinzugefügt*.

Bei der Steuerung von Gültigkeiten hingegen werden auch bestehende Gültigkeitsobjekte *geändert*. Zumeist wird die aktuelle Version eines Objektes mit einer nach oben offenen Gültigkeit versehen. Wird zu diesem Objekt beispielsweise eine weiterentwickelte Version freigegeben, so wird die Ende-Gültigkeit der bislang aktuellen Version auf die Anfangs-Gültigkeit der neuen Version gesetzt, deren Ende-Gültigkeit ihrerseits dann nach oben offen ist.

Im Allgemeinen bedeutet die Änderung einer Kantenbedingung von c zu c' die Änderung der Kante $u \xrightarrow{c} v$ zu $u \xrightarrow{c'} v$. Dies hat selbstverständlich Auswirkungen auf alle OLL-Katalogeinträge $x \xrightarrow{\bar{c}} y$, denen ein Pfad d in G^c zugrunde liegt mit $d = \langle x \xrightarrow{c_1} \ldots \xrightarrow{c_k} u \xrightarrow{c} v \xrightarrow{c_{k+1}} \ldots \xrightarrow{c_n} y \rangle$ und $\bar{c} = c_1 \wedge \ldots \wedge c_k \wedge c \wedge c_{k+1} \wedge \ldots \wedge c_n$.

Es gilt nun, diese Einträge zu finden und für eine Anpassung der OLL-Kataloge zu sorgen. An einem kleinen Beispiel lässt sich zeigen, wie das Verfahren hierfür auszusehen hat.

Abbildung 6.10a) zeigt die Ausgangssituation. Es sind keine Bedingungen an den Kanten angebracht, folglich ist auch der OLL-Katalogeintrag $x \rightharpoonup y$ ohne Be-

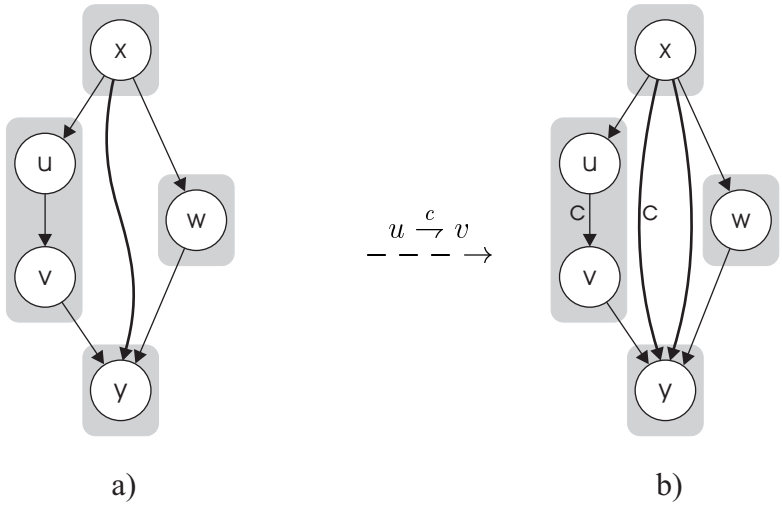


Abbildung 6.10: Änderung der Kante $u \rightharpoonup v$ in a) zu $u \xrightarrow{c} v$ in b) zusammen mit der Auswirkung auf den OLL-Katalog.

dingung. Von Bedeutung ist der Zähler, welcher bei der Initialisierung des OLL-Katalogs (und auch bei Strukturänderungen) auf die Anzahl der Pfade in G^c gesetzt wird, die zu diesem Eintrag führen. Im Beispiel der Abbildung 6.10a) hat der Zähler für $x \rightharpoonup y$ den Wert 2, denn es existieren zwei mögliche Pfade von x nach y in G^c: $d_1 = \langle x \rightharpoonup u \rightharpoonup v \rightharpoonup y \rangle$ und $d_2 = \langle x \rightharpoonup w \rightharpoonup y \rangle$.

Offensichtlich kann der OLL-Eintrag nicht einfach zu $x \xrightarrow{c} y$ abgeändert werden, denn dadurch würde der OLL-Eintrag für d_2 verloren gehen. Stattdessen muss der Zähler des bereits existierenden OLL-Katalogeintrags um Eins verringert werden und ein neuer Eintrag für den geänderten Pfad mit dem Zählerstand 1 hinzugefügt werden. Das Resultat zeigt Abbildung 6.10b).

Für die Anpassung der OLL-Katalogeinträge gilt im Allgemeinen:

1. Steht der Zähler des „alten“ OLL-Katalogeintrages vor der Änderung auf Eins, so wird der Eintrag gelöscht, andernfalls wird der Zähler um Eins heruntergezählt.

2. Vor dem Hinzufügen eines neuen OLL-Katalogeintrages muss überprüft werden, ob bereits ein solcher Eintrag existiert. Falls ja, so wird lediglich dessen Zählerstand um Eins erhöht, falls nein, so wird der Eintrag erzeugt.

Das Vorgehen zum Auffinden „alter“ Katalogeinträge ist eng verwandt mit dem in Abschnitt 6.5.2.2 beschriebenen Verfahren für Strukturerweiterungen: Zunächst

wird die Produktstruktur vom Knoten u aus *rückwärts* traversiert, um alle in Frage kommenden Ausgangsknoten (sowie die Bedingungen an den dabei durchlaufenen Kanten) aufzusammeln. Anschließend muss – ähnlich zur Initialisierung – der Graph von v aus expandiert werden, um alle möglichen Zielknoten von OLL-Katalogeinträgen zu finden. Dabei werden auch die Änderungen auf den jeweils lokalen OLL-Katalogen durchgeführt. Diese Änderungen werden den Aufrufern zurückgegeben, so dass diese in der Lage sind, bei Bedarf (d. h. falls der lokale OLL-Katalog betroffen ist) die Änderungen ebenfalls nachzuziehen.

Da der Algorithmus keine wesentlichen neuen Erkenntnisse birgt, ist er nur im Anhang B wiedergegeben.

6.5.2.4 Migration von Teilstrukturen

Bei der Migration von Substrukturen der Produktstruktur werden Knoten und Kanten von einem Standort s_1 an einen anderen Standort s_2 verschoben. Dabei geht die Verantwortlichkeit für die migrierte Teilstruktur an den „empfangenden“ Partner über.

Wie bereits in Abschnitt 6.5.2.1 motiviert, sind Migrationen in PDM-Systemen eine recht seltene Ausnahme, die allenfalls in frühen Produktentwicklungsphasen auftreten können. Hier ist es noch denkbar, dass ein Bauteil zur Detailkonstruktion an einen externen Partner oder Zulieferer übergeben wird bzw. dass das Ergebnis einer solchen Fremdbeauftragung wieder in das eigene System integriert werden soll.

Sobald jedoch ein Partner oder Zulieferer ein Bauteil produziert und dieses auch in das Produkt eingebaut wird, kann dieses Bauteil nur mit großem Aufwand (Vertragsänderungen zwecks Produkthaftung etc.) migriert werden. Dieser Sonderfall wird hier nicht betrachtet.

Soll ein Zulieferer für ein Bauteil gewechselt werden, so werden hierfür typischerweise zusätzliche Teil-Produktstrukturen angelegt und über die Gültigkeitssteuerung entsprechend konfiguriert. Somit findet hier keine Migration, sondern eine Erweiterung der Produktstruktur statt (vgl. Abschnitt 6.5.2.2).

Die Anpassung der OLL-Kataloge bei einer Migration von Teil-Strukturen kann auf ein ähnliches Verfahren abgebildet werden, wie es für die Strukturerweiterungen bereits vorgestellt wurde (vgl. auch Abbildung 6.7).

Sei v der oberste Knoten der zu migrierenden, zusammenhängenden Teilstruktur T_{PS} und $u \xrightarrow{c} v$ eine Kante in G^c. Dann müssen zunächst wiederum durch Rückwärts-Traversierung des Produktstrukturgraphen beginnend bei v alle Partitions-Ausgangsknoten ermittelt werden. Mit diesen wird der komplette Teilbaum beginnend bei v traversiert, um alle bislang gültigen OLL-Katalogeinträge

zu finden und zu entfernen. Anschließend wird die Teilstruktur T_{PS} migriert. Diese geänderte Struktur muss nun erneut traversiert werden (beginnend bei v, wobei u in die Menge der Ausgangsknoten aufgenommen werden muss, um auch den potentiell neuen Serverübergang $u \rightharpoonup v$ zu berücksichtigen), wobei die neuen OLL-Katalogeinträge generiert werden.

Auf eine Angabe des exakten Algorithmus kann auf Grund der Ähnlichkeit zu den Änderungsalgorithmen verzichtet werden.

6.5.2.5 Löschen von Teilstrukturen

Wie bereits mehrfach ausgeführt wurde, ist in PDM-Systemen das Löschen von Objekten nicht gestattet. Kommen OLL-Kataloge in anderen Anwendungen zum Einsatz, so kann das Löschen prinzipiell auf die bereits beschriebenen Änderungsvorgänge zurückgeführt werden.

Das Löschen eines Kanten-Objektes kann durch Zuweisung einer nie zutreffenden Gültigkeit für diese Kante simuliert werden. Ein Knoten-Objekt kann quasi gelöscht werden, indem sämtliche Relationen, an welchen dieses Objekt beteiligt ist, ungültig gesteuert werden. Die Objekte werden dadurch zwar nicht wirklich gelöscht, sie können aber in keiner Benutzeraktion als Ergebnis auftreten. Für größere Löschvorgänge bietet es sich an, auch die OLL-Kataloge zu löschen und anschließend komplett neu aufzubauen.

6.5.3 Rekursionsverteilung mit OLL-Katalogen

Abschnitt 6.4 beschreibt die prinzipielle Verwendung von OLL-Katalogen in rekursiven Aktionen. Die detaillierte algorithmische Darstellung ist nun hier angegeben.

Bei der Ausführung einer rekursiven Aktion unter Verwendung der OLL-Kataloge muss zwischen einem Anfrage-Master (das ist der Server, welcher den Wurzelknoten der zu expandierenden Teil-Struktur speichert) und den Slaves unterschieden werden. Der PDM-System-Client, an welchem die Anfrage für die Struktur gestellt wird, initiiert den Expansionsvorgang durch einen Aufruf der Funktion 'XMultiLevelExpand' (Algorithmus 6.5.4), welche die Aufrufe der Funktionen 'masterMLE' und 'slaveMLE' koordiniert (Algorithmen 6.5.5 und 6.5.6).

6.5.3.1 Koordination der Expansion

Algorithmus 6.5.4: (Expansion der Produktstruktur)

XMultiLevelExpand(in node u, in assignment $\mathcal{A}$)

```
1  subgraph ← (∅, ∅)
2  remotenodes ← ∅
3  masterMLE^(f(u))(u, A,subgraph,remotenodes)
4  for all s ∈ S in parallel do
5      R_s = {v ∈ remotenodes : f(v) = s}
6      subgraph_s ← (∅, ∅)
7      if (¬(R_s == ∅)) then
8          slaveMLE^(s)(R_s, A, subgraph_s)
9          Merge(subgraph, subgraph_s)
10     endif
11 enddo
12 return subgraph
```

Ablaufbeschreibung

Der Aufruf von XMultiLevelExpand erfolgt mit dem zu expandierenden Knoten u und der Belegung $\mathcal{A}$ von Strukturoptionen und Gültigkeiten, die der Anwender gewählt hat. Nach der Initialisierung (Zeilen 1 und 2) wird die lokale Expansion angestoßen (Zeile 3), die einen vorläufigen Subgraph sowie die Menge der entfernten Knoten zurückgibt, die noch expandiert werden müssen. Die Teilgraphen aller entfernten Server werden parallel angefordert (vgl. Zeilen 4 – 11). Die Merge-Funktion (Zeile 9) erzeugt lediglich die Vereinigung aller resultierender Teilgraphen (d. i. $V = \bigcup_{s \in S} \tilde{V}_s, E = \bigcup_{s \in S} \tilde{E}_s$ mit $(\tilde{V}_s, \tilde{E}_s)$ ist der resultierende Teilgraph von Server s).

6.5.3.2 Expansion am Master

Algorithmus 6.5.5: (Expansion (Master) entsprechend OLL-Katalog)

masterMLE(in node u, in assignment $\mathcal{A}$,
out ($\tilde{V_s}, \tilde{E_s}$), out nodes remotenodes)

```
1  Ṽ_s ← {u}
2  Ẽ_s ← ∅
3  successors ← {u}
4  for all x ∈ successors do
5      successors ← successors \{x}
6      for all x -c-> y ∈ E ∪ OLL|A(c) do
7          if x -c-> y ∈ E then
8              Ẽ_s ← Ẽ_s ∪ {x -c-> y}
9          endif
10         if (f(x) == f(y)) then
11             Ṽ_s ← Ṽ_s ∪ {y}
12             if (¬marked_visited(y)) then
13                 successors ← successors ∪ {y}
14             endif
15         else
16             remotenodes ← remotenodes ∪ {y}
17         endif
18     enddo
19     mark_visited(x)
20 enddo
21 return
```

Ablaufbeschreibung

Die Funktion 'masterMLE' wird mit dem zu expandierenden Zusammenbau u sowie der Belegung $\mathcal{A}$ aufgerufen. Ergebnis sind (1) der Teilgraph, der aus der lokalen Expansion von u resultiert, sowie (2) weitere noch zu expandierende Knoten, die sich an entfernten Servern befinden.

Die beiden verschachtelten Schleifen (beginnend in den Zeilen 4 und 6) traversieren die lokale Substruktur. Dabei werden (vgl. Zeilen 7 und 8) alle lokal gespeicherten Kanten inklusive derer, die einen Serverübergang darstellen, sowie die lokalen Knoten (vgl. Zeilen 10 und 11) aufgesammelt. Zielknoten von Serverübergängen werden separat zusammengefasst (Zeile 16). Um zu vermeiden, dass mehrfach verwendete Knoten unnötigerweise auch mehrfach expandiert werden, erhalten besuchte Knoten eine Markierung (vgl. Zeile 19).

6.5.3.3 Expansion an den Slaves

Algorithmus 6.5.6: (Expansion (Slave) entsprechend OLL-Katalog)

slaveMLE(in R_s, in assignment $\mathcal{A}$, out ($\tilde{V}_s, \tilde{E}_s$))

```
1  Ṽ_s ← R_s
2  Ẽ_s ← ∅
3  successors ← R_s
4  for all x ∈ successors do
5      successors ← successors \{x}
6      for all x -c-> y ∈ E ∪ OLL|A(c) do
7          if x -c-> y ∈ E then
8              Ẽ_s ← Ẽ_s ∪ {x -c-> y}
9          endif
10         if (f(x) == f(y)) then
11             Ṽ_s ← Ṽ_s ∪ {y}
12             if (¬marked_visited(y)) then
13                 successors ← successors ∪ {y}
14             endif
15         endif
16     enddo
17     mark_visited(x)
18 enddo
19 return
```

Ablaufbeschreibung

Im Gegensatz zur Funktion 'masterMLE' wird 'slaveMLE' mit einer *Menge* von Knoten aufgerufen, die jeweils Wurzelknoten einer lokalen Teilstruktur sind. Die Funktion selbst arbeitet analog zu Algorithmus 6.5.5, außer dass keine weiteren entfernten Knoten aufgesammelt werden müssen (d. h. es entfallen die Zeilen 15 und 16 im Algorithmus 6.5.5).

6.6 Komplexitätsbetrachtungen der Algorithmen

6.6.1 Vorbemerkungen

Die Komplexität der im Abschnitt 6.5 beschriebenen Algorithmen kann prinzipiell unter mehreren Aspekten betrachtet werden.

Typischerweise wird bei der Komplexitätsanalyse die *Laufzeit* (Berechnungskosten) in den Mittelpunkt gestellt. Dabei wird letztlich eine Aussage über die Anzahl der Elementaroperationen einer (fiktiven) Implementierung getroffen.

Ein weiterer Gesichtspunkt ist der *Platzbedarf*, der bei der Ausführung des Algorithmus anfällt. Als Maß dient hierbei die Anzahl der benötigten Speicherzellen des zugrunde gelegten Rechnermodells.

Algorithmen, die in verteilten Umgebungen ablaufen, können auch nach ihrem *Kommunikationsaufkommen* zwischen den verteilten Berechnungsorten bewertet werden. Die Anzahl der Kommunikationen (und eventuell auch das übertragene Datenvolumen) ist dabei für die Qualität eines Algorithmus ausschlaggebend.

Entsprechend dem Gedanken der Minimierung von Kommunikationen zwischen Standorten in weltweit verteilten Entwicklungsumgebungen ist für den in dieser Arbeit vorgestellten Ansatz die Betrachtung des Kommunikationsaufkommens von größter Bedeutung. Die jeweils lokal erforderliche Laufzeit sowie der für die Berechnungen benötigte Speicherplatz (sofern er nicht zum übertragenen Datenvolumen beiträgt) spielen in den vorgestellten Algorithmen nur eine untergeordnete Rolle.

In den folgenden Betrachtungen sei $G^c = (V, E, \mathcal{C})$ ein DACG, $|S|$ bezeichne die Anzahl der Standorte (gleichbedeutend mit Server), auf die G^c verteilt ist, und $|P|$ sei die Anzahl der Partitionen von G^c.

6.6.2 Betrachtung der Algorithmen

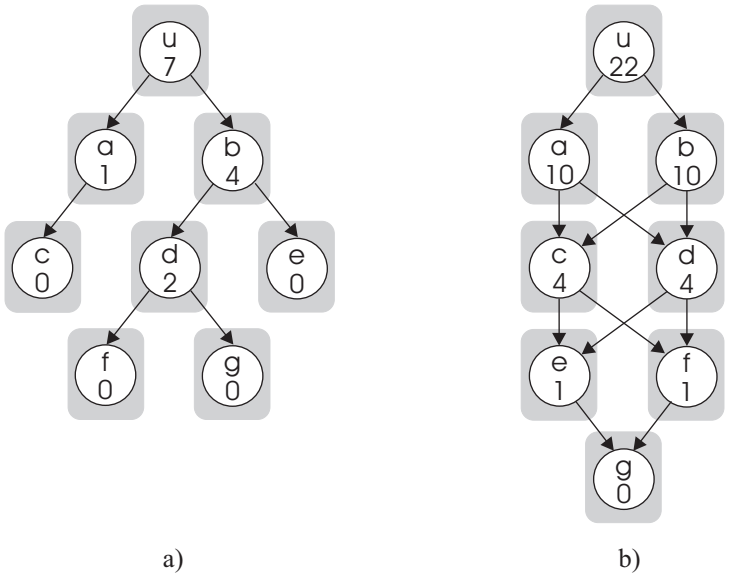
6.6.2.1 Initialisierung

Aus der Definition 6 (OLL-Katalog) geht hervor, dass bei der Konstruktion des OLL-Katalogs alle *Pfade* in G^c zu betrachten sind. Einige Pfade werden dabei zu OLL-Katalog-Einträgen führen, andere nicht. Um eine Komplexität des Initialisierungsvorganges hinsichtlich des Kommunikationsverhaltens angeben zu können, ist demnach die Anzahl der rekursiven Init-Aufrufe im Algorithmus 6.5.1 entlang der Pfade von Bedeutung. Diese lässt sich relativ einfach ermitteln:

Sei $succ(u)$ die Menge aller direkten Nachfolger von u (vgl. Definition 2). Dann kann die Anzahl der (rekursiven) Init-Aufrufe $calls(u)$, die ein Knoten $u \in V$ quasi „verursacht“, wie folgt berechnet werden:

$$calls(u) = \begin{cases} 0, \text{falls } succ(u) = \emptyset \\ \sum_{u_i \in succ(u)} calls(u_i) + |\{u_i \in succ(u) | f(u) \neq f(u_i)\}| \text{ sonst.} \end{cases}$$

Anhand von zwei Beispielen lässt sich die Berechnung von $calls(u)$ demonstrieren. Abbildung 6.11a) zeigt einen Baum, wobei jeder Knoten eine separate Partition darstellen soll, in Abbildung 6.11b) ist ein komplexerer Graph zu sehen. Es gilt jeweils $|P| = |S| = |V| = 8$.



a) b)

Abbildung 6.11: a) DACG als Baum, b) als beliebiger (dichter) Graph. In jedem Knoten i ist jeweils $calls(i)$ mit angegeben.

Offensichtlich hängt die Anzahl der bei der Initialisierung benötigten Kommunikationen von der Gestalt des DACGs ab. Für Bäume gilt $calls(root) = |P| - 1$ (wobei $root$ den Wurzelknoten bezeichne), d. h. die Anzahl der Kommunikationen kann mit $O(|P|)$ abgeschätzt werden. Bei komplexeren Graphen dagegen (vgl. Teilbild b) der Abbildung 6.11) wächst $succ()$ exponentiell mit der Tiefe τ des Graphen, d. h. die Anzahl der Kommunikationen ist in $O(2^\tau)$.

Betrachten wir nun noch das Datenvolumen, welches bei jeder Kommunikation übertragen werden muss. Im Parameter $borders$ der Init-Routine (vgl. Algorith-

mus 6.5.1) werden maximal $|S|$ viele Partitions-Ausgangsknoten übergeben, folglich kann das Datenvolumen pro Kommunikation mit $O(|S|)$ angegeben werden. Insgesamt, d. h. für alle Kommunikationen zusammen, ergibt sich ein Datenvolumen in $O(|S| * |P|)$ für Bäume; komplexe („entartete“) Graphen führen auch hier wiederum zu exponentieller Komplexität.

Da OLL-Katalog-Einträge $u \xrightarrow{c} v$, die entsprechend Algorithmus 6.5.1 am Server $f(v)$ erzeugt werden, auch am Server $f(u)$ gespeichert werden, muss dieser OLL-Katalog-Eintrag an den Server $f(u)$ übertragen werden. Dazu nutzt der Algorithmus den rekursiven Wiederaufstieg aus der Init-Routine. Im schlechtesten Fall wird dabei im Mittel jede Kante $|S|$-mal übertragen. Für Bäume entsteht damit ein Datenvolumen in $O(|S|^2 * |P|)$, für „entartete“ Graphen gilt auch hier wieder exponentielle Komplexität.

Das Auftreten exponentieller Komplexität mag zunächst auf eine unpraktikable Vorgehensweise im Algorithmus 6.5.1 hinweisen. In der Praxis jedoch sind Produktstrukturen in der Art, wie sie in Abbildung 6.11 Teil b) gezeigt ist, nicht anzutreffen. *Mehrfachverwendungen* von Einzelteilen, insbesondere die Mehrfachverwendung von zusammengesetzten Baugruppen sind sehr selten, so dass Produktstrukturen annähernd Baumeigenschaft aufweisen. Damit ist für die Initialisierung ein Kommunikationsaufwand nur wenig schlechter als $O(|P|)$, und für das Datenvolumen etwa $O(|S|^2 * |P|)$ zu erwarten.

6.6.2.2 Änderung der Produktstruktur

Der Update-Algorithmus 6.5.3 benützt die Init-Routine 6.5.1. Je nach Ausprägung der Struktur wird diese Routine auch mehrmals gerufen. Zunächst gilt es, die Anzahl dieser Aufrufe zu bestimmen.

Sei $pred(u)$ die Menge aller direkten Vorgänger von u (vgl. Definition 2). Dann kann die Anzahl der Pfade $backpaths(u)$, die von der Wurzel zu u führen, wie folgt berechnet werden:

$$backpaths(u) = \begin{cases} 1, \text{falls } pred(u) = \emptyset \\ \sum_{u_i \in pred(u)} backpaths(u_i) \text{ sonst.} \end{cases}$$

Für eine neu hinzukommende Kante $u \xrightarrow{c} v$ (Erweiterung der Produktstruktur) beziehungsweise bei der Änderung der Bedingung c an einer Kante $u \xrightarrow{c} v$ definiert $backpaths(u)$ somit die Anzahl der Init-Aufrufe am Knoten v. In Bäumen gilt für alle $u \in V \quad backpaths(u) = 1$. In „entarteten“ Graphen (vgl. Abbildung 6.11b) wächst $backpaths(u)$ wiederum exponentiell mit der Tiefe des Graphen an.

Um die Gesamtzahl der benötigten Kommunikationen des Update-Algorithmus zu bestimmen, wird zusätzlich noch die Anzahl der Serverübergänge bei der Bestimmung der $backpaths(u)$ vielen Pfade benötigt. Analog zur Funktion $calls(u)$ definieren wir eine Funktion $called(u)$, welche diese Anzahl ermittelt:

$$called(u) = \begin{cases} 0, \text{falls } pred(u) = \emptyset \\ \sum_{u_i \in pred(u)} called(u_i) + |\{u_i \in pred(u) | f(u) \neq f(u_i)\}| \text{ sonst.} \end{cases}$$

Entsprechend zur Funktion $calls()$ wächst $called()$ für „entartete“ Graphen exponentiell, für Bäume dagegen ist die Funktion durch $|P|$ nach oben begrenzt.

Die Gesamtzahl der Kommunikationen errechnet sich nun aus der Summe von $called(u)$ und $backpaths(u) * calls(v)$. Für Bäume kann der Kommunikationsaufwand daher wie bei der Initialisierung mit $O(|P|)$ abgeschätzt werden, für komplexe Graphen ist wieder exponentieller Aufwand anzugeben.

Für das Datenvolumen, welches durch die Update-Routine transferiert werden muss, gelten die gleichen Aussagen, die für die Initialisierung gemacht wurden (vgl. Abschnitt 6.6.2.1).

6.6.2.3 Expansion unter Verwendung von OLL-Katalogen

Heute erhältliche PDM-Systeme besitzen für Multi-Level-Expansionen einen Kommunikationsaufwand in $O(|E|)$ für baumartige Produktstrukturen, und exponentiellen Aufwand für „entartete“ Graphen (vergleiche dazu auch Abschnitt 2.3.1).

Unter Verwendung des in dieser Arbeit vorgestellten OLL-Katalogs kann der Kommunikationsaufwand auf $O(|S|)$ beschränkt werden, gleichgültig, ob der zugrunde liegende Graph Baumeigenschaften aufweist oder nicht! Entsprechend Algorithmus 6.5.4 wird der Algorithmus *masterMLE* (Alg. 6.5.5) genau einmal gerufen, der Algorithmus *slaveMLE* (Alg. 6.5.6) maximal $|S| - 1$-mal. Berücksichtigt man zusätzlich noch die parallele Ausführung der $|S| - 1$ Kommunikationen, so ist mit *konstantem* Zeitaufwand bei in $|S|$ linearem Platzbedarf zu rechnen.

Der für die Initialisierung und die Änderungen zu treibende Aufwand wird folglich durch die wesentlich häufiger auftretenden Expansions-Operationen (vgl. Tabelle 2.1) mehr als kompensiert.

6.6.3 Alternative Vorgehensweisen

Insbesondere für die Initialisierung des OLL-Katalogs lassen sich alternative Algorithmen angeben. Die entscheidende Frage dabei ist, ob gerade für speziellere Produktstrukturen mit häufig auftretenden Mehrfachverwendungen von zusammengesetzten Baugruppen der Kommunikationsaufwand verringert werden kann, und mit welchen anderen Kosten dies „bezahlt“ werden muss.

Generell gilt es, das Wechselspiel aus *Platzbedarf* und *Kommunikationskosten* oder *Laufzeit* zu berücksichtigen. Weniger Kommunikation kann etwa bedeuten, dass zumindest temporär mehr Speicherplatz zur Verfügung gestellt werden muss. Umgekehrt gilt, weniger bereitgestellter Speicher führt möglicherweise zu mehr Kommunikation. Beispielsweise kann paralleles Anfragen von Daten an mehreren Servern nur dann tatsächlich zu Performance-Verbesserungen gegenüber sequentieller Anforderung beitragen, falls die - möglicherweise gleichzeitig eintreffenden - Antworten auch quasi parallel abgegriffen, d. h. in den Speicher geschrieben werden können. Dazu muss selbstredend genügend Speicher bereitgestellt werden. Bei sequentieller Anforderung genügt dagegen Speicherplatz für jeweils *eine* (d. h. die volumenmäßig größte) Antwort.

Unter diesem Aspekt werden im Folgenden noch Varianten für die Initialisierung des OLL-Katalogs diskutiert.

6.6.3.1 Nutzung des Wissens über Mehrfachverwendung

Für einen konkret vorliegenden Graph sind die Knoten, die mehrere Väter besitzen, bekannt. Dieses Wissen kann man während der Initialisierung ausnützen, um die Traversierung der mehrfach verwendeten Substrukturen zu verhindern. Wir unterscheiden im Folgenden zwischen *passiver* und *aktiver* Nutzung dieses Wissens.

Passive Nutzung:

Die Grundidee der passiven Nutzung des Wissens über Mehrfachverwendung besteht darin, bereits traversierte Substrukturen von mehrfach verwendeten Knoten zu erkennen und bereits erzeugte OLL-Katalog-Einträge in diesen Substrukturen zur Generierung weiterer Einträge heranzuziehen. Dazu wird der OLL-Katalog zunächst wie folgt erweitert: Zu jedem Eintrag $u \rightharpoonup v$ wird eine Liste von Partitions-Ausgangsknoten gehalten, die zwischen u und v traversiert werden. Die Einträge haben also die Form $u \rightharpoonup v(b_1, b_2, \ldots, b_n)$.

Der Initialisierungsvorgang aus Algorithmus 6.5.1 wird nun folgendermaßen leicht abgewandelt:

Beim Besuch eines Partitions-Ausgangsknotens u wird geprüft, ob u früher bereits besucht wurde, d. h. ob es schon Kanten im OLL-Katalog am Server $f(u)$ gibt, die von u ausgehen. Falls ja, so kann man diese bestehenden Kanten nutzen, um die neuen Kanten zu erzeugen: Sei (b_a, b_b, b_c) die Liste der Ausgangsknoten, die als Input-Parameter der Init-Routine übergeben wurde. Für alle bereits existierenden OLL-Einträge am Server $f(u)$ mit oben genannter Form teste nun, ob Pfade $b_a \rightharpoonup b_b \rightharpoonup b_c \rightharpoonup u \rightharpoonup b_1 \rightharpoonup \ldots \rightharpoonup b_n$ existieren, welche die Eigenschaften erfüllen, um einen gültigen OLL-Katalog-Eintrag $b_a \rightharpoonup b_n(b_b, b_c, u, \ldots, b_{n-1})$ zu erstellen (Achtung: Es müssen auch alle Teilpfade auf OLL-Eigenschaft getestet werden, d. h. Pfade beginnend bei b_b und auch bei b_c, aber auch endend bei b_1 oder b_2 etc.). Falls ja, so nehme diese Einträge in die Liste der new_edges (vgl. Algorithmus 6.5.1) auf. Eine Traversierung des schon einmal besuchten Teilbaums kann jetzt entfallen.

Ein Problem dieses Ansatzes ist jedoch, dass Server, die Anteile (und damit Zielknoten neuer OLL-Katalog-Einträge) der Substruktur speichern, jetzt explizit benachrichtigt werden müssen, dass neue OLL-Katalog-Einträge hinzukommen. D. h. wird ein Teilbaum n-mal verwendet, so werden auch sämtliche partizipierenden Server n-mal benachrichtigt.

Insgesamt gesehen stellt die passive Nutzung des Wissens über Mehrfachverwendung nur eine geringfügige Verbesserung dar.

Aktive Nutzung:

Im Gegensatz zur passiven Nutzung, bei welchem die Mehrfachverwendung erst ausgenutzt wurde, nachdem der Teilbaum bereits traversiert wurde, wird hier mit der Traversierung solange gewartet, bis ein mehrfach verwendeter Knoten v von allen seinen Vorgängern zur Initialisierung aufgefordert wurde, d. h. bis v über alle vorhandenen Pfade erreicht wurde.

Dieser Ansatz setzt eine Parallelisierung der Vorgehensweise voraus (ansonsten wäre das Warten ja eine Blockierung). Da DACGs per Definition azyklisch sind, können dabei keine Deadlocks (Blockierungen) entstehen.

Die Vorgehensweise wird an einem einfachen Beispiel klar: Sei v der mehrfach verwendete Knoten, d. h. es gebe eine Kante $u_1 \rightharpoonup v$ und $u_2 \rightharpoonup v$. Der erste „Traversierungsversuch“ komme mit den Ausgangsknoten (b_1, b_2), der zweite mit (b_3, b_4, b_5). Erst wenn beide Pfade hin zu v betrachtet werden, wird mit der *gemeinsamen* Traversierung fortgefahren. Folglich wird die Init-Routine nicht mit *einer Menge* von Ausgangsknoten im Parameter *borders* aufgerufen, sondern mit einer *Menge von Mengen* von Ausgangsknoten. Somit können zu allen Ausgangsknoten des darüberliegenden Teils des Graphen in nur einem Durchlauf des mehrfach verwendeten Teilbaums die OLL-Katalog-Einträge ermittelt werden. Dabei werden die Aktivitäten, die bislang auf der (einzigen) Menge von Partitions-

Ausgangsknoten durchgeführt werden, *parallel* auf jeder dieser Mengen von Ausgangsknoten durchgeführt.

Jeder Serverübergang wird nun genau einmal traversiert. Die Anzahl der Kommunikationen lässt sich folglich auf $O(|P|)$ senken.

Unproblematisch ist dieses Vorgehen jedoch ebenfalls nicht. Betrachtet man erneut die Graphen, die in Algorithmus 6.5.1 eine exponentielle Anzahl an Kommunikationen zwischen den Servern hervorgerufen haben, so führen diese Graphen nun zu einem exponentiellen Platzbedarf im Parameter „Menge von Mengen von Partitions-Ausgangsknoten“, denn in diesen Mengen verbergen sich die (exponentiell vielen) Pfade, die von der Wurzel der Struktur zu dem mehrfach verwendeten Teilbaum führen.

Fazit: Man kann durch die aktive Nutzung von Mehrfachverwendungswissen zwar die Anzahl der Kommunikationen auf lineare Komplexität drücken, jedoch verlagert man das Problem dann in den Platzbedarf, der von linear zu exponentiell wechselt.

6.6.3.2 Vorgezogene Shortcut-Berechnung

Die an jedem Standort *lokale* Laufzeit der Algorithmen 6.5.1 (Init) und 6.5.3 (Update) lässt sich relativ leicht optimieren. Bei der mehrfachen Traversierung mehrfach verwendeter Knoten müssen an jedem von der Mehrfachverwendung betroffenen Standort die dort vorliegenden Partitionen mehrfach traversiert und dabei die Pfade von den Eingangsknoten zu den Ausgangsknoten mit den zugehörigen Bedingungen ermittelt werden. Diese Arbeit lässt sich auch einmalig *vor* der Initialisierung durchführen, so dass während der Initialisierung (und auch während des Updates) auf diese „vorberechneten Shortcuts“ zugegriffen werden kann.

Die Anzahl der Kommunikationen ändert sich durch diese Optimierung jedoch nicht, auch nicht das Volumen der zu übertragenden Daten.

6.6.3.3 Zentrale Berechnung des OLL-Katalogs

Den Gedanken der vorgezogenen Shortcut-Berechnung aus Abschnitt 6.6.3.2 kann man noch etwas weiter ausbauen: Mittels dieser Shortcuts könnte auf einem zentralen, vorab bestimmten Rechner (im Folgenden als Init-Koordinator bezeichnet) der OLL-Katalog für jeden Standort zentral berechnet werden.

Die Shortcuts jeder Partition P_{s_i} werden dazu lokal in einer Menge $\{(u \xrightarrow{c} v) \mid u \in V^{\top}(P_{s_i}) \wedge v \in V^{\perp}(P_{s_i}) \wedge \exists\, d = \langle u \xrightarrow{c_1} u_1 \xrightarrow{c_2} \ldots \xrightarrow{c_k} v \rangle$ mit $c = c_1 \wedge c_2 \wedge \ldots \wedge c_k\}$ gespeichert. Diese Menge wird zusammen mit den Serverübergängen $E^{\bowtie}(P_{s_i})$ an

den Init-Koordinator geschickt, der daraus einen reduzierten DACG G^c_{red} aufbaut. Abbildung 6.12 zeigt den reduzierten DACG passend zum DACG in Abbildung 6.1.

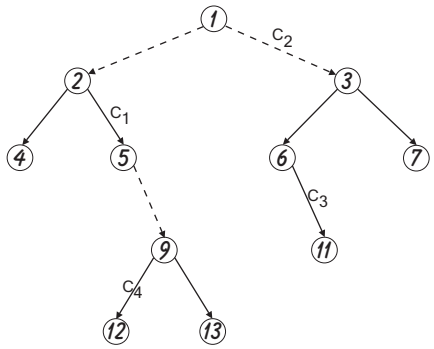


Abbildung 6.12: Reduzierter DACG zu Abbildung 6.1. (Gestrichelte Kanten symbolisieren Shortcuts, durchgezogene Kanten stehen für Serverübergänge)

Dieser reduzierte DACG genügt, um – beispielsweise mit einem angepassten Warshall-Algorithmus, wie er in Abschnitt 6.6.4.1 gezeigt wird – den kompletten OLL-Katalog aufzubauen. Anschließend muss für jeden Standort der OLL-Teilkatalog bestimmt werden, welcher nur Einträge enthält, die relevant für den jeweiligen Standort sind. Diese Teilkataloge werden dann noch an die Standorte verschickt.

Dieses Verfahren benötigt maximal zwei Kommunikationen zwischen jedem Server und dem Init-Koordinator: Eine Kommunikation für das Versenden der Shortcuts und Serverübergänge, und eine weitere für den Empfang des lokalen OLL-Katalogs. Da die Berechnung des OLL-Katalogs lokal, d. h. zentral, erfolgt, bedeuten Mehrfachverwendungen kein erhöhtes Kommunikationsaufkommen! Der Kommunikationsaufwand kann deshalb mit $O(|S|)$ abgeschätzt werden.

Für Graphen, die praktisch relevante Produktstrukturen darstellen, ist der Gewinn relativ gering im Vergleich zum Vorgehen im Algorithmus 6.5.1, welcher nahe bei $O(|P|)$ liegt. Für entartete Produktstrukturen jedoch stellt die zentrale Berechnung eine vielversprechende Alternative dar, auch wenn die Praktikabilität[29] in der Praxis von Fall zu Fall untersucht und bewertet werden muss.

[29]Es ist durchaus denkbar, dass 1. kein Partner/Zulieferer die Rolle des Koordinators übernehmen will (Leistungserbringung evtl. für Konkurrenz?) und 2. die Partner/Zulieferer nur ungern von einer zentralen Instanz abhängig sind.

6.6.4 Vergleich mit bekannten Algorithmen

6.6.4.1 Bildung der transitiven Hülle nach Warshall

Der OLL-Katalog stellt eine Teilmenge der transitiven Hülle eines DACGs dar (vgl. Definition 6). Daher bietet sich ein Vergleich der Initialisierung eines OLL-Katalogs (vgl. Algorithmus 6.5.1) mit dem Warshall-Algorithmus (vgl. [CLR96, Sed95]) zur Bildung der transitiven Hülle in gewöhnlichen gerichteten Graphen an.[30]

Der Warshall-Algorithmus basiert auf folgender Aussage (vgl. [Sed95]):

> *Falls ein Weg existiert, um von Knoten x nach Knoten y zu gelangen, und ein Weg, um von Knoten y nach Knoten z zu gelangen, so existiert auch ein Weg, um von Knoten x nach Knoten z zu gelangen.*

Eine Implementierung dieser Aussage ist (für Graphen, die in einer Adjazenzmatrix a dargestellt sind) im Folgenden angegeben:

```
1  for y := 1 to V do
2      for x := 1 to V do
3          if a[x, y] then
4              for z := 1 to V do
5                  if a[y, z] then a[x, z] := true;
```

Damit ermöglicht der Warshall-Algorithmus die Berechnung der transitiven Hülle in $O(V^3)$ Schritten. Es stellt sich nun die Frage, weshalb die Komplexität des Init-Algorithmus 6.5.1 hiervon abweicht.

Betrachten wir zunächst die Verteilung des Graphs, d. h. eine auf mehrere Standorte verteilte Adjazenzmatrix, bei zentraler Ausführung des Warshall-Algorithmus. Jeder Zugriff auf ein Element der Adjazenzmatrix kann folglich eine Kommunikation zu einem entfernten Standort benötigen. Die Komplexität hierfür liegt, wie leicht zu erkennen ist, in $O(V^3)$, gleichgültig, ob der Graph Baumeigenschaft aufweist oder „entartet“ ist (vgl. Abbildung 6.11). Da der Warshall-Algorithmus zum einen keine „Pfadinformationen“ speichert (d. h. alle Tests in den Zeilen 3 und 5 bedingen Zugriffe auf möglicherweise entfernt liegende Einträge der Adjazenzmatrix) und zum andern zentral kontrolliert wird (d. h. es können auch beim Auftreten von Partitionen mit mehr als nur einem Knoten keine Kommunikationen eingespart werden), kann auch ein Graph, der im Init-Algorithmus

[30] Weitere Algorithmen zur Bildung der transitiven Hülle (auch mit Performance-Vergleichen) finden sich in [ADJ90, AJ90, IRW93].

in $O(|E|)$ (genauer: $O(|P|)$ mit $|P| \leq E$) abgehandelt wird, nicht besser als in $O(V^3)$ ablaufen.

Einen weiteren Aspekt liefert die Definition der transitiven Hülle für DACGs (vgl. Definition 3). Der Warshall-Algorithmus findet quasi das *Vorhandensein eines Weges* zwischen zwei Knoten x und y, im Gegensatz dazu müssen bei der Initialisierung des OLL-Katalogs *alle Wege* zwischen diesen Knoten (zusammen mit den Bedingungen, die auf diesen Wegen gelten) gefunden und festgehalten werden. In der OLL-Katalog-Initialisierung ist also die Anzahl der vorhandenen Pfade ausschlaggebend, nicht jedoch die Anzahl der Kanten und Knoten im DACG. Für entsprechend „entartete“ Graphen ist damit eine Kommunikations-Komplexität exponentieller Größenordnung und damit deutlich schlechter als $O(V^3)$ anzugeben.

Wie bereits in Abschnitt 6.6.3.3 bemerkt, kann ein angepasster Warshall-Algorithmus für die zentrale Berechnung des OLL-Katalogs herangezogen werden. Ziel ist es, anstatt für eine Kante zwischen zwei Knoten u und v nur eine 1 in der Adjazenzmatrix zu verzeichnen, Informationen über den bzw. die Pfade von u nach v zu speichern, aus welchen am Ende des Algorithmus die OLL-Katalog-Einträge abgelesen werden können.

Dazu muss die Adjazenzmatrix erweitert werden, so dass sie Einträge der Form $\{(u_1 - u_2, c_1 \wedge c_2), (u_3, c_3)\}$ aufnehmen kann. Dieser Eintrag in der Matrix am Element $[u, v]$ bedeutet, dass ein Pfad von u nach v über die Knoten u_1 und u_2 existiert, und dabei die Bedingungen c_1 und c_2 erfüllt sein müssen, und ein weiterer Pfad über u_3 mit der Bedingung c_3 existiert. Initial wird diese modifizierte Adjazenzmatrix mit den Informationen aus dem DACG (besser noch ist die Verwendung des reduzierten DACGs) gefüllt.

Der Warshall-Algorithmus wird nun dahingehend geändert, dass die Pfade sukzessive „länger“ werden, wobei die Bedingungen gegebenenfalls erweitert werden:

```
1  for y := 1 to V do
2      for x := 1 to V do
3          if a[x, y] ≠ ∅ then
4              for z := 1 to V do
5                  if a[y, z] ≠ ∅ then a[x, z] := {(d − d', c ∧ c') |
6                      ∃(d, c) ∈ a[x, y] ∧ ∃(d', c') ∈ a[y, z]};
```

Aus der erweiterten Adjazenzmatrix können nun alle Pfade des (reduzierten) DACGs abgelesen werden. Die Anfangs- und Endknoten der Pfade, welche die OLL-Katalog-Eigenschaft erfüllen, werden als OLL-Kante zusammen mit der ermittelten Bedingung in den OLL-Katalog übernommen.

Anmerkung: Der angepasste Algorithmus kann noch weiter optimiert werden, so dass nicht *alle* transitiven Kanten ermittelt werden: Im OLL-Katalog können keine Kanten enthalten sein, deren Anfangsknoten lokale Nachfolger besitzen. Diese Kanten können bereits bei der Berechnung ausgeschlossen werden, indem in Zeile 3 die Bedingung erweitert wird zu $(f(x) \neq f(y))$ ***and*** $(a[x, y] \neq \emptyset)$. Die Bedingung in Zeile 5 hingegen darf nicht gleichermaßen verändert werden, da sonst Pfade zu potentiellen Zielknoten von OLL-Kanten abgeschnitten würden.

6.6.4.2 Kürzeste Wege

Bereits mehrfach wurde darauf hingewiesen, dass die Komplexität der Erzeugung des OLL-Katalogs von der Anzahl der *Pfade* im DACG abhängt. Eine Ähnlichkeit mit Algorithmen, die ebenfalls auf Pfaden operieren, ist somit naheliegend. Besonders Algorithmen zur Bestimmung von kürzesten Wegen zwischen je zwei Knoten des Graphs adressieren auf den ersten Blick eine verwandte Problematik.

Tatsächlich jedoch ist die Bestimmung der kürzesten Wege eng verwandt mit der Bildung der transitiven Hülle: Der Floyd-Warshall-Algorithmus [CLR96, Sed95], der die kürzesten Wege in $O(|V|^3)$ ermittelt, ist stark an den bereits beschriebenen Warshall-Algorithmus angelehnt. Für die Berechnung des OLL-Katalogs können folglich keine neuen Erkenntnisse gewonnen werden.

6.6.5 Ausblick

Sicherlich gibt es hinsichtlich der Erzeugung von OLL-Katalogen und deren Änderung insbesondere im Hinblick auf „Sonderfälle“ von DACGs weiterführende Überlegungen, die den Rahmen dieser Arbeit sprengen würden. Auf einen derartigen Sonderfall sei hier dennoch kurz hingewiesen:

Sei G^c ein DACG, bei welchem jeder Knoten eine eigene Partition darstellt (für Produktstrukturen ist dies nicht sinnvoll, im Allgemeinen jedoch denkbar). Betrachten wir einen Pfad $d = \langle u_1 \rightharpoonup u_2 \rightharpoonup u_3 \rightharpoonup \ldots \rightharpoonup v \rangle$ in diesem DACG. Dann gilt, dass am Server $f(u_1)$ neben dem Knoten u_1 selbst auch die Information $u_1 \rightharpoonup u_2$ vorhanden ist, und am Server $f(u_3)$ liegt neben u_3 auch die Information $u_2 \rightharpoonup u_3$ vor. Die beiden Server $f(u_1)$ und $f(u_3)$ können deshalb offensichtlich *ohne Kommunikation mit dem Server* $f(u_2)$ über die zu erzeugende OLL-Kante $u_1 \rightharpoonup u_3$ entscheiden!

Eine analoge Aussage gilt selbstverständlich für die Knoten u_4 auf dem Server $f(u_4)$, u_6 auf $f(u_6)$ usw. Jede zweite Ebene in der Struktur kann folglich „ausgelassen“ werden. Am Ende der Initialisierung muss diesen Servern lediglich der für sie relevante Teil des OLL-Katalogs übermittelt werden.

Interessant können derartige weitergehende Betrachtungen z. B. auch hinsichtlich des Ausfalls eines Servers während der Initialisierung oder einer Änderungsoperation sein, sie sind jedoch nicht Gegenstand dieser Arbeit.

6.7 Umsetzung in relationalen Datenbanken

Die effiziente Verwendung von OLL-Katalogen kann nur gewährleistet werden, wenn sie unmittelbar in die SQL-Anfragen integriert werden können. In den folgenden Abschnitten wird beschrieben, wie Konfigurationssteuerung auf OLL-Katalogen funktioniert, und wie die rekursiven Expansionen unter Verwendung der OLL-Kataloge in SQL-Anfragen abgebildet werden.

Für die Initialisierungs- und Änderungsalgorithmen geben wir keine Repräsentation in SQL an, da diese die Performance der rekursiven Aktionen nicht beeinflussen. Eine Implementierung dieser Algorithmen in einer geeigneten Programmiersprache (C, Java, etc.) stellt keine große Herausforderung dar und bleibt deshalb ohne weitere Betrachtung.

6.7.1 OLL-Kataloge in relationalen Tabellen

Die Informationen, die im OLL-Katalog gespeichert werden, umfassen auch die Informationen, die ein Objekt der *uses*-Relation enthält. Hinzu kommt noch die Kardinalität, die als Integerwert abgelegt wird. Die Bedingungen, die den OLL-Einträgen zugeordnet werden, erfordern jedoch noch eine eingehendere Betrachtung:

Sei $u \xrightarrow{c} v$ eine Kante des Produktstrukturgraphen G^c. Sind dieser Kante mehrere Strukturoptionen $\chi_1, \chi_2, \ldots, \chi_n$ zugeordnet, so ist v Bestandteil von u, falls der Benutzer mindestens eine Strukturoption $\chi_i, 1 \leq i \leq n$ ausgewählt hat. Zusätzlich müssen dieser Kante Gültigkeiten $\Psi_1, \Psi_2, \ldots, \Psi_m$ zugewiesen sein, so dass die Kante zu dem Zeitpunkt ψ, den der Benutzer vorgegeben hat, gültig ist, d. h. für ein $\Psi_i = [\Psi_i^{start}, \Psi_i^{end})$ mit $1 \leq i \leq m$ muss gelten $\Psi_i^{start} \leq \psi < \Psi_i^{end}$. Es gilt also $c = (\bigvee_{i=1}^{n} \chi_i) \wedge (\bigvee_{i=1}^{m} \Psi_i)$.

Für Objekte der *uses*-Beziehung wurde diese Konjunktion aufgebrochen und jede der beiden Disjunktionen jeweils auf ein mengenwertiges Attribut ('StrcOpts' und 'Effectivities', vgl. Abbildung 2.12) abgebildet.

Sei nun $u \xrightarrow{c_1} v \xrightarrow{c_2} w$ ein Pfad in G^c dergestalt, dass $u \xrightarrow{c_3} w$ einen gültigen Eintrag im OLL-Katalog darstellt (auf die Angabe der Kardinalität kann bei den folgenden Betrachtungen verzichtet werden). Entsprechend der Definition 6 gilt $c_3 = c_1 \wedge c_2$ und damit:

$$
\begin{aligned}
c_3 &= \left(\left(\bigvee_{i=1}^{n}\chi_i\right) \wedge \left(\bigvee_{i=1}^{m}\Psi_i\right)\right) \wedge \left(\left(\bigvee_{i=k}^{l}\chi_i\right) \wedge \left(\bigvee_{i=1}^{l}\Psi_i\right)\right) \\
&= \left(\left(\bigvee_{i=1}^{n}\chi_i\right) \wedge \left(\bigvee_{i=k}^{l}\chi_i\right)\right) \wedge \left(\left(\bigvee_{i=1}^{m}\Psi_i\right) \wedge \left(\bigvee_{i=1}^{l}\Psi_i\right)\right)
\end{aligned}
$$

Bricht man die Konjunktion wieder auf, so dass die Teilausdrücke entweder nur Strukturoptionen oder nur Gültigkeiten enthalten, so liegt jeweils dennoch eine Konjunktion von Disjunktionen vor. Mit der von der *uses*-Beziehung bekannten Abbildung von Disjunktionen auf mengenwertige Attribute kann dies nicht unmittelbar dargestellt werden. Folgende Erweiterung schafft Abhilfe:

Das Attribut 'Strcopt' enthält bei OLL-Einträgen eine *Menge von Mengen*, wobei die Elemente der „inneren" Mengen jeweils die Disjunktionen darstellen, die „äußere" Menge steht für die Konjunktion (vgl. Abbildung 6.13).

...	Strcopt	Effectivities	...
	$\Big\{ \underbrace{\{\chi_1, \chi_2\}}_{Disjunktion}, \underbrace{\{\chi_1, \chi_3\}}_{Disjunktion}, \underbrace{\{\chi_4, \chi_5, \chi_6\}}_{Disjunktion} \Big\}$ $\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{Konjunktion}$	$\Big\{ \underbrace{\{\Psi_1, \Psi_2\}}_{Disjunktion}, \underbrace{\{\Psi_1, \Psi_3, \Psi_4\}}_{Disjunktion} \Big\}$ $\underbrace{\qquad\qquad\qquad\qquad}_{Konjunktion}$	

Abbildung 6.13: Abbildung von Bedingungen an OLL-Kanten auf mengenwertige Attribute

6.7.2 Konfigurationssteuerung für OLL-Kataloge

Entsprechend der Table Function zur Konfigurationssteuerung der *uses*-Beziehung (vgl. Abbildung 3.13) kann auch eine Table Function definiert werden, die auf der Darstellung der OLL-Kataloge entsprechend Abbildung 6.13 operiert. Abbildung 6.14 zeigt eine entsprechende Funktionsdefinition, welche mithilfe der Übersetzungsfunktionen (vgl. Anhang A.2) erstellt wurde.

Das Ergebnis dieser Table Function ist ein Ausschnitt aus dem OLL-Katalog, welcher nur Einträge enthält, die konform zu den übergebenen Strukturoptionen und der angegebenen Gültigkeit sind.

```
CREATE FUNCTION t_OLL (options {id VARCHAR(24)}, effcty VARCHAR(10))
RETURNS
TABLE (obid VARCHAR(24), pred VARCHAR(24), succ VARCHAR(24), ...)
LANGUAGE SQL RETURN
SELECT o.obid, o.pred, o.succ, ...
FROM o IN OLL
WHERE (o.StrcOpts IS EMPTY OR options IS EMPTY
       OR (FOR_ALL opts IN o.StrcOpts :
       (opts INTERSECT options) IS NOT EMPTY))
AND (o.Effectivites IS EMPTY OR effcty IS NULL
     OR (FOR_ALL effs IN o.Effectivites : EXISTS(SELECT * FROM effs
     WHERE effcty BETWEEN effs.Start AND effs.End)))
```

Abbildung 6.14: Berücksichtigung der Konfigurationssteuerung im OLL-Katalog über eine *Table Function*

6.7.3 Expansion mit OLL-Katalogen

Interessant für die Umsetzung mittels SQL sind die beiden Algorithmen 6.5.5 (masterMLE) und 6.5.6 (slaveMLE) aus Abschnitt 6.5.3. Der Koordinationsalgorithmus 6.5.4 kann dagegen in jeder geeigneten Programmiersprache (C, Java, etc.) geschrieben werden.

Um auch hier analog zu Kapitel 4 rekursives SQL einsetzen zu können, müssen die Algorithmen etwas angepasst werden. Betrachten wir zunächst den Algorithmus 6.5.5 zur Expansion am Master:

- Das Ergebnis der Rekursion muss wiederum eine Menge homogener Tupel darstellen. Es ist also nicht möglich, neben dem Graph noch eine Menge entfernter Knoten zurückzugeben. Das Aufsammeln der *remotenodes* (vgl. Zeile 16) lässt sich jedoch einfach durch Rückgabe der traversierten OLL-Kanten realisieren, da in diesen Einträgen die entfernten Knoten referenziert werden.
- Die im Algorithmus angegebene Optimierung bei mehrmaligem Besuch eines Knotens muss der Rekursions-Implementierung des Datenbanksystems überlassen werden.

Zusätzlich muss noch folgende Einschränkung berücksichtigt werden: In Abschnitt 4.4.1 wurde bei der frühzeitigen Zugriffsregelauswertung darauf geachtet, im Ergebnis keine Kanten zurückzuliefern, deren Zielknoten nicht im Ergebnis enthalten sind. In verteilten Umgebungen können wir diesen Test nur lokal durchführen, d. h. nur für Kanten, deren Zielknoten ebenfalls lokal vorliegen. Kanten,

die einen Serverübergang darstellen (auch OLL-Kanten!), *müssen* in das Zwischenergebnis aufgenommen werden, auch wenn noch nicht definitiv bestimmt werden kann, ob das Zielobjekt im Ergebnis sein wird. Würde die Kante fehlen, so könnte – falls der Zielknoten im Ergebnis liegt – der Ergebnisgraph nicht korrekt rekonstruiert werden.

Die Expansion am Master lässt sich durch das rekursive SQL-Statement aus Abbildung 6.15 ausdrücken. Die Verwendung des OLL-Katalogs erfolgt dabei analog zur *uses*-Beziehung.

Die Expansion an den Slaves verläuft nach dem gleichen Muster. Der Unterschied besteht lediglich darin, dass (1) die Expansion mit einer *Menge* von Startobjekten angestoßen werden kann, und (2) keine entfernten Knoten zurückgegeben werden, d. h. die OLL-Kanten werden zwar traversiert, aber nicht in das Ergebnis aufgenommen. In Abbildung 6.16 sind die Änderungen gegenüber Abbildung 6.15 skizziert.

Selbstverständlich können die beiden hier vorgestellten rekursiven Anfragen in Table Functions versteckt werden. Der Vorteil dieser Lösung besteht darin, dass besonders bei entfernten Anfragen nicht das doch recht komplexe SQL-Statement übertragen werden muss, es genügt stattdessen lediglich der Funktionsaufruf mit den entsprechenden Parametern.

6.7.4 Hinweise zur Regelauswertung

Im Abschnitt 5.5 wurden die Probleme bei der Auswertung der ∀rows conditions und der tree aggregate conditions beschrieben. Selbstverständlich kann der Einsatz von OLL-Katalogen die dort angesprochenen Probleme auch nicht lösen. Im Gegenteil, zunächst mag es scheinen, als ob die frühzeitige Regelauswertung kaum noch durchführbar wäre:

Sei $d = \langle u \overset{c_1}{\rightharpoonup} n_1 \overset{c_2}{\rightharpoonup} n_2 \ldots n_k \overset{c_{k+1}}{\rightharpoonup} v \rangle$ ein Pfad im DACG G^c und $u \overset{c}{\rightharpoonup} v$ eine diesem Pfad entsprechende Kante im OLL-Katalog. Seien o.B.d.A. $u \overset{c_1}{\rightharpoonup} n_1$ und $n_k \overset{c_{k+1}}{\rightharpoonup} v$ Serverübergänge. Ferner sei n_k für den anfragenden Benutzer auf Grund einer row condition nicht zugreifbar.

Das Verfahren zur Verwendung des OLL-Katalogs wird nun bei einem Multi-Level-Expand des Knotens u auch eine Anfrage an den Server $f(v)$ stellen, da zu diesem Zeitpunkt nicht klar ist, dass der Knoten n_k – und damit auch dessen Subgraph, zu dem auch v gehört – nicht zugreifbar ist.

Versagt hier also das Prinzip, möglichst kurze Antwortzeiten durch möglichst wenig Datenübertragung zu erzielen? Die Antwort lautet: Nein! Die Anfrage an den Server $f(v)$ wurde *parallel* zu anderen Anfragen (insbesondere zu jener an den

```
WITH  ProdStr(LVL,TYPE,OBID,PN,NC,LOC,
      REV,SEQ,ASY,MSTR,ASDB,MDB,VER,VDB,...) AS
(
      SELECT 0, 'ASY', OBID, PartNumber, NULL, Location, Revision,
      Sequence, NULL(6),...
      FROM TABLE(t_Assembly($User, 'display')) AS Assembly
      WHERE PartNumber=<PN> AND Revision=<Rev> AND Sequence=<Seq>
      AND Assembly_permits('expand', Assembly.obid, $User)

        UNION ALL

      SELECT ProdStr.LVL+1, 'USE', uses.OBID, NULL(5), uses.pred,
      uses.succ, uses.predLoc, uses.succLoc, NULL(2), ...
      FROM TABLE(t_uses($UsrKonfig, $UsrEff)) AS uses, ProdStr
      WHERE uses.pred=ProdStr.OBID
      AND (uses.predLoc = uses.succLoc AND
      (AssmMstr_permits('display', uses.succ, $User)
      OR CmpMstr_permits('display', uses.succ, $User))
      OR uses.predLoc <> uses.succLoc)
      AND Assembly_permits('expand', uses.pred, $User)

        UNION ALL

      SELECT ProdStr.LVL+1, 'OLL', OLL.OBID, NULL(5), OLL.pred,
      OLL.succ, OLL.predLoc, OLL.succLoc, NULL(2), ...
      FROM TABLE(t_OLL($UsrKonfig, $UsrEff)) AS OLL, ProdStr
      WHERE OLL.pred=ProdStr.OBID
      AND (OLL.predLoc = OLL.succLoc AND
      (AssmMstr_permits('display', OLL.succ, $User)
      OR CmpMstr_permits('display', OLL.succ, $User))
      OR OLL.predLoc <> OLL.succLoc)
      AND Assembly_permits('expand', OLL.pred, $User)

        UNION ALL

      SELECT ProdStr.LVL+1, 'ASM', am.OBID, am.PartNumber,
      am.Nomenclature, am.Location, NULL(8), ...
      FROM TABLE(t_AssmMstr($User, 'display')) AS am, ProdStr
      WHERE am.OBID=ProdStr.MSTR

        UNION ALL

      ...
)
SELECT DISTINCT *
FROM ProdStr
ORDER BY LVL
```

Abbildung 6.15: Rekursives SQL-Statement (Ausschnitt) zur lokalen Expansion (Master) der Produktstruktur eines Zusammenbaus entsprechend den Zugriffsrechten und den Konfigurationsvorgaben eines Benutzers.

```
WITH  ProdStr(LVL,TYPE,OBID,PN,NC,LOC,
      REV,SEQ,ASY,MSTR,ASDB,MDB,VER,VDB,...) AS
(
      SELECT 0, 'ASY', OBID, PartNumber, NULL, Location, Revision,
      Sequence, NULL(6),...
      FROM TABLE(t_Assembly($User, 'display')) AS Assembly
      WHERE (PartNumber, Revision, Sequence) IN {(<PN>, <Rev>, <Seq>)}
      AND Assembly_permits('expand', Assembly.obid, $User)
        UNION ALL
      ...
)
SELECT DISTINCT *
FROM ProdStr
WHERE TYPE <> 'OLL'
ORDER BY LVL
```

Abbildung 6.16: Rekursives SQL-Statement (Ausschnitt) zur lokalen Expansion (Slave).

Server $f(n_k)$) gestellt, wodurch in der Regel keine nennenswerte Verzögerung zu erwarten ist. Sicherlich werden Daten wieder unnötigerweise übertragen, doch ist dies hierbei der zu zahlende Preis für a) die Parallelität der Anfragen sowie b) die Bündelung der Anfragen an einen Server. Der Gesamtnutzen wird dadurch jedoch nur unwesentlich beeinflusst.

6.8 Integration des OLL-Katalogs in die Architektur eines PDM-Systems

Abbildung 6.17 zeigt die Architektur aus Abschnitt 3.5.6 erweitert um den OLL-Katalog sowie dessen Management-Komponente und den Structured Object Builder.

Die Komponenten *Translator* und *Table Function Generator* arbeiten wie in Abschnitt 3.5.5 beschrieben. Ebenfalls unverändert in der Funktion ist auch der *Object Wrapper*.

Der *Query Generator* wird nun nicht mehr für rekursive Anfragen eingesetzt. Für einfache Anfragen, z. B. für die Suche nach Zusammenbauten mit frei wählbaren Eigenschaften, setzt der Query Generator jetzt nicht mehr auf die Basis-Tabellen, sondern auf die Table Functions, die zur Regelauswertung eingeführt wurden.

Der OLL-Katalog wird an jedem Standort in der Datenbank des PDM-Systems abgelegt. Der *OLL Catalog Manager* dient der Initialisierung und Aktualisierung

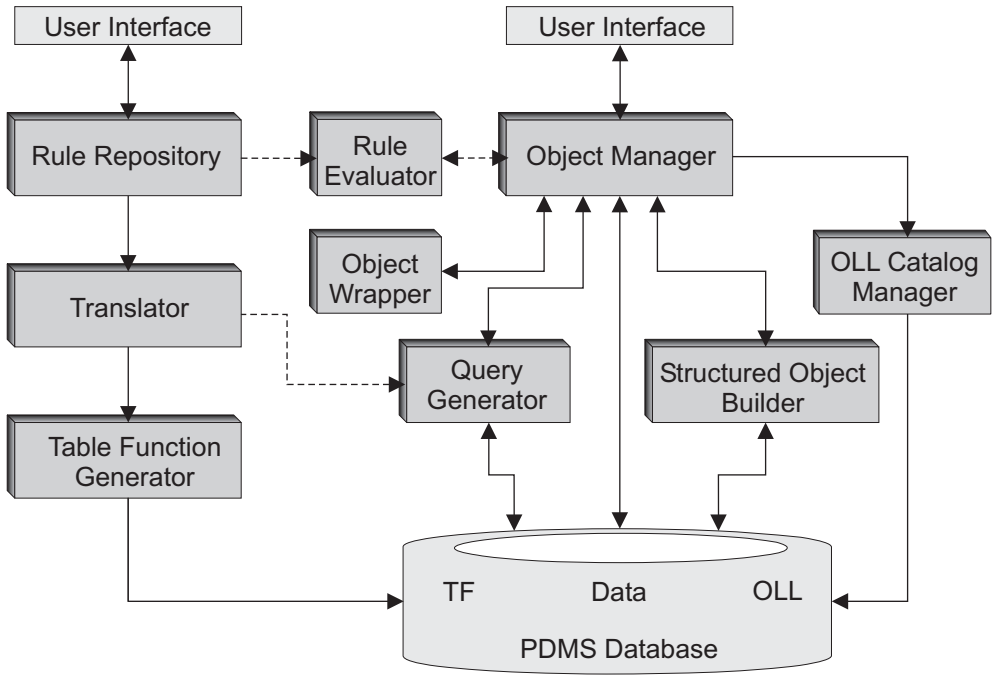


Abbildung 6.17: Architektur der Anfragekomponente mit OLL-Katalog

des OLL-Kataloges nach Änderungen durch den Objekt-Manager. In ihm werden die Algorithmen aus dem Abschnitt 6.5.2 implementiert.

Eine weitere neue Komponente, der *Structured Object Builder*, wurde eingeführt, um die Verwendung des OLL-Katalogs in den Expansions-Algorithmen zu verdeutlichen. Dieser Baustein stellt rekursive Anfragen an die Datenbank entsprechend den Abbildungen 6.15 und 6.16, bzw. ruft die Table Functions an der Datenbank, welche diese Algorithmen implementieren. Die Interpretation des lokalen Ergebnisses sowie die Aufrufe an entfernte Datenbanken werden im Falle des Anfrage-Masters ebenfalls vom *Structured Object Builder* vorgenommen, der auch die Teil-Ergebnisse dieser Anfragen zum Gesamt-Ergebnis zusammensetzt.

6.9 Beweis der Vollständigkeit des OLL-Katalogs

Die Korrektheit des OLL-Katalogs erschließt sich aus der Vollständigkeit des Katalogs, d. h. der Katalog enthält alle benötigten Kanten, um einmaligen Zugriff auf entfernte PDM-Server zu garantieren.

Zu zeigen: Die Verwendung des OLL-Katalogs führt dazu, dass bei der Expansion eines beliebigen Knotens jeder beteiligte PDM-Server genau einmal angesprochen

werden muss. Wir zeigen, dass keine Partition existiert, die durch die Definition des OLL-Katalogs nicht berücksichtigt wird, d. h. bei der Expansion unter Zuhilfenahme des OLL-Katalogs werden keine Teile der Produktstruktur „vergessen“.

Beweis-Verfahren: Widerspruchsbeweis.

Annahme: Sei $G^c = (V, E, \mathcal{C})$ ein DACG und E' ein OLL-Katalog zu G^c. Es existiere eine Partition P mit Startknoten v, die bei der Expansion eines übergeordneten Knotens u mit der erweiterten Version des Multi-Level-Expands (vgl. Algorithmus 6.5.4) nicht erreichbar ist. O.B.d.A., falls es in einem Pfad von u zu einem 'Blattknoten' mehrere solche nicht erreichbaren Partitionen mit den Startknoten v_i gibt, so wähle für v jenes v_i, welches von u in G^c mit den wenigsten Serverübergängen erreicht wird (d. h. alle Knoten zwischen u und v sind von u aus erreichbar!).

Beweis:
Da P bei der Expansion von u „vergessen“ wird, kann in E keine Kante existieren, welche die Partition von u mit P verknüpft. Folglich ist P in G^c nur indirekt mit der Partition von u verbunden. Somit existiert in G^c ein Pfad $\langle u \xrightarrow{c_1} n_1 \xrightarrow{c_2} \ldots \xrightarrow{c_k} n_k \xrightarrow{c_{k+1}} v\rangle, k \geq 1$ mit: $f(n_k) \neq f(v)$ (nach Voraussetzung) und es existiert ein $i \in [1,k] \mid f(n_i) \neq f(u)$. Es existieren nun zwei Fälle:

1. Fall: $\exists i \in [1,k[\mid f(n_i) = f(v)$

   Folglich gibt es auch ein $j = max\{i | i \in [1,k[\wedge f(n_i) = f(v)\}$. Nach der Definition des OLL-Katalogs gilt dann $n_j \xrightarrow{c} v \in E'$. Da nach Voraussetzung bei der Expansion von u der Knoten n_j erreicht wird, wird auch v erreicht (und damit P expandiert). Widerspruch zur Annahme.

2. Fall: $\nexists i \in [1,k[\mid f(n_i) = f(v)$

   Sei $n_0 = \begin{cases} u, \text{falls } f(u) \neq f(n_i), 1 \leq i \leq k \\ n_j | j = \max\{i | 1 \leq i \leq k \wedge f(n_i) = f(u)\} \text{ sonst.} \end{cases}$

   Aus der Voraussetzung folgt zunächst, dass n_0 von u aus erreichbar ist (falls $n_0 \neq u$, andernfalls trivial). Nach der Definition des OLL-Katalogs gilt $n_0 \xrightarrow{c} v \in E'$, und somit ist bei der Expansion von u der Knoten v erreichbar. Widerspruch zur Annahme. □

Kapitel 7

Effizienzanalyse mittels Simulation

7.1 Simulation

7.1.1 Einführung und Überblick

Simulation ist ein „schillerndes“ Wort, das für eine ganze Reihe von Untersuchungstechniken (und manches mehr) verwendet wird. So wird beispielsweise in der Produktentwicklung das Verhalten eines Bauteils während eines Bearbeitungsprozesses (etwa im Zerspanungsprozess) simuliert, Strömungseigenschaften eines Flugzeuges können durch Simulation ermittelt und überprüft werden, und Flugsimulatoren können zur Aus- und Weiterbildung von Piloten eingesetzt werden, indem sie die Flugeigenschaften von Flugzeugen „nachspielen“. Im Folgenden werden diese Arten der Simulation keine Rolle spielen. Vielmehr steht die Simulation von Computer-Anwendungssystemen im Mittelpunkt [Krü75, Nee87].

Das Ziel dieser Art von Simulation ist die quantitative Bewertung der Leistung eines Rechensystems. Dabei wird die Bearbeitung einer vorgegebenen *Last* durch eine *modellierte Maschine* nachvollzogen, wobei charakteristische *Größen* beobachtet und anschließend ausgewertet werden.

Alternativ zu einer Simulation kann die Bewertung eines Rechensystems auch durch Messungen – oftmals als *Monitoring* bezeichnet – an einem realen System erfolgen. Selbstverständlich kann eine derartige Leistungsbewertung nur durchgeführt werden, wenn das System bereits implementiert ist und mit der für die Messung gewünschten Last eingesetzt wird. Schwachpunkte können damit allerdings nur im Nachhinein festgestellt werden, nötige Verbesserungen an den Systemen sind zu diesem Zeitpunkt nur schwer und typischerweise mit hohem Kostenaufwand durchführbar.

Die Vorteile simulativer Analysen im Gegensatz zum Monitoring liegen damit auf der Hand: Im Vergleich zum realen System können Simulationsprogramme zumeist mit relativ geringem Aufwand implementiert werden. Setzt man solche Programme bereits in der Planungsphase des realen Systems ein, so können *Vorhersagen* über das zu erwartende Verhalten des Systems gemacht werden. Auch sind Simulationen von „Was-wäre-wenn…"-Szenarien möglich, so dass mehrere Alternativen beispielsweise hinsichtlich der System-Architektur evaluiert werden können und Fehlentscheidungen vermieden bzw. frühzeitig korrigiert werden können, noch bevor sie das reale System beeinflussen.

Im Weiteren erweist sich das Monitoring von realen Systemen oft auch als schwer durchführbar, beispielsweise wenn sich die Beobachtung eines Systems über mehrere Tage, Wochen oder gar Monate hinziehen würde, um einen längerfristigen Trend feststellen zu können. Simulationen können hier helfen, äquivalente Aussagen in wesentlich kürzerer Zeit zu erhalten, indem das Verhalten des realen Systems quasi *im Zeitraffer* durchgespielt wird.

Simulationen besitzen allerdings auch einen Nachteil: Sie arbeiten auf einem *Modell*, das mehr oder weniger gut der Realität entspricht. Zu stark von der Realität abstrahierende Modelle können zu Ergebnissen führen, die ebenfalls realitätsfern sind, d. h. die Qualität des Simulationsergebnisses hängt stark von der Qualität des Modells ab. Hochwertige, exakte und damit komplexere Modelle jedoch erhöhen den Implementierungs- und Simulationsaufwand erheblich und schmälern damit auch den genannten Vorteil, billiger und zeitsparend implementierbar zu sein! Es gilt folglich, für jedes konkrete Simulationsszenario eine gute Mischung aus Realitätstreue und akzeptablem Aufwand zu finden.

In der vorliegenden Arbeit wird die Technik der Simulation verwendet, um die Tauglichkeit und das Potential des OLL-Katalog-Ansatzes zusammen mit rekursiver Anfrageauswertung nachzuweisen, ohne eine reale Implementierung vornehmen zu müssen. Eine solche Implementierung wäre ohne Unterstützung eines kommerziellen PDM-Systemherstellers auf Grund der Komplexität der PDM-Systeme nicht in akzeptabler Zeit durchführbar, so dass Messungen an realen Implementierungen nicht möglich sind. In einer Simulation jedoch, in welcher nur die leistungsbestimmenden Aspekte des Ansatzes implementiert werden müssen, kann der Nutzen des neuen Ansatzes sehr wohl nachgewiesen werden.

7.1.2 Ereignisorientierte Simulation

Ein Rechensystem kann als *zustandsdiskretes System* aufgefasst werden. Unter dem Zustand eines Systems versteht man dabei den Zustand, Inhalt oder Wert aller Komponenten eines Rechensystems zu einem gegebenen Zeitpunkt: den Zustand

aller Register, Zähler, Statusbits, Speicherzellen des Arbeitsspeichers, aber auch die Auslastung des Netzes, des Prozessors etc.

Durch Aktionen, die das Rechensystem ausführt, werden Zustandsänderungen vorgenommen. Die Zustandsänderungen finden dabei an diskreten Zeitpunkten statt. Jede Aktion (mit anderen Worten: jeder Auslöser einer Zustandsänderung) wird dabei als *Ereignis* oder *Event* bezeichnet.

Bei der ereignisorientierten Simulation imitiert das Simulationsprogramm durch die Abarbeitung einer Folge von Ereignissen (dies entspricht der Last des Systems) exakt definierte Zustandsänderungen. Dazu müssen zunächst die Zustände des zu simulierenden Systems auf die Datenstrukturen des Simulators abgebildet werden, die Zustandsänderungen werden von sogenannten *Ereignis-Routinen* des Simulators vorgenommen. Während eines Simulationslaufes (d. h. Ausführung des Simulators) können Zustände beobachtet und für die nachfolgende Auswertung aufgezeichnet werden.

In der Literatur werden zwei Klassen ereignisorientierter Simulatoren genannt, die hier kurz gegenübergestellt werden sollen.

7.1.2.1 Event-Scheduling

Im Mittelpunkt von Simulationen nach dem Event-Scheduling-Ansatz steht die *Ereignisliste*. Sie ist vergleichbar einem Kalender aus zukünftigen, für diskrete Zeitpunkte vorgemerkten Ereignissen (siehe Abbildung 7.1).

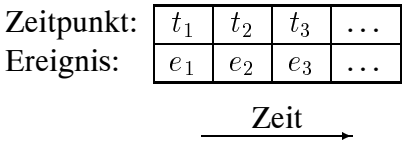


Abbildung 7.1: Ereignisliste

Eine *Simulationshauptschleife* arbeitet die Ereignisliste ab, indem für das zeitlich nächste Ereignis die zugehörige Ereignisroutine gerufen wird und das Ereignis anschließend aus der Liste entfernt wird.

Durch die Ereignisroutinen können weitere Events „geplant“ werden. Diese sind dazu in der zeitlich korrekten Reihenfolge in die Ereignisliste einzufügen. Die Simulation endet, wenn keine geplanten Events anstehen, d. h. die Ereignisliste leer ist.

Der Event-Scheduling-Ansatz lässt sich prinzipiell in jede Programmiersprache einbetten (Beispiele siehe [P+88]). Es gibt jedoch auch Sprachen, die diesen Ansatz mit speziellen Anweisungen der Art

PLAN <event_type>, <event_time>

unmittelbar unterstützen. In diesen Sprachen muss die Simulationshauptschleife nicht explizit programmiert werden, sie ist als Bestandteil des Laufzeitsystems standardmäßig vorhanden. Beispiele für solche Sprachen sind SIMSCRIPT [KVM68] und SLAM [Pri84]. Für FORTRAN gibt es eine Erweiterung namens GASP [Pri74], die ebenfalls den Event-Scheduling-Ansatz verfügbar macht.

7.1.2.2 Process-Interaction

Um die Dynamik eines Modells zu spezifizieren, muss man nicht notwendigerweise jedem einzelnen Ereignis „nachrennen“. Jeweils eine Menge von Ereignissen gehören typischerweise zusammen, sie sind quasi Ereignisse „im Leben“ von wohlidentifizierbaren Objekten des Modells. Für jedes dieser Objekte lässt sich ein Verhaltensmuster, auch *Prozessmuster* genannt, angeben, welches das Verhalten des Objektes entlang der Zeit beschreibt. Das „Leben“ eines Objektes kann demnach mit dem Ablaufen eines Prozesses gleichgesetzt werden. Abbildung 7.2 zeigt am Beispiel eines Bankschalters zwei Objekttypen (Kunde und Bediener) mit ihren zugehörigen stark vereinfachten Prozessmustern.

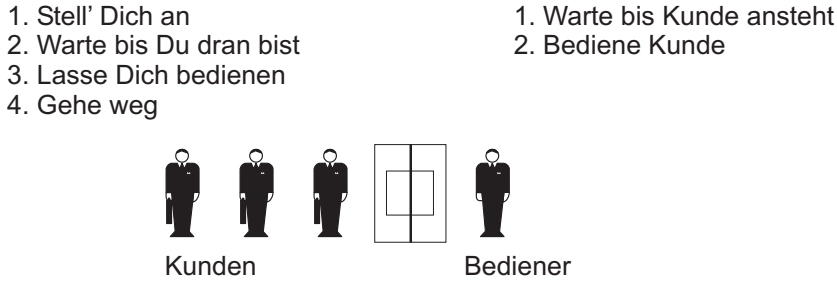


Abbildung 7.2: Process-Interaction; vereinfachtes Prozessmuster am Beispiel eines Bankschalters

Viele derartige Prozesse laufen – entsprechend der realen Welt – typischerweise parallel ab. Dabei sind sie nicht unabhängig voneinander. Zumindest gelegentlich haben sie „etwas miteinander zu tun“, d. h. sie müssen interagieren (im Beispiel stellen „Lasse Dich bedienen“ und „Bediene Kunde“ eine Interaktion dar).[31]

Während des Ablaufes einer Simulation können Prozesse gestartet, angehalten und wieder angestoßen werden, sie können auch auf das Eintreten eines Ereignisses (z. B. auf das Verfügbarwerden eines Betriebsmittels) warten.

[31] Daraus leitet sich auch der Begriff *Process Interaction* ab!

Das Denken in parallelen Prozessen wird als natürlicher empfunden und ist weniger komplex als das Denken in Einzelereignissen des Event-Scheduling-Ansatzes. Nachteilig jedoch ist, dass der Process-Interaction-Ansatz nur in Programmiersprachen verwendet werden kann, die eine gewisse Parallelität unterstützen, d. h. es muss ein Koroutinen- oder Thread-Konzept vorhanden sein. GPSS [Sch91] ist eine spezielle Sprache hierfür, die sich besonders zur Spezifikation von Warteschlangen-Problemen eignet. Für C++ existiert beispielsweise eine Erweiterung in Form einer Programmbibliothek, CSIM18 [Sch96b], die den Process-Interaction-Ansatz verfügbar macht bzw. die Handhabung der für diese Art der Simulation erforderlichen Objekte und Prozesse stark vereinfacht.

7.2 Simulation von PDM-Systemarchitekturen

Auf abstraktem Level betrachtet kann jedes Anwendungsprogramm als „Bedienstation“ entsprechend Abbildung 7.2 aufgefasst werden: Benutzer – sie entsprechen den Kunden – setzen Aufträge ab, die vom System – dem Bediener – ausgeführt werden müssen. Kommen mehr Aufträge an als vom System bearbeitet werden können, so „stauen“ sich die Aufträge in einer Warteschlange.

Es ist damit durchaus naheliegend, für die Simulation von PDM-Systemarchitekturen den Process-Interaction-Ansatz zu verwenden. In den folgenden Abschnitten werden der Simulationsaufbau, die Simulationsdurchführung sowie die dabei erzielten Resultate erläutert.

7.2.1 Simulationsaufbau

Ziel der Simulation ist es, ein PDM-System zu simulieren, welches in weltweit verteilten Umgebungen eingesetzt wird und einen OLL-Katalog zusammen mit rekursiver Anfrageauswertung einsetzt. Dabei sollen die Netzbelastung sowie die zu erwartenden Antwortzeiten des Systems in verschiedenen Umgebungen beobachtet werden. Um den Benefit gegenüber heute verfügbaren Systemen zu zeigen, ist es sinnvoll, parallel dazu auch heutige Architekturen zu simulieren und die Ergebnisse gegenüberzustellen.

7.2.1.1 Komponenten des Simulators

Für die Simulation genügt es, folgende Komponenten eines PDM-Systems (vgl. Architektur in Abbildung 6.17) zu modellieren:

Object Manager implementiert die Methoden für die Aktivitäten des PDM-Systems (z. B. *Query* und *Expand* für heute verfügbare Systeme).

Structured Object Builder enthält die neuen Algorithmen zur Verwendung des OLL-Katalogs bei Expansionen der Produktstruktur.

Datenbankmanagementsystem simuliert die Anfragebearbeitung des PDM-DBMS.

Fileserver wird nur benötigt, falls in der Simulation auch Filetransfers berücksichtigt werden sollen.

Clients erzeugen die zu simulierende Last. Ihre Aufgaben können über separat abgelegte Profile individuell gesteuert werden.

Auf Grund des geplanten Einsatzes in Weitverkehrsnetzen muss eine Simulation neben den Komponenten des PDM-Systems auch Komponenten eines WANs (und auch LANs) berücksichtigen. Dazu zählen:

Leitungen zwischen den Standorten. Es wird davon ausgegangen, dass je zwei Standorte miteinander kommunizieren können, ohne einen dritten als Vermittler zu benötigen.

Gateways zur Verbindung der Leitungen zu einem Netzwerk.

Netzwerkadapter zur Verbindung der PDM-Komponenten mit dem Netz (vergleichbar einer Netzwerkkarte im PC).

Messages als Informationsträger für den Datenaustausch unter Verwendung des Netzwerks.

7.2.1.2 Konfiguration des Simulationsszenarios

Die Eigenschaften des zu simulierenden Systems lassen sich in einer Parameterdatei konfigurieren. Zunächst wird die Anzahl der Standorte festgelegt. Im Anschluss daran werden für jeden Standort die Anzahl der Benutzer und Object Manager angegeben. Jeder Standort kann auch über einen eigenen Datenbankserver verfügen.[32] Für jeden Object Manager ist noch der zu verwendende Datenbankserver anzugeben. Die durchschnittliche Bearbeitungszeit der Anfragen am Datenbankserver kann ebenfalls festgelegt werden.

[32] Existiert nur ein Datenbankserver im gesamten System, so lässt sich damit der zentrale Fall aus Kapitel 4 simulieren.

Auch die Parameter der Netzverbindungen werden über die Parameterdatei konfiguriert. Latenzzeiten und Übertragungsraten (für WAN und LAN jeweils getrennt) sowie die Frame- und Paketgrößen (für einfache Anfragen sowie für das Request/Acknowledge-Handling) sind die zu definierenden Steuergrößen.

Abbildung 7.3 zeigt eine Beispiel-Konfiguration, wie sie in ähnlicher Form in mehreren Simulationstestläufen verwendet wurde.

Die Art der Last, d. h. das Prozessmuster für die Clients, wird in einer separaten Konfigurationsdatei angegeben. Ein Prozessmuster legt die Aktivitäten (Login, Expansionen, Queries, Check-Out etc.) der Clients in ihrer zeitlichen Reihenfolge fest. Da in realen Umgebungen die Benutzer nicht nur mit dem PDM-System arbeiten, sondern typischerweise auch z. B. CAx-Werkzeuge zum Einsatz kommen und die Aktionen in den verschiedenen Systemen wechselweise durchgeführt werden, müssen zwischen zwei PDM-Aktivitäten „Pausen“ simuliert werden. Die Länge dieser Verzögerungen ist zufällig verteilt, die Erwartungswerte werden mit in der Konfigurationsdatei angegeben.

In einer weiteren Konfigurationsdatei wird die den Expansionsvorgängen zu Grunde liegende Produktstruktur beschrieben. Die Partitionen der Struktur werden dabei – im Gegensatz zur Realität – nicht fest an einen Datenbankserver gebunden, sondern *relativ zum lokalen Netz* des Object Managers adressiert, der die Expansion kontrolliert (Master): Die Partition mit dem Toplevel-Element wird immer als lokal vorhanden angesehen, die weiteren Partitionen werden dann entsprechend ihrer Bezeichnung in der Konfigurationsdatei auf die vorhandenen Datenbankserver verteilt. Beispiel: Liegt eine Partition aus der Sicht eines Masters des Subnetzes N_1 auf dem Datenbankserver im Subnetz N_2, so liegt genau dieselbe Partition aus der Sicht eines Masters des Subnetzes N_3 auf dem Datenbankserver im Subnetz N_4. Ist in N_4 kein Datenbankserver installiert, so wird jener in N_5 verwendet (oder in N_6, falls auch N_5 ohne Datenbankserver konfiguriert wurde usw.). Bei identischer Konfiguration *aller* Subnetze bedeutet dies, dass auch die Simulationsergebnisse aller Subnetze identisch sind und daher die Beobachtung auf Aktivitäten von Benutzern in *einem* Subnetz beschränkt werden kann, obwohl ein verteiltes System mit potentiell hoher Last zu Grunde liegt. Kürzere Simulations- und Auswertungs-Zeiten sind die Folge.

7.2.1.3 Protokoll des Simulators

In jedem Simulationslauf werden eine Vielzahl von Daten erfasst und für die spätere Auswertung aufgezeichnet. Dazu zählen:

- Anzahl der Kommunikationen sowie der Pakete, die über jede einzelne Leitung übertragen werden

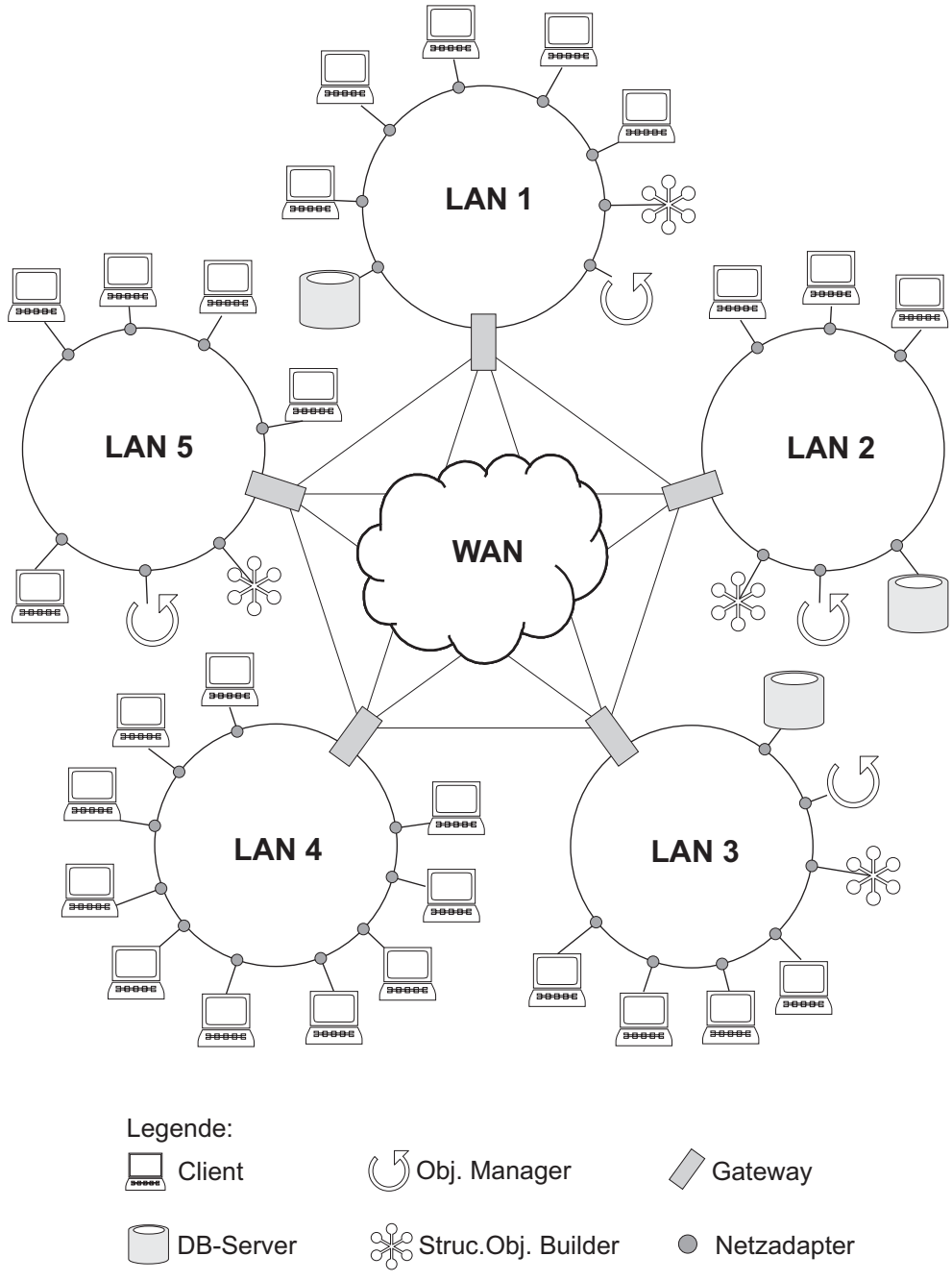


Abbildung 7.3: Konfiguration eines Simulators (Beispiel)

- Auslastung des Netzes
- Antwortzeit für ein komplettes Prozessmuster
- Antwortzeit für jede einzelne Aktivität (nach Typen sortiert)
- Auslastung der Object Manager und Datenbankserver

Die Protokollierung erfolgt bei Zustandsänderungen. Beispielsweise wird beim Versenden einer Nachricht vom Client zum Object Manager die Anzahl der gesendeten Pakete zusammen mit dem Sendezeitpunkt vom Netzadapter des Clients aufgezeichnet. Ebenso protokolliert jedes Gateway, das die Nachricht passiert, das Weiterreichen sowie den zugehörigen Zeitpunkt dieser Aktivität. Die dabei entstehenden Folgen von Beobachtungen werden im Anschluss an die Simulation verdichtet und zu aussagekräftigen Zahlen aufbereitet.

7.2.1.4 Implementierung

Die Implementierung der beschriebenen Komponenten erfolgte in der Programmiersprache C++ unter Verwendung der Programmbibliothek CSIM18. Diese Bibliothek stellt unter anderem Objekte für die Simulationsbeobachtung und -auswertung zur Verfügung, ebenso werden Mechanismen zum Starten, Beenden und Synchronisieren von Prozessen sowie Klassen und Methoden für das Event-Handling bereitgestellt. Verschiedene Zufallszahlen-Generatoren sind ebenfalls verfügbar.

Das Quellprogramm des Simulators umfasst etwa 4000 Zeilen Programmcode. Ein zu simulierendes Szenario wird mit etwa 60 Parametern definiert.

7.2.1.5 Test des Simulators

Die Überprüfung des Simulators auf sinnvolle Ergebnisse lässt sich anhand der Beispielszenarien aus Abschnitt 2.3.2.2 durchführen. Stellvertretend werden hier nur die Ergebnisse des Multi-Level-Expands aus dem Szenario mit den Parametern $\tau = 3, \nu = 9, \sigma = 2/3, T_{Lat} = 0.3s, dtr = 1024kBit/s$ vorgestellt.

Die naive Methode erforderte rein rechnerisch für den Multi-Level-Expand etwa 385 Sekunden, die optimierte Version lediglich knapp fünf Sekunden. Die Simulation ermittelte 390 Sekunden bzw. ebenfalls knapp fünf Sekunden für die gleiche Aktion. Die Abweichung für die naive Methode erklärt sich dadurch, dass die Simulation auch Einflüsse lokaler Netze berücksichtigt, die im mathematischen Modell jedoch keine Beachtung finden.

Da die übrigen Szenarien aus Abschnitt 2.3.2.2 vergleichbar gute Resultate liefern, kann der Simulator als zuverlässig angesehen werden.

7.2.2 Simulationsdurchführung und Ergebnisse

Für die Simulationsdurchführung wurde ein im Vergleich zur Realität kleines Basis-Szenario gewählt. Tests mit größeren Szenarien haben gezeigt, dass die Ergebnisse sehr ähnlich sind, die Simulationszeit jedoch entsprechend höher ausfällt. Deshalb werden für sämtliche im Folgenden beschriebenen Simulationsläufe folgende Basis-Parameter festgelegt:

Simuliert wird ein PDM-System in einem Entwicklungsverbund bestehend aus fünf Partnern. Jeder Partner betreibt ein lokales Netz, in welchem jeweils 25 Anwender mit dem PDM-System arbeiten. Jeder Partner kann mit den anderen Partnern direkt kommunizieren. Auf den Netzen wird eine Paketgröße von 4 Kilobytes angenommen. Jeder Anwender führt während des Simulationslaufes 40 primitive Datenbankanfragen sowie 40 Multi-Level-Expansionen aus (zufällig verteilt auf einen simulierten Arbeitstag). Die Datenstruktur, die den Expansionen zu Grunde liegt, entspricht der aus Abbildung 7.4.

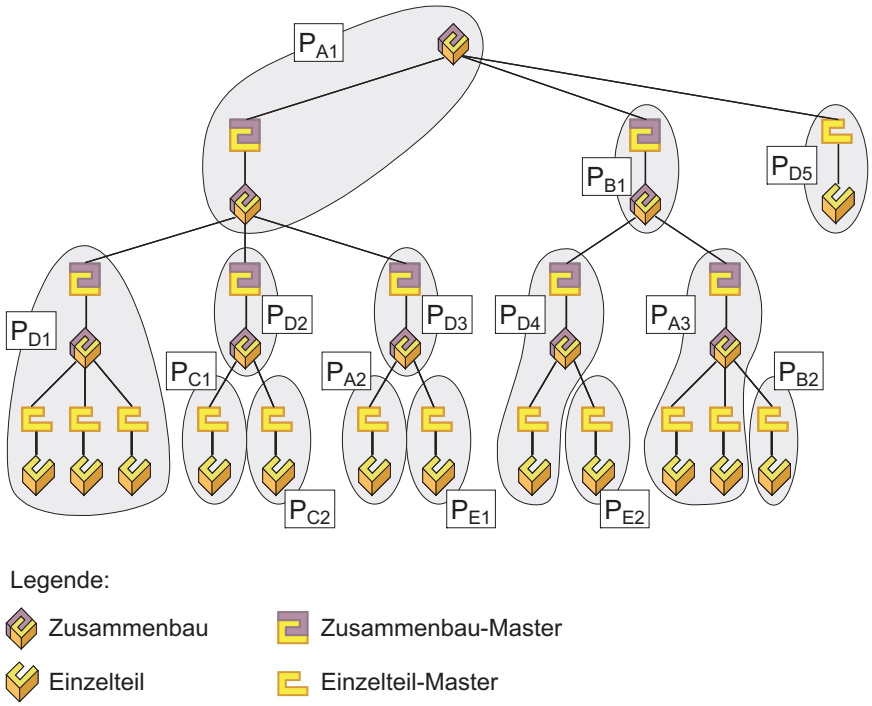


Abbildung 7.4: Partitionierte Produktstruktur für die Simulationen

In Abbildung 7.5 sind die weiteren Netzwerk-Parameter sowie der simulierte Datenbank-Typ (verteilte Datenbank oder zentrale Datenbank) angegeben.

Szenario	T_{Lat} in ms		dtr in kBit/s		verteilte	zentrale
	WAN	LAN	WAN	LAN	DB	DB
1	600	3	512	102400	x	
2	600	3	512	102400		x
3	300	3	1024	102400	x	
4	300	3	1024	102400		x
5	150	3	10240	102400	x	
6	150	3	10240	102400		x

Abbildung 7.5: Netzparameter und Datenbank-Verteilungstyp

Jedes Szenario wird sowohl nach heute üblichem Verfahren als auch mit den Verbesserungen durch Rekursion in der Anfrage und Verwendung des OLL-Katalogs simuliert. Jede Simulation wird dabei 1000 Mal wiederholt, so dass pro Szenario 5.000.000 Expansionen und ebenso viele primitive Anfragen simuliert werden. Dabei lassen sich recht exakte Mittelwerte erzielen, „Ausreißer“ können erkannt und eliminiert werden.

7.2.2.1 Simulation mit verteilter Datenbank

Die Simulationsergebnisse der Szenarien 1, 3 und 5 hinsichtlich Kommunikationsverhalten und Antwortzeiten sind in den Abbildungen 7.6 und 7.7 dargestellt.

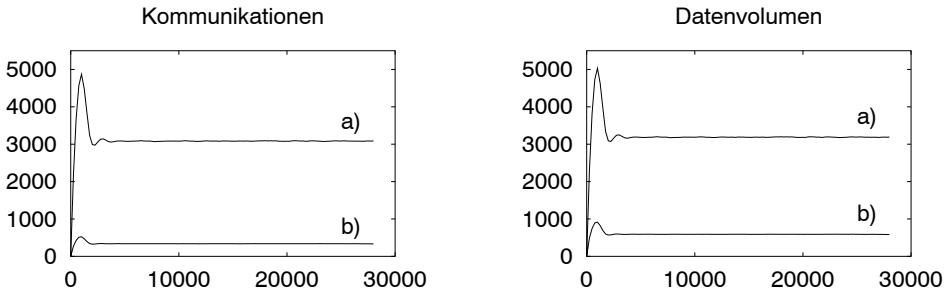


Abbildung 7.6: Kommunikationsaufkommen und Datenvolumen in den Szenarien 1, 3 und 5

Simulationsergebnisse, die auf einem PDM-System nach heute üblicher Architektur basieren, sind in den Diagrammen und Tabellen mit „a)“ gekennzeichnet, die Ergebnisse mit den angeführten Optimierungen mit „b)“.

Das Diagramm „Kommunikationen“ in Abbildung 7.6 zeigt die Anzahl der WAN-Kommunikationen über der simulierten Zeit. Es ist deutlich zu sehen, dass zu

Szenario 1	Antwortzeit gesamt		Antwortzeit Expand	
	in s	95%-KI	in s	95%-KI
Simulation a)	2607.45	$+/-$ 0.55	63.48	$+/-$ 0.00
Simulation b)	124.34	$+/-$ 0.05	1.76	$+/-$ 0.00
Einsparung in %	95.23		97.23	

Szenario 3	Antwortzeit gesamt		Antwortzeit Expand	
	in s	95%-KI	in s	95%-KI
Simulation a)	1336.02	$+/-$ 0.08	32.67	$+/-$ 0.00
Simulation b)	67.71	$+/-$ 0.02	0.98	$+/-$ 0.00
Einsparung in %	94.93		97.00	

Szenario 5	Antwortzeit gesamt		Antwortzeit Expand	
	in s	95%-KI	in s	95%-KI
Simulation a)	663.51	$+/-$ 0.01	16.24	$+/-$ 0.00
Simulation b)	34.06	$+/-$ 0.01	0.51	$+/-$ 0.00
Einsparung in %	94.87		96.86	

Abbildung 7.7: Antwortzeiten der Szenarien 1, 3 und 5

Beginn der Simulation das Kommunikationsaufkommen rapide ansteigt, danach wieder stark abfällt und sich schließlich auf einen relativ konstanten Wert einschwingt. Dieses Überschwingen zu Beginn ist damit zu erklären, dass sich alle Benutzer innerhalb kurzer Zeit in das PDM-System einloggen und mit ihrer ersten Aktivität beginnen. Die zufällig verteilten Pausen zwischen den einzelnen Aktivitäten führen dann im Weiteren zum „eingeschwungenen Zustand“ des Systems.[33]

Das gleiche Verhalten ist auch für die Anzahl der transportierten Pakete zu beobachten (siehe Diagramm „Datenvolumen“ in Abbildung 7.6).

Ein Vergleich der Ergebnisse aus den Simulationen a) und b) zeigt, dass durch die Optimierungen die Anzahl der Kommunikationen etwa auf ein Neuntel, das Datenvolumen auf weniger als ein Fünftel gesenkt werden können! In den Antwortzeiten (vgl. Abbildung 7.7) schlägt sich diese Verbesserung noch deutlicher nieder, beispielsweise in Szenario 1: Über das gesamte Profil (d. h. alle 40 primitiven Anfragen zusammen mit den 40 Multi-Level-Expansionen) benötigt die Simulation a) mehr als 45 Minuten Antwortzeit, mit den Optimierungen wird diese Zeit auf gut zwei Minuten gesenkt – eine Einsparung von über 90%! Betrachtet man

[33] Am Ende der Simulation, d. h. wenn alle Clients nach und nach ihre Arbeit beenden, geht die Anzahl der Kommunikationen auf Null zurück. Dieser Ausschwingvorgang spielt für die Betrachtungen hier momentan keine Rolle und wurde daher in den Diagrammen ausgeblendet.

nur die Antwortzeiten für Expansionen, so fällt das Verhältnis sogar noch positiver aus. Der Grund hierfür liegt in den simulierten Benutzerprofilen: Primitive Anfragen erfahren durch die Verwendung von OLL-Katalogen und rekursiven Anfragen in Multi-Level-Expansionen nur insofern eine geringfügige Verbesserung, als die Netze weniger belastet sind und somit hin und wieder Verzögerungen (z. B. auf Grund von Kollisionen) entfallen. Die für diese Anfragen anfallende Antwortzeit bleibt also nahezu konstant, Einsparungen sind hauptsächlich über die Expansionen zu erzielen.

Die Spalten mit der Bezeichnung „95%-KI“ in Abbildung 7.7 erfordern noch eine besondere Betrachtung: Simulationen liefern nur angenäherte Ergebnisse, niemals jedoch exakte Werte. Um die Qualität eines Simulationsergebnisses zu bestimmen, bedient man sich deshalb sehr häufig einer Methodik aus der Stochastik, den sogenannten *Konfidenz-Intervallen* (vgl. [Lan92]). Ein 95%-Konfidenz-Intervall (kurz: 95%-KI) besagt, dass mit einer Wahrscheinlichkeit von 95% der tatsächliche Wert innerhalb dieses Intervalls liegt. Für das Szenario 1 beispielsweise bedeutet dies, dass der tatsächliche Wert der gesamten Antwortzeiten in a) zu 95% im Intervall $[2607.45 - 0.55, 2607.45 + 0.55]$ liegt (gerundet auf zwei Stellen). Aus den Werten in Abbildung 7.7 geht hervor, dass die Simulation genügend oft wiederholt wurde, um genügend kleine Konfidenz-Intervalle zu bekommen, so dass die aus den Ergebnissen abgeleiteten Aussagen korrekt und die Vergleiche der Ergebnisse sinnvoll sind.

Die beobachteten Antwortzeiten sind hauptsächlich auf Übertragungszeiten zurückzuführen. Die Bearbeitungsdauer der Methodenserver und Datenbankserver bleibt weitgehend unberücksichtigt (lediglich ein „Pauschal-Aufwand“ von einigen Millisekunden pro Server-Aktivität wird jeweils eingerechnet).

7.2.2.2 Simulation mit zentraler Datenbank

Die Simulationsergebnisse der Szenarien mit zentraler Datenhaltung (Szenarien 2, 4 und 6) sind in den Abbildungen 7.8 und 7.9 zu sehen.[34]

Da in diesen Szenarien nur *ein* Subnetz für die Datenhaltung zuständig ist, müssen alle Anfragen und Ergebnisse zu Benutzern in anderen Subnetzen über das WAN transportiert werden. Bei flüchtiger Betrachtung würde man vielleicht sowohl beim naiven als auch beim optimierten Ansatz mit einem Anstieg des Kommunikationsaufkommens und des Datenvolumens rechnen. Die Simulationsergebnisse zeigen jedoch, dass dieser Anstieg nur beim naiven Ansatz eintritt – beim

[34] Auf die Angabe der Konfidenzintervalle wurde hier bewusst verzichtet; sie liegen in der selben Größenordnung wie in Abbildung 7.7.

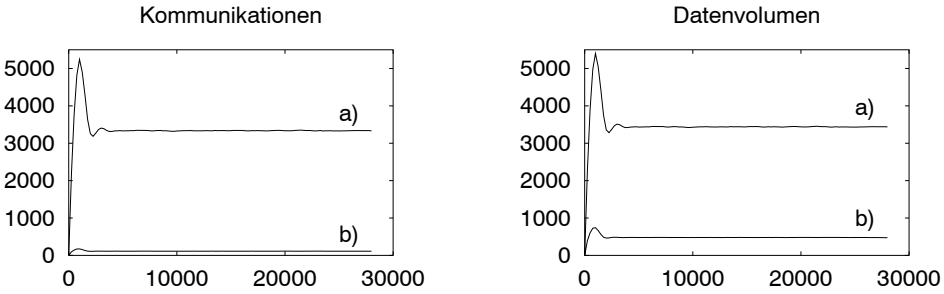


Abbildung 7.8: Kommunikationsaufkommen und Datenvolumen in den Szenarien 2, 4 und 6

Szenario 2	gesamt in s		Expand in s	
	lokal	remote	lokal	remote
Simulation a)	123.35	3658.63	2.98	88.27
Simulation b)	7.91	146.90	0.10	1.99
Einsparung in %	93.59	95.98	96.64	97.75

Szenario 4	gesamt in s		Expand in s	
	lokal	remote	lokal	remote
Simulation a)	124.88	1824.64	3.02	44.65
Simulation b)	7.91	76.83	0.10	1.04
Einsparung in %	93.67	95.79	96.69	97.67

Szenario 6	gesamt in s		Expand in s	
	lokal	remote	lokal	remote
Simulation a)	125.54	886.25	3.03	21.74
Simulation b)	7.91	33.44	0.10	0.43
Einsparung in %	93.70	96.23	96.70	98.02

Abbildung 7.9: Antwortzeiten der Szenarien 2, 4 und 6

optimierten Ansatz dagegen ist sogar eine leichte Reduktion des Kommunikationsaufkommens und Datenvolumens zu beobachten (vgl. Abbildungen 7.8 und 7.6)!

Die Ursache ist in der Verteilung der Struktur zu finden: Im Falle der Datenverteilung geht jeder Standort davon aus, die Partitionen A_i lokal zur Verfügung zu haben, der Rest wird als entfernt angenommen. Das bedeutet, dass für einen Multi-Level-Expand eines Benutzers nach der optimierten Methode (d. h. mit

OLL-Katalogen) eine Anfrage an den lokalen Datenbankserver und 4 Anfragen an entfernte Instanzen gesendet werden. Für je einen Multi-Level-Expand aus allen fünf simulierten Netzen sind somit $5 * 4 = 20$ entfernte Anfragen nötig. Bei zentraler Datenhaltung hingegen genügen von jedem Standort aus eine Anfrage an den Datenbankserver, d. h. insgesamt werden für jeweils einen Multi-Level-Expand aus den fünf Netzen nur 4 entfernte Anfragen benötigt (ein Standort kann auf die Daten lokal zugreifen, deshalb sind nur 4 und nicht 5 entfernte Anfragen erforderlich).

Ähnliche Überlegungen gelten auch für das Datenvolumen. Im verteilten Fall werden pro Multi-Level-Expand 25 Objekte jeweils lokal ermittelt, die vier entfernten Standorte liefern insgesamt 56 Objekte. Für die Übermittlung dieser Objekte und der korrespondierenden Datenbank-Anfragen sind 12 Pakete erforderlich. Bei je einem Expansionsvorgang in jedem Netz werden demnach $5 * 12 = 60$ Pakete übertragen. Wird dagegen eine zentrale Datenhaltung eingesetzt, so kann die komplette Struktur in 11 Paketen an die entfernten Standorte transportiert werden. Berücksichtigt man noch die dazu erforderliche Anfrage, so sind insgesamt für jeweils einen Multi-Level-Expand aus jedem der fünf Netze $4 * (11 + 1) = 48$ Pakete über das WAN zu übertragen.

Eine andere Verteilung der Daten kann selbstverständlich zu anderen Ergebnissen führen: Je höher der Prozentsatz an lokalen Daten im verteilten Fall ist, desto höher wird im Falle zentraler Datenhaltung das zu übertragende Datenvolumen.

Eine genauere Analyse des Kommunikationsverhaltens zeigt, dass mit den in dieser Arbeit vorgestellten Optimierungen bei zentraler Datenhaltung das Kommunikationsaufkommen (über das WAN) auf unter 5%, das Datenvolumen auf unter 20% im Vergleich zu den „naiven“ Vorgehensweisen gesenkt werden können! Entsprechend bessere Antwortzeiten sind daher zu erwarten.

Die Tabellen in Abbildung 7.9 zeigen die mittleren Antwortzeiten jeweils für das komplette Profil sowie für eine Expansion. Dabei wird unterschieden zwischen Anwendern, die in dem Subnetz mit der zentralen Datenbank angesiedelt sind, die also lokal auf die Datenbank zugreifen können (Spalte „lokal“), und Benutzern, die nur über das WAN auf die Datenbank zugreifen können (Spalte „remote“).

Erwartungsgemäß sind die Antwortzeiten für die Benutzer im Netz der zentralen Datenbank nahezu unabhängig von der Qualität des Wide Area Netzes, über welches die entfernten Benutzer zugreifen. In diesem lokalen Netz sind daher die beobachteten Zeiten nahezu konstant. Die geringfügige Verschlechterung, z. B. von 123.35 auf 125.54 Sekunden für das komplette Profil in Szenario 2 bzw. 6, ist damit zu erklären, dass die Anfragen der entfernten Benutzer auf Grund der schnelleren WAN-Verbindung in Szenario 6 in kürzeren Abständen an die zentrale Datenbank gestellt werden – das lokale Netz, in welchem sich die Datenbank

befindet, muss folglich die gleiche Last in kürzerer Zeit bewältigen und gerät an die Kapazitätsgrenzen bzw. überschreitet diese, so dass auch für die Benutzer vor Ort etwas längere Antwortzeiten anfallen.

Für die entfernten Benutzer lassen sich wie erwartet die Antwortzeiten durch bessere Netze senken – eine „Verdopplung“ der Netzqualität (d. h. Halbierung der Latenzzeit und Verdopplung der Datentransferrate) bewirkt dabei etwa eine Halbierung der Antwortzeit, sowohl in Simulation a) als auch in Simulation b). Leider ist in der Praxis diese Netzverbesserung nicht beliebig fortführbar, je nach räumlicher Ausdehnung des Netzes und Lokalisation der Standorte muss immer mit einer Latenzzeit gerechnet werden, die nicht mit der von lokalen Netzen vergleichbar ist.

7.2.2.3 Erweiterte Simulationsszenarien

Nach den bereits beschriebenen Szenarien bleiben noch die Fragen offen, wie sich ein „optimiertes“ PDM-System verhält, wenn

- die Strukturen größer werden oder
- im Falle von zentraler Datenhaltung in entfernten Subnetzen ausschließlich Clients eingesetzt werden (ohne Structured Object Builder und Object Manager).

Mit zwei weiteren Simulationen lassen sich auch Vorhersagen für diese Fälle angeben.

Betrachten wir zunächst den zweiten Punkt. Die Frage nach Subnetzen ausschließlich mit Clients ist durchaus berechtigt, denn auf den ersten Blick scheint es aus der Sicht des Kommunikationsaufkommens gleichgültig zu sein, ob für einen Multi-Level-Expand der *Client* mit nur *einer* Anfrage an den entfernten Object Manager auskommt (wie im zweiten Punkt angenommen), oder ob ein lokal angesprochener Object Manager *eine* Kommunikation zum entfernten Datenbankserver benötigt (mit einer rekursiven Datenbank-Anfrage). Es wäre also zu erwarten, dass die Antwortzeiten dieser beiden Varianten sehr ähnlich sein würden.

Als Grundlage für ein derartiges Szenario (im Folgenden als Szenario 7 bezeichnet) dienen die Parameter aus Szenario 6. Geändert wurden lediglich die Subnetze, die keinen Datenbankserver enthalten; sie besitzen ausschließlich Clients (im Szenario 6 waren auch dort Object Manager und Structured Object Builder installiert).

Das Simulationsergebnis bestätigt zunächst den beschriebenen ersten Eindruck. Die Anzahl der Kommunikationen über das WAN sowie das dabei transportierte Datenvolumen sind in beiden Teil-Szenarien („primitiv“ und optimiert) absolut identisch und entsprechen erwartungsgemäß den Werten des Szenarios 6 bei optimierter Anfrage-Strategie (vgl. Abbildung 7.8). Wie ein Vergleich der Werte in der Tabelle in Abbildung 7.10 zeigt, differieren dennoch die Antwortzeiten der beiden Varianten a) und b) zum Teil erheblich.

Szenario 7	gesamt in s		Expand in s	
	lokal	remote	lokal	remote
Simulation a)	126.32	152.61	3.04	3.37
Simulation b)	7.92	34.04	0.10	0.43
Einsparung in %	93.73	77.69	96.71	87.24

Abbildung 7.10: Antwortzeiten des Szenarios 7

Die Ursache hierfür liegt in der äußerst unterschiedlichen Belastung des lokalen Netzes, in welchem sich die zentrale Datenbank befindet. Abbildung 7.11 zeigt, dass das Kommunikationsaufkommen im Teil-Szenario a) mehr als zehnmal höher ist als in b), und das Datenvolumen in a) etwa das Vierfache dessen in b) beträgt!

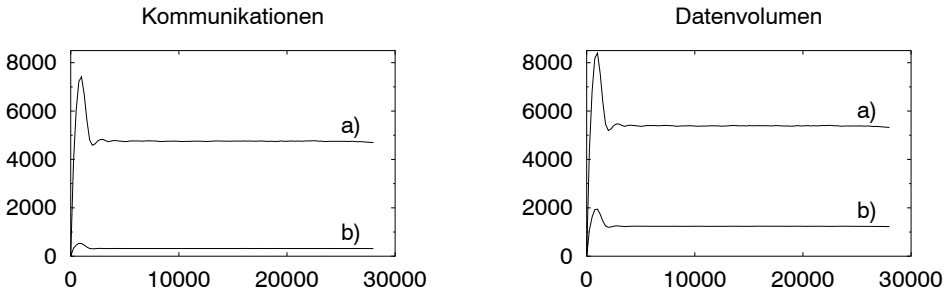


Abbildung 7.11: Kommunikationsaufkommen und Datenvolumen im Subnetz der Datenbank im Szenario 7

Das erhöhte Kommunikationsaufkommen in der Simulation a) führt offensichtlich zu den vergleichsweise längeren Antwortzeiten. Es gilt also auch in Szenarien wie dem hier gezeigten der Grundsatz, möglichst wenige Kommunikationen und möglichst geringes transportiertes Datenvolumen zu erzeugen.

In den bisher simulierten Szenarien wurde eine relativ kleine Produktstruktur expandiert. Die folgende Simulation (im Weiteren als Szenario 8 bezeichnet) basiert auf dem Szenario 3, die Größe der Produktstruktur jedoch wird verfünffacht: Vier

(beliebige) Blattknoten der Datenstruktur aus Abbildung 7.4 werden dazu lediglich durch die gesamte abgebildete Struktur ersetzt.

Berücksichtigt man den *konstant bleibenden Anteil* von Anfragen an entfernte Datenbankserver, die durch die Query-Aktionen initiiert werden, so ist – für die Simulation der „naiven“ PDM-System-Architektur – im WAN etwa das 4,8-fache Kommunikationsaufkommen im Vergleich zu Szenario 3 zu erwarten. Gleiches gilt auch für das übertragene Datenvolumen. In Abbildung 7.12 sind die beobachteten Werte aus den Szenarien 3 und 8 sowie die erwarteten Werte gegenübergestellt.

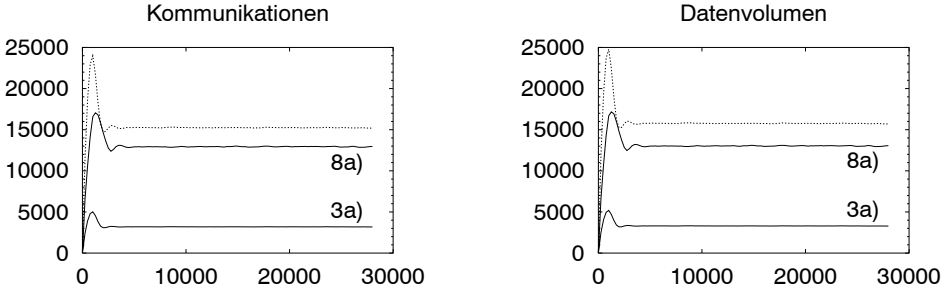


Abbildung 7.12: Kommunikationsaufkommen und Datenvolumen des Szenarios 8a im Vergleich zu Szenario 3a und den erwarteten Werten (gepunktete Linien)

Offensichtlich liegen die erwarteten Werte deutlich über den beobachteten Werten! Der „Fehler“ liegt allerdings in der Darstellung der Ergebnisse: Das bislang problemlose „Abschneiden“ der Ausschwingphase in den Diagrammen führt hier zur Fehlbetrachtung. In Abbildung 7.13 sind die kompletten Kurven zusammen mit den erwarteten Werten dargestellt.

Es ist nun deutlich zu sehen, dass die Kurve aus Szenario 8 von der Kurve mit den erwarteten Werten geschnitten wird! Die Erklärung hierfür ist, dass pro Zeiteinheit zwar weniger Kommunikationen beobachtet wurden als erwartet, jedoch benötigte die Simulation insgesamt mehr Zeiteinheiten. Die Bearbeitung der vorgegebenen Last im Szenario 8 dauert folglich wesentlich länger als in Szenario 3! Der Grund liegt in der Überbelastung der Netze, die – trotz „unendlich“ großer Puffer im Simulator – nicht mehr in der Lage sind, alle Kommunikationsanforderungen in der gleichen Zeit wie in Szenario 3 zu erfüllen. Analoge Aussagen gelten zwangsläufig auch für das übertragene Datenvolumen.

Abbildung 7.14 zeigt abschließend noch die Ergebnisse der Teil-Szenarien 8a und 8b im direkten Vergleich.

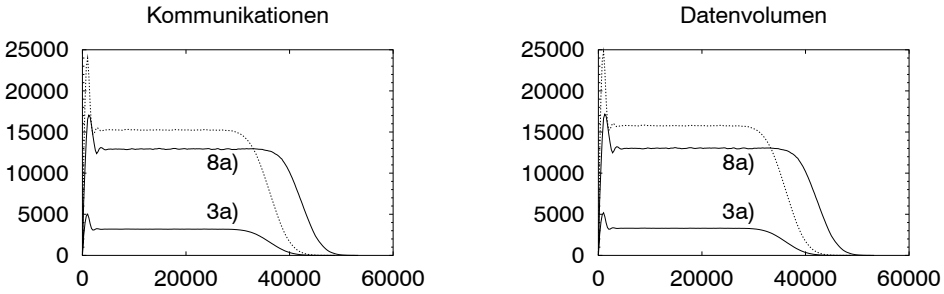


Abbildung 7.13: Gesamtes Kommunikationsaufkommen und Datenvolumen des Szenarios 8a im Vergleich zu Szenario 3a und den erwarteten Werten (gepunktete Linien)

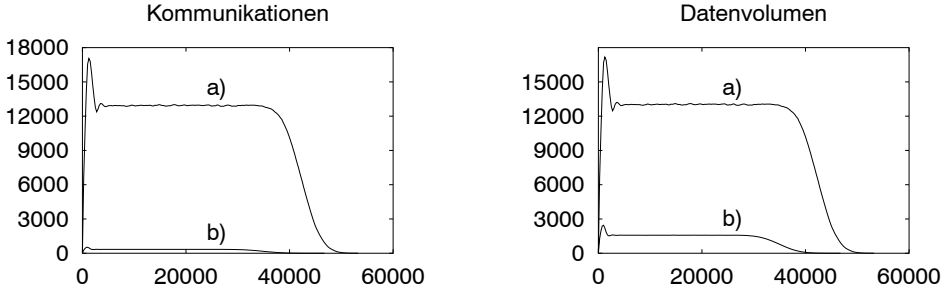


Abbildung 7.14: Gesamtes Kommunikationsaufkommen und Datenvolumen des Szenarios 8

Das Kommunikationsaufkommen der Simulation 8b) entspricht exakt dem des Szenarios 3b! Obwohl die Struktur um ein mehrfaches größer ist, werden – entsprechend der Idee des OLL-Katalogs – nicht mehr Kommunikationen benötigt. Im Vergleich zum herkömmlichen Expansionsverfahren werden in diesem Szenario mehr als 95% der Kommunikationen eingespart!

Das zu übertragende Datenvolumen steigt zwangsläufig entsprechend der Größe der Produktstruktur an. Die Verwendung rekursiver Anfragen zusammen mit OLL-Katalogen führt aber auch hier zu einer Einsparung von über 85%.

Das verbesserte Kommunikationsverhalten schlägt sich erwartungsgemäß auch hier in den Antwortzeiten der Benutzer nieder (vgl. Tabelle in Abbildung 7.15).

Szenario 8	gesamt in s	Expand in s		
		∅	Min	Max
Simulation a)	7353.55	168.12	155.47	583.66
Simulation b)	92.16	1.58	1.56	3.54
Einsparung in %	98.75	99.06		

Abbildung 7.15: Antwortzeiten im Szenario 8

7.3 Fazit

Die im Abschnitt 7.2.2 beschriebenen Simulationen zeigen, dass die Verwendung von rekursivem SQL in Kombination mit den OLL-Katalogen zu deutlichen Performance-Steigerungen führen kann. Die beobachteten Werte drücken im Wesentlichen den Übertragungsaufwand in lokalen sowie Weitverkehrs-Netzen aus, hinzukommende Aufwände z. B. durch Berechnungen oder Externspeicherzugriffen wurden in den Simulationen nicht fokussiert berücksichtigt.

Ebenfalls ohne Beachtung blieb das Auftreten von Kollisionen und der damit verbundene Aufwand für erneutes Senden von Datenpaketen. In der Simulation wurde für alle Bestandteile des Simulators ein beliebig großer Eingangs- und Ausgangspuffer angenommen, so dass zwar Verzögerungen auf Grund langer Warteschlangen (z. B. an den Gateways) auftreten können, Kollisionen von Datenpaketen jedoch werden nicht simuliert. In der Realität können Kollisionen zu erheblich schlechteren Antwortzeiten führen, als die Simulationsergebnisse zeigen, besonders bei der Simulation heute üblicher Architekturen!

Für realistische Szenarien ist bei Multi-Level-Expansionen durchaus mit Einsparungen von mehr als 95% zu rechnen, im Mix mit anderen Aktionen können (je nach Anteil von Multi-Level-Expansionen) 75-90% im Vergleich zu heutigen PDM-System-Architekturen möglich sein. Eine einfache Implementierung der Datenbankzugriffskomponente auf dem relationalen Datenbankmanagementsystem DB2 UDB V7.2 der Firma IBM führte zu ähnlich guten Ergebnissen und untermauert damit die in der Simulation gewonnenen Erkenntnisse.

Teil III

Vergleich mit anderen Lösungsansätzen und Zusammenfassung

Kapitel 8

Diskussion verwandter Lösungsansätze

Gleichwohl die Bedeutung von PDM-Systemen in der Industrie mehr und mehr zunimmt, sind Forschungsarbeiten in diesem – zugegebenermaßen sehr speziellen – Umfeld bislang kaum zu finden. Insbesondere der Performance-Aspekt in *verteilten* PDM-Systemen spielt dabei nahezu keine Rolle.

Im Folgenden werden verschiedene Ansätze und Arbeiten, die mehr oder weniger verwandt sind mit den in den vorangegangenen Kapiteln beschriebenen Mechanismen, kurz beschrieben und verglichen.

8.1 Mechanismen zur Zugriffssteuerung in kooperativen Systemen

8.1.1 Zugriffsberechtigungen in EDMS

EDMS [Pel00, PMAS93] gehört zu der Klasse der Dokumentenmanagementsysteme [GSSZ99, Kam00] und wurde von der *Product Data Management Group* an der *Helsinki University of Techology* in Zusammenarbeit mit KONE, einem finnischen Hersteller von Aufzugs-Anlagen, entwickelt. Der Fokus lag zunächst auf der Verwaltung von Dokumenten, die bei der Produktentwicklung anfallen. In einer späteren Phase sollte dann auch eine Produktstruktur-Verwaltung hinzugefügt werden.

Die Architektur des Systems ist absolut zentralistisch ausgelegt: Datenbankserver und EDMS-Server sind in einem zentralen Verbund angeordnet, lediglich die

Clients können an verschiedenen Standorten angesiedelt werden und über TCP/IP mit dem EDMS-Server kommunizieren.

Dokumente haben prinzipiell ähnliche Eigenschaften wie Produktstrukturen. Sie können sich beispielsweise aus verschiedenen Teil-Dokumenten zusammensetzen; analog zur Konfiguration von Produkten müssen auch Dokumente konfiguriert werden können – z. B. sollte ein Handbuch eines Produktes nur die Funktionen beschreiben, die in dem gefertigten und gelieferten Produkt tatsächlich verfügbar sind, die Beschreibung aller anderen, nicht gewählten „Sonderausstattungen“ ist schlichtweg überflüssig.

Systeme zur Verwaltung von Dokumenten werden wie PDM-Systeme von vielen Anwendern benutzt. Auch hier gilt es, entsprechend den Aufgaben und Befugnissen der Benutzer feingranulare Zugriffsrechte vergeben zu können. EDMS stellt hierfür sogenannte *Autorisierungs-Routinen* zur Verfügung [PAMS94, Pel00]. Vor der Ausführung einer Aktion durch den EDMS-Server wird eine derartige, aktions-spezifische Routine ausgeführt, die über die Zulässigkeit der Aktion entscheidet. Ein Beispiel einer Autorisierungs-Routine für die Aktion „Erzeugen einer Sperre auf einer Dokument-Version“ ist in Abbildung 8.1 dargestellt.

```
event lockDocVersion(user, version)
  if user = ‘‘super_user‘‘ then
     accept;
  end;
  if version.current_stat = ‘‘draft‘‘ then
     accept;
  else
     reject ‘‘must be in state ’draft’‘‘;
  end;
end;
```

Abbildung 8.1: Beispiel einer Autorisierungs-Routine in EDMS

Autorisierungs-Routinen werden in Abhängigkeit des Benutzers und des betrachteten Objektes aufgerufen. Innerhalb der Routinen dürfen auch Datenbank-Anfragen ausgeführt werden.

Die Autorisierungs-Routinen durchlaufen entweder ein `accept`-Statement (vergleichbar mit `return true`) oder ein `reject`-Statement. Wird das `reject`-Statement ausgeführt, so wird die vom Benutzer geforderte Funktion am EDMS-Server nicht ausgeführt. Stattdessen erhält der Client die im `reject`-Statement angegebene Fehlermeldung.

Die Auswertungsstrategie der Zugriffsregeln in EDMS entspricht im Wesentlichen der heutiger PDM-Systeme. Folglich handelt es sich auch hier um eine relativ späte Regelauswertung hinsichtlich *anfragender* Operationen wie Query und Multi-Level-Expand[35]. Zwischen dem Datenbankmanagementsystem und dem EDMS-Server werden also möglicherweise viele Dokumente (bzw. die Meta-Information dazu) übertragen, die am EDMS-Server anschließend wieder „aussortiert“ werden müssen. In zentral organisierten Szenarien, für die EDMS offensichtlich entworfen wurde, sind keine Performance-Probleme zu erwarten. Für ein (weltweit) verteilt arbeitendes PDM-System allerdings stellen Autorisierungs-Routinen nach dem Muster von EDMS keine adäquate Zugriffsregelauswertung dar.

Aus der Beschreibung des EDMS-Systems geht nicht hervor, ob die Autorisierungs-Routinen generiert werden oder manuell erzeugt werden müssen. Eine Formalisierung der Zugriffsregeln ähnlich der Beschreibung durch die vier Merkmale *Benutzer*, *Objekttyp*, *Aktion* und *Bedingung* (vgl. Abschnitt 3.2.2 und [SWW00]) findet nicht statt.

8.1.2 Zugriffskontrolle in EDICS

EDICS ist ein *kooperatives Design-Environment* zur Entwicklung von Kontroll-Systemen [Mac96] und ist den CSCW-Anwendungen zuzuordnen. Entwicklungsumgebungen wie EDICS müssen unter anderem die Entwicklung verteilter und gemeinsam genutzter Designs ermöglichen, wobei auch die Zugriffskontrolle auf die erzeugten Daten eine Schlüsselrolle für die *kooperative Nutzung* spielt.

Basis des Systems ist eine Design-Datenbank, auch als *Informationsraum* bezeichnet. Da in EDICS die entwickelten Produkte *nicht* in verschiedenen Sichten angezeigt werden sollen, wird ein objektorientiertes Datenbankmanagementsystem einem relationalen System vorgezogen, auch wenn in [Mac96] insbesondere das Fehlen von Standards, adäquater Unterstützung assoziativer Anfragen sowie Unterstützung von Views und Zugriffskontrolle bemängelt wird.

Die Zugriffskontrolle in EDICS dient zwei Aspekten: Zum einen soll die Korrektheit parallel ausgeführter Aktionen gewährleistet werden (Integrität der Daten), zum anderen sollen die Befugnisse einzelner Benutzer kontrolliert werden. Dazu wird unterschieden zwischen *physikalischer Zugriffskontrolle* und *logischer Zugriffskontrolle*. Bei der physikalischen Zugriffskontrolle werden *Sperren* auf gemeinsam genutzte Objekte auf Datenbankebene gesetzt. Gesperrte Objekte können dabei von anderen Benutzern in deren Design-Prozess angezeigt, nicht je-

[35] In Dokumentenmanagementsystemen entspricht der Multi-Level-Expand dem Zusammensetzen eines Dokuments aus seinen Teil-Dokumenten.

doch geändert oder gelöscht werden. Die logische Zugriffskontrolle demgegenüber setzt auf die *Auswertung von Objekt-Zuständen* und gibt dabei im wesentlichen folgende Regeln vor:

(a) Objekte, die ein bestimmter Benutzer nicht sehen darf, werden von diesem niemals gemeinsam mit anderen Benutzern verwendet

(b) Objekte, die von einem bestimmten Benutzer weder geändert noch gelöscht werden dürfen, können gemeinsam mit anderen Benutzern verwendet werden, ohne Sperren darauf anzufordern

Besonders Regel (b) soll dazu dienen, unnötige Sperranforderungen zu vermeiden und damit das Kommunikationsaufkommen zwischen Client und Server zu reduzieren.

Um die logische Zugriffskontrolle unter Berücksichtigung dieser beiden Regeln anwenden zu können, müssen offensichtlich vier Zugriffsberechtigungen für die Benutzer definiert werden:

1. Verhindern jeglicher Operation auf einem Objekt
2. Erteilen der Leseberechtigung auf den Inhalt eines Objektes
3. Erteilen der Änderungsberechtigung auf den Inhalt eines Objektes
4. Erteilen der Löschberechtigung für ein Objekt

Eine Auswertungsstrategie der Zugriffskontrolle ist aus der Beschreibung von EDICS nicht ersichtlich und somit auch nicht mit den hier vorgestellten Ansätzen vergleichbar. Außerdem ist für PDM-Systeme die bereitgestellte Granularität der Zugriffskontrolle absolut nicht ausreichend.

8.1.3 Auswertung von EXPRESS-Integrity-Constraints

Unter der Führung der ISO (International Standard Organization) wird STEP (STandard for the Exchange of Product data) entwickelt, dessen Ziel die Spezifikation einer Norm für die eindeutige Darstellung sowie den Austausch computerinterpretierbarer Produkt-Informationen über den gesamten Produktlebenszyklus ist (vgl. [Mas92, Owe93]). STEP besteht aus mehreren Teilen, unter anderem wird auch die Datenmodellierungssprache EXPRESS [Spi94] definiert. Diese Sprache ermöglicht – neben vielen anderen Aspekten – die Definition von Constraints zusammen mit der Definition einer Entität.

In [SBY97] wird beschrieben, wie derartige Constraints effizient ausgewertet werden können. Als Voraussetzung wird angenommen, dass alle Instanzen eines Typs in genau einer Datenbank gespeichert sind (d. h. keine horizontale Verteilung und auch keine Replikation, verschiedene Typen dürfen jedoch sehr wohl in verschiedenen Datenbanken liegen).

Es existieren drei Klassen von Integrity Constraints in EXPRESS:

- UNIQUE: Der Wert eines einzelnen Attributs oder einer Kombination von Attributen muss innerhalb des Typs eindeutig sein.
- WHERE: Definition einer Wertebereichs-Regel, die für jede Instanz eines Typs gelten muss. Hiermit werden die gültigen Werte eines Attributs (oder einer Kombination von Attributen) festgelegt.
- RULE: Jede beliebige Funktionalität kann hier Verwendung finden, insbesondere können auch Datenbank-Anfragen enthalten sein.

Für UNIQUE-Constraints kann eine effiziente Auswertung über entsprechende Indexe erfolgen. WHERE-Constraints sind ebenfalls relativ kostengünstig, da jede Objekt-Instanz separat getestet werden kann (unabhängig von anderen Objekten).

Teuer – und damit auch besonders interessant für die Optimierung – können RULE-Constraints werden, insbesondere dann, wenn zu ihrer Auswertung Datenbankanfragen durchgeführt werden müssen.

Eine RULE in EXPRESS mit zwei Datenbankanfragen hat beispielsweise folgende Form:

```
RULE p-match FOR (point);
LOCAL
  first-oct, seventh-oct: SET OF POINTS := [];
END-LOCAL
  first-oct := QUERY (temp<*point|(temp.x>0) AND (temp.y>0) AND (temp.z>0));
  seventh-oct := QUERY (temp<*point|(temp.x<0) AND (temp.y<0) AND (temp.z<0));
WHERE
  SIZEOF(first-oct) = SIZEOF(seventh-oct);
END-RULE;
```

Die erste Query (Zeile 5) ermittelt alle Punkte (Instanzen der Klasse point), die im ersten Oktant eines dreidimensionalen Koordinatensystems liegen und speichert diese im Array first-oct. Die zweite Query ermittelt die Punkte, die im siebten Oktant liegen, und speichert sie in seventh-oct. Die RULE testet letztlich, ob beide Arrays die gleiche Größe haben, d. h. ob im ersten und siebten Oktant gleich viele Punkte liegen.

Liegen die Daten in einer Datenbank, so kann man die Queries als Datenbank-Anfragen schreiben (in abstrakter, aber selbsterklärender Syntax):

```
query ('SELECT * FROM point WHERE (point.x>0) AND (point.y>0) AND (point.z>0)');
query ('SELECT * FROM point WHERE (point.x<0) AND (point.y<0) AND (point.z<0)');
```

Da die Variablen first-oct und seventh-oct nur als Parameter für SIZEOF dienen, könnte die RULE umformuliert und damit billiger ausgewertet werden:

```
RULE p-match FOR (point);
LOCAL
  temp1, temp2: INTEGER;
END-LOCAL
  temp1:=query ('SELECT COUNT (*) FROM point
                 WHERE (point.x>0) AND (point.y>0) AND (point.z>0)');
  temp2:=query ('SELECT COUNT (*) FROM point
                 WHERE (point.x<0) AND (point.y<0) AND (point.z<0)');
WHERE
  temp1=temp2;
END-RULE;
```

Der erste, auch im Beispiel angewandte, primitive Ansatz zur Optimierung liegt klar auf der Hand: Die Ergebnisse von Datenbank-Anfragen in den RULEs sollten so klein wie möglich sein. Es gilt zum einen, die Projektion auf die tatsächlich benötigten Attribute zu reduzieren (diese können durch Analyse der RULE ermittelt werden), zum anderen sind zur Berechnung von Aggregationen entsprechende Aggregat-Funktionen einzusetzen, anstatt diese quasi „manuell“ zu bestimmen.

Eine weitergehende, komplexere Optimierung kann stattfinden, indem bei mehreren vorhandenen Datenbank-Anfragen innerhalb einer RULE deren gegenseitige Abhängigkeiten analysiert werden. Damit lässt sich (1) die optimale Auswertungsreihenfolge und (2) die minimale Menge von auszuwertenden Regeln (insbesondere für Änderungen oder Einfügungen) bestimmen.

Die grundlegenden Ideen dieser Effizienzsteigerung sind:

1. **Feinere Granularität der Datenabhängigkeit:** EXPRESS definiert RULEs auf der Ebene der Entitäten, nicht der Attribute von Entitäten. Für Änderungsoperationen beispielsweise kann dies bedeuten, dass RULEs überprüft werden, obwohl sich die Attribute, auf die sich die RULEs beziehen, gar nicht geändert haben! Mit der feineren Granularität können exakt die RULEs ermittelt und ausgewertet werden, die auf Grund der Änderung tatsächlich fehlschlagen könnten und deshalb geprüft werden müssen.

2. **Gemeinsame Nutzung von Anfrageergebnissen durch mehrere Regeln:** Sind mehrere RULEs zu evaluieren, so können zwei oder mehrere dieser

RULEs identische Datenbank-Anfragen enthalten. Hier bietet es sich an, eine solche Anfrage nur einmalig auszuwerten und das Ergebnis den identischen Anfragen direkt zur Verfügung zu stellen.

3. **Ordnung von Anfragen und RULEs:** Sind in einen Constraint-Test mehrere RULEs und/oder Anfragen involviert, so kann durch geschicktes Anordnen der Anfragen möglicherweise *frühzeitig* die Verletzung einer RULE „entdeckt“ und die Auswertung somit abgebrochen werden. Je früher die Verletzung auftritt, um so weniger Anfragen und RULEs müssen ausgewertet werden.

4. **Inkrementelle Evaluierung:** War der Inhalt der Datenbank *vor* dem Einfügen oder Ändern von Objekten regelkonform, so genügt es, beim Einfügen oder Ändern nur die neuen bzw. geänderten Objekte auf die betreffenden RULEs zu testen.

Um die optimale Auswertungs-Reihenfolge der Anfragen und RULEs zu finden, wird ein Dependency-Graph aufgebaut. Es handelt sich dabei um einen gerichteten, azyklischen Graph, dessen Knoten Attribute, Datenbank-Anfragen (aus den RULEs) und die RULEs selbst darstellen. Die Kanten dazwischen beschreiben, welche Attribute in welcher Anfrage verwendet werden bzw. welche Anfrageergebnisse in welcher RULE auftreten. Abbildung 8.2 zeigt den Dependency-Graph der RULE p-match.

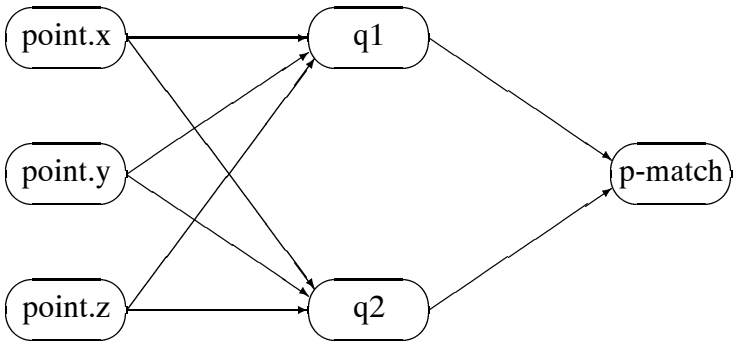


Abbildung 8.2: Dependency-Graph der RULE p-match

Über geeignete Kostenfunktionen, die die Ausführungskosten sowie die Wahrscheinlichkeit des Fehlschlagens einer RULE berücksichtigen, kann aus dem Dependency-Graph eine im Mittel optimale, d. h. kostengünstigste Auswertungsreihenfolge der Anfragen und RULEs bestimmt werden.

Weitergehende Ausführungen zur konkreten Auswertung der RULEs, z. B. unter Verwendung eines relationalen Datenbankmanagementsystems, sind in [SBY97] nicht enthalten.

Die in EXPRESS beschriebenen Constraints legen fest, unter welchen Bedingungen ein Objekt „zu einem definierten Typ“ gehört. Vergleichbar hierzu legen die in der vorliegenden Arbeit beschriebenen Zugriffsregeln fest, unter welchen Bedingungen ein Objekt zu einem Anfrage-Ergebnis gehört. Der Unterschied liegt nun darin, dass Constraints für Objekte definiert werden, also quasi mit ihnen fest verbunden sind, während Zugriffsregeln nicht nur von den Objekten selbst, sondern auch von den Benutzern sowie der auszuführenden Aktion abhängen, also quasi dynamisch auszuwählen sind. Die Auswertungsreihenfolge der RULEs und deren Datenbank-Anfragen kann demnach einmalig bestimmt und dann eingehalten werden. Es ist durchaus denkbar, eine derartige Optimierung auch bei der Generierung der Table-Functions (vgl. Abschnitt 3.5.5.2) durchzuführen, der erzielbare Gewinn ist jedoch als sehr gering einzuschätzen gegenüber der in dieser Arbeit vorgeschlagenen frühzeitigen Zugriffsregelauswertung, denn in [SBY97] wird nur die Auswertungsreihenfolge, nicht aber der Auswertungsort bzw. -zeitpunkt selbst optimiert.

8.2 Optimierung des Zugriffsverhaltens

8.2.1 Berücksichtigung monotoner Prädikate

[RHM84] beschreibt einen Ansatz zur Performance-Optimierung von Anfragen an Produktstrukturen. Ziel ist es, mittels möglichst *kleiner Ergebnismengen* das kommunizierte Datenvolumen zu reduzieren. Als typisch werden dabei Anfragen der folgenden Form angenommen: „Ermittle alle Teile eines Flugzeugs, die max. 1 Meter von der Instrumententafel entfernt sind.“ Allgemeiner ausgedrückt: „Ermittle alle Teile P in der Struktur H, die das Prädikat **<***pred***>** erfüllen!“. Analog kann selbstverständlich auch die Negierung dieser Anfrage ausgeführt werden.

Grundlage für die Minimierung der Ergebnismenge sind *Regeln*, die auf der Klasse der *monotonen Prädikate* aufbauen. Die Monotonie eines Prädikates bezogen auf eine Struktur wird dabei wie folgt definiert: Navigiert man in einer Struktur abwärts (d. h. vom Zusammenbau hin zum Einzelteil), dann wird die Chance, das Prädikat zu erfüllen immer besser (abwärts-monoton) oder immer schlechter (aufwärts-monoton).

Für ein abwärts-monotones Prädikat p gilt folglich: Erfüllt ein Part P das Prädikat p, dann erfüllen es auch alle Subparts von P! Hat man also entsprechend oben

genannter Anfrage ein auf p passendes Part P gefunden, so kann ohne weitere Prüfung auf p der komplette Teilbaum von P in das Anfrage-Ergebnis übernommen werden. Umgekehrt kann die Navigation durch einen Teilbaum abgebrochen werden, falls ein Part P ein aufwärts-monotones Prädikat nicht erfüllt, denn die Parts in der Substruktur erfüllen das Prädikat entsprechend der Definition auch nicht.

Die Monotonie eines Prädikates p ist abhängig vom Attribut, das p testet: *ordnungs-erhaltend* ist ein Attribut, falls der Wert des Attributs für jedes Subpart kleiner ist als der des übergeordneten Zusammenbaus. Beispiele hierfür sind etwa das Gewicht von Teilen und die Hüll-Geometrie. *Invers ordnungs-erhaltend* ist ein Attribut, falls der Wert für jedes Subpart größer ist als der des übergeordneten Zusammenbaus (gilt z. B. für die Struktur-Tiefe).

Einige Beispiele abwärts-monotoner Prädikate sind:

- $attr(P) < const$, falls das Attribut $attr$ ordnungs-erhaltend ist.
- $attr(P) > const$, falls das Attribut $attr$ invers ordnungs-erhaltend ist.
- P ist ein Subpart von P_0 (für ein beliebiges aber festes P_0).
- AND– und OR–Verknüpfung abwärts-monotoner Prädikate.
- Negierung aufwärts-monotoner Prädikate.

Beispiele für aufwärts-monotone Prädikate:

- $attr(P) > const$, falls das Attribut $attr$ ordnungs-erhaltend ist.
- $attr(P) < const$, falls das Attribut $attr$ invers ordnungs-erhaltend ist.
- AND– und OR–Verknüpfung abwärts-monotoner Prädikate.
- Part P enthält Region R.
- Part P schneidet oder enthält Region R.

Voraussetzung für die vorteilhafte Ausnutzung der Monotonie-Eigenschaft von Prädikaten ist die Verwendung einer *instanzbasierten* Produktstruktur. Derartige Produktstrukturen enthalten bei mehrfach verwendeten Substrukturen für jedes Vorkommen (gleichbedeutend mit *Instanz*) instanz-spezifische Information, beispielsweise die geometrische Einbaulage bezüglich des übergeordneten Zusammenbaus. Die Angabe der *Einbau-Menge* (d. h. wie oft die Substruktur in ein Produkt eingebaut wird) ist – im Gegensatz zu *Stücklisten* – nicht ausreichend!

Problematisch erscheint dieser Ansatz für die geometrische Überprüfung von Bauteilen anhand des Digital Mockups (DMU). Das „Abschneiden“ von Teilbäumen führt zu Informationsverlust, der zu fehlerhaften Analysen führen kann. Für die Ausführung eines DMUs kann lediglich der Vorteil resultierend aus abwärtsmonotonen Prädikaten durch Einsparung unnötiger Tests genutzt werden. Für verteilte Umgebungen, wie sie insbesondere in den Kapiteln 5 und 6 betrachtet wurden, sind damit allerdings nur marginale Performance-Verbesserungen zu erzielen.

8.2.2 Anwendung von Replikation

Antwortzeiten von Datenbank-Anfragen in verteilten Umgebungen lassen sich mitunter durch Replikation der Daten verringern. Häufig benötigte Daten werden dabei an mehreren Standorten redundant gespeichert, so dass die Benutzer dort lokal – und damit sehr schnell – auf diese Daten zugreifen können.

Problematisch wird die redundante Datenhaltung bei häufigen Änderungen. Die Anpassung der Replikate erfordert Kommunikation zwischen den Standorten und führt neben der erhöhten Netzlast auch zu Mehraufwand an den betroffenen Standorten. Auch sind mögliche Netz-Partitionierungen zu berücksichtigen, d. h. unter Umständen sind Standorte nicht immer erreichbar – derartige Zustände müssen erkannt und in die Verfahren zur Konsistenzsicherung der Replikate einbezogen werden.

Der Einsatz von Replikaten erfordert daher bereits im Vorfeld genaue Analysen, um vorhersagen zu können, ob die erzielten Einsparungen für Anfragen nicht durch den Mehraufwand für Änderungen kompensiert werden. Ein umfassender Überblick über die Thematik findet sich beispielsweise in [Dad96].

Im Falle von PDM-Systemen stellt die Replikation von Produktstruktur-Informationen aus zwei Gründen *keine* befriedigende Lösung des Performance-Problems dar:

Zum einen sind Partner und Zulieferer häufig Konkurrenten (beispielsweise in verschiedenen Projekten). Daten und Informationen über ihre Produkte bilden einen wesentlichen Teil ihres Kapitals. Deshalb wird die redundante Datenhaltung – besonders über Unternehmensgrenzen hinweg – aus unternehmenspolitischen und strategischen Gründen abgelehnt.

Der zweite Grund liegt in der Datenmenge, die besonders in Großprojekten anfällt. In derartigen Projekten können mehrere hundert Partner und Zulieferer beteiligt sein, so dass zumindest eine vollständige Replikation der Daten nicht sinnvoll erscheint. Einzelne, sehr eng kooperierende Partner und Zulieferer mögen

sich eventuell noch auf eine teilweise Replikation verständigen (sofern nicht andere Gründe dagegen sprechen), jedoch unterliegen die Daten besonders in der Entwicklungs-Phase häufigen Änderungen, so dass auf Grund des dabei anfallenden Aufwandes nicht mit deutlichen Performance-Steigerungen zu rechnen ist.

8.2.3 Anwendung von Prefetching

Unter Prefetching versteht man die Bereitstellung von Daten noch bevor diese von der datenverarbeitenden Einheit angefordert werden. Prefetching wird beispielsweise angewendet, um die relativ langen Zugriffszeiten des Hauptspeichers zu umgehen: Anstatt auf einen Cache-Miss zu warten, der die Anforderung von Daten aus dem Hauptspeicher initiiert, werden Daten bereits vorab in den Cache transferiert, um dem vergleichsweise schnellen Prozessor Idle-Zyklen (oder auch einen Prozess-Wechsel) zu sparen (vgl. [VL97, VL00]). Auch im Datenbank-Umfeld werden Prefetch-Technologien eingesetzt, insbesondere um Kommunikationen zwischen Clients und dem Datenbankserver einzusparen [BPS00].

Die zentrale Frage beim Prefetching ist, *welche* Daten *wann* anzufordern sind: Werden die falschen Daten angefordert, so wird lediglich Zusatzaufwand generiert. Werden die Daten zum falschen Zeitpunkt übertragen, so werden sie möglicherweise vor Gebrauch aus dem Cache verdrängt (zu frühe Anforderung in Verbindung mit Least Recently Used Strategie des Caches), oder aber die Daten sind zum Zeitpunkt des gewünschten Zugriffs noch nicht vorhanden (zu späte Anforderung).

Interessant für Prefetching im Zusammenhang mit strukturierten Objekten, die in relationalen Datenbanksystemen abgelegt werden, ist [BPS00]. Die Autoren beschreiben einen Ansatz, in welchem Prefetching in Abhängigkeit des aktuellen *Kontextes* durchgeführt wird. Ein einfaches Beispiel (siehe Abbildung 8.3) möge das Prizip verdeutlichen.

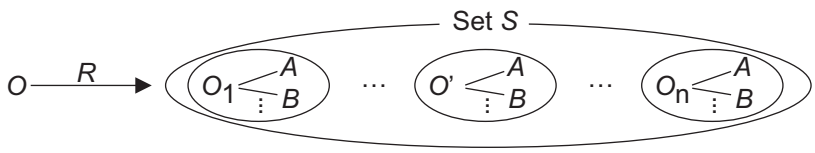


Abbildung 8.3: Primitives Beispiel kontextbasierten Prefetchings

Eine Applikation greife ausgehend vom Objekt O auf die Relation R zu. Das Ergebnis ist eine Menge S von Objekten, deren Attribute (die den *Zustand* definieren) auf mehrere Tabellen verteilt seien. Da die Applikation nicht notwen-

digerweise auf alle Attribute der Objekte zugreift, wird, um nicht unnötige Daten bereitzustellen, das Prefetching verzögert ausgeführt: Anstatt die kompletten Zustände *aller* Objekte in S vorab bereitzustellen, werden zunächst nur deren Objekt-IDs übertragen (in *einem* Round-Trip zur Datenbank). Wird nun das Objekt O' angefasst und auf dessen Attribut A zugegriffen, so wird nicht nur der Wert des Attributs A am Objekt O' von der Datenbank an den Client übertragen, sondern es werden alle Werte dieses Attributs *aller Objekte in* S in einem weiteren Round-Trip ermittelt und transferiert.

Dieses Verfahren eignet sich zur Einsparung von Kommunikationen, falls die Applikation in nachfolgenden Arbeitsschritten auf mehrere Objekte in S sowie deren Attribut A zugreift. Nach [BPS00] entspricht dies einem in der Praxis sehr häufig auftretenden Zugriffsmuster.

Zu beachten ist, dass im Beispiel auf O' *im Kontext der Menge* S, die über R von O bestimmt ist, zugegriffen wurde. Wird O' ohne diesen Kontext, d. h. ohne vorherige Navigation von O zu S angefasst, so wird das beschriebene Prefetching folglich *nicht* ausgeführt.

Die Verwendung von OLL-Katalogen bei der Ausführung von Multi-Level-Expansionen kann als eine besondere Art des Prefetchings verstanden werden. Dazu betrachtet man – entsprechend dem in heutigen PDM-Systemen implementierten Verfahren – den Multi-Level-Expand als eine Folge von Single-Level-Expansionen, die an einem Server S_1 ausgeführt werden und Daten der Server $S_2, \ldots, S_n$ benötigen. Über die OLL-Kataloge sowie die rekursiven Anfragen an den entfernten Servern $S_2, \ldots, S_n$ werden alle erforderlichen Daten an S_1 vorab bereitgestellt, so dass die Single-Level-Expansionen in tieferen Leveln keine Zugriffe auf entfernte Server initiieren müssen.

Real betrachtet führen allerdings die rekursiven Datenbank-Anfragen an den Servern $S_1, S_2, \ldots, S_n$ selbst einen lokalen Multi-Level-Expand durch – hier findet sogenanntes *Function Shipping* statt, wohingegen Prefetching *Data Shipping* darstellt.

Mit dem in [BPS00] beschriebenen Prefetch-Verfahren lassen sich Multi-Level-Expansionen nicht mit der erforderlichen Effizienz unterstützen. Der Benefit des Prefetchings basiert auf dem Prinzip, Antwortzeiten von *interaktiven* Benutzer-Aktionen dadurch zu verkürzen, dass Daten für Folge-Aktionen schon beschafft werden, bevor der Benutzer diese Aktionen anstößt. Mehrfaches Ansprechen eines entfernten Servers kann aber nicht ausgeschlossen werden. Bei rekursiv und automatisch ausgeführten Single-Level-Expansionen erzielt man mit diesem Verfahren folglich nicht den gewünschten Effekt, da der Benutzer nicht interaktiv am Ablauf teilnimmt, sondern nur am Gesamtresultat interessiert ist.

8.2.4 Indexe über Pfad-Ausdrücke

OLL-Kataloge besitzen eine gewisse Ähnlichkeit zu Indexen über Pfad-Ausdrücken. In [BK89, MS86, YM98] werden drei verschiedene Arten derartiger Indexe diskutiert, die im Folgenden kurz vorgestellt werden (vergleiche auch die Diskussion in [Keß95]).

Typische Pfad-Ausdrücke besitzen die Form $C_0.A_1.A_2 \ldots A_n$. Dabei ist C_0 ein Klassenname (bzw. ein Variablenname für eine Instanz einer Klasse), A_i ist ein strukturiertes Attribut von C_{i-1} mit Instanzen der Klasse C_i als Wertebereich ($1 \leq i < n$), und A_n ist entweder ein primitives (unstrukturiertes) Attribut der Klasse C_{n-1} oder ein strukturiertes Attribut mit dem Wertebereich C_n. Die Klasse C_0 wird als *Startklasse*, C_{n-1} als *Zielklasse* bezeichnet.

Ein *Nested Index* wird für jeweils eine Start- und Zielklasse aufgebaut und entspricht einer Menge von Paaren (ν, O), wobei ν einen Wert im Wertebereich des Attributs A_n darstellt, und O eine Menge von Objekt-IDs von Objekten in C_0 dergestalt ist, dass ν und jede Objekt-ID in O in einer Instanz eines Pfad-Ausdruckes auftreten. Ein Nested Index assoziiert folglich jeden Wert des Attributes A_n aus der Zielklasse mit den Objekt-IDs der Startklasse.

Im Unterschied dazu enthalten *Pfad-Indexe* nicht nur die Objekt-ID der Startklasse, sondern auch die IDs aller Objekte, die im Pfad zwischen Start- und Zielklasse enthalten sind. Pfad-Indexe können folglich als Mengen (ν, P) aufgefasst werden, wobei ν wiederum einen Wert des Attributs A_n darstellt, und P eine Menge von Pfaden repräsentiert.

Der dritte vorgestellte Index-Typ ist der sogenannte *Multi-Index*. Ein Multi-Index ist eine Menge von Nested Indexen basierend auf den n Teilpfad-Ausdrücken $C_i.A_{i+1}, 0 \leq i < n$, wobei C_i den Wertebereich des Attributs A_i bestimmt. Hier wird also nicht für jeden Pfad zwischen einer Start- und Zielklasse ein Index aufgebaut, sondern für alle an diesem Pfad beteiligten Klassen.

OLL-Kataloge können am ehesten mit den Nested Indexen verglichen werden. Allerdings werden Nested Indexe über einem Pfad mit fester, vorgegebener Folge von Klassen C_0, C_1, …, C_{n-1} definiert. Produktstrukturen sind jedoch rekursiv über einige wenige Klassen definiert – der Pfad von der Start- zur Zielklasse kann also nicht nach obigem Muster beschrieben werden. Erweitert man die Definition eines Pfad-Ausdruckes dergestalt, dass lediglich Start- und Zielklasse sowie Eigenschaften der Objekte im Pfad dazwischen definiert werden müssen, so kommt dies der Definition 6 (vgl. Abschnitt 6.3) bereits sehr nahe.

Ein Unterschied ergibt sich allerdings noch in der Verwendung: Nested Indexe sind richtungsgebunden, d. h. man sucht zu einem Attributwert am „Ende“ eines

Pfades die zugehörigen Objekte am „Anfang“ des Pfades. Anfragen mit dem umgekehrten Weg („welche Attributwerte am „Ende“ des Pfades gibt es zu den Objekten am „Anfang“ des Pfades?“) werden nicht unterstützt. OLL-Kataloge hingegen unterstützen beide Richtungen, denn sowohl der Verwendungsnachweis wie auch der Expansionsvorgang können über OLL-Kataloge effizient durchgeführt werden.

8.3 Zusammenbau komplexer Objekte

8.3.1 Zeigerbasierte Join-Methoden

Der Zusammenbau komplexer Objekte ist verwandt mit Methoden für *pointer-based joins* insbesondere in objekt-orientierten und objekt-relationalen Datenbankmanagementsystemen [Kos00, MGS+94]. Die Materialisierung von tief strukturierten Objekten in derartigen Systemen ist vergleichbar mit einer Multi-Level-Expansion in PDM-Systemen. Ziel der zeiger-basierten Join-Verfahren ist eine effiziente Verknüpfung von derartigen komplexen Objekten, die nicht über (wertbasierte) Fremdschlüssel sondern mittels Objekt-IDs referenziert werden. Am bekanntesten sind die drei Varianten basierend auf nested-loop join, merge-join und hybrid hash join [Gra93, SC90].

Ausgangsbasis seien zwei Relationen R und S, wobei jedes Tupel in R einen Zeiger auf ein Tupel in S enthält. Das Verfahren entsprechend nested-loop join durchläuft R und ermittelt parallel zu jedem R-Tupel das zugehörige S-Tupel. Shekita und Carey kommen in [SC90] bezüglich dieser Klasse von Verfahren zu dem Schluss, dass es „für objekt-orientierte Datenbanksysteme nicht ratsam ist, nur pointer-basierte Join-Algorithmen zu unterstützen“.

Die dem merge join entsprechende Variante sortiert zunächst R nach den Zeigern auf S-Tupel (beispielsweise über die Hintergrundspeicher-Adresse, auf die diese Zeiger verweisen) und ermittelt anschließend alle referenzierten S-Tupel *en bloc*. Dadurch können unnötige, mehrfache Zugriffe auf dasselbe S-Tupel vermieden werden.

In der Variante nach dem hybrid hash join wird R derart partitioniert, dass jede Partition nur noch R-Tupel mit Zeigern auf S-Tupel *auf einer Seite des Hintergrundspeichers* enthält. Für jede Partition in R wird damit nur noch genau ein Zugriff auf den Hintergrundspeicher benötigt.

Liegen die Relationen R und S (und möglicherweise noch weitere Relationen, deren Tupel nach dem gleichen Prinzip referenziert werden) auf unterschiedlichen Servern, so ist das nested-loop-Verfahren vergleichbar mit der Multi-Level-

Expand-Auswertung heutiger verteilter PDM-Systeme. Auch die beiden verbesserten Verfahren können jedoch nicht verhindern, dass – besonders bei zyklischer Verteilung der Daten – entfernte Server mehrfach angesprochen werden müssen, selbst wenn die Kontrolle über den Join nicht von einem zentralen Server ausgeübt wird, sondern von einem Server zum nächsten (konform zur Datenverteilung) übergeben werden kann.

8.3.2 Der Assembly-Operator

In [KGM91, MGS+94] wird ein *Assembly-Operator* beschrieben, der den Zusammenbau strukturierter Objekte auch in verteilten Umgebungen ermöglicht. Der Ansatz wurde im Rahmen des Revelation-Projektes [GM88] auf dem Volcano Query Processing System [Gra94] implementiert.

In Revelation referenzieren Objekte andere Objekte mittels Objekt-Identifikatoren (OIDs). Anfragen können entweder primitiv über das Runtime-System ausgeführt werden, oder aber im Vorfeld in äquivalente Ausdrücke einer sogenannten *Komplex-Objekt-Algebra* transformiert werden.[36] Ein Optimierer ersetzt die Operatoren der Komplex-Objekt-Algebra (z. B. einen Join-Operator) in physikalische Algebra-Operatoren (z. B. Hash-Join-Operator). Der Assembly-Operator zählt zu den physikalischen Algebra-Operatoren, denen kein Operator der Komplex-Objekt-Algebra entspricht (vergleichbar einem Sort-Operator relationaler Datenbankmanagementsysteme). Er dient damit quasi der Bereitstellung von Daten für weitere Operatoren der physikalischen Algebra.

Zweck des Assembly-Operators ist die effiziente Transformation einer Menge von komplexen Objekten aus deren Hintergrundspeicherrepräsentation in schnell traversierbare Hauptspeicherstrukturen. Unter einem *komplexen Objekt* wird dabei ein oder mehrere Objekte bzw. Fragmente, die zu einem Objekt gehören und über Referenzen miteinander verbunden sind, verstanden. Verteilte Produktstrukturen sind in diesem Sinne komplexe Objekte, die Autoren von [KGM91] betrachten jedoch eher „kleine“ komplexe Objekte wie Personen- und zugehörige Adress-Daten, die in einer objektorientierten Datenbank gespeichert werden. Indem die OIDs in Hauptspeicheradressen umgewandelt werden, reduziert sich das Navigieren in komplexen Objekten auf das Verfolgen von Zeigern im Hauptspeicher.

Der Assembly-Operator ist in der Lage, *innerhalb eines komplexen Objektes* mehrfach verwendete (d. h. mehrfach referenzierte) Sub-Objekte zu erkennen und nur einmalig zu bearbeiten. Werden jedoch mehrere komplexe Objekte zusammengebaut, die *gemeinsame* Sub-Objekte verwenden, so werden diese Sub-

[36] Dieser Vorgang wird als *Revelation* bezeichnet und gibt dem Projekt seinen Namen.

Objekte – sofern sie an anderen Standorten liegen als das sie referenzierende Objekt – mehrfach angefordert und, falls sie sich bereits im Hauptspeicher befinden, verworfen. Die Autoren von [MGS+94] gehen dabei davon aus, dass die Kosten des mehrfachen Übertragens von Sub-Objekten geringer sind als die zur Koordination benötigten Kommunikationen zwischen den beteiligten Servern.

Eine Optimierung dahingehend, dass entfernte Standorte möglichst nur einmalig angefragt werden, findet nicht statt: Trifft man bei der Navigation in der Struktur auf eine nicht-aufgelöste Referenz auf ein entferntes Objekt, so wird für genau diese Referenz ein Assembly-Auftrag an den entfernten Server gesendet. Je nach Auswertungsstrategie (vgl. Abschnitt 5.4) muss möglicherweise dieser entfernte Server selbst für die Auflösung dortiger Referenzen sorgen, oder der ursprüngliche Anfrage-Server ist dafür verantwortlich. In beiden Fällen ist mehrfaches Kommunizieren entfernter Server nicht auszuschließen. Eine Erweiterung des Assembly-Operator-Ansatzes um die OLL-Kataloge aus Kapitel 6 kann diesen gravierenden Nachteil beheben.

8.3.3 Instantiierung von View-Objekten

Unter *View-Objekten* versteht man komplexe Objekte, die aus einer Anfrage an eine Datenbank mit anschließender Konvertierung des Ergebnisses in eine verschachtelte Struktur resultieren. Lee und Wiederhold bemängeln in [LW94], dass in relationalen Systemen Anfrageergebnisse als flache Relation zurückgegeben werden. Im Falle von ursprünglich verschachtelten Objekten, die zur Darstellung in relationalen Datenbanken „flachgeklopft“ werden, führen mehrfach verwendete Subtupel zu Duplikaten in der Datenbank – und damit auch in Anfrage-Ergebnissen. Folglich werden Daten möglicherweise mehrfach vom Server zum Client übertragen. Grundlage dieser Aussage ist die Abbildung eines komplexen Objektes auf *eine* Tabelle, bezeichnet als *single flat relation* (SFR).

Um die Kommunikation zwischen Datenbankserver und Client zu optimieren, werden in [LW94] zwei neue Vorschläge präsentiert. Der Erste ermittelt und überträgt eine Menge von *Relationenfragmenten* (RF), wobei jedes Fragment einem Subobjekttyp des komplexen Objektes (mit entsprechenden Verweisen auf das Superobjekt und auf eventuell weitere vorhandene Subobjekte) entspricht. Das ursprünglich komplexe Objekt wird hier in ein Geflecht von Einzelobjekten zerlegt.

Der zweite Ansatz überträgt direkt eine geschachtelte Relation, bezeichnet als *single nested relation* (SNR), basiert aber auf RFs, die ja sämtliche Informationen enthalten, um mittels Joins, Selektionen und Projektionen die dazu passende SNR zu generieren. Der Unterschied der beiden Methoden RF und SNR liegt darin, dass

im RF-Ansatz die *Fragmente* an den Client übertragen werden, die dieser zum komplexen Objekt zusammensetzen muss, im SNR-Ansatz hingegen übernimmt das „Einschachteln“ bereits der Server und übertägt so das komplexe Objekt als verschachtelte Struktur.

Da im RF-Ansatz Referenzen auf Subobjekte eingesetzt werden, anstatt Subobjekte „flachzuklopfen“ oder „einzuschachteln“, ist dieser Ansatz bezogen auf Subobjekte redundanzfrei! Mehrfach verwendete Subobjekte werden nicht mehrfach gespeichert, sondern lediglich mehrfach referenziert. Nicht zuletzt deshalb schneidet der RF-Ansatz bei den Performance-Untersuchungen in [LW94] zumeist am Besten ab und wird daher hinsichtlich der Effizienz vor SNR und SFR eingeordnet.

Wie in Abschnitt 2.4.2.1 ausgeführt, bestehen auch Produktstrukturen aus miteinander verknüpften eigenständigen Objekten. Die Abbildung auf Relationen (vgl. Datenbank-Schema aus Abschnitt 2.5.1) kann folglich mit der Fragmentierung des RF-Ansatzes verglichen werden. Die Ergebnisse aus [LW94] bestätigen, dass die gewählte Darstellungsart zusammen mit der rekursiven Anfragestrategie die bestmögliche Effizienz gewährleistet.

8.4 Verteilte Berechnung transitiver Hüllen

Die Definition des OLL-Katalogs (vgl. Definition 6 in Abschnitt 6.3) bezieht sich auf die transitive Hülle des DACGs, der die Produktstruktur repräsentiert. Die Algorithmen zur Initialisierung und Anpassung der OLL-Kataloge an den jeweiligen Standorten müssen den gesuchten Ausschnitt der transitiven Hülle ermitteln bzw. aktualisieren. Die dazu präsentierten Algorithmen (vgl. z. B. Algorithmus 6.5.1) arbeiten verteilt – das Hauptaugenmerk lag dabei weniger auf dem Verfahren zur Bildung der transitiven Hülle selbst als auf der korrekten Berechnung des OLL-Katalogs, also der gesuchten Teilmenge der transitiven Hülle.

Für die verteilte Berechnung transitiver Hüllen existieren mehrere Algorithmen, von denen einige in [CCH93] detailliert beschrieben sind. Prinzipiell lassen sie sich in *iterative* und *direkte* Verfahren unterteilen. Iterative Verfahren berechnen die transitive Hülle *breadth-first* unter Zuhilfenahme einer Schleife mit algebraischen Operatoren, die sukzessive alle neuen Kanten erzeugen. Abbildung 8.4 zeigt den naiven und semi-naiven Ansatz, die beide zu den iterativen Verfahren zählen.

Direkte Verfahren berechnen die transitive Hülle *depth-first*, d. h. auch der Warshall-Algorithmus (vgl. Abschnitt 6.6.4.1) fällt in diese Kategorie.

Parallele Verfahren zur Berechnung der transitiven Hülle lassen sich ebenfalls in zwei Hauptkategorien einteilen, je nach „Art“ der Parallelität:

power := R;
union := R;
repeat
    old_union := union;
    power := π(power $\bowtie$ R);
    union := union $\cup$ power
until (union $-$ old_union) $= \emptyset$

a)

delta := R;
union := R;
repeat
    power := π (delta $\bowtie$ R);
    delta := power $-$ union;
    union := union $\cup$ delta
until delta $= \emptyset$

b)

Abbildung 8.4: a) Naives und b) semi-naives Verfahren zur Berechnung der transitiven Hülle einer Relation R.

- Zuordnung unterschiedlicher Operationen auf verschiedene Prozessoren
- Zuordnung unterschiedlicher Daten auf verschiedene Prozessoren (Daten-Fragmentierung)

Die Kombination dieser beiden Ansätze ist ebenfalls denkbar. Am gebräuchlichsten ist jedoch die zweite Kategorie, die noch weiter unterteilt werden kann in *Hash-basierte* Fragmentierung und *semantische* Fragmentierung (vergleiche auch [Now01]). Bei der Hash-basierten Fragmentierung bestimmt eine Hash-Funktion – ausgeführt auf dem Join-Attribut der Relation – den Speicherort, bei der semantischen Fragmentierung bestimmt die Semantik eines Subgraphen dessen Speicherort. Verteilte Produktstrukturen, wie sie in der vorliegenden Arbeit Verwendung finden, gehören in die Kategorie der semantischen Fragmentierung: Der unter den Partnern und Zulieferern vereinbarte Workshare (Semantik!) bestimmt den Speicherort. Aus diesem Grund wird im Folgenden ein Verfahren basierend auf semantischer Datenfragmentierung vorgestellt.

Der Disconnection Set Approach

Die grundlegende Idee des Disconnection Set Approaches [HAC90a, HAC90b] lässt sich an einem einfachen Beispiel aufzeigen:

Man betrachte ein Eisenbahnnetz, das die Städte in Europa miteinander verbindet, und eine Anfrage nach der schnellsten Verbindung zwischen Amsterdam und Rom. Der Graph, der das Eisenbahnnetz darstellt, sei entsprechend der geographischen Gebiete (z. B. Niederlande, Deutschland, Italien etc.) fragmentiert. Des Weiteren sei die Anzahl der Grenzorte zwischen den Staaten relativ gering im Vergleich zur Anzahl der Städte innerhalb der Staaten.

Die ursprüngliche Verbindungs-Anfrage kann nun in mehrere Teilanfragen aufgespaltet werden: Finde einen Weg von Amsterdam zur niederländisch-deutschen

Grenze, von dort einen Weg zur südlichen Grenze Deutschlands, von dort einen Weg zur nördlichen italienischen Grenze, und schließlich von dort einen Weg nach Rom. Diese Anfragen besitzen alle die gleiche Struktur, beziehen sich nur auf jeweils *ein* Fragment und können parallel ausgeführt werden. Um die schnellste Verbindung von Amsterdam nach Rom zu finden, müssen die Teilergebnisse noch geeignet zusammengeführt werden. Abbildung 8.5 skizziert ein entsprechendes Szenario.

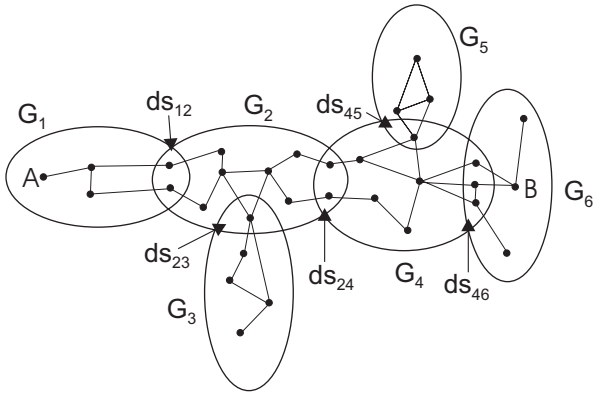


Abbildung 8.5: Prinzip des Disconnection Set Approaches: Staaten werden durch Teilgraphen $G_1, \ldots, G_6$ repräsentiert, Grenzorte befinden sich in Disconnection Sets $ds_{12}, \ldots, ds_{46}$.

Ein „Fragment-Connection-Graph“ (kurz: FCG, vgl. Abbildung 8.6) repräsentiert die Verknüpfung der einzelnen Fragmente. Jedes Fragment entspricht einem Knoten im FCG, die Disconnection Sets entsprechen den Kanten. Über diesen Graph kann ermittelt werden, welche Fragmente und Disconnection Sets in einer verteilten Anfrage involviert sind und mittels (parallelen) Teil-Anfragen angefasst werden müssen.

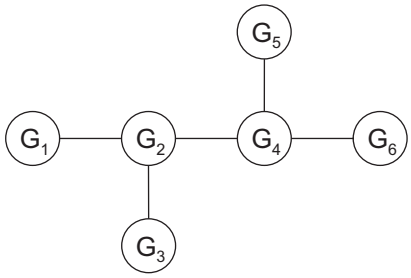


Abbildung 8.6: Fragment-Connection-Graph zu Abbildung 8.5

Das Verfahren funktioniert gut für Fragen nach Erreichbarkeit und kürzesten Pfaden. Mit leichten Modifikationen kann es auch im PDM-Umfeld (wenn auch nicht optimal) angewendet werden:

Die Fragmente des Disconnection Set Approaches enthalten die Knoten des DACGs, die dem Workshare der Partner entsprechen, sowie die zugehörigen Kanten. Die Disconnection Sets enthalten im Gegensatz zum originalen Ansatz *keine* Knoten, sondern die Kanten, die je zwei Knoten in unterschiedlichen Fragmenten verbinden. Im Fragment-Connection-Graph entsprechen die Knoten den Partnern und Zulieferern, Kanten können interpretiert werden als *ist-Zulieferer-von*-Relation bzw. in der anderen Richtung als *ist-Auftraggeber-von*-Relation.

Hier zeigt sich nun bereits die Schwäche dieses Ansatzes: Die Information im Fragment-Connection-Graph ist – bezogen auf den Objekt-Level – sehr unpräzise, denn es ist nicht ersichtlich, *welche Teile* ein Auftraggeber von seinem Zulieferer erhält. In einer Multi-Level-Expansion bedeutet dies, dass – bei paralleler Anfrage-Auswertung mit dem Disconnection Set Approach – *alle* Teile des Zulieferers expandiert und an den Anfrager zurückgegeben werden müssen. Offensichtlich werden dabei wiederum zu viele Daten übertragen. Der in der vorliegenden Arbeit vorgestellte OLL-Katalog vermeidet dies, indem *exakte* Informationen über die verknüpften *Objekte*, nicht nur über die verknüpften *Partitionen* gespeichert werden.

8.5 Routing in Netzwerken

Auch Routing-Tabellen in Netzwerkumgebungen scheinen auf den ersten Blick mit den OLL-Katalogen verwandt zu sein. Vereinfacht ausgedrückt speichern Routing-Tabellen Informationen über die günstigsten Kommunikationspfade zwischen zwei Stationen im Netzwerk. Der Aufbau und die Anwendung solcher Tabellen wird im Folgenden am Beispiel des Routing-Protokolls *Open Shortest Path First* (OSPF, vgl. [CFM99, Hel97, Moy98]) kurz erläutert und den OLL-Katalogen gegenübergestellt.

Abbildung 8.7 zeigt einen Teil eines komplexen Netzwerks. Es besteht aus sechs „elementaren“ Netzen N_1 – N_6, sowie acht Routern R_1 – R_8. Zusammengehörende Ausschnitte des gesamten Netzwerkes formen sogenannte *Areas*. Die Grenze einer Area bilden jeweils ein oder mehrere Router, die auch als *Area Border Router* (ABR) bezeichnet werden (R_3, R_4, R_6 und R_7). Router, die innerhalb einer Area liegen, werden als *interne Router* bezeichnet. Neben den Routern sind die Kosten abgebildet, die bei der Kommunikation anzusetzen sind. Eine Kommunikation von Router R_5 zu Router R_6 beispielsweise kostet 7 Einheiten (umgekehrt

sind 8 Einheiten zu veranschlagen!), die Kommunikation von Router R_6 zu Netz N_5 kostet 3 Einheiten. Kommunikationen von einem Netz zu einem Router werden üblicherweise als kostenlos angenommen (in der Abbildung wurde auf die Angabe der *0* der Übersichtlichkeit halber verzichtet).

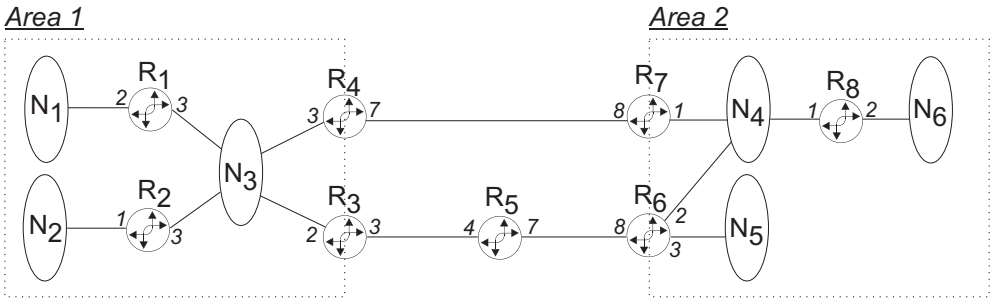


Abbildung 8.7: Netzwerk mit verschiedenen Areas

In OSPF existiert implizit die sogenannte *Backbone Area*. Ihr gehören alle ABRs sowie Router, die an einen ABR anschließen (R_5), an. Dieser Backbone ist zuständig für die Verteilung von Routing-Informationen zwischen Nicht-Backbone Areas.

In einer Konfiguration entsprechend Abbildung 8.7 müssen alle Router mit der benötigten Routing-Information versorgt werden: Jeder Router muss in der Lage sein, anhand der Zieladresse des weiterzuleitenden Datenpakets unter Verwendung der Routing-Tabelle den nächsten Router bzw. das nächste Netz auf dem Weg zum Ziel zu ermitteln und das Paket dorthin zu senden. Ein Auszug zweier Routing-Tabellen ist in Abbildung 8.8 dargestellt.

Router R_3

Ziel	Next	Kosten
N_1	R_1	4
N_2	R_2	3
N_3	–	2

Router R_6

Ziel	Next	Kosten
N_4	–	2
N_5	–	3
N_6	R_8	4

Abbildung 8.8: Auszug aus der Routing-Tabelle für R_3 und R_6

Der Aufbau der Routing-Tabellen erfolgt nach folgendem Schema:

Jeder interne Router und jeder ABR hat Kenntnis über die Topologie genau der Area, zu welcher er gehört. Dies lässt sich z. B. durch eine Matrix-Darstellung des Teilgraphen entsprechend der Area vornehmen. Zeilen und Spalten repräsentieren

Router und Netze, die Matrix-Elemente stehen für die entsprechenden Kommunikationskosten. Für die Kommunikation *innerhalb* einer Area kann basierend auf dieser Darstellung für jeden Router R_i ein *shortest path tree* ermittelt werden, der die jeweils billigste Kommunikation zwischen R_i und den Netzen und weiteren Routern derselben Area repräsentiert. Aus diesem Baum können die Einträge für die Routing-Tabelle leicht abgeleitet werden.

Für die Kommunikation zwischen Netzen in verschiedenen Areas wird nun der Backbone benötigt. ABRs kennen neben der Topologie der Area, zu welcher sie gehören, auch die Topologie des Backbones. Damit kann jeder ABR zunächst die Kommunikationskosten (ebenfalls über einen shortest path tree) zu allen entfernten ABRs berechnen. R_3 benötigt beispielsweise 10 Einheiten (3+7), um mit R_6 zu kommunizieren (über R_5).

Um nun z. B. am Router R_1 die korrekte Routing-Information für Kommunikationen in Netze entfernter Areas (z. B. N_4) bereitzustellen, fehlt nun lediglich die Information über die entfernten Areas – diese liegt an den entfernten ABRs bereit: Jeder ABR, der nach oben beschriebenem Verfahren seine „Area-interne“ Routing-Tabelle bereits aufgebaut hat (vgl. Abbildung 8.8), kann Informationen über die erreichbaren Area-internen Zielnetze sowie die anfallenden Kosten für die Kommunikation mit ihnen nach außen weitergeben. R_6 gibt beispielsweise die Information $\{(N_4,2), (N_5,3), (N_6,4)\}$ an Router R_3 weiter. Dieser addiert die Kosten für die Kommunikation mit R_6 und erhält damit $\{(N_4,12), (N_5,13), (N_6,14)\}$. Mit dieser Information kann R_3 nun seine Routing-Tabelle erweitern; des Weiteren reicht R_3 diese Information auch an alle internen Router der Area 1 weiter. Diese erweitern damit ihren Kenntnisstand bezüglich der Topologie des Netzwerkes. Da mehrere ABRs in Area 1 existieren, können einem internen Router nun mehrere Möglichkeiten offeriert werden, um mit einem entfernten Netz zu kommunizieren, z. B. mit N_4: Ein Routing über den ABR R_3 kostet 12 Einheiten, ein Routing über R_4 lediglich acht (jeweils zuzüglich der Kosten für die Area-interne Kommunikation mit den ABRs). Über den – jetzt erweiterten – shortest path tree können die internen Router ihren optimalen Pfad finden und ihre Routing-Tabelle entsprechend erweitern.

Änderungen in der Topologie des Netzwerkes (z. B. innerhalb einer Area) werden durch entsprechende Algorithmen in den Routing-Tabellen nachgezogen (vgl. [Moy98]).

Nach diesen Betrachtungen fällt auf, dass Routing-Tabellen und OLL-Kataloge nur auf sehr abstraktem Level Gemeinsamkeiten aufweisen: Sie dienen beide zur Optimierung der Kommunikationskosten. Im Detail sind die Ziele jedoch sehr verschieden. Routing-Tabellen werden benutzt, um *ein Paket* auf dem bestmöglichen (d. h. billigsten) Weg von einem Sender zum Empfänger zu leiten. OLL-

Kataloge hingegen ermöglichen die *parallele Bearbeitung* ursprünglich sequentieller Anfragen und damit Kommunikationsvorgängen, wobei Kommunikationen, die zwischen dem gleichen Sender und Empfänger ablaufen, zusätzlich noch zusammengefasst werden. Es handelt sich hier also um eine applikationsspezifische Optimierung, wohingegen Routing-Tabellen die Kommunikation als solche und unabhängig von der kommunizierenden Applikation optimieren.

Des Weiteren werden Routing-Tabellen im Wesentlichen über Shortest-Path-Algorithmen (z. B. Dijkstra-Algorithmus, vgl. [CLR96, Moy98]) berechnet. OLL-Kataloge repräsentieren dagegen die Erreichbarkeit von Knoten unter definierten Bedingungen.

8.6 Versionierung und Temporale Datenbanken

In der Vergangenheit war die Modellierung von Versionen sowie deren systemseitige Unterstützung Gegenstand einiger Forschungsaktivitäten [HMNR95, Kä92, KW95, RR96, TOC93]. [Kat90] enthält eine Zusammenfassung von Versionierungsmodellen im Bereich der „Engineering-Datenbanken“, [CW98] beschreibt ähnliche Modelle am Beispiel der Software-Konfiguration.

Alle Versionierungsmodelle – so unterschiedlich sie auch sein mögen – definieren letztendlich, welche Operationen neue Versionen erzeugen, welche Beziehungen zwischen verschiedenen Versionen bzw. zwischen Versionen und ihrem Master existieren, und welche Semantik sich hinter einer Version verbirgt. Für die in der vorliegenden Arbeit durchgeführten Performance-Betrachtungen spielen diese Aspekte keine Rolle, denn die Verteilungsinformation bezüglich der Daten ist zu den Modellinformationen hinsichtlich der Versionen quasi „orthogonal“, d. h. der Einsatz eines bestimmten Versionierungsmodells hat praktisch keinen Einfluss auf die Performance von Multi-Level-Expansionsvorgängen.

Im Zusammenhang mit der Versionierung sind Ansätze zu sehen, die temporale Aspekte in Datenbankmanagementsysteme integrieren [JS99, Myr97]. Ein sehr detaillierter Ansatz ist TSQL2 [SAA$^+$94, Sno00], eine komplette Erweiterung von SQL2 um zeitliche Funktionalitäten. Hierbei lassen sich Versionen als *Änderungen über der Zeit* beschreiben und beispielsweise für Anfragen auf Zustände „in der Vergangenheit“ nutzen. Auch für derartige temporale Erweiterungen gilt, dass die Aspekte *Zeit* und *Verteilung* orthogonal betrachtet werden können – ein Zusammenhang hinsichtlich der Performance verteilter Multi-Level-Expansionen besteht nicht.

Kapitel 9

Zusammenfassung

Die verteilte Entwicklung von Produkten ist bereits heute in vielen weltweit tätigen Konzernen keine Seltenheit mehr und wird in den kommenden Jahren immer mehr an Bedeutung gewinnen. Im Rahmen der Globalisierung der Märkte sind Unternehmen immer mehr darauf angewiesen, Produkte in immer kürzer werdenden Zeitabständen bei gleichzeitig steigenden Qualitätsanforderungen zu entwickeln. Firmenzusammenschlüsse und Allianzen für die Durchführung großer Projekte beispielsweise in der Luft- und Raumfahrt sind nicht zuletzt deshalb an der Tagesordnung.

Parallel zu dieser Entwicklung hat auch die Komplexität insbesondere technischer Produkte stark zugenommen. Die bei der Entwicklung derartiger Produkte anfallende Datenmenge ist ohne Computerunterstützung nicht mehr überschaubar. Deshalb werden zunehmend Produktdatenmanagement-Systeme eingesetzt, die ein Informationszentrum für alle an der Entwicklung beteiligten Personen darstellen (es ist damit zu rechnen, dass in Zukunft auch nachgelagerte Prozessschritte wie Produktion, Vertrieb und Wartung auf die Daten in den PDM-Systemen zurückgreifen werden). Die PDM-Systeme werden typischerweise auf der Basis relationaler Datenbankmanagementsysteme aufgebaut, in welchen sie die Meta-Information über sämtliche Produktdaten ablegen.

In lokalen Umgebungen können PDM-Systeme bereits erfolgreich eingesetzt werden. Dagegen verhindern in geographisch verteilten Umgebungen, wie sie oftmals bei weltweit agierenden Unternehmen anzutreffen sind, die langen Antwortzeiten insbesondere der in den Produktdaten navigierenden Aktionen einen produktiven Einsatz. Als Ursache für diese Performance-Probleme wurden (a) die Übertragung von Daten, auf die der Benutzer kein Zugriffsrecht besitzt, sowie (b) eine inadäquate Verwendung der Datenbank resultierend in vielen Kommunikationen zwischen den verteilten Standorten identifiziert.

Augenscheinlich primitive Lösungsversuche wurden in der Arbeit beleuchtet und als nicht adäquat bewertet. Schnellere Netzwerke können nur vorübergehend etwas Besserung verschaffen, auf Grund der hohen Latenzzeiten in Weitverkehrsnetzen können jedoch nicht die benötigten Einsparungen erzielt werden. Auch der Einsatz objektorientierter anstatt relationaler Datenbankmanagementsysteme führt nicht zu den erforderlichen Verbesserungen, denn das inperformante Kommunikationsverhalten heutiger PDM-Systeme gründet sich nicht auf systembedingte Schwächen relationaler Datenbankmanagementsysteme.

Das Ziel dieser Arbeit war die Entwicklung einer Architektur für verteilt eingesetzte PDM-Systeme, die unter Anwendung einer zu entwickelnden, neuen Strategie die Auswertung rekursiver Navigationsoperationen auf Produktdaten akzeptable Antwortzeiten garantieren. Dieses Ziel konnte erfolgreich verwirklicht werden – Simulationen und auch eine prototypische Implementierung der Zugriffsstrategie bestätigen dies. Die wichtigsten Ergebnisse der Arbeit lassen sich wie folgt kurz zusammenfassen:

Frühzeitige Regelauswertung

Zugriffsregeln beschreiben, welcher Benutzer auf welchen Objekten welche Operation unter welcher Bedingung ausführen darf. Diese Bedingung muss möglichst frühzeitig, d. h. auf dem Datenbankmanagementsystem, ausgewertet werden, um unnötigen Datenverkehr über das WAN zu vermeiden. Dabei sind eventuell vorhandene Strukturoptionen und Gültigkeiten zu berücksichtigen.

In der Arbeit wurden hierfür verschiedene Regeldarstellungen und Evaluierungsalternativen geprüft: Zugriffsmechanismen heutiger Datenbankmanagementsysteme sowie Zugriffskontroll-Listen wurden als für PDM-Systeme nicht einsetzbar eingestuft, da sie die benötigte Funktionalität nicht abdecken bzw. mit zu hohem Änderungsaufwand behaftet sind. Es wurde gezeigt, dass maximaler Gewinn prinzipiell durch die direkte Abbildung der Bedingungen in äquivalente SQL-WHERE-Klauseln erzielt werden kann. Dazu wurde in der Arbeit eine Grammatik für die Regelbedingungen sowie ein darauf arbeitender Übersetzer definiert. Voraussetzung für einen effizienten Einsatz ist jedoch, dass die Übersetzung sowie die erforderliche Anfrage-Modifikation nicht zur Laufzeit durchgeführt werden.

Favorisiert wird daher ein Ansatz, welcher die SQL-WHERE-Klauseln in vordefinierten, parametrisierten Views hinterlegt. Derartige Views sind im aktuellen SQL-Standard SQL:1999 nicht enthalten, in der Arbeit wurde jedoch gezeigt, dass z. B. das Datenbankmanagementsystem IBM DB2 UDB V7.2 ein äquivalentes Konzept in den sogenannten Table-Functions anbietet, mit deren Hilfe die gewünschte Funktionalität implementiert werden kann. Um den Programmieraufwand bei Änderungen des Regelwerkes zu minimieren, wurde ein Algorithmus zur Generierung der Table-Functions entwickelt.

Strukturoptionen und Gültigkeiten wurden in der Arbeit auf spezielle Zugriffsregeln zurückgeführt und damit in die Strategie der frühzeitigen Regelauswertung nahtlos integriert. Zur Darstellung dieser Konzepte wurden mengenwertige Attribute eingesetzt, die zwar von SQL:1999 noch nicht ausreichend unterstützt werden, jedoch für SQL:2003 bereits in der Planung sind.

Verteilte Rekursion mit OLL-Katalogen

Mehrstufige Navigationen in der Produktstruktur, wie sie beispielsweise von dem Multi-Level-Expand durchgeführt werden, können als rekursive Aktionen aufgefasst werden. In der Arbeit wurde gezeigt, dass durch die Ausführung dieser Navigationen quasi unmittelbar auf der Datenbank durch Verwendung rekursiven SQLs in Umgebungen mit *zentraler* Datenhaltung enorme Einsparungen hinsichtlich der Antwortzeiten erzielt werden können.

Bei *verteilter* Datenhaltung bedingt die zentrale Auswertung rekursiver Anfragen ein ständiges Nachfordern von Daten an entfernten Standorten (Data Shipping). Folglich besteht auch hier das Problem der häufigen Kommunikationen. Deshalb haben wir in der Arbeit auch die Rekursionsauswertung verteilt, d. h. die Rekursionsberechnung findet nicht an einem einzigen Standort statt, sondern jeder beteiligte Standort führt selbst die rekursive Auswertung auf den lokal verfügbaren Daten durch (Function Shipping).

In der Literatur werden zu derartigen verteilten Berechnungen einige Ansätze beschrieben. Allen gemein ist das große Problem, dass mehrfache Kommunikationen zwischen zwei oder mehreren Servern während der Rekursionsauswertung nicht verhindert werden können. Kaskadierende Aufrufe zwischen den Standorten können sogar dazu führen, dass berechnete Teilergebnisse quasi „im Kreis herum“ übertragen werden und damit unnötige Netzlast generiert wird.

Mit Hilfe der in dieser Arbeit entwickelten Object-Link-and-Location-Kataloge ist es nun gelungen, jeden an der verteilten Berechnung beteiligten Server genau einmal zu kontaktieren, wobei die resultierenden Teilergebnisse der Standorte den Konfigurationsvorgaben und Zugriffsrechten des anfragenden Benutzers entsprechen! Für die theoretische Grundlage der OLL-Kataloge wurde in der Arbeit eine neue Art von gerichteten azyklischen Graphen, der Directed Acyclic Condition Graph (DACG), eingeführt. Diese Graphen ermöglichen durch die Angabe von Bedingungen an den Kanten eine adäquate Beschreibung varianter Produktstrukturen.

Um die Produktstruktur mit jedem beliebigen Knoten als Startknoten sowohl *abwärts* (für Expansionen) als auch *aufwärts* (für Verwendungsnachweise) traversieren zu können, sind die OLL-Katalogeinträge bidirektional ausgelegt. Im Sinne der Kommunikationsminimierung haben wir uns für die verteilte Speicherung der

OLL-Kataloge entschieden – zusätzliche Kommunikationen mit einem zentralen Katalog-Server entfallen somit.

Die Analyse der Algorithmen für den Aufbau und die Pflege der OLL-Kataloge hat gezeigt, dass in praktisch relevanten Szenarien der Aufwand, der zur Aktualisierung der verteilten Kataloge getrieben werden muss, linear in der Anzahl der Partitionen und damit akzeptabel ist.

Für den Nachweis der Effizienz der OLL-Kataloge im Zusammenspiel mit frühzeitiger Regelauswertung wurde neben der Entwicklung eines mathematischen Berechnungsmodells auch ein ereignisorientierter Simulator konzipiert und implementiert. Die Simulationen mehrerer Szenarien haben gezeigt, dass bei navigierenden Operationen Antwortzeit-Einsparungen bis zu 95%, im Mix mit anderen Aktionen (je nach Anwendertyp) immerhin 80-90% im Vergleich zu heutigen PDM-Systemen in realistischen Umgebungen zu erwarten sind. Auch eine prototypische Implementierung auf der Basis des Datenbankmanagementsystems IBM DB2 UDB V7.2 führte zu vergleichbar guten Ergebnissen.

Ausblick

Das Prinzip der in dieser Arbeit vorgestellten OLL-Kataloge ist nicht beschränkt auf das Spezialgebiet der PDM-Systeme. Auch Applikationen anderer Anwendungsbereiche können davon profitieren: Bei der verteilten Software-Entwicklung beispielsweise können verteilt entwickelte Module, die zu einem konfigurierbaren Software-Paket zusammengesetzt werden, ebenfalls über OLL-Kataloge effizient zusammengeführt werden. Fragmente verteilt gespeicherter XML-Dokumente lassen sich ebenso mit OLL-Katalogen referenzieren. Die Suche nach Textstellen innerhalb solcher Dokumente etwa lässt sich damit leicht auf alle Fragmente parallelisieren und deshalb effizienter gestalten als bei herkömmlicher, sequentieller Abarbeitung.

In dieser Arbeit konnten nicht sämtliche Aspekte von PDM-Systemen bis ins Detail behandelt werden. Wir haben uns auf einen Ausschnitt beschränkt, der eine gewisse „Basisfunktionalität“ darstellt – mit der Weiterentwicklung der Systeme und deren breiterem Einsatz werden selbstverständlich auch weitere Anforderungen entstehen. Insbesondere wird vermehrt auf *sichere Kommunikation* Wert gelegt, auch die Kopplung verschiedener PDM-Systeme wird eine Rolle spielen. Gerade hierbei wartet in den nächsten Jahren noch viel Arbeit, bis flexible, effizient arbeitende weltweit verteilte Entwicklungsverbunde realisiert werden können.

Teil IV

Anhang

Anhang A

Struktureller Aufbau und Übersetzung von Regel-Bedingungen

A.1 Grammatik von Regel-Bedingungen

$G_{condition} = (N, T, P, S)$ mit

$N =$ {condition, rowcond, struccond, expr, term, factor, function, param-list, parameter, match-expr, pattern-expr, context-var }

$T =$ {„TRUE“, „FALSE“, „NOT“, „BETWEEN“, „AND“, „OR“, „LIKE“, „InList“, „INTERSECTS“, „IS“, „EMPTY“, „ANY“, „(“, „)“, „,“, „:“}

$P =$ *siehe Seite 220*

$S =$ condition

condition	→	rowcond
	|	struccond
rowcond	→	„TRUE“
	|	„FALSE“
	|	„NOT“ rowcond
	|	„(“ rowcond „)“
	|	rowcond „AND“ rowcond
	|	rowcond „OR“ rowcond
	|	signed-expr („=“ | „<>“ | „<“ | „>“ | „<=“ | „>=“) signed-expr
	|	signed-expr „BETWEEN“ signed-expr „AND“ signed-expr
	|	signed-expr „NOT“ „BETWEEN“ signed-expr „AND“ signed-expr
	|	match-expr „LIKE“ pattern-expr
	|	signed-expr „InList“ „(“ param-list „)“
	|	attrib-name „INTERSECTS“ context-var
	|	attrib-name „IS“ „EMPTY“
	|	attrib-name „BETWEEN“ „ANY“ attrib-name „AND“ attrib-name
struccond	→	„∃“ rel-type obj-type
	|	„∃“ rel-type obj-type „:“ rowcond
signed-expr	→	expr
	|	„−“ expr
	|	„+“ expr
expr	→	term
	|	term „+“ expr
	|	term “−“ expr
term	→	factor
	|	factor „∗“ term
	|	factor „/“ term
factor	→	function
	|	constant
	|	attrib-name
	|	context-var
	|	„(“ signed-expr „)“
function	→	function-name „(“ param-list „)“
param-list	→	signed-expr
	|	signed-expr „,“ param-list
match-expr	→	attrib-name
	|	function
pattern-expr	→	string-constant
context-var	→	„$User“
	|	„$UsrConfig“
	|	„$UsrEffectivity“

A.2 Übersetzung in SQL-konforme Bedingungen

ENV beschreibt die Ersetzungstabelle, in welcher festgehalten wird, wie Objekt-Attribute aus DB-Tupel-Attributen hervorgehen (Berechnungsvorschrift $o.x = f(t.a, t.b)$ für Wertersetzung und auch transitive Attribute, $o.x = t.y$ oder $o.x = id(t.y)$ bei Umbenennung), wie Funktionen in der Bedingung auf Funktionen im SQL-Statement abgebildet werden (falls z. B. eine Funktion einen Namen hat, der auch ein reservierter Bezeichner in SQL ist, muss die äquivalente SQL-Funktion anders benannt werden), und welche Werte Umgebungsvariablen ($User etc.) besitzen.

Der Objekt-Typ, auf den sich die zu übersetzende Bedingung in der Regel bezieht, wird im Parameter τ übergeben.

Die Übersetzer-Funktionen sind über der Struktur der Bedingungen definiert (vgl. Anhang A.1).

condition-trans: condition × ENV × *obj-type* → *SQL-condition*
rcond-trans: rowcond × ENV × *obj-type* → *SQL-condition*
scond-trans: struccond × ENV × *obj-type* → *SQL-condition*
sigexpr-trans: signed-expr × ENV × *obj-type* → *SQL-expression*
expr-trans: expr × ENV × *obj-type* → *SQL-expression*
term-trans: term × ENV × *obj-type* → *SQL-term*
factor-trans: factor × ENV × *obj-type* → *SQL-factor*
func-trans: function × ENV × *obj-type* → *SQL-function*
paramlist-trans: param-list × ENV × *obj-type* → *SQL-parameter list*
ctxvar-trans: context-var × ENV × *obj-type* → *SQL value*
mexpr-trans: match-expr × ENV × *obj-type* → *SQL match-expression*
pexpr-trans: pattern-expr × ENV × *obj-type* → *SQL pattern-expression*

Hinweis: Zeichen und Zeichenfolgen, welche die Übersetzungsfunktionen ausgeben, sind fettgedruckt.

condition-trans(*rowcond*, ϱ, τ) := *rcond-trans*(*rowcond*, ϱ, τ)

condition-trans(*struccond*, ϱ, τ) := *scond-trans*(*struccond*, ϱ, τ)

rcond-trans(TRUE, ϱ, τ) := $(\mathbf{1 = 1})$

rcond-trans(FALSE, ϱ, τ) := $(\mathbf{1 = 0})$

rcond-trans(NOT *rowcond*, ϱ, τ) := **NOT** *rcond-trans*(*rowcond*, ϱ, τ)

rcond-trans(**(** *rowcond* **)**, ϱ, τ) := **(** *rcond-trans*(*rowcond*, ϱ, τ) **)**

rcond-trans(*rowcond*$_1$ AND *rowcond*$_2$, ϱ, τ) :=
rcond-trans(*rowcond*$_1$, ϱ, τ) **AND** *rcond-trans*(*rowcond*$_2$, ϱ, τ)

rcond-trans(*rowcond*$_1$ OR *rowcond*$_2$, ϱ, τ) :=
rcond-trans(*rowcond*$_1$, ϱ, τ) **OR** *rcond-trans*(*rowcond*$_2$, ϱ, τ)

rcond-trans(*signed-expr*$_1$ = *signed-expr*$_2$, ϱ, τ) :=
sigexpr-trans(*signed-expr*$_1$, ϱ, τ) = *sigexpr-trans*(*signed-expr*$_2$, ϱ, τ)

rcond-trans(*signed-expr*$_1$ <> *signed-expr*$_2$, ϱ, τ) :=
sigexpr-trans(*signed-expr*$_1$, ϱ, τ) <> *sigexpr-trans*(*signed-expr*$_2$, ϱ, τ)

rcond-trans(*signed-expr*$_1$ < *signed-expr*$_2$, ϱ, τ) :=
sigexpr-trans(*signed-expr*$_1$, ϱ, τ) < *sigexpr-trans*(*signed-expr*$_2$, ϱ, τ)

rcond-trans(*signed-expr*$_1$ > *signed-expr*$_2$, ϱ, τ) :=
sigexpr-trans(*signed-expr*$_1$, ϱ, τ) > *sigexpr-trans*(*signed-expr*$_2$, ϱ, τ)

rcond-trans(*signed-expr*$_1$ <= *signed-expr*$_2$, ϱ, τ) :=
sigexpr-trans(*signed-expr*$_1$, ϱ, τ) <= *sigexpr-trans*(*signed-expr*$_2$, ϱ, τ)

rcond-trans(*signed-expr*$_1$ >= *signed-expr*$_2$, ϱ, τ) :=
sigexpr-trans(*signed-expr*$_1$, ϱ, τ) >= *sigexpr-trans*(*signed-expr*$_2$, ϱ, τ)

rcond-trans(*signed-expr*$_1$ BETWEEN *signed-expr*$_2$ AND *signed-expr*$_3$, ϱ, τ) :=
sigexpr-trans(*signed-expr*$_1$, ϱ, τ) **BETWEEN**
sigexpr-trans(*signed-expr*$_2$, ϱ, τ) **AND** *sigexpr-trans*(*signed-expr*$_3$, ϱ, τ)

rcond-trans(*signed-expr*$_1$ NOT BETWEEN *signed-expr*$_2$ AND *signed-expr*$_3$, ϱ, τ) :=
sigexpr-trans(*signed-expr*$_1$, ϱ, τ) **NOT BETWEEN**
sigexpr-trans(*signed-expr*$_2$, ϱ, τ) **AND** *sigexpr-trans*(*signed-expr*$_3$, ϱ, τ)

rcond-trans(*match-expr* LIKE *pattern-expr*, ϱ, τ) :=
mexpr-trans(*match-expr*, ϱ, τ) **LIKE** *pexpr-trans*(*pattern-expr*, ϱ, τ)

rcond-trans(*signed-expr* InList (*param-list*), ϱ, τ) :=
sigexpr-trans(*signed-expr*, ϱ, τ) **InList (** *paramlist-trans*(*paramlist*, ϱ, τ) **)**

rcond-trans(*attrib-name* INTERSECTS *context-var*, ϱ, τ) :=
(*factor-trans* (*attrib-name*, ϱ, τ) **INTERSECT**
factor-trans (*context-var*, ϱ, τ) **) IS NOT EMPTY**

rcond-trans(*attrib-name* IS EMPTY, ϱ, τ) :=
factor-trans(*attrib-name*, ϱ, τ) **IS EMPTY**

rcond-trans(*factor* BETWEEN ANY *attrib-name*$_1$.*attrib-name*$_2$
AND *attrib-name*$_1$.*attrib-name*$_3$, ϱ, τ) :=
EXISTS(SELECT * FROM z IN $\varrho(\tau)$**.** *factor-trans*(*attrib-name*$_1$, ϱ, τ)
WHERE *factor-trans*(*factor*, ϱ, τ) **BETWEEN**
z.*factor-trans*(*attrib-name*$_2$, ϱ, τ) **AND z.***factor-trans*(*attrib-name*$_3$, ϱ, τ)**)**

scond-trans(∃ *rel-type obj-type*, ϱ, τ) :=
EXISTS (SELECT * FROM ϱ(*rel-type*) **JOIN** ϱ(*obj-type*)
ON ϱ(*rel-type*)**.right** = ϱ(*obj-type*)**.obid**
WHERE ϱ(*rel-type*)**.left** = $\varrho(\tau)$**.obid)**

scond-trans(∃ *rel-type obj-type : rowcond*, ϱ, τ) :=
EXISTS (SELECT * FROM ϱ(*rel-type*) **JOIN** ϱ(*obj-type*)
ON ϱ(*rel-type*)**.right** = ϱ(*obj-type*)**.obid**
WHERE ϱ(*rel-type*)**.left** = $\varrho(\tau)$**.obid**
AND *rcond-trans*(*rowcond*, ϱ,*obj-type*)**)**

sigexpr-trans(*expr*, ϱ, τ) := *expr-trans*(*expr*, ϱ, τ)

sigexpr-trans($-$*expr*, ϱ, τ) := $-$ *expr-trans*(*expr*, ϱ, τ)

sigexpr-trans($+$*expr*, ϱ, τ) := $+$ *expr-trans*(*expr*, ϱ, τ)

expr-trans(*term*, ϱ, τ) := *term-trans*(*term*, ϱ, τ)

expr-trans(*term*$+$*expr*, ϱ, τ) := *term-trans*(*term*, ϱ, τ) $+$ *expr-trans*(*expr*, ϱ, τ)

expr-trans(*term*$-$*expr*, ϱ, τ) := *term-trans*(*term*, ϱ, τ) $-$ *expr-trans*(*expr*, ϱ, τ)

term-trans(*factor*, ϱ, τ) := *factor-trans*(*factor*, ϱ, τ)

term-trans(*factor*$*$*term*, ϱ, τ) := *factor-trans*(*factor*, ϱ, τ) $*$ *term-trans*(*term*, ϱ, τ)

term-trans(*factor*/*term*, ϱ, τ) := *factor-trans*(*factor*, ϱ, τ) / *term-trans*(*factor*, ϱ, τ)

factor-trans(*function*, ϱ, τ) := *func-trans*(*function*, ϱ, τ)

factor-trans(*constant*, ϱ, τ) := *constant*

factor-trans(*attrib-name*, ϱ, τ) := ϱ(*τ.attrib-name*)

factor-trans(*context-var*, ϱ, τ) := *ctxvar-trans*(*context-var*, ϱ, τ)

factor-trans((*signed-expr*), ϱ, τ) := **(** *sigexpr-trans*(*signed-expr*, ϱ, τ) **)**

func-trans(*function-name*(*param-list*), ϱ, τ) :=
ϱ(*function-name*)**(** *paramlist-trans*(*param-list*, ϱ, τ) **)**

paramlist-trans(*signed-expr*, ϱ, τ) := *sigexpr-trans*(*signed-expr*, ϱ, τ)

paramlist-trans(*signed-expr, parameter-list*, ϱ, τ) :=
sigexpr-trans(*signed-expr*, ϱ, τ)**,** *paramlist-trans*(*parameter-list*, ϱ, τ)

ctxvar-trans(*$User*, ϱ, τ) := ϱ(*$User*)

ctxvar-trans(*$UsrConfig*, ϱ, τ) := ϱ(*$UsrConfig*)

$$\textit{ctxvar-trans}(\textit{\$UsrEffectivity}, \varrho, \tau) := \varrho(\textit{\$UsrEffectivity})$$

$$\textit{mexpr-trans}(\textit{attrib-name}, \varrho, \tau) := \varrho(\tau.\textit{attrib-name})$$

$$\textit{mexpr-trans}(\textit{function}, \varrho, \tau) := \textit{func-trans}(\textit{function}, \varrho, \tau)$$

$$\textit{pexpr-trans}(\textit{string-constant}, \varrho, \tau) := \textit{string-constant}$$

Anhang B

Algorithmen zur inkrementellen Anpassung der OLL-Kataloge

B.1 Änderung von Kantenbedingungen

Die Algorithmen B.1.1, B.1.2 und B.1.3 beschreiben das Vorgehen zur Anpassung der OLL-Kataloge an den verschiedenen Standorten bei der Änderung einer Bedingung an einer Kante im DACG. Die beiden Algorithmen B.1.4 und B.1.5 enthalten hierzu Hilfsroutinen.

Algorithmus B.1.1: (Änderung einer Kantenbedingung)

UpdateEdge(in vertex u, in vertex v, in condition c, in condition c')

```
1  ...
2  if (f(u) == f(v)) then
3      dset ← Update-Up^(f(u)) (u, ⟨v, c, c'⟩, ⊤, ⊤, ∅, ∅, +∞)
4  else
5      dset ← Update-Up^(f(u)) (u, ⟨v, ε, ε⟩, c, c', {⟨u, c, 0⟩}, {⟨u, c', 0⟩}, +∞)
6  endif
7  ...
```

Algorithmus B.1.2: (Aufwärts-Traversierung nach Änderung einer Kantenbedingung)

Update-Up(in vertex x, in ⟨vertex u, condition prec, condition prec′⟩,
in condition $\bar{c}$, in condition $\hat{c}$, in borders B, in borders $\hat{\mathrm{B}}$, in int multi_used)

```
1  remove_edges ← ∅
2  new_edges ← ∅
3  multi_used_indicator ← false
4  if (B == ∅) then
5      seq ← 0
6  else
7      seq ← min(B[z.seq_number]) − 1
8  endif
9  if (∃ w --c--> x ∈ E) then
10     for all w --c--> x ∈ E do
11         pred.push ⟨w --c--> x, c̄, ĉ⟩
12     enddo
13 else
14     (p_edges, p_edges′) ← Update-Down^(f(u))(u, prec, prec′, B, B̂, multi_used)
15     for all u --c′--> v ∈ p_edges do
16         if (f(u) == f(x)) then
17             RemoveFromOLL(u --c′--> v)
18         else
19             remove_edges ← remove_edges ∪ {u --c′--> v}
20         endif
21     enddo
22     for all u --c′--> v ∈ p_edges′ do
23         if (f(u) == f(x)) then
24             AddToOLL(u --c′--> v)
25         else
26             new_edges ← new_edges ∪ {u --c′--> v}
27         endif
28     enddo
29 endif
30 while (pred.test ! = NULL) do
31     ⟨x --c--> y, c̄, ĉ⟩ ← pred.pop
32     if (f(x) == f(y)) then
33         if (∃ w --c′′--> x ∈ E) then
34             for all w --c′′--> x ∈ E do
35                 pred.push ⟨w --c′′--> x, c̄ ∧ c, ĉ ∧ c⟩
```

```
36              if (seq == 0) then
37                  prec ← prec ∧ c
38                  prec' ← prec' ∧ c
39              endif
40          enddo
41      else
42          (p_edges, p_edges') ← Update-Down^(f(u)) (u, prec, prec',
43                                                    B, B̂, multi_used)
44          for all u -c-> v ∈ p_edges do
45              if (f(u) == f(x)) then
46                  RemoveFromOLL(u -c-> v)
47              else
48                  remove_edges ← remove_edges ∪ {u -c-> v}
49              endif
50          enddo
51          for all u -c-> v ∈ p_edges' do
52              if (f(u) == f(x)) then
53                  AddToOLL(u -c-> v)
54              else
55                  new_edges ← new_edges ∪ {u -c-> v}
56              endif
57          enddo
58      endif
59  else
60      multi_used' ← (multi_used_indicator?seq : multi_used)
61      if (¬∃b ∈ B|f(x) == f(b)) then
62          (p_edges, p_edges') ← Update-Up^(f(x)) (x, ⟨u, prec, prec'⟩, c̄ ∧ c, ĉ ∧ c,
63                  B ∪ {⟨x, c̄ ∧ c, seq⟩}, B̂ ∪ {⟨x, ĉ ∧ c, seq⟩}, multi_used')
64      else
65          (p_edges, p_edges') ← Update-Up^(f(x)) (x, ⟨u, prec, prec'⟩, c̄ ∧ c, ĉ ∧ c,
66                  B, B̂, multi_used')
67      endif
68      multi_used_indicator ← true
69      for all u -c-> v ∈ p_edges do
70          if (f(u) == f(y)) then
71              RemoveFromOLL(u -c-> v)
72          else
73              remove_edges ← remove_edges ∪ {u -c-> v}
74          endif
75      enddo
76      for all u -c-> v ∈ p_edges' do
77          if (f(u) == f(y)) then
```

```
78              AddToOLL(u -c-> v)
79          else
80              new_edges ← new_edges ∪ {u -c-> v}
81          endif
82      enddo
83  endif
84 enddo
85 return (remove_edges, new_edges)
```

Algorithmus B.1.3: (Abwärts-Traversierung nach Änderung einer Kantenbedingung)

```
Update-Down(in vertex u, in condition prec, in condition prec',
        in borders B, in borders B̂, in int multi_used)
1  remove_edges ← ∅
2  new_edges ← ∅
3  sorted_B ← [ ]
4  sorted_B̂ ← [ ]
5  if (B == ∅) then seq ← 1
6  else
7      sorted_B ← sort(B) on B[z.seq_number] desc
8      sorted_B̂ ← sort(B) on B[z.seq_number] desc
9      seq ← max(B[z.seq_number]) + 1
10 endif
11 if (prec == ε) then
12     for all ⟨v, c, q⟩ ∈ sorted_B in descending order do
13         if ((v -c-> u ∈ OLL) and q < multi_used) then
14             RemoveFromOLL(v -c-> u)
15             if (f(u) ≠ f(v)) then
16                 remove_edges ← remove_edges ∪ {v -c-> u}
17             endif
18         endif
19         if (f(u) == f(v)) then
20             break
21         endif
22     enddo
23     for all ⟨v, c, q⟩ ∈ sorted_B̂ in descending order do
24         if ((v -c-> u ∉ E) and q < multi_used) then
25             AddToOLL(v -c-> u)
26             if (f(u) ≠ f(v)) then
27                 new_edges ← new_edges ∪ {v -c-> u}
```

```
28            endif
29        endif
30        if (f(u) == f(v)) then
31            break
32        endif
33    enddo
34 endif
35 for all u -c-> p ∈ E do
36    succ.push ⟨u -c-> p, prec, prec'⟩
37 enddo
38 while (succ.test != NULL) do
39    ⟨u -c-> v, c̄, ĉ⟩ ← succ.pop
40    if (f(u) == f(v)) then
41        for all v -c'-> w ∈ E do
42            succ.push ⟨v -c'-> w, c̄ ∧ c, ĉ ∧ c⟩
43        enddo
44    else
45        B' ← (B[z.cond ← z.cond ∧ c̄ ∧ c]\
46            {b ∈ B : f(b) == f(u)}) ∪ {⟨u, c, seq⟩}
47        B̂' ← (B̂[z.cond ← z.cond ∧ ĉ ∧ c]\
48            {b ∈ B : f(b) == f(u)}) ∪ {⟨u, c, seq⟩}
49        if (marked_visited(u -c-> v) and multi_used = +∞) then
50            (p_edges, p_edges') ← Update-Down^(f(v))(v, ε, ε, B', B̂', seq)
51        else
52            (p_edges, p_edges') ← Update-Down^(f(v))(v, ε, ε, B', B̂', multi_used)
53        endif
54        mark_visited(u -c-> v)
55        for all x -c-> y ∈ p_edges do
56            if (x == u) then
57                RemoveFromOLL(x -c-> y)
58            else
59                remove_edges ← remove_edges ∪ {x -c-> y}
60            endif
61        enddo
62        for all x -c-> y ∈ p_edges' do
63            if (x == u) then
64                AddToOLL(x -c-> y)
65            else
66                new_edges ← new_edges ∪ {x -c-> y}
67            endif
68        enddo
```

69 **endif**
70 **enddo**
71 **return** *(remove_edges, new_edges)*

Algorithmus B.1.4: (Entfernen einer Kante aus dem lokalen OLL-Katalog)

RemoveFromOLL(in edge $u \xrightarrow{c} v$)
1 **seek for** $\langle u \xrightarrow{c} v, k\rangle$ **in** OLL
2 **if** $(k == 1)$ **then**
3 $\quad OLL \leftarrow OLL \setminus \{\langle u \xrightarrow{c} v, 1\rangle\}$
4 **else**
5 $\quad OLL \leftarrow OLL[\langle u \xrightarrow{c} v, k\rangle \leftarrow \langle u \xrightarrow{c} v, k-1\rangle]$
6 **endif**
7 **return**

Algorithmus B.1.5: (Hinzufügen einer Kante zum lokalen OLL-Katalog)

AddToOLL(in edge $u \xrightarrow{c} v$)
1 **if** $(u \xrightarrow{c} v \notin OLL)$ **then**
2 $\quad OLL \leftarrow OLL \cup \{\langle u \xrightarrow{c} v, 1\rangle\}$
3 **else**
4 $\quad OLL \leftarrow OLL[\langle u \xrightarrow{c} v, k\rangle \leftarrow \langle u \xrightarrow{c} v, k+1\rangle]$
5 **endif**
6 **return**

Literaturverzeichnis

[ABD+89] M. Atkinson, F. Bancilhon, D. J. DeWitt, K. R. Dittrich, D. Maier, St. B. Zdonik. The Object-Oriented Database System Manifesto. In: *Proc. 1st Int'l Conf. on Deductive and Object-Oriented Databases (DOOD'89)*. Kyoto, Japan, Dezember 1989, Seiten 223–240.

[ADJ90] R. Agrawal, S. Dar, H. V. Jagadish. Direct transitive closure algorithms: Design and performance evaluation. *ACM Transactions on Database Systems*, 15(3), 1990:Seiten 427–258.

[AJ90] R. Agrawal, H. V. Jagadish. Hybrid transitive closure algorithms. In: *Proc. 16th Int'l Conf. on Very Large Databases (VLDB'90)*. Brisbane, Australien, August 1990, Seiten 326–334.

[AMKP85] H. Afsarmanesh, D. McLeod, D. Knapp, A. C. Parker. An Extensible Object-Oriented Approach to Databases for VLSI/CAD. In: *Proc. 11th Int'l Conf. on Very Large Databases (VLDB'85)*. Stockholm, 1985, Seiten 13–24.

[ANS99a] ANSI/ISO/IEC 9075-2:1999 (E). *Database Language SQL – Part 2: Foundation (SQL/Foundation)*, September 1999.

[ANS99b] ANSI/ISO/IEC 9075-4:1999 (E). *Database Language SQL – Part 4: Persistent Stored Modules (SQL/PSM)*, September 1999.

[ASL89] A. M. Alashqur, S. Y. W. Su, H. Lam. OQL: A Query Language for Manipulating Object-Oriented Databases. In: *Proc. Int'l Conf. on Very Large Databases (VLDB'89)*. Amsterdam, August 1989, Seiten 433–442.

[BK89] E. Bertino, W. Kim. Indexing Techniques for Queries on Nested Objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2), Juni 1989:Seiten 196–214.

[BPS00] P. A. Bernstein, S. Pal, D. Shutt. Context-based prefetch – an optimization for implementing objects on relations. *The VLDB Journal*, (9), 2000:Seiten 177–189.

[CBB+97] R. G. G. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, D. Wade. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[CCH93] Filippo Cacace, Stefano Ceri, Maurice A. W. Houtsma. A survey of parallel execution strategies for transitive closure and logic programs. *Distributed and Parallel Databases*, 1(4), 1993:Seiten 337–382.

[CFM99] R. Coltun, D. Ferguson, J. Moy. OSPF for IPv6. Network Working Group, Request for Comments 2740, Dezember 1999.

[CIM97] CIMdata, Inc., CIMdata World Headquarters, Ann Arbor, MI 48108 USA. *Product Data Management: The Definition. An Introduction to Concepts, Benefits, and Terminology*, 4. Auflage, September 1997.

[CLR96] T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Introduction to Algorithms*. MIT-Press, 1996.

[Clu98] S. Cluet. Designing OQL: Allowing Objects to be Queried. *Information Systems*, 23(5), 1998:Seiten 279–305.

[CW98] R. Conradi, B. Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2), Juni 1998:Seiten 232–282.

[Dad96] P. Dadam. *Verteilte Datenbanken und Client/Server-Systeme*. Springer-Verlag, 1996.

[DDW02] A. Dhillon, C. DiMinico, A. Woodfin. *Optical Fiber and 10 Gigabit Ethernet*. 10 Gigabit Ethernet Alliance, May 2002.

[DKA+86] P. Dadam, K. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, G. Walch. A DBMS Prototype to Support Extended NF2 Relations: An Integrated View on Flat Tables and Hierarchies. In: *Proc. ACM SIGMOD Int'l Conf. on Management of Data*. Washington, D.C., 1986, Seiten 356–367.

[EM99] A. Eisenberg, J. Melton. SQL:1999, formerly known as SQL3. *ACM SIGMOD Record*, 28(1), März 1999:Seiten 131–138.

[EWP+99] W. Eversheim, M. Weck, S. Pühl, P. Ritz, K. Sonnenschein, M. Walz. Auftrags- und Dokumentenverwaltung bei der technischen Produktentwicklung. In: *Integration von Entwicklungssystemen in Inge-*

nieuranwendungen (Herausgeber M. Nagl, B. Westfechtel). Springer-Verlag, 1999, Seiten 17–46.

[Fit90] M. C. Fitting. *First-order logic and automated theorem proving*. Springer, New York, Heidelberg, 1990.

[For98a] UMTS Forum. The Path towards UMTS – Technologies for the Information Society. *UMTS Forum Report*, 2, 1998.

[For98b] UMTS Forum. UMTS/IMT-2000 Spectrum. *UMTS Forum Report*, 6, Dezember 1998.

[For00] UMTS Forum. Enabling UMTS/Third Generation Services and Applications. *UMTS Forum Report*, 11, Oktober 2000.

[GM88] G. Graefe, D. Maier. Query Optimization in Object-Oriented Database Systems: A Prospectus. In: *Advances in Object-Oriented Database Systems. Proc. 2nd Int'l Workshop on Object-Oriented Database Systems*. (LNCS 334, Springer-Verlag), September 1988, Seiten 358–363.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2), 1993:Seiten 73–170.

[Gra94] G. Graefe. Volcano – An Extensible and Parallel Query Evaluation System. *Transactions on Knowledge and Data Engineering*, 6(1), Februar 1994:Seiten 120–135.

[GSSZ99] J. Gulbins, M. Seyfried, H. Strack-Zimmermann. *Dokumenten-Management - Vom Imaging zum Business-Dokument*. Springer-Verlag, 2. Auflage, 1999.

[HAC90a] M. A. W. Houtsma, P. M. G. Apers, S. Ceri. Distributed Transitive Closure Computations: The Disconnection Set Approach. In: *Proc. 16th Int'l Conf. on Very Large Databases (VLDB'90)*. Brisbane, 1990, Seiten 335–346.

[HAC90b] M. A. W. Houtsma, P. M. G. Apers, S. Ceri. Complex transitive closure queries on a fragmented graph. In: *Proc. 3rd Int'l Conf. on Database Theory*. (LNCS 470, Springer-Verlag), Paris, 1990, Seiten 470–484.

[Hel97] G. Held. *Understanding Data Communications. From Fundamentals to Networking*. Wiley, 2. Auflage, 1997.

[Hew93] Hewlett Packard. *Engineering Data Management - Das Umfeld für die fortschrittliche Produktentwicklung*, 1993.

[HK03] M. Hitz, G. Kappel. *UML@Work. Von der Analyse zur Realisierung*. dpunkt.verlag, Heidelberg, 2. Auflage, 2003.

[HMNR95] T. Härder, B. Mitschang, U. Nink, N. Ritter. Workstation/Server-Architekturen für datenbankbasierte Ingenieuranwendungen. *Informatik Forschung und Entwicklung*, 10(2), 1995:Seiten 55–72.

[IBM01] IBM Corporation. *IBM DB2 Universal Database – SQL Reference – Version 7*, 2001.

[IRW93] Y. Ioannidis, R. Ramakrishnan, L. Winger. Transitive Closure Algorithms Based on Graph Traversal. *ACM Transactions on Database Systems*, 18(3), September 1993:Seiten 512–576.

[JRR91] S. Jablonski, B. Reinwald, T. Ruf. Eine Fallstudie zur Datenverwaltung in CIM-Systemen. *Informatik Forschung und Entwicklung*, 6(2), 1991:Seiten 71–78.

[JS99] C. S. Jensen, R. T. Snodgrass. Temporal Data Management. *IEEE Transactions on Knowledge And Data Engineering*, 11(1), 1999:Seiten 36–44.

[Kä92] W. Käfer. *Geschichts- und Versionsmodellierung komplexer Objekte. Anforderungen und Realisierungsmöglichkeiten am Beispiel des NDBS PRIMA*. Dissertation, Universität Kaiserslautern, Fachbereich Informatik, 1992.

[Kam00] U. Kampffmeyer. Dokumenten-Management - Trends und Ausblicke im DMS-Bereich. *it Fokus*, (11), 2000:Seiten 22–29.

[Kat90] R. H. Katz. Toward a Unified Framework for Version Modelling in Engineering Databases. *ACM Computing Surveys*, 22(4), Dezember 1990:Seiten 375–408.

[Keß95] U. Keßler. *Flexible Speicherungsstrukturen und Sekundärindexe in Datenbanksystemen für komplexe Objekte*. Dissertation, Universität Ulm, 1995.

[Kel90] Udo Kelter. Group-Oriented Discretionary Access Controls for Distributed Structurally Object-Oriented Database Systems. In: *Proc. European Symposium on Research in Computer Security (ESORICS 90)*. Toulouse, Oktober 1990, Seiten 23–33.

[KGM91] T. Keller, G. Graefe, D. Maier. Efficient Assembly of Complex Objects. In: *Proc. ACM SIGMOD Int'l Conf. on Management of Data*. Denver, Colorado, 1991, Seiten 148–157.

[KM94] A. Kemper, G. Moerkotte. *Object-Oriented Database Management - Applications in Engineering and Computer Science*. Prentice Hall, 1994.

[Kos00] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32(4), Dezember 2000:Seiten 422–469.

[Krü75] S. Krüger. *Simulation; Grundlagen, Techniken, Anwendungen*. de Gruyter, 1975.

[KVM68] P. J. Kiviat, R. Villanueva, H. M. Markowitz. *The SIMSCRIPT II programming language*. Prentice-Hall, 1968.

[KW95] A. Koschel, M. Wallrath. Ist die Zukunft der Datenbanken objektorientiert? *EDM-Report*, 2, 1995.

[Lan92] H. Langendörfer. *Leistungsanalyse von Rechensystemen: Messen, Modellieren, Simulation*. Carl Hanser Verlag München, Wien, 1992.

[LE78] A. H. Lightstone, H. B. Enderton. *Mathematical logic: An introduction to model theory*. Plenum Pr. New York, 1978.

[Les02] P. Lescuyer. *UMTS – Grundlagen, Architektur und Standard*. dpunkt.verlag, Juli 2002.

[LW94] B. S. Lee, G. Wiederhold. Efficiently Instantiating View-Objects From Remote Relational Databases. *The VLDB Journal*, 3(3), Juli 1994:Seiten 289–323.

[Mac96] M. Machura. Managing Information in a Co-operative Object Database System. *Software Practice & Experience*, 26(5), Mai 1996:Seiten 545–579.

[Mas92] H. Masson. *STEP Part 1: Overview and Fundamental Principles*. ISO TC184/SC4, 1992.

[MGS+94] D. Maier, G. Graefe, L. Shapiro, S. Daniels, T. Keller, B. Vance. Issues in Distributed Object Assembly. In: *Distributed Object Management (Proc. Int'l Workshop on Distributed Object Management, August 1992, Edmonton, Canada)* (Herausgeber M. T. Özsu, U. Dayal, P. Valduriez). Morgan Kaufmann Publishers, 1994, Seiten 165–181.

[Moy98] J. Moy. OSPF Version 2. Network Working Group, Request for Comments 2328, STD 54, April 1998.

[MP] Metaphase. http://www.plmsol-eds.com/metaphase/index.shtml.

[MS86] D. Maier, J. Stein. Indexig in an Object-Oriented DBMS. In: *Proc. Int'l Workshop on Object-Oriented Database Systems*. IEEE Computer Society Press, 1986, Seiten 171–182.

[MS93] J. Melton, A. R. Simon. *Understanding SQL: A complete guide*. Morgan Kaufmann Publishers, Inc., 1993.

[MS02] J. Melton, A. R. Simon. *SQL:1999; Understanding Relational Language Components*. Morgan Kaufmann Publishers, Inc., 2002.

[MW00] A. Meier, T. Wüst. *Objektorientierte und objektrelationale Datenbanken - Ein Kompaß für die Praxis*. dpunkt.verlag, Heidelberg, 2. Auflage, 2000.

[Myr97] Th. Myrach. Realisierung zeitbezogener Datenbanken: Ein Vergleich des herkömmlichen relationalen Datenmodells mit einer temporalen Erweiterung. *Wirtschaftsinformatik*, 39(1), 1997:Seiten 35–44.

[Nee87] F. Neelamkavil. *Computer simulation and modelling*. Wiley, 1987.

[Nit89] S. Nittel. Relationale und objektorientierte Datenbanksysteme für CIM-Applicationen - ein Vergleich. *CIM Management*, (6), 1989:Seiten 11–14.

[Now01] J. Nowitzky. Partitionierungstechniken in Datenbanksystemen. *Informatik Spektrum*, 24(6), Dezember 2001:Seiten 345–356.

[Oli02] V. Oliva. *Ethernet – The Next Generation WAN Transport Technology*. 10 Gigabit Ethernet Alliance, Mai 2002.

[OMG01] OMG. *OMG Unified Modeling Language Specification. Version 1.4*, September 2001.

[Ora01a] Oracle Corporation. *Oracle9i Application Developer's Guide – Fundamentals*, Juni 2001.

[Ora01b] Oracle Corporation. *Oracle9i PL/SQL User's Guide and Reference*, Juni 2001.

[Ora01c] Oracle Corporation. *Oracle9i SQL Reference*, Juni 2001.

[OW96] T. Ottmann, P. Widmayer. *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag Heidelberg, 1996.

[Owe93] J. Owen. *STEP: An Introduction*. Information Geometers, Winchester, UK, 1993.

[P+88] B. Page, et al. *Simulation und moderne Programmiersprachen; Modula-2, C, Ada*. Springer, 1988.

[PAMS94] H. Peltonen, K. Alho, T. Männistö, R. Sulonen. An Authorization Mechanism For a Document Database. In: *Proc. ASME Database Symposium*. Minneapolis, 1994, Seiten 137–143.

[Pel00] H. Peltonen. *Concepts and an Implementation for Product Data Management*. Dissertation, Helsinki University of Technology, Department of Computer Science and Engineering, 2000.

[PMAS93] H. Peltonen, T. Männistö, K. Alho, R. Sulonen. An Engineering Document Management System. In: *Proc. ASME Winter Annual Meeting*. Paper 93-WA/EDA-1. New Orleans, Louisiana 1993.

[Pri74] A. A. B. Pritsker. *The GASP IV simulation language*. Wiley, 1974.

[Pri84] A. A. B. Pritsker. *Introduction to simulation and SLAM II*. Wiley, 1984.

[PT85] P. Pistor, R. Traunmüller. A Data Base Language for Sets, Lists, and Tables. IBM Wiss. Zentrum Heidelberg, Techn. Rep. TR 85.10.004, Oktober 1985.

[PT86] P. Pistor, R. Traunmüller. A Database Language for Sets, Lists, and Tables. *Information Systems*, 11(4), 1986:Seiten 323–336.

[RHM84] A. Rosenthal, S. Heiler, F. Manola. An Example of Knowledge-Based Query Processing in a CAD/CAM DBMS. In: *Proc. 10th Int'l Conf. on Very Large Databases (VLDB'84)*. Singapur 1984, Seiten 363–370.

[RPK+99] B. Reinwald, H. Pirahesh, G. Krishnamoorthy, G. Lapis, B. Tran, S. Vora. Heterogeneous Query Processing through SQL Table Functions. In: *Proc. 15th Int'l Conf. on Data Engineering*. Sydney, Australien, März 1999, Seiten 366–373.

[RPM96] K. Ranneberg, A. Pfitzmann, G. Müller. Sicherheit, insbesondere mehrseitige IT-Sicherheit. *it+ti – Informationstechnik und Technische Informatik*, 38(4), 1996:Seiten 7–10.

[RR96] R. Ramakrishnan, D. J. Ram. Modeling Design Versions. In: *Proc. 22nd Int'l Conf. on Very Large Databases (VLDB'96)*. Mumbai, Indien, 1996, Seiten 556–566.

[RTG97a] M. Ritter, P. Tran-Gia. Mechanismen zur Steuerung und Verwaltung von ATM-Netzen - Teil 1: Grundlegende Prinzipien. *Informatik-Spektrum*, 20(4), August 1997:Seiten 216–224.

[RTG97b] M. Ritter, P. Tran-Gia. Mechanismen zur Steuerung und Verwaltung von ATM-Netzen - Teil 2: Modellierung und Leistungsbewertung. *Informatik-Spektrum*, 20(5), Oktober 1997:Seiten 276–285.

[SAA+94] R. T. Snodgrass, I. Ahn, G. Ariav, et al. TSQL2 Language Specification. *ACM SIGMOD Record*, 23(1), März 1994:Seiten 65–96.

[SBY97] Sang K. Cha Sang B. Yoo. Integrity maintenance in a heterogeneous engineering database environment. *Data & Knowledge Engineering*, 21(3), Februar 1997.

[SC90] E. J. Shekita, M. J. Carey. A performance evaluation of pointer-based joins. In: *Proc. ACM SIGMOD Int'l Conf. on Management of Data*. Atlantic City, NJ, Mai 1990, Seiten 300–311.

[SC97] V. Srinivasan, D. T. Chang. Object persistence in object-oriented applications. *IBM SYSTEMS JOURNAL*, 36(1), 1997:Seiten 66–87.

[Sch91] T. J. Schriber. *An introduction to simulation using GPSS/H*. Wiley, 1991.

[Sch96a] W. Schäfer. IT-Sicherheits-Anforderungen in Dienstleistungsunternehmen. *it+ti – Informationstechnik und Technische Informatik*, 38(4), 1996:Seiten 11–14.

[Sch96b] H. Schwetman. *CSIM18 – The Simulation Engine*. Mesquite Software, Inc., 1996.

[Sch99] J. Schöttner. *Produktdatenmanagement in der Fertigungsindustrie. Prinzip – Konzepte – Strategien*. Carl Hanser Verlag, München Wien, 1999.

[Sed95] R. Sedgewick. *Algorithmen*. Addison Wesley, 1995.

[SHK+97] A. Schill, R. Hess, S. Kümmel, D. Hege, H. Lieb. *ATM-Netze in der Praxis*. Addison Wesley, 1997.

[Sno00] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann Publishers, San Francisco, California, 2000.

[Spi94] Ph. Spidy. *STEP Part 11: The EXPRESS Language Reference Manual*. ISO TC184/SC4 N65, 1994.

[SWW00] O. Stiemerling, M. Won, V. Wulf. Zugriffskontrolle in Groupware – Ein nutzerorientierter Ansatz. *Wirtschaftsinformatik*, 42(4), 2000:Seiten 318–328.

[Tan89] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall International Editions, 2. Auflage, 1989.

[TOC93] G. Talens, C. Oussalah, M. F. Colinas. Versions of Simple and Composite Objects. In: *Proc. 19th Int'l Conf. on Very Large Databases (VLDB'93)*. Dublin, August 1993, Seiten 62–72.

[Tol00] Bruce Tolley. *Strategic Directions Moving the Decimal Point: An Introduction to 10 Gigabit Ethernet*. Cisco Systems, Inc., 2000.

[Tür03] C. Türker. *SQL:1999 & SQL:2003*. dpunkt.verlag, Heidelberg, Februar 2003.

[VDI99] VDI, Verein Deutscher Ingenieure, VDI-Gesellschaft Entwicklung Konstruktion Vertrieb (Herausgeber). *Datenverarbeitung in der Konstruktion – Einführung und Wirtschaftlichkeit von EDM/PDM-Systemen*. Beuth Verlag GmbH, Berlin, 1999.

[VL97] S. P. VanderWiel, D. J. Lilja. When Caches Aren't Enough: Data Prefetch Techniques. *IEEE Computer*, 30(7), 1997:Seiten 23–30.

[VL00] S. P. VanderWiel, D. J. Lilja. Data Prefetch Mechanisms. *ACM Computing Surveys*, 32(2), Juni 2000:Seiten 174–199.

[Woh00] P. Wohlmacher. Sicherheitsanforderungen und Sicherheitsmechanismen bei IT-Systemen. *EMISA-Forum, Mitteilungen der GI-Fachgruppe „Entwicklungsmethoden für Informationssysteme und deren Anwendung"*, Heft 1, 2000:Seiten 16–25.

[YM98] C. T. Yu, W. Meng. *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann Publishers, San Francisco, California, 1998.