Interner Bericht

Vergleichsverfahren für Systementwurfsgraphen

Konzepte, Algorithmen, Implementation

Rainer Gerten
Marcus Powarzynski

214/91

Fachbereich Informatik

Universität Kaiserslautern · Postfach 3049 · D-6750 Kaiserslautern

INTERNER BERICHT

Vergleichsverfahren für Systementwurfsgraphen

Konzepte, Algorithmen, Implementation

Rainer Gerten
Marcus Powarzynski

214/91

Herausgeber:

AG Programmiersprachen und Compilerbau
Leiter: Prof. Dr.-Ing. H.-W. Wippermann
Fachbereich Informatik
Universität Kaiserslautern

September 1991

0. Vorwort

In Laufe der letzten Jahrzehnte ist der Prozeß der Softwareentwicklung methodisiert und zum Teil auch formalisiert worden. I.a. unterteilt man den Vorgang in grobe Stufen, Entwicklungsphasen genannt. Jede dieser Phasen betrachtet den entstehenden Entwurf des Projekts aus verschiedenen Sichtweisen. Aus dieser Sichtweise resultieren etliche Modelle und Darstellungsformen und mit ihnen auch verschiedene rechnergestützte Entwicklungswerkzeuge. In frühen Phasen sind beispielsweise Datenflußdiagramme eine nützliche Darstellungsform, in späteren konkrete Algorithmenbeschreibungen.

Entwurfsänderungen im Laufe der Entwicklungszeit müssen in allen betroffenen Ebenen neu formuliert werden, eine automatisierte phasenübergreifende Behandlung ist daher i.a. nicht oder nur teilweise möglich. Um effizienter und weniger fehleranfällig arbeiten zu können, wurden aus diesem Grund in letzter Zeit Ansätze gemacht, den gesamten Softwareentwicklungsprozeß von der Anforderungsanalyse bis hin zur Wartungsphase einem einheitlichen Konzept und einer einheitlichen Darstellungsform zu unterwerfen, die sich darüberhinaus zur Realisation auf Rechnersystemen eignen.

Der vorliegende Bericht entstand im Rahmen eines solchen Projekts. Es wurden eine allumfassende Systementwurfssprache und die dazugehörigen Konzepte entwickelt, die sämtliche Entwurfsphasen und die wichtigsten -prinzipien zu unterstützen vermögen.

Es liegen bereits zwei Arbeiten zu diesem Projekt vor. Sie stellen im wesentlichen neben der eigentlichen Definition der Systementwurfssprache zwei Entwicklungswerkzeuge vor, die auf einer einheitlichen Datenbasis operieren [GK-91, Kel-90].

Ein Bereich innerhalb der Forschungen ist die Wiederverwendung von Softwareentwürfen. Schon existierende Lösungen sollen bei der Entwicklung eines neuen Entwurfs durch Vergleich und Bewertung des Grades der Ähnlichkeit ausgewählt und dem Entwickler nutzbar gemacht werden.

Dieser Bericht beschäftigt sich mit einem Kernpunkt der Wiederverwendung, dem Vergleich zweier Softwareentwürfe.

Es werden zunächst grundsätzliche Konzepte ausgearbeitet, die den Ähnlichkeitsaspekt unter verschiedenen Gesichtspunkten charakterisieren. Daraufhin werden Algorithmen konstruiert, die verschiedenartige Vergleichsfunktionen realisieren und zu einer Gesamtfunktion kombinieren. Um zu einem späteren Zeitpunkt die Leistungsfähigkeit dieser Funktionen in der Praxis untersuchen zu können, liegt darüberhinaus ein lauffähiges Programm vor.

Die weitere Ausarbeitung gliedert sich folgendermaßen:

Kapitel 1,	**Einleitung**, motiviert und führt in die Thematik ein.
Kapitel 2,	**Der Systementwurfsgraph**, umreißt die in diesem Zusammenhang wichtigen Konzepte der bereits in [GK-91] vorgestellten Systementwurfssprache.
Kapitel 3,	**Die Vergleichsverfahren**, ist der Haupteil der Ausarbeitung. Hier werden alle Vergleichsfunktionen, einschließlich der Gesamtvergleichsfunktion, vorgestellt.
Kapitel 4,	**Implementation**, gibt einen Überblick über das erstellte Programm.
Kapitel 5,	**Resümee,** rundet den Bericht mit abschließenden Worten ab.

Inhaltsübersicht

Inhaltsverzeichnis

1. Einleitung

Wiederverwendung (engl.: Reusability) von Softwareentwürfen soll den Softwareentwicklungsprozeß effizienter gestalten.

Solange sich Wiederverwendung auf die Programm- bzw. Modulebene beschränkt und darüber hinaus nur manuell (vom Anwender) betrieben wird, gestaltet sie sich sehr ineffizient. Jeder Programmierer weiß um die Mühen, Module in Bibliotheken zu suchen, zu entscheiden, ob sie brauchbar sind und schließlich die Anpassung und Einbindung ins eigene Projekt vorzunehmen. Es ist oft weniger aufwendig, Module komplett neu zu entwickeln.

Effiziente Wiederverwendung muß weitergehen. Sie muß Entwicklungsphasen-übergreifend sein und (zumindest teil-) automatisiert. Bisherige Ansätze hierzu waren in ihrer Leistungsfähigkeit eingeschränkt, da sich konventionelle Softwareentwicklungsumgebungen i.a. **mehrerer** Datenbasen entsprechend der Entwicklungsphasen zur Speicherung der Projekte bedienen. Reusability-Tools müssen bisher mittels Converter auf die verschiedenen Datenformate zugreifen, eine Erkennung wiederverwendbarer Teile über Phasengrenzen hinweg ist konzeptionell daher schwierig.

Durch Schaffung einer **einheitlichen** Darstellungsform von Softwareentwürfen (siehe Vorwort), in der alle Phasen integriert sind, gewinnt die Entwicklung eines Reusability-Systems neues Interesse. Es besteht nun unter den gegebenen Voraussetzungen die Möglichkeit, ein allumfassendes Wiederverwendungssystem zu gestalten, das über Phasen- bzw. Projektgrenzen und auch gruppenübergreifend arbeitet.

In der neuen Softwareentwicklungsumgebung könnte die Einbindung des Reusability-Systems aus der Sicht des Anwenders sich folgendermaßen offenbaren:

Während das Projekt in der Entwicklungsphase ist, wird es ständig im Hintergrund mit schon bestehenden Lösungen **verglichen**. Dabei wird der Grad der Ähnlichkeit untersucht und bewertet. Aufgrund dieses Grades wird entschieden, welche Kandidaten als Vorschläge präsentiert werden und in welcher Reihenfolge dies geschieht. Dem Benutzer werden **interaktiv Vorschläge** in einer Liste **angeboten**, deren Aufbau die Rangfolge der möglichen Lösungen widerspiegelt. Er kann aus dem Angebot Teilentwürfe auswählen und sie modifizieren. Das System markiert Abweichungen zwischen Vorschlag und Problemansatz, um die **Anpassung** an das aktuelle Problem zu unterstützen.

Ein Kernpunkt ist der Vergleich und die Bewertung des Grades der Ähnlichkeit zweier Entwürfe.

In diesem Bericht geht es zunächst darum, den intuitiven Begriff von Ähnlichkeit (im Sinne von Eignung zur Wiederverwendung) aus verschiedenen Betrachtungspunkten heraus zu beleuchten und anhand einiger Sichtweisen und Methoden zu formalisieren. Daher werden **mehrere,** in zwei Gruppen eingeteilte Ansätze vorgeschlagen, wie die Bewertung zu treffen ist. Die erste der beiden vorgestellten Gruppen versucht den Vergleich über die **Inhalte**, die zweite über die **Struktur** der Entwürfe vorzunehmen. Aus diesen Ansätzen werden Konzepte und schließlich konkrete Vergleichsverfahren entwickelt. Jedes Verfahren wird bezüglich seiner Leistungsfähigkeit und Güte einge-

schätzt. Die Untersuchungen führen zu einer abschließenden Betrachtung, wie die Einzelverfahren sinnvoll zu einem Gesamtverfahren kombiniert werden können. Die Realisierbarkeit (Beachtung von Zeitkomplexitäten) wird dabei berücksichtigt. Die Kombination der Einzelverfahren soll die Aussagekraft des Gesamtverfahrens vergrößern, denn so wird ein breites Spektrum der Aspekte, die die Ähnlichkeit charakterisieren, abgedeckt.

2. Der Systementwurfsgraph

2.1. Einleitung

Im Vorwort wurde erwähnt, daß zu Anfang dieses Projekts die Schaffung einer Softwareentwicklungssprache stand. Sie soll im folgenden **kurz** vorgestellt werden, wobei auf die für den vorliegenden Bericht relevanten Aspekte natürlich der Schwerpunkt gelegt wird. Für detaillierte Erläuterungen wird die Lektüre von [GK-91] und [Ger-90] empfohlen.

Die mit der Softwareentwurfssprache geschaffene Darstellungsform erfüllt die Forderung, **sämtliche** Informationen, die während der Projektentwicklung anfallen, modellieren zu können. Sie weist deshalb sowohl Konzepte zur Beschreibung von **Algorithmen** (für die späten Phasen), hier wurde ein imperativer Ansatz gewählt, als auch zur Darstellung **informeller Daten** (für frühere Phasen) auf. Ein wichtiger Punkt dabei ist, daß die informellen Daten ebenso behandelt werden wie die Algorithmenbeschreibungen (Kontroll- bzw. Datenstrukturen), z.B. dem gleichen Parameterkonzept unterworfen sind. Man spricht von **Orthogonalität**. Die Verwaltung und Behandlung informeller Daten ist also ins Gesamtkonzept eingebettet, was bei herkömmlichen Programmiersprachen nicht der Fall ist. Auch wenn im folgenden aus Anschaungsgründen hauptsächlich die algorithmische Sichtweise hervorgehoben wird, meist sogar nur die der Kontrollstrukturen, sollte sich der Leser ständig vor Augen halten, daß sehr viel allgemeinere Beschreibungen möglich sind.

Die Softwareentwurfssprache erlaubt die **strukturierte** und **abstrahierende** Modellierung von Information. Es existiert eine textuelle sowie eine graphische Darstellung. Die graphische Darstellung ist zur Veranschaulichung zu bevorzugen, da so Strukturen gut ersichtlich sind. Solche Graphen werden im folgenden als **Softwareentwurfsgraphen** bezeichnet. Der Graph ist gerichtet, insofern sind Begriffe von Baumstrukturen[1], wie Sohn und Vater, sinnvoll. Es gibt einen ausgezeichneten Knoten innerhalb jedes (Teil-) Graphen, die Wurzel. Wir zeichnen Väter oberhalb ihrer Söhne, die Verfeinerung des Entwurfs steigt von oben nach unten, die Abstraktion von unten nach oben. Dementsprechend lassen sich obere Ebenen eines Graphen früheren, untere späteren Entwicklungsphasen eines Entwurfs zuordnen.

Mit **Projektgraph** bezeichnet man einen Graphen, der sämtliche Beschreibungen zu einem Projekt enthält. Vor allem im Hinblick auf die Wiederverwendungsfunktion ist es wichtig, auf alle schon existierenden Projektbeschreibungen zugreifen zu können. Aus diesem Grunde macht man sämtliche Projektgraphen zu Teilgraphen eines übergeordneten Graphen, dem **Weltgraph**.

---

[1]Trotzdem ist der Graph kein Baum, die graphentheoretischen Eigenschaften des SEG sind allgemeiner als die von Bäumen, siehe [GK-91], Kap. 3.3.

2.2. Konstrukte und Konstruktklassen

Wir wollen uns nun mit dem Aufbau von Softwareentwurfsgraphen näher beschäftigen.

Jeder Knoten des Graphen beinhaltet im wesentlichen eine Knotenidentifikation (im einfachsten Fall einen Namen) und ein Konstrukt. Konstrukte wiederum können auf Knoten verweisen. Auf diese Weise entsteht die Struktur des Graphen.

Die Konstrukte selbst lassen sich, je nach ihrer Beschreibungsart, in drei Klassen einteilen. Es gibt

- **operationale** Konstrukte, sie beschreiben operationale und funktionale Informationen der Zielumgebung,
- **strukturelle** Konstrukte, sie beschreiben Datenstrukturen und
- **informelle** Konstrukte, sie beschreiben die Entwicklungs- und Funktionsumgebung eines geplanten Softwareprojekts (z.B. Betriebssystem, Ein- und Ausgabemedien), Anforderungen (Pflichtenheft), Compileroptionen oder Kommentare u.v.m.

Je nachdem, welcher Klasse ein Konstrukt angehört, spricht man auch von dem zugehörigen Knoten von einem operationalen, strukturellen bzw. informellen Knoten. Auch Teilgraphen können mit Klassennamen benannt werden, man richtet sich dann nach der Klasse der Wurzel, auch wenn andere Knoten des Teilgraphen anderen Klassen angehören (was durchaus sinnvoll und nötig ist, wie in einem der späteren Beispiele deutlich wird).

In vorliegendem Bericht wird der in [GK-91] vorgestellte Formalismus zur textuellen Beschreibung der Graphen verwendet. Sie wird im folgenden, neben der graphischen Darstellung, aufgezeigt.

Die Bezeichnung von Konstrukten setzt sich aus einem Funktor, gefolgt von den Komponenten des Konstruktes in Klammern, zusammen. Die Konstruktklasse wird als Index an den Funktor gehängt (o: operational, s: strukturell, i: informell). Im folgenden soll der Index x eine der drei Möglichkeiten vertreten.

Folgende Konstrukte sind möglich:

a) Die Sequenz

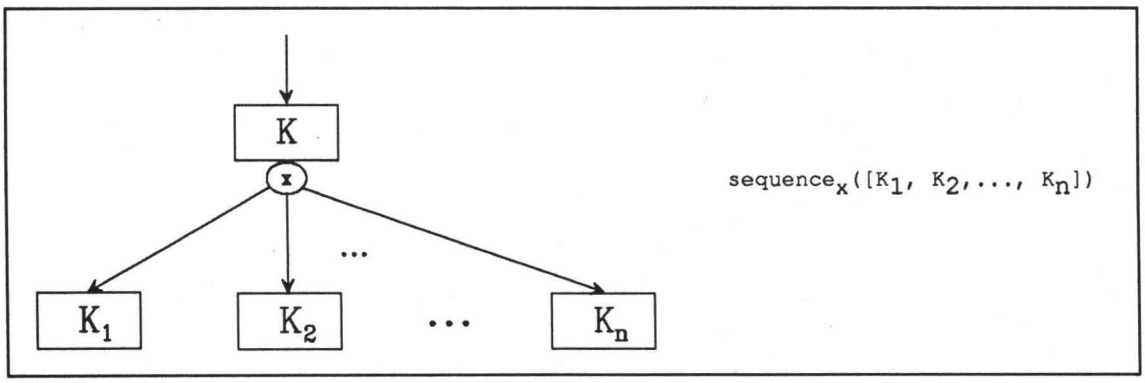


ABB. 2.2.1. DAS SEQUENZKONSTRUKT

Die Sequenz dient zur Darstellung einer Folge von Anweisungen (operationale Interpretation, entspricht in etwa einem Block in PASCAL), Datenstrukturen (strukurelle Interpretation, entspricht in etwa dem **record** in PASCAL) oder einer Zusammenfassung informeller Daten (informelle Interpretation). Sequenzenkonstrukte bestehen demnach aus der Angabe der Klasse und einer Liste von Verweisen auf einen oder mehrere Sohnknoten ("K_i"), die der Reihe nach, von links nach rechts zu behandeln sind.

b) Die Selektion

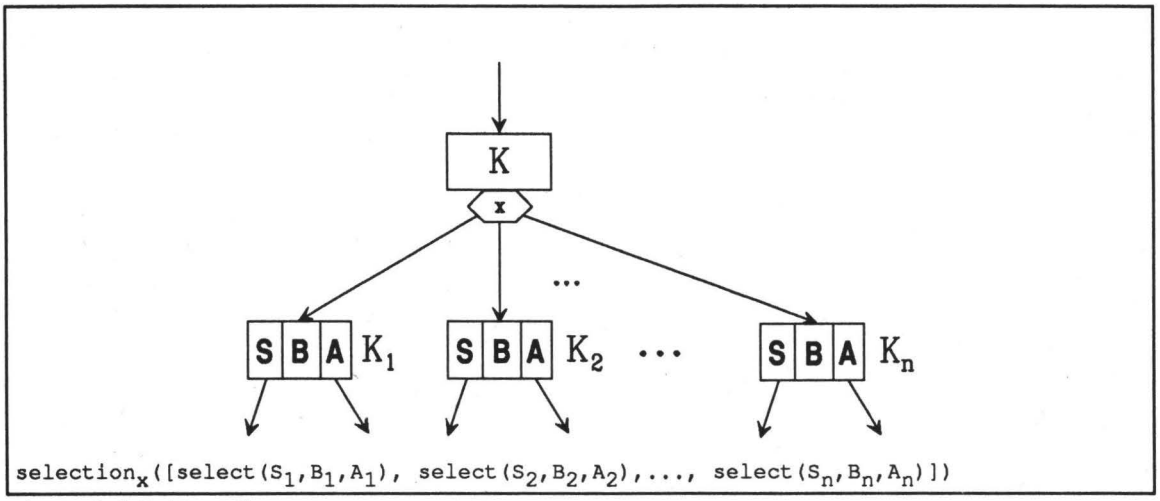


ABB. 2.2.2. DAS SELEKTIONSKONSTRUKT

Die Selektion dient zur Auswahl einer Aktion (operationale Interpretation), eines Elements einer Menge (strukturelle Interpretation, entspricht einem union in C bzw. einer Variante in PASCAL) oder eines informellen Datums (informelle Interpretation). Die Sohnknoten von Selektionen ("K_i") bestehen aus drei Komponenten,

- dem Selektor ("S_i"), er wertet eine Bedingung aus ,
- einem Bedingungsparameter ("B_i"), auf den wir nicht näher eingehen müssen,
- einem Aktionsteil ("A_i"), der dann ausgewertet wird, wenn die Bedingung des Selektors zu wahr ausgewertet wurde.

Die operationale Selektion kann demnach mit einem geschachtelten **if-then-else** verglichen werden, die strukturelle mit einem **variant-record** aus PASCAL.

c) Die Iteration

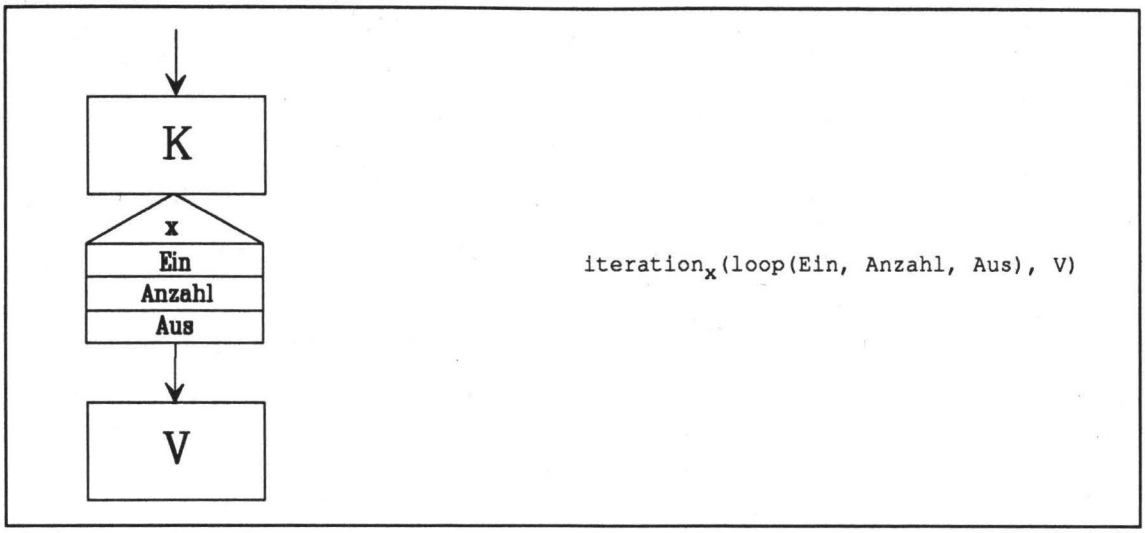


ABB. 2.2.3. DAS ITERATIONSKONSTRUKT

Die Iteration dient zur Realisierung von bedingten Schleifen (operationale Interpretation) oder Arrays (strukturelle Interpretation). Sie besteht aus einem Satz von drei Schleifenbedingungen (zusammengefaßt mit der Bezeichnung "loop") und einem Verweis auf den Schleifenkörper (Sohnknoten "V"). Die Semantik der operationalen Iteration verlangt jeweils die Validierung der Eingangsbedingung ("Ein") vor dem Schleifeneintritt, die der Ausgangsbedingung ("Aus") nach dem Schleifendurchlauf und die der Zählbedingung ("Anzahl"). Auf diese Weise sind quasi die bekannten **for-**, **repeat-until-** und **while-** Anweisungen in einem Konstrukt vereint. Schleifenbedingungen können auch fehlen, sie werden dann nicht beachtet. Die strukturelle Interpretation verwendet nur die Zählbedingung als Größenangabe des Arrays.

Die folgenden Konstrukte machen die Knoten, in denen sie stehen, zu Endknoten, da sie auf keine weiteren Knoten verweisen.

d) Die Rekursion

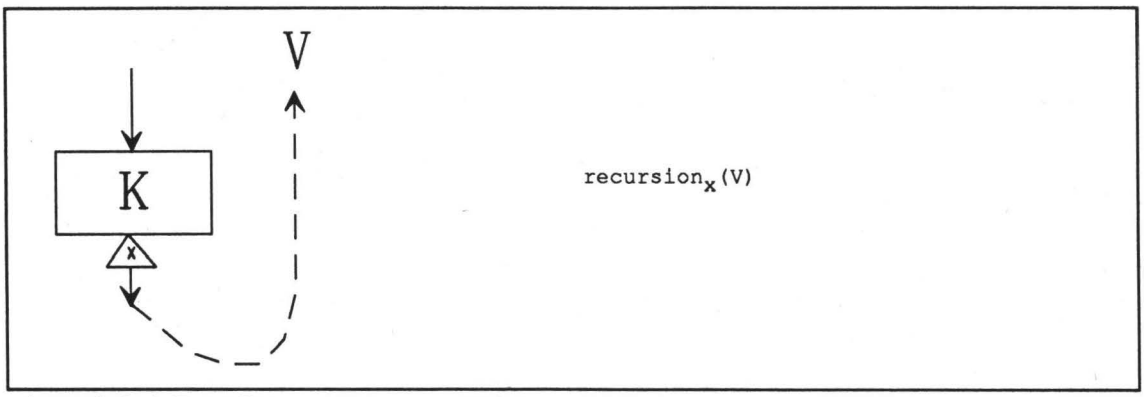


ABB. 2.2.4. DAS REKURSIONSKONSTRUKT

Die Rekursion ermöglicht rekursive Aufrufe innerhalb von Teilgraphen (operationale Interpretation) bzw. die Definition rekursiver Datenstrukturen (z.B. Listen). Es wird der Name des referenzierten Knotens angegeben. Die Rekursion ist das einzige Konstrukt, das auf einen Knoten oberhalb verweist. Dieser Verweis wird nicht als eigentliche Kante angesehen (deshalb gestrichelt dargestellt), um den Graphen zyklenfrei zu halten.

e) Das Atom

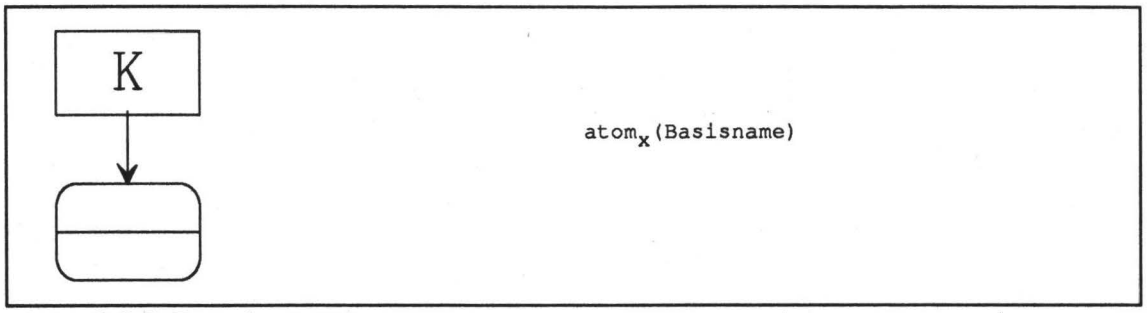


ABB. 2.2.5. DAS ATOMKONSTRUKT

Atome sind die kleinsten Elemente der Softwareentwurfssprache, ihnen ist eine Basis, in der die Semantik des Atomes beschrieben ist, zugeordnet. Operationale Atome beschreiben primitive Operationen, strukturelle Grunddatentypen. Es gibt keinen festen Satz von Atomen, ihre Zahl ist erweiterbar.

f) Das nil-Konstrukt

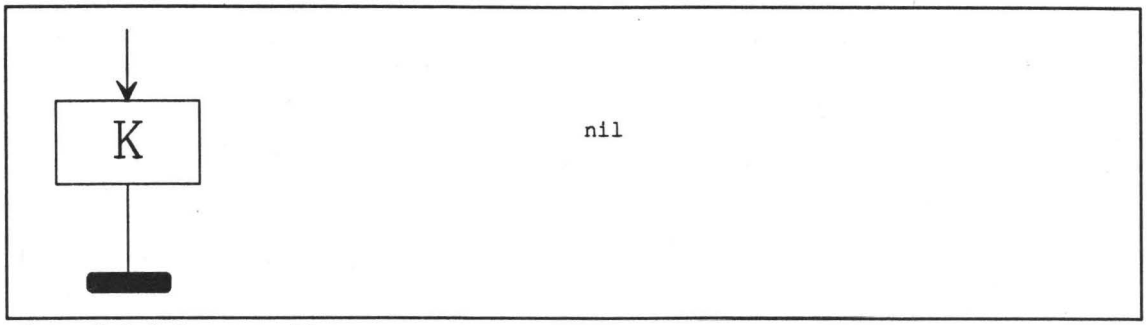


ABB. 2.2.6. DAS NIL-KONSTRUKT

Das nil-Konstrukt dient dazu, unvollständige Entwürfe oder Prototypen semantisch korrekt sein zu lassen. Es kann später durch jedes andere Konstrukt ersetzt werden.

g) Die Instanz

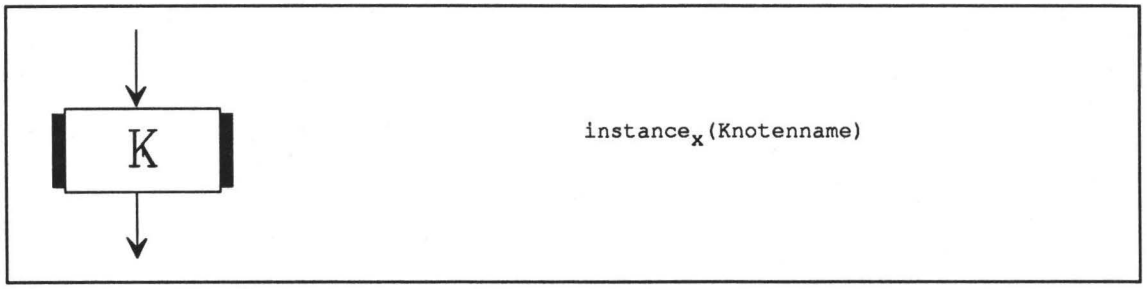


ABB. 2.2.7. DAS INSTANZ-KONSTRUKT

Die Instanz verweist auf einen Teilgraphen und ermöglicht so die Instanziierung desselben, ohne ihn erneut anzugeben. Hierzu noch einige Worte später.

Die möglichen Konstrukte (mit Ausnahme des nil-Konstruktes), samt ihrer Interpretation infolge der Konstruktklasse, sind in folgender Tabelle zusammengefaßt.

Konstruktor	*Operationale Interpretation*	*Strukturelle Interpretation*	*Informelle Interpretation*
Sequenz	Sequenz	Verbund	Aggregation
Selektion	Verzweigung	Vereinigung	Alternative
Iteration	Iteration	Feld	--
Instanz	Makro/Prozedur	Variable	--
Rekursion	Rekursiver Aufruf	Rek. Datenstruktur	--
Atom	Basis-Funktion	Basis-Typ	Kommentar

2.3. Knoten und Knotenverweise

Kommen wir zurück zu den Knoten. Sie bestehen aus vier Komponenten, nämlich

- dem Knotennamen,
- dem Knotentyp,
- einer sog. formalen Parameterliste (FPL) und
- dem Knotenkonstrukt.

```
node(Knotenname, Knotentyp, FPL, Konstrukt)
```

Beispiel:

```
node("Projekt 1", "root", [fp("in, "x"), fp("inout", "y")], ...)
```

ABB. 2.3.1. TEXTUELLE DARSTELLUNG VON (EXPLIZITEN) KNOTEN

Knotenname und -typ dienen zur Identifikation des Knotens. Der Typ dient zur Klassifizierung von Knoten hinsichtlich ihrer Bedeutung für bearbeitende Werkzeuge. Er ist für unsere Betrachtungen weniger wichtig. Die formale Parameterliste gibt die formalen Parameter[2] des Knotens bzw. des darunterliegenden Teilgraphens an. Ihre Angabe ist nur bei Atomknoten (das sind Knoten, deren Konstrukte Atome sind) zwingend.

[2]Jeder Parameter wird in der Form *fp(Zugriffsrecht, Parametername)* angegeben. Mögliche Zugriffsrechte sind "in", "out" und "inout".

Wird ein Knoten behandelt, so ist er von diesem Zeitpunkt an bekannt und kann später über seinen Namen referenziert werden, auch als Parameter. Daher können nicht nur Datenstrukturen ("Variable"), sondern auch Kontrollstrukturen ("Prozeduren") als Parameter übergeben werden. Wird also ein Knoten behandelt, so entspricht dies der gleichzeitigen Definition der Struktur, die der darunterliegende Teilgraph beschreibt, als auch der Instanziierung, so daß die später behandelten Knoten diese Struktur benutzen können. Einer Instanz auf einen Knoten entspricht einer erneuten Instanziierung dieser Struktur (sie kann durch Angabe von Parametern in einer anderen Form vorliegen), die jetzt unter dem Namen des instanziierenden Knotens bekannt gemacht worden ist.

Die Kanten des Graphen werden durch ein eigenes Konstrukt, dem Knotenverweis[3], modelliert, soweit es sich bei auf den verwiesenen Knoten um einen Knoten, wie er oben beschrieben wurde, handelt. Man spricht dann von **expliziten** Knoten. Schachtelt man Konstrukte direkt ineinander ohne den Umweg über Knotenverweise, so spricht man von **impliziten** Knoten. Implizite Knoten haben keinen Namen, Typ bzw. formale Parameterliste. Sie können dementsprechend weder als Parameter übergeben werden, noch als Typendeklaration fungieren. Um sich durch diese Unterscheidung nicht zu sehr verwirren zu lassen, stellt man sich am besten immer explizite Knoten vor, d.h. Konstrukte enthalten als Unterkonstrukte Verweise auf Knoten, in denen dann erst die "eigentlichen" Unterkonstrukte stehen.

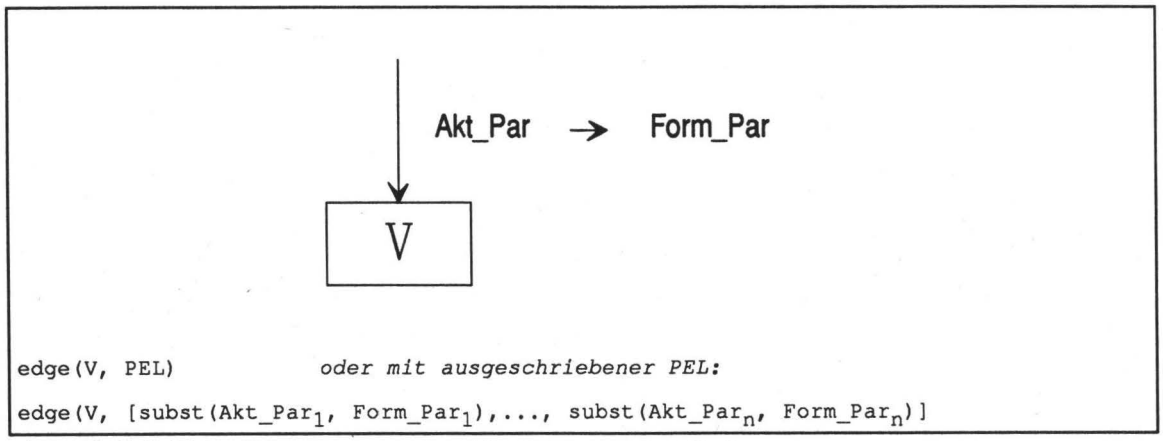


ABB. 2.3.2. DARSTELLUNGEN DES KNOTENVERWEISES

[3]Der Knotenverweis enthält außer der Angabe des Knotens, auf den verwiesen wird noch die sog. **Parameterersetzungsliste** (PEL), deren Funktion später erläutert wird.

Bsp. der text. Darstellung von

a) impliziten Knoten (die beiden Sohnknoten der Sequenz sind implizite Knoten)

```
node("K", Typ, [], sequence_o([Konstrukt des 1. Sohnknotens,
                               Konstrukt des 2. Sohnknotens])
```

b) expliziten Knoten

```
node("K",   Typ, [], sequence_o(edge(K_1, PEL), edge(K_2, PEL)))
node("K_1", Typ, [], Konstrukt von Sohnknoten K_1)
node("K_2", Typ, [], Konstrukt von Sohnknoten K_2)
```

ABB. 2.3.3. BEISPIEL ZU IMPLIZITEN UND EXPLIZITEN KNOTEN

2.4. Geltungsbereich von Namen und Parameterkonzept

Ein einmal behandelter Knoten ist nicht an jeder Stelle des Graphen bekannt, er besitzt einen eingeschränkten **Geltungsbereich**. Dieses Konzept ist aus strukturierten Programmiersprachen bekannt. Es gelten folgende Regeln:

Der Geltungsbereich eines Knotens erstreckt sich nach rechts und nach unten, jedoch nicht nach links und oben.

Das bedeutet einerseits, daß ein Knotenname den Knoten auf gleicher Ebene rechts davon, aber auch Unterstrukturen solcher Knoten, bekannt sind. Das bedeutet aber auch andererseits, daß Knoten aus Unterstrukturen in darüberliegenden Knoten nicht bekannt sind (Information-Hiding!).

Dementsprechend kann ein und derselbe Knotennamen mehrmals auftrteten, der neue **überdeckt** dann den alten. Verläßt man den Geltungsbereich des neuen, gilt wieder der alte.

Die folgenden Beispiele sollen den Sachverhalt verdeutlichen.

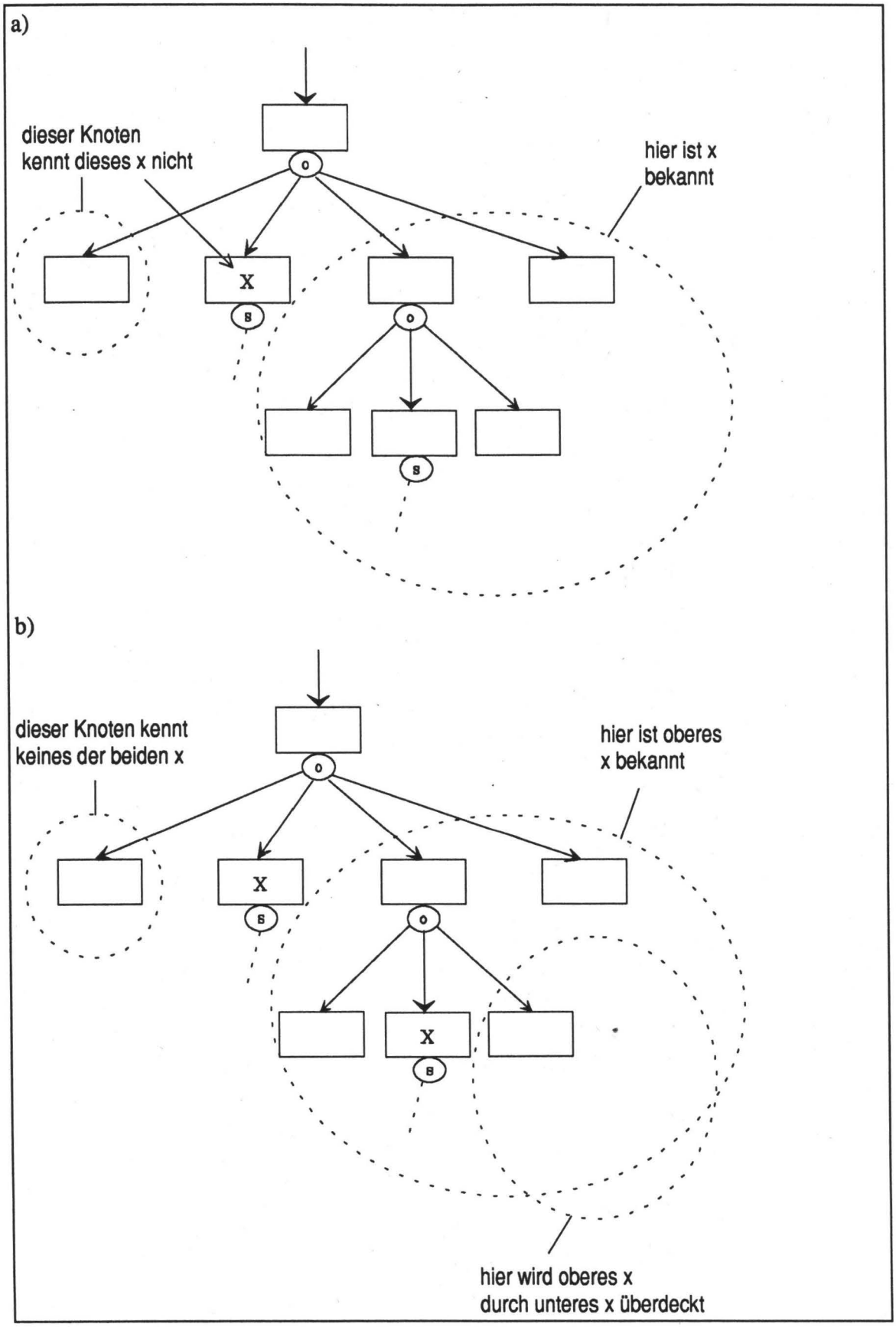


ABB. 2.4.1. BEISPIELE ZUM GELTUNGSBEREICH

Das Parameterkonzept

Die Datenflüsse zwischen Teilgraphen werden durch Parameter geregelt. Veranschaulicht man sich den Sachverhalt anhand der operationalen Interpretation, so kann jeder operationale Teilgraph als Modul bzw. Prozedur gesehen werden (siehe Bild). Zur Übergabe von aktuellen Parametern dienen Knotennamen, die dem Teilgraphen bekannt sind, deren Geltungsbereich den Teilgraphen also beinhaltet. Ein Knotenverweis auf einen solchen Teilgraphen kann als Aufruf interpretiert werden. Entsprechend bekannter Programmiersprachen gibt es auch hier ein Konzept zur Übergabe von aktuellen nach formalen Parametern. Und zwar ermöglicht jeder Knotenverweis, wie schon angedeutet, die Angabe einer Parameterersetzungsliste (PEL), siehe auch die Abbildung hierzu. Dort stehen die Ersetzungsvorschriften von aktuellen nach formalen Parameter. Die Parameterersetzungsliste ist also Schnittstelle zwischen der Umgebung, in der das Modul eingesetzt wird ("außen") und der Umgebung innerhalb des Moduls selbst ("innen"). Daten können von außen nach innen (im Graphen: von oben nach unten) und umgekehrt fließen.

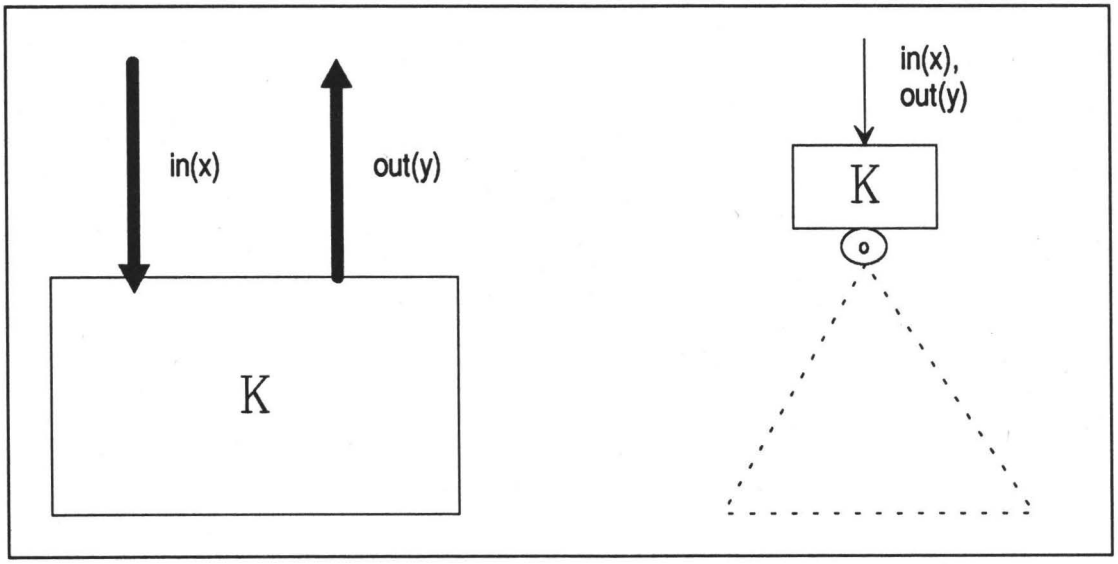


ABB. 2.4.2. VERGLEICH: MODUL UND OPERATIONALER TEILGRAPH

Bsp. einer Parameterübergabe

a) im Graphen:

```
node(..., ..., ..., Konstrukt_o(... edge("K", [subst(a, x), subst(b, y)])))
node("K", Typ, [fp("in", "x"), fp("in", "y")], Konstrukt_o(......))
```

b) in Pascal:

```
procedure K(x, y)
      begin
            ...
      end
      ...
begin (* main *)
      ...
K(a, b);
      ...
end.        (* main *)
```

ABB. 2.4.3. BEISPIEL ZUR PARAMETERÜBERGABE

3. Die Vergleichsverfahren

3.1. Einleitung

In diesem Kapitel werden die einzelnen Vergleichsverfahren und deren Kombination zu einem Gesamtverfahren vorgestellt.

Wir nehmen eine Einteilung in zwei Gruppen, den **semantischen** und den **syntaktischen Verfahren**, vor. Dementsprechend wurde die Gliederung dieses Kapitels vorgenommen. Nach dieser Einleitung werden zunächst die semantischen, dann die syntaktischen und schließlich das Gesamtverfahren besprochen.

Jedes Einzel- wie auch das Gesamtverfahren fungiert auf zwei Teilgraphen. Der erste ist ein im Entstehen befindlicher Entwurf, der zweite eine bereits existierende Lösung, ein Kandidat zur Wiederverwendung im ersten. Diese bereits existierende Lösung wurde vom Wiederverwendungswerkzeug, in das das Gesamtvergleichsverfahren eingebettet ist, ausgewählt. Es bedient sich hierzu des **Weltgraphen**, ein Graph, der alle bisherigen Projekte als Teilgraphen in sich vereinigt.

Lösung und Kandidat werden von Verfahren zu Verfahren verschiedenartig untersucht, um zu einer Ähnlichkeitsaussage zu gelangen. Dabei verfolgt jedes Einzelverfahren eine andere **Strategie** bzw. **Sichtweise**, nach der die Ähnlichkeit beurteilt wird. Durch eine geschickte **Kombination** der Einzelverfahren will man so ein möglichst breites Spektrum der Aspekte, die den Ähnlichkeitscharakter ausmachen, abdecken, um in der Gesamtbewertung eine hohe Aussagekraft zu erzielen.

Die Qualität der Ähnlichkeitsaussage hängt von der tatsächlichen Eignung des untersuchten, schon bestehenden Teilgraphen für seine Wiederverwendung innerhalb des im Entwurf befindlichen Teilgraphen ab. Sie wird mit Hilfe eines **Ähnlichkeitskoeffizienten** getroffen. Dieser ist eine reele Zahl zwischen 0 und 1 einschließlich, wobei Zahlen nahe 0 eine kleine, solche nahe 1 eine große Übereinstimmung andeuten sollen. Es sind in erster Linie weniger die absoluten Werte als vielmehr die Relationen zwischen zwei Koeffizienten aussagekräftig. Ein **zuverlässiges** Verfahren liefert im Vergleich mit dem entstehenden Teilgraphen für einen, der sich weniger für eine Wiederverwendung eignet auch einen geringeren Koeffizienten, als für einen, der sich besser eignet.

Auf diese Weise kann eine **Rangfolge** zwischen den Kandidaten aufgestellt werden, mit deren Hilfe die Entscheidung über Präsentation und Reihenfolge getroffen werden kann. Um die Anzahl der Vorschläge in einem überschaubaren Rahmen zu halten, werden nur solche Kandidaten in die Liste der Vorschläge übernommen, die einen Ähnlichkeitskoeffizienten erzielt haben, der über einer gewissen **Schranke** liegt.

Das **Gesamtverfahren** arbeitet dann zufriedenstellend, wenn es die Zahl der weniger geeigneten Kandidaten in der Vorschlagsliste klein hält und gleichzeitig die Zahl der gut geeigneten Kandidaten, die nicht in die Liste übernommen werden, in vertretbarem Rahmen hält. Da die Vorschlagsliste zwecks endgültiger Wahl schließlich vom Anwender bearbeitet wird, wird die vornehmliche Strategie die sein, eher zu viele (d.h. auch

weniger geeignete) Kandidaten in die Liste mit aufzunehmen, als Gefahr zu laufen, geeignete überhaupt nicht vorzuschlagen.

Die Vergleichsverfahren, die aufgrund der Natur ihrer Sichtweise nur wage Aussagen treffen können, sind daher so gestaltet, daß sie eher zu zu hohen als zu niedrigen Koeffizienten neigen.

Mit Hilfe dieser Überlegungen läßt sich eine grundsätzliche Methodik, wie die Einzelverfahren zu **kombinieren** sind, herausarbeiten, und zwar dahingehend, daß die Einzelverfahren aufgrund ihrer Zuverlässigkeit bzw. danach, wie erfolgversprechend sie sind, geordnet werden. D.h., Verfahren, die in der Mehrheit der Fälle einen Ähnlichkeitskoeffizienten liefern, der eine gute Aussage über die Eignung zur Wiederverwendung macht, erhalten eine hohe, solche, die nur wage Aussagen treffen, eine niedere Anwendungspriorität. Konkret gesprochen erhalten die semantischen Verfahren die höhere Priorität und zwar in der Reihenfolge ihrer Vorstellung in dieser Ausarbeitung, die syntaktischen die niedrigere.

Stehen nun zwei Teilgraphen zum **Vergleich** an, so wird zunächst das Verfahren mit der höchsten Priorität angesetzt. Ergibt sich ein hoher Koeffizient (der Vergleich hatte Erfolg), so kann mit großer Wahrscheinlichkeit eine gute Eignung zur Wiederverwendung angenommen und der Kandidat in die Vorschlagsliste übernommen werden. Erhält man einen niedrigen Koeffizienten (der Vergleich scheiterte), so ist nur schwer eine Aussage zu machen. Zumindest ist die Eignung zur Wiederverwendung nicht völlig auszuschließen, da die Sichtweise des entsprechenden Verfahrens eventuell die tatsächlich bestehende Ähnlichkeit einfach nicht erfaßt hat. Deshalb wird nun das nächste Verfahren, in der Hoffnung auf einen besseren Wert, angewendet. Je weiter man so die Prioritätenleiter der angewendeten Verfahren hinabsteigen muß, desto unsicherer werden die Ähnlichkeitsaussagen und es kann vorkommen, daß unnütze Vorschläge gemacht werden. Dieser Nachteil ist akzeptabel, denn durch diese Vorgehensweise wird die Gefahr, geeignete Kandidaten zu verlieren, stark eingeschränkt.

Eine Alternative aus den Partialkoeffizienten der Einzelverfahren zu einem Gesamtkoeffizienten zu kommen ist, grundsätzlich immer alle Verfahren anzuwenden und die Ergebnisse geschickt (im einfachsten Fall etwa durch gewichtete Mittelwertbildung) zu einem zu verrechnen. Diese Möglichkeit wurde im Verlauf der Arbeit aber im voraus ausgeschlossen, da sie einen nicht vertretbaren Zeitaufwand mit sich bringt. Tatsächlich verhält es sich nämlich so, daß die Verfahren, die in der Prioritätenliste weiter unten stehen auch die höhere Zeitkomplexität haben. Sie sollten also deshalb nur dann angewendet werden, wenn die vorhergehenden, weniger kostenintensiveren, scheiterten.

Wie man nun genau sinnvolle Kriterien formulieren kann, die darüber entscheiden, ob ein Einzelverfahren scheiterte oder nicht, wird näher bei der Beschreibung des Gesamtverfahrens erläutert.

Kommen wir nun zur Besprechung der Einzelverfahren selbst. Die semantischen Verfahren versuchen eine Ähnlichkeitsaussage aufgrund der Gegenüberstellung der Inhalte der zu untersuchenden Teilgraphen zu machen. Konkret werden Knotenidentifikationen und informelle Daten unter verschiedenen Aspekten (z.B. ob auf beiden Seiten auftretende Worte synonym sind) verglichen. Man erhält i.a. eine sehr zuverlässige Ähnlichkeitsaussage, die kostengünstig berechenbar ist. Schlug die Gesamtheit der se-

mantischen Verfahren fehl, wird mit Hilfe der syntaktischen Verfahren versucht, eine Ähnlichkeit in der syntaktischen Struktur der Graphen zu finden. Da die Struktur der Graphen stark mit der durch sie modellierten Problemlösung zusammenhängt, wird also nach ähnlichen Problemlösungen (im einfachen Fall "Programmen") gesucht. Dies ist verständlicherweise sehr aufwendig und unsicher, da für einen zuverlässigen Vergleich die genaue Kenntnis und das Verständnis der Vorgehensweise in den Problemlösungen nötig wäre. So werden in den syntaktischen Verfahren oft unüberprüfte Annahmen zur syntaktischen Struktur gemacht, sie haben dann den Charakter von Heuristiken.

3.2. Die semantischen Vergleichsverfahren

3.2.1. Einleitung

Im ersten Teil des Gesamtverfahrens wird mittels der semantischen Vergleichsverfahren versucht, eine Ähnlichkeit herzuleiten. Dazu werden die **Inhalte** der zu untersuchenden Teilgraphen gegenübergestellt. Diese Inhalte sind sämtliche textuelle Informationen, die sich aus den Teilgraphen extrahieren lassen, im wesentlichen die Knotenidentifikationen (Name, Typ etc.) und informelle bzw. umgebungsspezifische Daten (z.B. Eigenschaftstabellen).

Vier Einzelverfahren werden vorgestellt und im Gesamtverfahren auch in dieser Reihenfolge angewendet:

(1. Textuelle Reduktion)
2. Signaturvergleich
3. Synonymvergleich
4. Teilwortvergleich

Die **textuelle Reduktion** wandelt zunächst die Rohdaten in eine brauchbare Form. Dazu wird unnütze Information gestrichen, die restliche in eine einheitliche Form gebracht (es werden u.a. Stammformen der Wörter gebildet). Im **Signaturvergleich** versucht man (semi-) formale Informationen auszunutzen. Dies können beispielsweise Spezifikationen sein, die einem vorgegebenen Muster entsprechen oder mathematische Formeln, die einem vorgegebenen Kalkül unterliegen. Der **Synonymvergleich** bewertet die Bedeutungsähnlichkeit von Worten mit Hilfe von Wörterbüchern. Konnte durch diese Sichtweise, der Gegenüberstellung auf Wortebene, keine befriedigende Übereinstimmung gefunden werden, untersucht der **Teilwortvergleich** die Eingaben zeichenweise und zählt Gemeinsamkeiten.

Von den vorgestellten Verfahren können sehr zuverlässige Ergebnisse erwartet werden. Ihre Zeitkomplexität ist vergleichsweise niedrig, die Algorithmen einfach. Trotzdem können durch sie nicht alle Sichtweisen des Ähnlichkeitscharakters abgedeckt werden.

Problemlösungen können ähnlich sein, obwohl sie verschiedene textuelle Information besitzen. Bezeichner beispielsweise können in Programmen frei gewählt werden.

Aus diesem Grund muß zusätzlich die **syntaktische Struktur** der Problemlösungen untersucht werden. Genau dies tun die syntaktischen Vergleichsverfahren. Wir werden uns mit ihnen im nächsten Teil beschäftigen.

Die im folgenden vorgestellten Algorithmen erwarten die schon extrahierten Inhalte eines Knotenpaares. Es ist die Aufgabe des Gesamtverfahrens, diese Eingangsdaten aus beiden Teilgraphen zu gewinnen und weiterzureichen.

3.2.2. Textuelle Reduktion

Die textuelle Reduktion ist kein Vergleichsverfahren im eigentlichen Sinne, sondern dient als vorbereitende Maßnahme für die nachfolgenden semantischen Verfahren. Weil ihre Ausgabe von den semantischen Verfahren genutzt wird, wurde ihre Beschreibung mit in dieses Kapitel aufgenommen.

Der Algorithmus erhält als Eingabe einen Text und führt ihn in eine Normalform über. Dieser Text stammt letztendlich aus den Inhalten eines Knotens (z.B. der Knotenname, Knotentyp etc.) der zu vergleichenden Teilgraphen. Die normierte Form soll die nachfolgenden Vergleiche erleichtern bzw. macht sie teilweise erst möglich. Um demnach den gesamten semantischen Vergleich durchzuführen, muß zunächst die textuelle Reduktion für die Inhalte beider Teilgraphen durchgeführt werden und dann die so erhaltenen Normalformen in die einzelnen Verfahren eingegeben werden.

Genauer gesagt werden textuelle Darstellungen mit dem gleichen Gehalt in eine einheitliche Ersatzdarstellung überführt und darüberhinaus Symbole, die keine vergleichbare semantische Information tragen, als ignorierbar markiert.

Der Algorithmus spaltet zunächst die Eingabe in einzelne Symbole, die er anhand von Trennzeichen erkennt, z.B. können Wörter durch Leerzeichen getrennt sein. Jedes so erhaltene Symbol wird nun der Reihe nach folgenden Operationen unterzogen:

1. Überführung von Groß- in Kleinschreibung:

   Das Wort wird Zeichen für Zeichen untersucht und sämtliche auftretenden Großbuchstaben werden in Kleinbuchstaben gewandelt, Kleinbuchstaben selbst bleiben unverändert. Damit werden Texte, die mit verschiedenen Konventionen bzgl. Groß- / Kleinschreibung erstellt wurden, vergleichbar gemacht.

2. Umwandeln von nationalen Umlauten in eine Ersatzdarstellung:

   Das Wort wird Zeichen für Zeichen nach Umlauten untersucht. Auftretende Umlaute werden ersetzt. Für die deutsche Sprache bedeutet dies, daß aus "ä" "ae", aus "ö" "oe" usw. wird. "ß" wird zu "ss" reduziert. Ebenso wird bei Fremdsprachen vorgegangen, z.B. wird für Buchstaben mit Akzenten eine entsprechende Ersatzdarstellung gewählt.

3. Expansion von Abkürzungen in ihre vollständige textuelle Darstellung:

   z.B. wird "u.a." zu "unter anderem", "s.o." zu "siehe oben" und "bsp" zu "beispiel" gewandelt. Dadurch wird verhindert, daß ein Vergleich an der unterschiedlichen Anwendung von Abkürzungen in beiden Texten scheitert.

4. Markierung ignorierbarer Symbole:

   Es können in der Eingabe Symbole auftreten, die keine für den Vergleich brauchbare semantische Information tragen. Diese werden markiert oder sogar aus der Eingabe gelöscht. Beispiele sind die Artikel (der, die, das) und Personennamen.

5. Ableiten der unmarkierten Wörter auf ihre Stammformen:

   Dies ist der letzte Schritt in der Normierung, der die Vergleichbarkeit weiter erhöht. Als Stammformen können gewählt werden: Für Substantive der Singular, für Verben der Infinitiv, für Adjektive die Grundform.

Schritt 1 und 2 können vom Zeichensatz bzw. der Sprache abhängig gemacht werden.

Für die letzten drei Schritte verwendet man günstigerweise je ein Wörterbuch zur Darstellung der nötigen Informationen. Dabei sind bei Schritt 4 eine Spalte im Wörterbuch vorzusehen, es handelt sich also um nichts anderes als einer Liste der ignorierbaren Worte. Schritt 3 und 5 benötigen zwei Spalten in ihrem Wörterbuch, in der linken Spalte stehen die Ausgangsformen, in der rechten die Ersatzdarstellungen bzw. die Stämme. Ausschnitte aus diesen Wörterbüchern könnten so aussehen:

```
.
.
.
der
die
das
.
.
.
```

ABB. 3.2.2.1. MÖGL. AUSSCHNITT AUS DEM WÖRTERBUCH IGNORIERB. SYMBOLE

```
.
.
.
u.a.        unter anderem
s.o.        siehe oben
bsp         beispiel
.
.
.
```

ABB. 3.2.2.2. MÖGLICHER AUSSCHNITT AUS DEM WÖRTERBUCH FÜR ABKÜRZUNGEN

```
.
.
.
werte       wert
ging        gehen
laenger     lang
.
.
.
```

ABB. 3.2.2.3. MÖGLICHER AUSSCHNITT AUS DEM STAMMWÖRTERBUCH

Um die Schritte 3 bis 5 sehr effizient zu machen, wird eine der bekannten Organisationsformen für Wörterbücher gewählt, die sich bei der **Suche** nach Einträgen kostengünstig verhalten (Binäre Bäume, Hashing o.ä.).

Abschließend ein Beispiel, das die Arbeitsweise der textuellen Reduktion illustrieren soll:

Eingegebener Text:

```
Die Änderungen sind w.o. definiert
```

Reduzierter Text:

```
aenderung sein wie oben definieren
```

Erfolgte Ersetzungen / Streichungen:

Schritt 1: "D" zu "d" (in "Die"), "Ä" zu "ä" (in "Änderungen")

Schritt 2: "ä" zu "ae" (in "änderungen")

Schritt 3: "w.o." zu "wie oben"

Schritt 4: "die" wurde gestrichen

Schritt 5: "aenderungen" zu "aenderung", "sind" zu "sein", "definiert" zu "definieren"

ABB. 3.2.2.4. BEISPIEL EINES PROBELAUF S ZUR TEXTUELLEN REDUKTION

Wir kommen nun zu den eigentlichen Verfahren, die die eben beschriebenen Berechnungen benötigen.

3.2.3. Signaturvergleich

Nachdem die textuelle Reduktion die Inhalte der zu vergleichenden Graphen in eine Normalform gebracht hat, ist ein erster Schritt, diese Inhalte nach (semi-) formalen Informationen, d.h. Signaturen, zu untersuchen. Denn ein Vergleich solcher Informationen kann im Gegensatz zur Verwendung völlig informeller Daten immer das Wissen über die zugrunde liegenden Formalismen bzw. Schemata benutzen - ein sicheres Ergebnis, d.h. eine hohe Erfolgsquote von "richtigen" Vergleichsergebnissen kann erwartet werden. Solche Informationen können z.B. mathematische Formeln (die einer gegebenen Spezifikation entsprechen) oder Eigenschaftstabellen sein.

Liegen in beiden Graphen Signaturen vor, so wird aus diesem Grund ein Vergleich der beiden vor allen anderen Verfahren stehen und die eventuell für diese Signaturen existierenden Ableitungs- und Vergleichsalgorithmen genutzt.

Liegt in einem der beiden Graphen keine oder liegen zwei Signaturen aus verschiedenen Klassen vor, so kann kein Signaturvergleich stattfinden und das Verfahren gibt den Partialkoeffizienten 0 zurück. Das Gesamtverfahren wird dann einen weiteren Versuch mit einem anderen Vergleichsverfahren starten.

Der Softwareentwurfsgraph sieht in seiner Definition Signaturen nicht vor, sie müssen von darüberstehenden Werkzeugen eingeführt werden. Man kann sie als informelle Teilgraphen gewisser Struktur anlegen (deren Wurzel durch einen vereinbarten Typ, etwa "Signatur 1" gekennzeichnet ist) oder auch in die Knotenidentifikationen (Name, Typ etc.) selbst einbinden.

Nachfolgend werden einige Klassen von Signaturen kurz angesprochen und danach für eine ausgewählte ein Vergleichsverfahren angegeben.

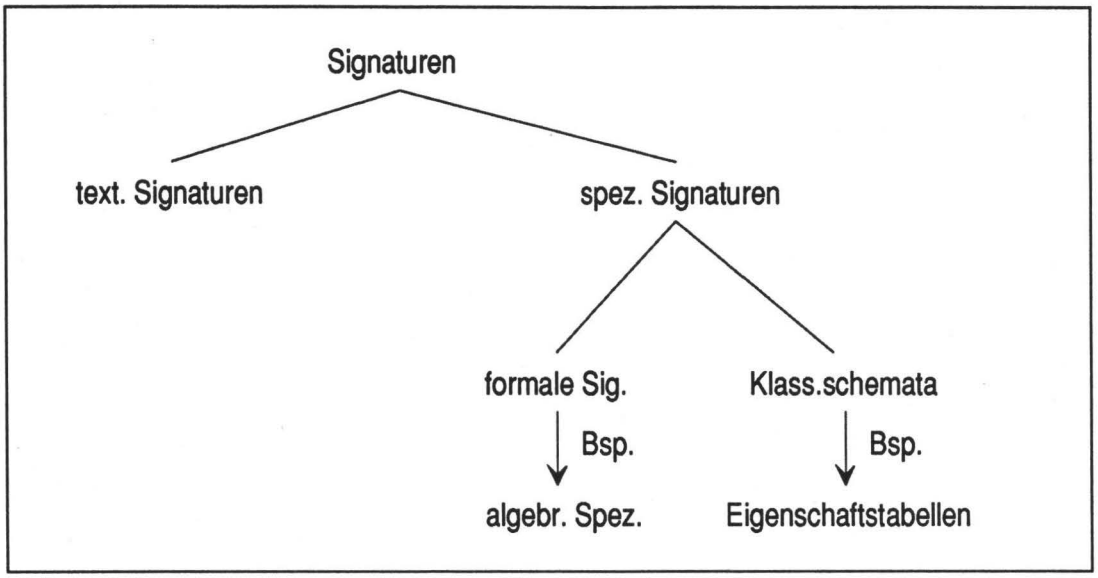


ABB. 3.2.3.1. KLASSIFIKATION VON SIGNATUREN

Grundsätzlich kann man zwischen **textuellen** und **spezifischen** Signaturen unterscheiden.

Textuelle Signaturen sind die allgemeingültigeren, das bedeutet aber auch, daß eine Vergleichsfunktion nur wage Aussagen über die Ähnlichkeit treffen kann, da allgemeingültiger Text erkannt werden muß. Spezifische Signaturen hingegen sind im Anwendungsbereich eingeschränkt, dementsprechend kann man jedoch weitergehende Annahmen (z.B. struktureller Art) über die Informationen in den Signaturen vornehmen und somit auch sichere und überschaubare Verfahren entwickeln.

Spezifische Signaturen schlüsseln sich weiterhin in **formale Signaturen** und **Klassifikationsschemata** auf.

Formale Signaturen (z.B. algebraische Spezifikationen: math. Kalküle, Prädikatenlogik etc.) sind bzgl. ihrer Aussagekraft durch formale Definitionen sehr leistungsfähig in eingeschränkten Umgebungen (z.B. math. Problemlösungen) und lassen den Entwurf guter Vergleichsalgorithmen zu, in den Bereichen, in denen eine formale Spezifikation

jedoch nicht in vertretbarem Rahmen erstellt werden kann bzw. nicht wünschenswert ist (z.B. Betriebsdatenverarbeitung), aber weniger zu gebrauchen.

Klassifikationsschemata sind zwar ebenfalls nicht allgemeingültiger Natur, bieten aber durch Schematisierung die Möglichkeit, relativ einfache und genaue Vergleichsverfahren (die sich eben an dem Schema orientieren) aufzustellen.

In dem vorliegenden Bericht wurde zunächst nur eine einfache Signatur näher untersucht, um erst einmal grundsätzliche Erfahrungen machen zu können. Wir wählten ein Klassifikationsschema, die Eigenschaftstabellen.

Der Aufbau der Eigenschaftstabellen ist der folgende:

Jeder Eigenschaft ist eine Zeile in der Tabelle zugeordnet, d.h. man benennt die Zeilen mit den Eigenschaften. Die Felder jeder Zeile sind mit den sogenannten **Ausprägungen**, das sind die Einzelwerte der jeweiligen Eigenschaft, gefüllt. Die Anzahl der ausgefüllten Felder kann somit von Zeile zu Zeile bzw. Tabelle zu Tabelle variieren. Ein Beispiel für eine Tabelle, die Eigenschaften eines Programmes widerspiegelt, soll dies verdeutlichen :

läuft mit kB RAM	128	640	1000
Betriebssysteme	MS-DOS	OS/2	
Autoren	Müller	Maier	Schulze

ABB. 3.2.3.2. BEISPIEL EINER EIGENSCHAFTSTABELLE

Wie Eigenschaftstabellen sinnvoll im Softwareentwurfsgraphen darzustellen sind, soll hier nicht näher diskutiert werden. Klar ist zumindest, daß die Knoten des Teilgraphens, der die Eigenschaftstabelle darstellt, informell sind. Man könnte ihn als Sequenz von Sequenzen modellieren.

Unabhängig von der Darstellungsweise muß die Möglichkeit von nil-Einträgen und deren Behandlung besprochen werden. Nil-Einträge an der Stelle einer Eigenschaft bedeuten eine vorgesehene, aber noch nicht näher bezeichnete Eigenschaft, nil-Einträge an Stelle von Werten dementsprechend noch nicht näher bezeichnete Werte. In beiden Fällen gehen wir davon aus, daß die nil-Einträge durch beliebige Dinge ersetzt werden können, d.h. ein Vergleich einer Eigenschaft mit einem nil-Eintrag fällt immer positiv aus und entsprechend ein Vergleich zwischen einem Wert und einem nil-Eintrag.

Der Vergleich zweier Eigenschaftstabellen setzt die Untersuchung in den Übereinstimmungen der Eigenschaften und weiterhin für jede gemeinsame Eigenschaft die der Übereinstimmungen in den Ausprägungen voraus.

Offensichtlich bieten sich für die genauere Vorgehensweise viele Möglichkeiten an. Um einigermaßen flexibel zu bleiben, wird folgende Methode vorgeschlagen:

Wir erheben für die Tabellen einige charakteristische Werte, die über Funktionen verrechnet werden. Die Freiheit in der Gestaltung dieser Funktionen läßt die genaue Bewertung der Übereinstimmungen zunächst offen. Konkretisierungen ergeben sich durch Versuche in der Praxis.

a) Für jede **gemeinsame** Eigenschaft in den Tabellen wird berechnet:

- Anzahl der Werte in Tabelle 1 für diese Eigenschaft
- Anzahl der Werte in Tabelle 2 für diese Eigenschaft
- Anzahl der gemeisamen Werte für diese Eigenschaft

b) Für die beiden Tabellen wird berechnet:

- Anzahl der Eigenschaften (Zeilen) in Tabelle 1
- Anzahl der Eigenschaften (Zeilen) in Tabelle 2
- Anzahl der gemeinsamen Eigenschaften der Tabellen

Die drei Werte aus Gruppe a) werden über eine Bewertungsfunktion f_B zu einer Bewertung der gemeinsamen Eigenschaft verrechnet. f_B soll zusätzlich normiert sein, d.h. ihr Wert soll wie alle Ähnlichkeitskoeffizienten zwischen 0 und 1 liegen. Alle Bewertungen der gemeinsamen Eigenschaften werden in der sog. Bewertungsliste zur weiteren Verarbeitung gesammelt.

Diese Bewertungsliste wird zusammen mit den Werten aus Gruppe b) mittels der Koeffizientenfunktion f_K verrechnet und das Ergebnis normiert. Es ist das Endergebnis, also der Ähnlichkeitskoeffizient des Signaturvergleichs.

Ein Beispiel soll den Algorithmus verdeutlichen:

Wir wählen:

$$f_B(\#Werte\_1, \#Werte\_2, \#gem\_Werte) := \frac{\#gem\_Werte}{\max(\#Werte\_1, \#Werte\_2)}$$

$$f_K(\#Zeilen\_1, \#Zeilen\_2, \#gem\_Eig, Bew\text{-}Liste) := \frac{\sum Bew\text{-}Liste}{\#gem\_Eig}$$

1. Tabelle:

läuft mit kB RAM	128	640	1000
Betriebssysteme	MS-DOS	OS/2	
Autoren	Müller	Maier	Schulze

2. Tabelle:

läuft mit kB RAM	640	1000	
nil	**nil**		
Autoren	Müller		
benötigt Peripherie	Platte	Drucker	Keyboard

#Zeilen_1 = 3, #Zeilen_2 = 4,

#gem_Eig = 3

gem. Eigenschaft "läuft mit kB RAM":

#Werte_1 = 3,

#Werte_2 = 2,

#gem_Werte = 2, also $f_B(3, 2, 2) = \frac{2}{\max(2,3)} = \frac{2}{3}$

gem. Eigenschaft "Autoren":

#Werte_1 = 3

#Werte_2 = 1

#gem_Werte = 1, also $f_B(3, 1, 1) = \frac{1}{\max(1,3)} = \frac{1}{3}$

gem. Eigenschaft "Betriebssysteme" (mit **nil**):

#Werte_1 = 2

#Werte_2 = 1

#gem_Werte = 1, also $f_B(2, 1, 1) = \frac{1}{\max(1,2)} = \frac{1}{2}$

Bew-Liste = $[\frac{2}{3}, \frac{1}{3}, \frac{1}{2}]$

Ähnlichkeitskoeffizient = $f_K(3, 3, 2, [\frac{2}{3}, \frac{1}{3}, \frac{1}{2}]) = \frac{\frac{2}{3}+\frac{1}{3}+\frac{1}{2}}{3} = 0.5$

Bei der vorgestellten Methode werden alle gemeinsamen Eigenschaften gleich bewertet. Eine Erweiterung des Verfahrens könnte eine gewichtete Bewertung vornehmen. Hierzu müßte die Summe aus dem Beispiel durch einen gewichteten Mittelwert ersetzt werden. Man könnte z.B. Zeilen mit starken Unterschieden in der Länge weniger stark gewichten als solche mit ähnlich großer Länge. Die hier vorgestellte Vorgehensweise ist einfach gehalten worden, um grundsätzliche Möglichkeiten des Vorgehens aufzuzeigen.

3.2.4. Synonymvergleich

Lagen in den Teilgraphen keine Signaturen vor, versucht das Gesamtverfahren nun die textuellen Informationen zu vergleichen. Dabei wäre eine reine Überprüfung auf zeichenweise Übereinstimmung zu grob. Symbole können auch dann als gleich gewertet werden, wenn sie die gleiche Bedeutung haben, also synonym sind. Insofern hätte man diesen Schritt ebenfalls in die textuelle Reduktion integrieren können, indem man Symbole, die **exakt** die gleiche Bedeutung haben auf ein ausgewähltes reduziert hätte. Doch wir wollen hier weiter gehen. Der Synonymvergleich soll auch Paare von Worten berücksichtigen, die **ähnlich** in ihrer Bedeutung sind. Die Ähnlichkeit wird durch einen Koeffizienten ausgedrückt. Ordnet man so den Symbolen der beiden normierten Eingaben paarweise einen Ähnlichkeitskoeffizienten zu, so braucht man diese nur noch zu einem Gesamtkoeffizienten zu verrechnen (z.B. durch Mittelwertbildung), der dann ein Maß für die Übereinstimmung der Ausgangstexte darstellt.

Es fragt sich also noch, wie die Bestimmung der einzelnen Koeffizienten bei zwei eingegebenen Worten zu bewerkstelligen ist.

Die Idee liegt nahe, ein Synonymwörterbuch, wie es aus einigen Textverarbeitungssystemen bekannt ist, zu verwenden. Solch ein Wörterbuch hat zwei Spalten: In der linken stehen Worte, in der rechten eine Liste von Worten, die zu diesem synonym sind. Jedem der Worte in der Liste ist ein Ähnlichkeitskoeffizient zugeordnet. Der Algorithmus müßte also lediglich die erste Eingabe in der linken Spalte suchen, die zweite danach in der zugeordneten Liste auffinden und schließlich den Koeffizienten auslesen. Dabei werden zeichenweise gleiche Eingaben als Sonderfälle behandelt und mit einem Koeffizienten von 1 bewertet. Selbstverständlich sind die Eingaben symmetrisch zu betrachten, d.h. schlägt die Suche in der linken Spalte fehl, so werden die beiden Eingaben vertauscht und ein weiterer Versuch angesetzt. Eine insgesamt fehlgeschlagene Suche führt zu einem Koeffizienten von 0.

Ein Beispiel soll diesen ersten Ansatz für den Algorithmus verdeutlichen (Die Koeffizienten im Wörterbuch sind willkürlich gewählt).

Wörterbuch:

```
                .
                .
                .
algorithmus         (verfahren, 0.90), (methode, 0.80), (programm, 0.50)
verfahren           (methode, 0.85), (programm 0.50)
methode             (programm 0.60)
                .
                .
                .
```

Beispielläufe:

1. Suche: **Eingabe:** ("programm", "verfahren")

"programm" ist nicht in linker Spalte, also tausche Eingaben

"verfahren" in linker Spalte gefunden, "programm" in Liste gefunden, also:

Ausgabe: Koeffizient = 0.5

2. Suche: **Eingabe:** ("algorithmus", "datenstruktur")

"algorithmus" in linker Spalte gefunden, "datenstruktur" jedoch nicht in Liste vorhanden, also:

Ausgabe: Koeffizient = 0

ABB. 3.2.4.1. BEISPIEL ZUM ERSTEN ANSATZ DES SYNONYMVERGLEICHS

Dieser erste Ansatz würde zwar zu guten Ergebnissen, aber auch zu sehr großen Wörterbüchern und einer komplizierten Verwaltung führen. Dies liegt daran, daß die Listen in den rechten Spalten sehr groß würden, da es wohl zu einem Wort eine Unmenge von Synonymen gibt. Worte würden in dem Wörterbuch vielfach auftreten, da Worte aus einer Liste ebenfalls eine gewisse Synonymität untereinander haben und deshalb selbst wiederum in der linken Spalte auftreten müssen. Die **Erweiterung** und **Modifikation** wird dadurch sehr aufwendig.

Wenn z.B im obigen Wörterbuch der Koeffizient des Paares ("algorithmus", "verfahren") von 0.90 auf 0.95 geändert werden soll, wie ändert sich dann der von ("verfahren", "methode") bzw. ("methode", "programm") ?

Eine Möglichkeit, diese Probleme zu umgehen ist, aus der Synonymität zweier Paare auf die einer dritten transitiv zu schließen und führt zu unserem zweiten Ansatz. Und zwar in etwa so: Sei A synonym zu B mit Koeffizient K_1 und B synonym zu C mit Koeffizient K_2, so sei A synonym zu C mit Koeffizient $f(K_1, K_2)$. f sei dabei eine geeignete Funktion zur Verrechnung von K_1 und K_2. Das Wörterbuch müßte nun in der zweiten Spalte nur ein Wort und in einer dritten den dazugehörigen Koeffizienten aufnehmen. Die Bestimmung der Ähnlichkeit zweier Eingaben läuft auf eine sukzessive Ableitung der Wörter und ständiger Anwendung von f heraus. Bei der Suche nach dieser Ableitung wird von der ersten Eingabe ausgegangen und die Worteinträge symmetrisch behandelt, d.h. in der ersten und zweiten Spalte gesucht.

Auch hier ein Beispiel:

Wörterbuch:

```
                    .
                    .
                    .
algorithmus         verfahren       0.90
verfahren           methode         0.85
methode             programm        0.60
                    .
                    .
                    .
```

Sei $f(K_1, K_2) = K_1 * K_2$.

Beispiellauf:

Eingabe: ("algorithmus", "programm")

Ableitung: "algorithmus", "verfahren", "methode", "programm"

Ausgabe: Koeffizient = 0.90*0.85*0.60 = 0.459

ABB. 3.2.4.2 BEISPIEL ZUM ZWEITEN ANSATZ DES SYNONYMVERGLEICHS

Am Beispiel erkennt man jedoch schon: Es dürfte schwer sein, eine geeignete Funktion f zu finden, die auch bei langen Ableitungen ein zufriedenstellendes Ergebnis liefert.

Die dritte und letzte Möglichkeit der Sache einigermaßen Herr zu werden sei im folgenden dargestellt. Es ist eine Mischform der beiden ersten Ansätze.

Man geht zunächst vom zweiten Ansatz aus. Für einige Paare von Wörtern, die ein unzufriedenstellendes Ergebnis bringen, führt man zusätzliche Einträge ein. Auf diese Weise treten in der linken Spalte Worte mehrmals auf. Dies bedeutet wiederum, daß bei der Suche mehrere verschiedene Ableitungswege möglich sind. In einem solchen Fall wird ein Ableitungsweg nach einem vorgegebenen Kriterium ausgewählt, z.B. die kürzeste Ableitung bzw. die mit dem größten (kleinsten) Gesamtkoeffizienten oder eine geschickte Kombination solcher Kriterien. Die Wahl des kürzesten Ableitungswegs als Auswahlkriterium führt dazu, daß bevorzugt Wege betrachtet werden, die von den zusätzlichen Einträgen herrühren und so die gewünschten Koeffizienten bei der Anwendung von f berücksichtigt werden. Eine manuelle Korrektur des Koeffizienten ist somit durch diese Erweiterung im Gegensatz zum zweiten Ansatz möglich.

Ein Beispiel hierzu:

Wörterbuch:

```
                    .
                    .
                    .
algorithmus                 verfahren       0.90
algorithmus                 programm        0.50
verfahren                   methode         0.85
methode                     programm        0.60
                    .
                    .
                    .
```

Sei $f(K_1, K_2) = K_1 * K_2$.

Das Auswahlkriterium bei meheren möglichen Ableitungswegen laute folgendermaßen:

Wähle den kürzesten Weg. Existieren mehrere kürzeste Wege, wähle daraus den mit dem höchsten Gesamtkoeffizienten.

Beispiellauf:

Eingabe: ("algorithmus", "programm")

Ableitungen: 1. "algorithmus", "verfahren", "methode", "programm"

2. "algorithmus", "programm"

Die zweite Ableitung ist die kürzere, also wird sie gewählt.

Ausgabe: Koeffizient = 0.50

(Korrrektur um 0.041 zum zweiten Ansatz)

ABB. 3.2.4.3. BEISPIEL ZUM DRITTEN ANSATZ DES SYNONYMVERGLEICHS

Da die Einträge der ersten und zweiten Spalte symmetrisch betrachtet werden, entsteht bei der Suche nach der Ableitung die aus Reduktionssystemen bekannte Problematik der Zykelbildung. Durch Markierung der schon verwendeten Zeilen im Wörterbuch kann man das Problem in Griff bekommen.

Zusammenfassend kann man sagen, daß alle drei vorgestellten Verfahren mit einem vertretbaren Aufwand zu realisieren sind. Für kleine Systeme und zur Erprobung, welche Bedeutung der Synonymvergleich im Gegensatz zu den anderen Vergleichsverfahren überhaupt hat, dürfte zunächst der erste, einfache Ansatz genügen. Aus diesem Grund wurde in der Implementation dieser Algorithmus gewählt. In der jetzigen Entwicklungsphase und zur Illustration ist er vollkommen zufriedenstellend. Stellt sich später jedoch der Synonymvergleich als ein sehr wichtiges Verfahren neben den anderen heraus und soll in der Praxis eingesetzt werden, muß auf jeden Fall der dritte Ansatz mit eventuell noch einigen Erweiterungen gewählt werden. Seine Leistungsfähigkeit ist stark von der Wahl der Mehrfacheinträge, der Koeffizientenverrechnungsfunktion f und dem Auswahlkriterium für Ableitungswege abhängig. Hierzu sind zu einem späteren Zeitpunkt genauere praktische Untersuchungen von Nöten.

3.2.5. Teilwortvergleich

Erkennt der Synonymvergleich nur eine geringe Ähnlichkeit zwischen den Eingaben, so wird der Teilwortvergleich durchgeführt. Er deckt die wichtigsten Fälle ab, in denen der Synonymvergleich aufgrund seiner Vorgehensweise einen zu niedrigen Koeffizienten liefert. Solche Fälle sind unterschiedliche Schreibweisen von Wörtern, Schreibfehler und Kurzschreibweisen.

Gerade die Benutzung von Kürzeln für längere Bezeichner o.ä. ist in der Informatik weit verbreitet. Es gibt jedoch kein zwingendes Reglement, wie diese Kürzel zu bilden sind. Manche Programmierer wählen z.B. für "Current Reference" das Kürzel "Curnt_refer", andere vielleicht "CurRef" usw. Aufgrund der großen Vielzahl solcher Symbole ist es nicht sinnvoll, solche Begriffe im Wörterbuch für Abkürzungen bei der textuellen Reduktion vorzusehen, noch einen Eintrag im Synonymwörterbuch einzufügen.

Ebenso ist es möglich, daß in den Eingaben Schreibfehler oder verschiedene Schreibweisen des gleichen Begriffes vorliegen (z.B. "Address" und "Adress") die trotz unerkannter Synonymität als ähnlich gewertet werden sollen. In all diesen Fällen greift der Teilwortvergleich: Er untersucht die Eingaben auf Zeichenebene und weist ihnen aufgrund der Anzahl von Übereinstimmungen und Vergleich der Anordnug der Zeichen einen Koeffizienten zu, der recht gut unsere Bedürfnisse befriedigt.

Die Entwicklung von effizienten Algorithmen zum Vergleich von Zeichenketten wird aufgrund der vielfältigen Anwendungsgebiete seit schon geraumer Zeit verfolgt. Dabei untersuchen die meisten Algorithmen die Vorkommen der einen in der anderen Zeichenkette ("string matching", siehe z.B. [KMP-77, Sun-90]), einige jedoch auch in wieweit zwei Zeichenketten eine gewisse Ähnlichkeitsforderung erfüllen ("approximate string matching"). Uns interessiert hier natürlich nur die zweite Klasse von Algorithmen. Das Verfahren zum Teilwortvergleich ist im wesentlichen ein solcher Algorithmus. Er soll im folgenden vorgestellt werden.

Zunächst besprechen wir das ausgewählte Ähnlichkeitskriterium. Es bringt gerade für die eingangs erwähnten Fälle gute Ergebnisse. Und zwar nehmen wir als Maß für die Ähnlichkeit die minimale Anzahl der Editierschritte, die nötig sind, um die erste eingegebene Zeichenkette in die zweite zu überführen. Als Editierschritt wird in diesem Zusammenhang das Einfügen, das Löschen und das Überschreiben gezählt. In der Literatur wird von der **Editier-Distanz** (edit-distance) gesprochen.

Angenommen, es läge ein Schreibfehler vor: Um das Wort "halllo" in "hallo" zu überführen ist nur ein Editierschritt notwendig, nämlich das Löschen eines "l".

oder: "bcdefgh" kann in drei Schritten folgendermaßen in "bxdyegh" überführt werden:

1. Überschreiben des "c" mit "x" ergibt "bxdefgh",
2. Einfügen des "y" an der vierten Stelle ergibt "bxdyefgh",
3. Löschen des "f" ergibt "bxdyegh"

ABB. 3.2.5.1. BEISPIEL ZUR BESTIMMUNG DER EDITIER-DISTANZ

Aus der Anzahl der Editierschritte muß nun ein Koeffizient berechnet werden, der unserer Forderung, daß er zwischen 0 und 1 liegen muß, genügt. Hierzu überlegt man sich, daß eine längere Zeichenkette in eine kürzere im schlechtesten Fall durch komplettes Überschreiben und anschließendes Löschen der überzähligen Zeichen bzw. umgekehrt eine kürzere in eine längere durch komplettes Überschreiben und anschließendes Einfügen der fehlenden Zeichen überführt werden kann. In beiden Fällen sind genausoviele Schritte notwendig, wie die längere Zeichenkette Zeichen hat. Also könnte obige Forderung dadurch erfüllt werden, daß die Anzahl der Schritte durch die größere der Längen der beiden Zeichenketten geteilt wird, die Normierung würde lauten:

$$\text{Koeffizient} = 1 - \frac{\text{Anz. Schritte}}{\max(|S_1|, |S_2|)}$$

BERECHNUNG DES KOEFFIZIENTEN AUS DER EDITIER-DISTANZ

$|S_1|$ bzw. $|S_2|$ sollen dabei die Längen der eingegebenen Zeichenketten S_1 und S_2 sein.

Dieser Koeffizient kann natürlich erneut Argument für eine weitere Berechnung sein, wenn ein anderes als das lineare Verhalten erwünscht ist.

Wir haben nun geklärt, wie man mit Hilfe der Editier-Distanz eine sinnvolle Definition für Ähnlichkeit zwischen Zeichenketten trifft und wie man aus der Editier-Distanz einen Koeffizienten, der unseren Forderungen genügt, berechnen kann. Nun kommen wir zum wichtigsten Teil, nämlich der Berechnung der Editier-Distanz selbst.

Hierzu wird ein Algorithmus aus [LV-89], Kapitel 2.1 (einer der erwähnten Artikel, die sich mit approximate string matching beschäftigen) kurz vorgestellt und danach geklärt, wie er für unsere Zwecke zu erweitern ist. Dieser Algorithmus benutzt die Methode des dynamischen Programmierens und leistet den wichtigsten Anteil zur Berechnug der Editier-Distanz. [Sel-80] ist einer der Artikel, in dem das Verfahren zum ersten Mal beschrieben wird. Die Beschreibung berücksichtigt jedoch einen weit allgemeineren Zusammenhang, daher beziehen wir uns auf den auf unseren Zweck spezialisierten und einfacheren Text [LV-89]. Trotz allem muß bei der Diskussion der Erweiterung des Algorithmus´ aus [LV-89] der ursprüngliche Artikel [Sel-80] zu Rate gezogen werden. Es

empfiehlt sich, zum besseren Verständnis und zum Beweis der Korrektheit des Verfahrens, beide Veröffentlichungen zu studieren.

Im folgenden werden die zwei Zeichenketten benannt, die eine soll Text ($t_1...t_n$), die andere Pattern ($a_1...a_m$) heißen. Wir gehen davon aus, daß das Pattern die kleinere von beiden ist, oder aber daß beide gleich groß sind (m <= n).

In Abhängigkeit von $t_1...t_n$ und $a_1...a_m$ wird nun eine Matrix $D_{i,j}$ (0<=i<=m, 0<=j<=n) konstruiert, wobei $d_{i,j}$ die kleinste Editier-Distanz zwischen $a_1...a_j$ und einem zusammenhängenden Teilwort des Textes angibt, das mit t_j endet. Der Aufbau der Matrix geschieht folgendermaßen: Die nullte Zeile wird mit Nullen, die nullte Spalte mit den Zahlen von 0 bis m initialisiert. Alle unausgefüllten Stellen $d_{i,j}$ werden folgendermaßen berechnet: $d_{i,j}$ ergibt sich aus dem Minimum dreier Zahlen: dem linken Nachbarn + 1, dem oberen Nachbarn + 1 und dem diagonal links oben liegenden Nachbarn. Zum diagonal links oben liegenden Nachbarn wird 1 hinzugezählt, wenn a_i von t_j verschieden ist, bevor das Minimum gebildet wird:

```
     for j := 0 to n do d0,j := 0          Initialisierung  der  0. Zeile
     for i := 0 to m do di,0 := i          Initialisierung  der  0. Spalte

for i := 1 to m do
     for j := 1 to n do
          di,j := if ai = tj     then min(di-1,j +1, di,j-1 +1, di-1,j-1)
                                 else min(di-1,j +1, di,j-1 +1, di-1,j-1 +1)
```

ABB. 3.2.5.2. ALGORITHMUS AUS [LV-89]

Dies sei am folgendem Beispiel illustriert:

Sei der Text GGGTCTA, das Pattern GTTC. Dann sieht die Matrix D folgendermaßen aus:

		G	G	G	T	C	T	A
	0	0	0	0	0	0	0	0
G	1	0	0	0	1	1	1	1
T	2	1	1	1	0	1	1	2
T	3	2	2	2	1	1	1	2
C	4	3	3	3	2	1	2	***2***

ABB. 3.2.5.3. BEISPIEL ZUR KONSTRUKTION DER MATRIX D NACH [LV-89]

Durch das Beispiel dürfte die Idee des Algorithmus´ klar sein: Pattern und Text werden beide schrittweise parallel von links nach rechts aufgebaut. Man übernimmt hierzu die beiden aktuellen Zeichen oder nimmt Löschungen bzw. Einfügungen in einer der Ketten vor, je nach dem welche Alternative die kostengünstigste ist. Den Aufbau kann man als Weg in der Matrix von der ersten zur letzten Zeile verfolgen. Der Weg durch die Matrix entspricht den Editierungen in den Zeichenketten:

- Geht man in der Matrix einen Schritt nach rechts, so überspringt man ein Zeichen im Text, dies entspricht einer Löschung im Text bzw. einer Einfügung im Pattern, was in beiden Fällen ein Editierschritt kostet. Deshalb wird zu $d_{i,j-1}$ eins hinzugezählt.
- Geht man in der Matrix einen Schritt nach unten, überspringt man im Pattern ein Zeichen, dies entspricht einer Löschung im Pattern bzw. einer Einfügung im Text, kostet also ebenfalls einen Editierschritt. Deshalb wird zu $d_{i-1,j}$ eins hinzugezählt.
- Geht man in der Matrix einen Schritt nach rechts unten, so kann das einmal ein Kopieren des Zeichens bedeuten, was nicht als Editierschritt gewertet wird. Oder aber es kann ein Überschreiben des Zeichens bedeuten, was einen Editierschritt kostet. Deshalb wird je nach dem, ob $a_i = t_j$ ist, zu $d_{i-1,j-1}$ nichts bzw. eins hinzugezählt.

Uns interessiert für unser Problem, bei dem aus dem Text das Pattern entstehen soll, immer nur das letzte Element der letzen Zeile (im folgenden W genannt), denn nur bei seiner Bildung ist der **gesamte** Text und das **gesamte** Pattern beachtet worden. Doch dieser Wert (im Beispiel 2) ist noch nicht unsere gewünschte Anzahl von Editierschritten zur Umwandlung des gesamten Textes, sondern bezieht sich nur auf einen Teil des Textes.

Wenn man Einfügungen im Text durch Leerräume im Text und Löschungen im Text durch Leerräume im Pattern darstellt, ist der kostengünstigste Weg, um vom Text zum Pattern zu gelangen, der folgende:

		G	G	G	T	C	T	A
		_	_	G	T	_	T	C

Der dazugehörige Weg in der Matrix:

		G	G	G	T	C	T	A
	0	0	0	0	0	0	0	0
G	1	0	0	0	1	1	1	1
T	2	1	1	1	0	1	1	2
T	3	2	2	2	1	1	1	2
C	4	3	3	3	2	1	2	2

ABB. 3.2.5.4. BEISPIEL: MATRIX D MIT DEM WEG DES AUFBAUS

Es sind also drei Löschungen und ein Überschreiben notwendig gewesen, insgesamt sind das vier Editierschritte. Obiger Algorithmus zählt mit W nur den letzten Lösch- und den Überschreibschritt, die beiden zusätzlich benötigten ersten Löschschritte werden nicht berücksichtigt. Wir halten also fest:

Der Algorithmus aus [LV-89] berechnet lediglich die Editier-Distanz zwischen dem Pattern und einem Ausschnitt $t_p...t_n$ des Textes, nicht unbedingt zwischen Pattern und dem gesamten Text. In [LV-89] interessiert man sich nur für die Editier-Distanz, nicht für die dazugehörigen Textausschnitte $t_p...t_n$.

In dem anderen Artikel, [Sel-80] wird in Theorem I, Punkt (iii) angegeben, wie man die Textausschnitte $t_p...t_n$ findet: Durch Rückverfolgen der Wege in der Matrix. Das heißt, ausgehend von $d_{m,n}$, untersucht man sukzessive die Vorgänger (den linken, oberen und diagonalen) des aktuellen $d_{i,j}$, um festzustellen, welcher das Minimum gebildet hatte. So gelangt man schließlich in die erste Zeile an die Stelle $d_{0,p}$.

Es kann mehrere Wege durch die Matrix geben, da die Minimumbildung nicht eindeutig ist. Uns interessiert von allen möglichen Wegen der, der die Anzahl der zusätzlichen Löschungen minimiert. Am obigen Bild erkennt man, daß die Anzahl der zusätzlichen Löschungen um so kleiner ist, je größer der Textausschnitt ist, d.h. je kleiner p ist. Um den richtigen Weg zu finden, geht man beim Zurückverfolgen am besten nach folgender Strategie vor: Trifft man auf Gabelungen, also können zwei oder

drei Vorgänger das Minimum gebildet haben, so wählt man den Weg, der am weitesten nach links führt, wählt also nach folgender Prioritätenliste: links, diagonal, oben. Bei $d_{0,p}$ angelangt erkennt man die Anzahl der zusätzlichen Löschungen: Sie entspricht p, denn $t_1...t_p$ ist genau der Textanfang, der gelöscht werden muß.

Der gesamte Algorithmus zum Teilwortvergleich hat folgende Gestalt:

```
algorithmus Teilwortvergleich

Eingabe:   (S1, S2),        die zu vergleichenden Zeichenketten
Ausgabe:   Koeffizient,     der Ähnlichkeitskoeffizient

1. Wähle die längere Zeichenkette als Text t1...tn, die andere als
   Pattern a1...am:

if |S1| > |S2|        then begin t1...tn := S1; a1...am := S2 end
                      else begin t1...tn := S2; a1...am := S1 end

2. Konstruiere die Matrix D:

     for j := 0 to n do d0,j := 0              Initialisierung
     for i := 0 to m do di,0 := i
for i := 1 to m do
     for j := 1 to n do
          di,j := if ai = ti      then min(di-1,j +1, di,j-1 +1, di-1,j-1)
                                  else min(di-1,j +1, di,j-1 +1, di-1,j-1 +1)
W := dm,n              letzter Wert der letzten Zeile

3. Rückverfolge den Weg:

(i,j) := (m,n)
while i > 0 do
     begin
     Min := di,j
     G := if ai = tj        then 0          Kopieren
                            else 1          Überschreiben
     (i,j) :=          if di,j-1 + 1 =  Min      then (i,j-1)      links
               else  if di-1,j-1 + G = Min       then (i-1,j-1)    diagonal
                                                 else (i-1,j)      oben
     end while
P := j                      Anzahl zusätzlicher Löschungen
Anz_Schritte := W + P       Editier-Distanz

                                Anz Schritte
Koeffizient = 1 -  ------------------------------
                   max(|S1|, |S2|)
```

ABB. 3.2.5.5. DER TEILWORTVERGLEICH

3.3. Die syntaktischen Vergleichsverfahren

3.3.1. Einleitung

Mit den vorgestellten semantischen Verfahren wird ein großes Feld der Möglichkeiten, eine Übereinstimmung in den Inhalten der Teilgraphen zu finden, abgedeckt. Bei deren Scheitern kann man daher davon ausgehen, daß weitere Versuche, mittels dieser Informationen in den Inhalten eine Ähnlichkeit herzuleiten, ebenfalls erfolglos bleiben werden.

Nun werden die syntaktischen Verfahren angewendet. Sie versuchen, aufgrund von Annahmen über die syntaktische **Struktur** eine Aussage über die Ähnlichkeit einer Problemlösung vorzunehmen. Dies kann dann notwendig werden, wenn die textuellen Inhalte der Entwürfe wenig Übereinstimmung zeigen, obwohl eine Wiederverwendung sinnvoll erscheint. Programme beispielsweise können durch die Freiheit, Bezeichner fast beliebig zu definieren, textuell sehr differieren, auch wenn sie ähnliche Problemlösungen beschreiben. (Man betrachte z.B. die Ausdrücke x := a*b und prod := faktor_1 * faktor_2). Die syntaktischen Verfahren sind i.a. wesentlich komplexer und unzuverlässiger als die semantischen. Nur die genaue Kenntnis und das Verständnis der Problemlösungen könnten zuverlässige Ergebnisse liefern, hierzu wäre aber eine "Intelligenz" nötig, die kaum von einem solchen System erwartet werden kann. Man kann dies mit einer Situation vergleichen, die jeder Programmierer kennt: Es ist sehr schwierig, sich in ein fremdes Programm einzuarbeiten und seine Arbeitsweise völlig zu verstehen.

Daher wird in keinem der im folgenden vorgestellten Verfahren versucht, die Semantik der Strukturen zu erkennen, sondern eher aufgrund deren äußerer Ähnlichkeit Rückschlüsse über die Wiederverwendbarkeit zu ziehen. An einigen Stellen werden sogar Daten bei der Beurteilung vernachlässigt bzw. unüberprüfte Annahmen gemacht. Einige erhalten so den Charakter von Heuristiken und diesbezüglich soll bei der Besprechung der Schwerpunkt mehr auf die Glaubwürdigmachung deren Leistungsfähigkeit als auf die genaue Ausformulierung der Algorithmen gelegt werden.

Im Gesamtverfahren wird beim syntaktischen Teil im wesentlichen in drei Schritten vorgegangen.

Zunächst wird als vorbereitende Maßnahme eine **Datenflußanalyse** (nächster Teil) der beiden Teilgraphen vorgenommen. Die anfallenden Ergebnisse werden an verschiedenen Stellen im weiteren Verlauf genutzt.

Daraufhin wird, beginnend von den Wurzelknoten bis hinunter zu den Atomen, Knotenpaar für Knotenpaar einem Vergleich unterzogen. Dabei wird die Struktur der Graphen noch vollständig beibehalten, es werden nur **gleiche** Hierarchieebenen gegenübergestellt. Es müssen Algorithmen für jedes mögliche Paar von Konstrukten bereitgestellt werden. Die wichtigsten werden in dem zweiten bis vierten Unterkapitel besprochen, und zwar:

- Vergleich zweier Sequenzen: **Permutation der Sequenz**
- Vergleich zweier Selektionen: **Permutation der Selektion**
- Vergleich zweier Iterationen: **Permutation der Iteration**

Alle anderen Kombinationen sind vergleichsweise schnell abgehandelt und werden daher innerhalb des Gesamtverfahrens erläutert.

Mit Permutation der Sequenz bzw. Selektion werden Methoden angesprochen, eine Ähnlichkeit der entsprechenden Konstrukte durch Vergleich ihrer Sohnknoten herzuleiten. Da die Reihenfolge der Sohnknoten u.U. variiert werden kann, wird durch gezielte Gegenüberstellung ihrer Permutationen die beste Übereinstimmung gesucht.

Die Permutation der Iteration beachtet die Eigenart des Iterationskonstruktes, die Kontrolle der Berechnung gleichzeitig mehreren Bedingungen (Eingangs-, Ausgangs- und Zählbedingung) zu unterwerfen.

Als dritter und letzter Schrittt wird, falls der zweite erfolgslos war, durch das Verfahren **Abstraktion der Hierarchie** versucht, eine Ähnlichkeit herzuleiten. Dazu werden beide Teilgraphen in eine Normalform überführt, die es erlaubt, anschließend auch **verschiedene** Hierarchieebenen des Entwurfs gegenüberzustellen.

Zum zweiten Schritt sollte noch gesagt werden, daß die Abarbeitung von den Wurzeln zu den Atomen rekursiv jeweils vom Vater zu den Sohnknoten geschieht. Jedes Paar Sohnknoten wird dem vollständigen Gesamtverfahren unterzogen, das in den Beschreibungen mit **VK** bezeichnet wird.

3.3.2. Datenflußanalyse

3.3.2.1. Einleitung

Wie die Normierung bei den semantischen Verfahren, so ist die Datenflußanalyse kein Vergleichsverfahren an sich, sondern ist als vorbereitende Maßnahme für die folgenden Verfahren zu sehen und deshalb wurde ihre Beschreibung in dieses Kapitel integriert. Der Algorithmus hat als Eingabe die Wurzel eines Teilgraphen und liefert zwei Ergebnisse:

- Das vollständige Datenflußmuster für jeden Knoten des angegebenen Teilgraphens, d.h. für jeden Knoten werden alle Ein- und Ausgabeparameter (die Schnittstelle des Teilgraphen unterhalb des Knotens zu dem darüberliegenden Graphen) erkannt und ihr Typ ("in","out" bzw. "inout") bestimmt.
- Die erlaubten Reihenfolgen (Permutationen) der Sohnknoten aller Sequenzen, die in diesem Teilgraph vorkommen, das sind jene Permutationen, die die Semantik der Sequenz erhalten.

Wir interessieren uns hauptsächlich für das zweite Ergebnis, denn es wird später bei einigen Verfahren genutzt und leistet einen sehr wichtigen Beitrag zur Erhöhung der Effizienz. Um nämlich beim Vergleich von Sequenzen bzw. Selektionen untereinander zu guten Ergebnissen zu kommen, müssen für beide Vaterknoten alle **erlaubten** Permutationen der Sohnknoten durchgespielt werden. Würde man das zweite Ergebnis nicht nutzen, so müßten **sämtliche** möglichen Permutationen betrachtet werden und davon gibt es im allgemeinen sehr viel mehr als erlaubte.

Das erste Ergebnis wird zur Zeit lediglich dafür verwendet, um das zweite zu erzeugen.

Der Algorithmus geht von "unten nach oben" vor, d.h. ausgehend von den Atomknoten werden aus den Datenflußmustern der Söhne jeweils die der Väter bestimmt, bis hin zur Wurzel des angegebenen Teilgraphens.

Die Muster der Atomknoten stehen in ihren formalen Parameterlisten, die Berechnung der Muster eines Knotens aus dem seiner Söhne erfolgt nach gewissen Regeln; nachfolgend ein Beispiel:

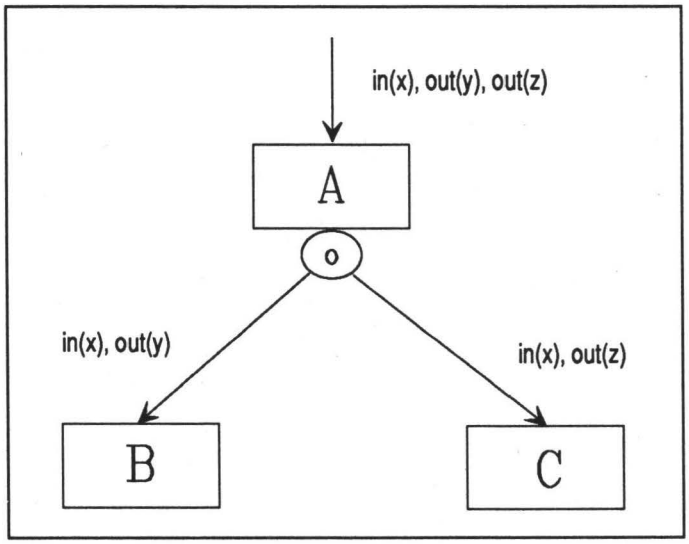


ABB. 3.3.2.1.1 BSP. DER DATENFLUßMUSTER EINER SEQUENZ

Sei obige operationale Sequenz gegeben. Hat man die Muster für Knoten B und C bereits berechnet, so ergibt sich in diesem Beispiel das für A aus der Vereinigung derer von B und C, denn da beide Knoten B und C ausgeführt werden, müssen nach außen hin alle in beiden vorkommenden Parameter mit allen vorkommenden Zugriffsrechten weitergereicht werden. Bei komplizierteren Situationen ist das Muster für A jedoch nicht so einfach zu bestimmen, wie wir in Teil 2 sehen werden.

Die Reihenfolge der Sohnknoten darf dann nicht frei variiert werden, wenn zwei Sohnknoten **abhängig** voneinander sind in dem Sinn, daß sie in einer Verbraucher-Erzeuger-

Relation stehen bzw. sich Konflikte im Geltungsbereich von Namen entwickeln könnten. Der erste Fall sei kurz illustriert:

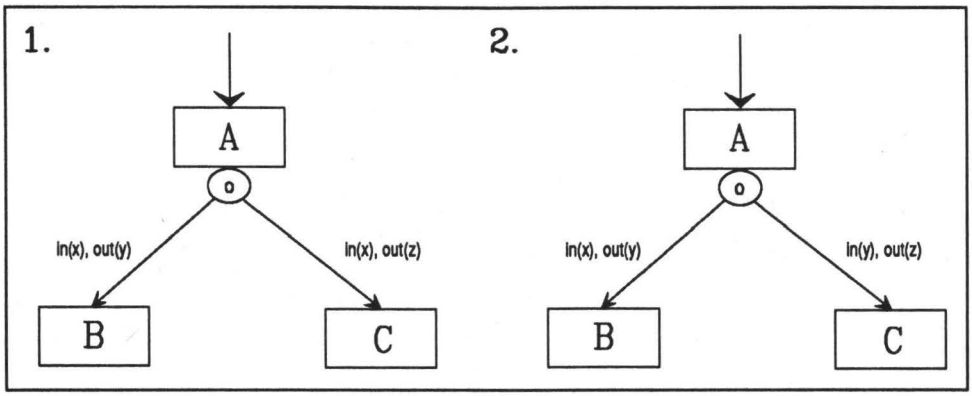


ABB. 3.3.2.1.2. BSP.: SEQUENZEN MIT BZW. OHNE VERTAUSCHBAREN SÖHNEN

Bei Sequenz Nr. 1 wird die Semantik durch Vertauschen von B und C nicht geändert, weil ihre Parameter nicht voneinander abhängig sind. Hingegen ist im 2. Beispiel ein Vertauschen verboten, denn liefe C vor B ab, so erhielte er seine Eingabe für y von außen und nicht mehr von B, wie ursprünglich. Bei Sequenzen kann es also je zwei Sohnknoten geben, deren Reihenfolge egal ist, d.h die Semantik erhält, aber auch solche, bei der sie unbedingt eingehalten werden muß, sonst würde die Semantik geändert.

Um dies zu vermerken, wird jedem Sohnknoten eine Liste der Nummern der Sohnknoten zugeordnet, die nach ihm folgen **müssen**, die, die auch vor ihm stehen können, tauchen in dieser Liste nicht auf. Wir nennen sie Folgelisten.

3.3.2.2. Der Algorithmus

1. Datenformate

Die Datenflußmuster werden in Parameterlisten gehalten, wobei neben den zwei Grundrechten "in" und "out" noch ein weiteres, "def" eingeführt wird. Es kann mit "in" und "out" in Kombination stehen, z.B. ist das Recht "indef" zugelassen. "def" soll die Definition eines Knotennamens innerhalb eines Geltungsbereiches vermerken, damit der Name als Parameter genutzt werden kann.

Die erlaubten Permutationen von Sequenzen werden in den sog. Folgelisten festgehalten: Sei K ein seq. Knoten und K_1,..., K_n seine Sohnknoten. Es wird nun jedem Sohnknoten K_i eine Liste zugeordnet. In diesen Listen stehen die Nummern j>i der Knoten K_j, die nach K_i folgen **müssen**, taucht eine Nummer j>i dort nicht auf, so können K_i und K_j auch vertauscht werden ohne den Erhalt der Semantik zu gefährden.

Wir betrachten ein Beispiel, es sei folgende operationale Sequenz und die dazugehörigen Folgelisten gegeben.

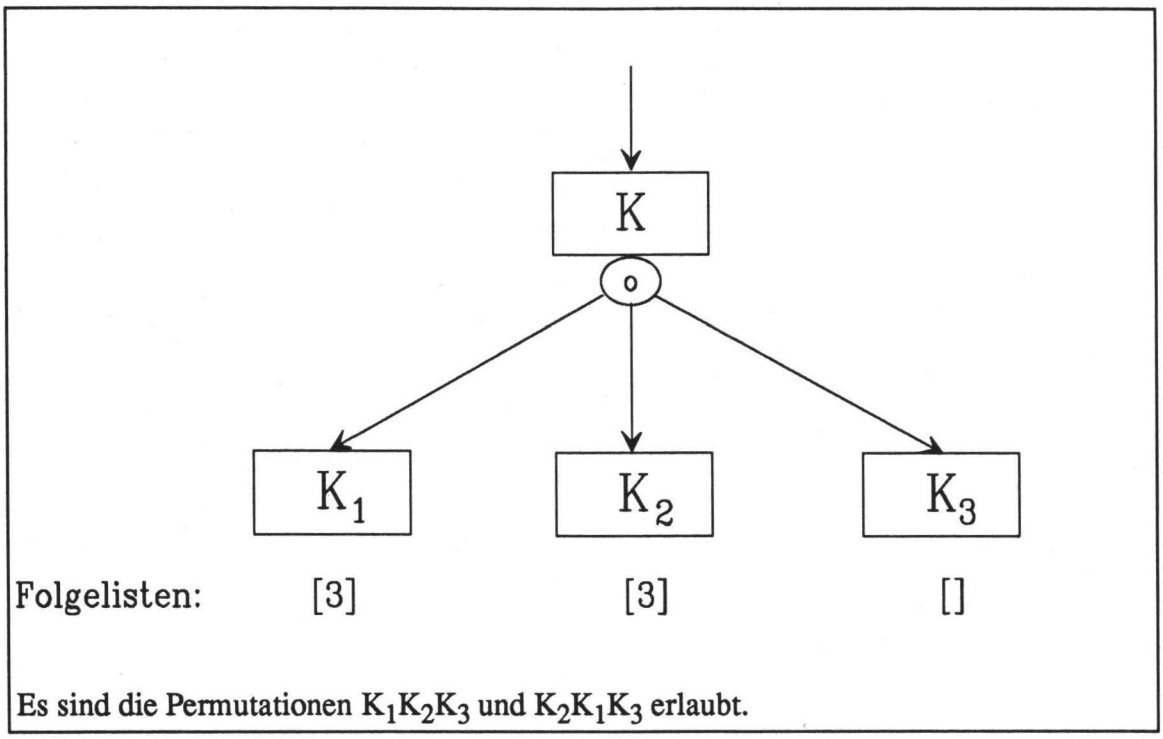


ABB. 3.3.2.2.1. BSP. ZUM ZUSAMMENHANG ZW. FOLGELISTEN UND ERLAUB. PERM.

2. Ablauf

Die Datenflußanalyse benötigt drei Durchläufe:

- Durchlauf 1: Berechnung der Datenflußmuster sämtlicher Knotentypen mit Ausnahme der Rekursion
- Durchlauf 2: wie oben, jedoch zusätzlich Berechnung der Datenflußmuster für alle Rekursionen
- Durchlauf 3: Berechnung der Folgelisten

Rekursionen müssen zweimal besucht werden, um ihr Datenflußmuster zu berechnen, deshalb benötigt der gesamte Algorithmus mindestens zwei Durchläufe. Durchlauf zwei und drei können zusammengefaßt werden (in der Implementation wurde dies gemacht), sie sind hier nur der besseren Gliederung wegen getrennt.

Zunächst sollen einige Funktionen bzw. Operationen definiert werden, die wir für den eigentlichen Algorithmus benötigen. Sie operieren mit Parameterlisten.

Def.: Ausblendefunktion P = **inPar**(Q)

In P werden nur die Parameter aus Q übernommen, die das Zugriffsrecht "in" oder ein mit "in" kombiniertes haben. Alle Parameter in P erhalten das Recht "in".

Def.: Ausblendefunktionen **outPar** und **defPar**

Hier gilt sinngemäß die Definition von inPar, jedoch nun jeweils auf die Rechte "out" bzw. "def" bezogen.

Def.: Ausblendefunktion P = **inoutPar**(Q)

In P werden nur die Parameter aus Q übernommen, die das Zugriffsrecht "in" oder "out" oder eines mit einem dieser Rechte kombiniertes haben, d.h. es werden die Parameter übernommen, deren Recht kein reines "def" ist, wobei das neue Recht in P das alte ohne "def" ist.

Def.: Ausblendefunktion P = **outdefPar**(Q)

Hier gilt sinngemäß die Definition von inoutPar, jedoch nun auf die Rechte "out" bzw. "def" bezogen.

Def.: Vereinigung von Parameterlisten $P = Q \bigcup R$

Zwei Parameterlisten Q und R werden zu P vereinigt, indem alle Parameter, die in Q oder R vorkommen, nach P übernommen werden. Dabei erhält jeder Parameter die Gesamtheit aller Zugriffsrechte, die in beiden Listen für ihn angegeben sind.

Def.: Differenz von Parameterlisten $P = Q \setminus R$

In P werden nur die Parameter aus Q übernommen, die nicht in R sind. Die Rechte werden jeweils ohne Änderung übernommen.

Der Algorithmus **datenflußanalyse**(K) errechnet für sämtliche Knoten des Teilgraphen mit Wurzel K die Datenflußmuster und für Sequenzen zusätzlich die Folgelisten. Die Ergebnisse werden für die gesamte Dauer des Vergleichs den Knoten zugeordnet und stehen somit den nachfolgenden eigentlichen syntaktischen Verfahren zur Verfügung.

Die drei hintereinander ablaufenden Durchläufe werden mit $\mathbf{flow_1}$, $\mathbf{flow_2}$ und $\mathbf{flow_3}$ bezeichnet. $flow_1$ und $flow_2$ sollen Funktionen sein mit einem Konstrukt bzw. Knoten als Argument und dem dazugehörigen Datenflußmuster als Funktionswert. $flow_2$ überschreibt die von $flow_1$ entstandenen Daten. $flow_3$ schließlich berechnet die Folgelisten daraus.

In der Definition von $flow_x$ werden keine Fallunterscheidungen für die einzelnen Konstruktklassen getroffen. D.h., daß für operationale, strukturelle und informelle Konstrukte die gleichen Berechnungsvorschriften gelten[4]. Die Ausarbeitung der Algorithmen bezieht sich jedoch größtenteils auf die operationalen Konstrukte, um anschaulich zu sein.

2. Identische Berechnungen für $flow_1$ und $flow_2$

Die folgenden Definitionen gelten für $flow_1$ und $flow_2$, es wird deshalb von **flow** ohne Index gesprochen.

1. Knoten

a) Atomknoten

Die Datenflußmuster der Atomknoten stehen in deren formalen Parameterlisten (FPL)[5]. Zusätzlich kommt noch der Knotenname als Parameter mit dem Zugriffsrecht "def" hinzu, um die Definition des Namens zu vermerken.

```
flow(node(Name,_,FPL,atom_x)) := FPL ∪ {"def" Name}
```

Man sollte sich klarmachen, daß es auch bei strukturellen und informellen Atomen Sinn macht, Parameter vorzusehen. Das strukturelle Atom kann z.B. die Größe der zu definierenden Struktur als Ein-, den dafür benötigten Speicherplatz als Ausgabeparameter haben. Ähnlich könnte in einem informellen Atom die Eingabe von Text für Tabellenplätze benötigt und als Ausgabe die Länge der insgesamten Information geliefert werden.

b) innere Knoten

Bei inneren Knoten wird zunächst das Datenflußmuster des Konstruktes bestimmt und dann durch die formale Parameterliste erweitert. Zusätzlich wird auch hier der Knotenname mit dem Recht "def" hinzugefügt.

```
flow(node(Name,_,FPL,Konstrukt)) := flow(Konstrukt) ∪ FPL ∪ {"def" Name}
```

Die Verknüpfung mit der formalen Parameterliste hat folgenden Hintergrund: Für innere Knoten muß die formale Parameterliste nicht zwingend angegeben werden. Wird

---

[4]Auf diese Art und Weise wird an vielen Stellen der Arbeit vorgegangen, Hintergrund hierfür ist die Orthogonalität, siehe Kap. 2.1.

[5]Die Angabe der FPL ist für Atomknoten zwingend, für alle anderen optional, siehe Kap. 2.3.

sie angegeben, so kann von ihrer Richtigkeit ausgegangen werden und muß benutzt werden. Denn es können dort Informationen auftauchen, die die Datenflußanalyse nicht erkennt, da der Teilgraph noch unvollständig sein, d.h. nil-Konstrukte enthalten kann. Vollständig muß die formale Parameterliste jedoch nicht sein und deshalb bringt obige Berechnung die besten Ergebnisse.

2. Konstrukte

a) nil-Konstrukte

Das nil-Konstrukt kann im Laufe des Softwareentwicklungsprozesses durch einen beliebigen Teilgraphen ersetzt werden, die Parameterliste ist also noch unbekannt und wird deshalb als leere Liste angenommen.

```
flow(nil) := []
```

b) Instanzen

Das Flußmuster wird von dem instanziierten Knoten übernommen, wobei jedoch "def" - Rechte gestrichen werden. Dies geschieht deshalb, weil die "def" - Rechte nur an den Stellen erscheinen sollen, die auch in den Geltungsbereich des definierten Namen fallen (siehe Punkt 5). Dem instanziierenden Knoten K bleibt nach der Semantik der Instanz aber die innere Struktur des instanziierten Teilgraphen verborgen, "def" - Rechte dürfen also nicht nach oben weitergereicht werden. Diese Überlegung gilt ähnlich für alle anderen Konstrukte außer natürlich für den Knotenverweis. Deshalb wird an diesen Stellen die Funktion **inoutPar** angewendet.

```
flow(instance(Name)) := inoutPar(flow(node(Name,_,_,_)))
```

c) Knotenverweise

Bei Kanten wird das Datenflußmuster direkt vom Sohnknoten übernommen, wobei jedoch eventuell Parameterersetzungen vorzunehmen sind.

```
flow(edge(Name,PEL)) := substPar(flow(node(Name,_,_,_)),PEL)
```

substPar(ParamListe, PEL) ist dabei die Funktion, die als Wert die mit Hilfe der Parameterersetzungsliste PEL veränderte ParamListe liefert.

d) Iterationen

Ein- und Ausgangsbedingung können einen von außen initialisierten Wert erhalten, ebenso kann die Anzahl der Durchläufe variabel gehalten sein. Daher müssen diese drei Parameter zu denen des Iterationskörpers hinzugefügt werden und zwar mit Zugriffsrecht "in".

```
flow(iteration_x(loop(Ein,Anzahl,Aus),V)) := inoutPar(flow(V)) U ["in" Ein,
                                                                  "in" Anzahl,
                                                                  "in" Aus]
```

e) Sequenzen

Gegeben sei eine Sequenz mit n Sohnknoten, die mit K_1,..., K_n bezeichnet seien.

Die Flußmuster der Knoten K_1,..., K_n werden im wesentlichen vereinigt, einige Parameter bzw. Rechte von einzelnen Parametern jedoch gestrichen. Und zwar nach folgenden Regeln:

1. Beachtung des Geltungsbereichs und der Namensüberdeckung:

   Wird in einem K_i ein Name x definiert (durch einen expliziten Knoten) und ist in der Sequenz sichtbar (hat das Recht "def"), so beziehen sich alle Parameter x der darauffolgenden Knoten K_j (j>i) auf diesen eben definierten Namen x und nicht auf einen davor definierten Namen x, da der neue den alten überdeckt. Der Parameter x der Knoten K_{i+1},..., K_n darf also nicht nach oben weitergereicht werden, da sie in den Geltungsbereich des neuen, nicht des alten Namens fallen (siehe Beispiel).

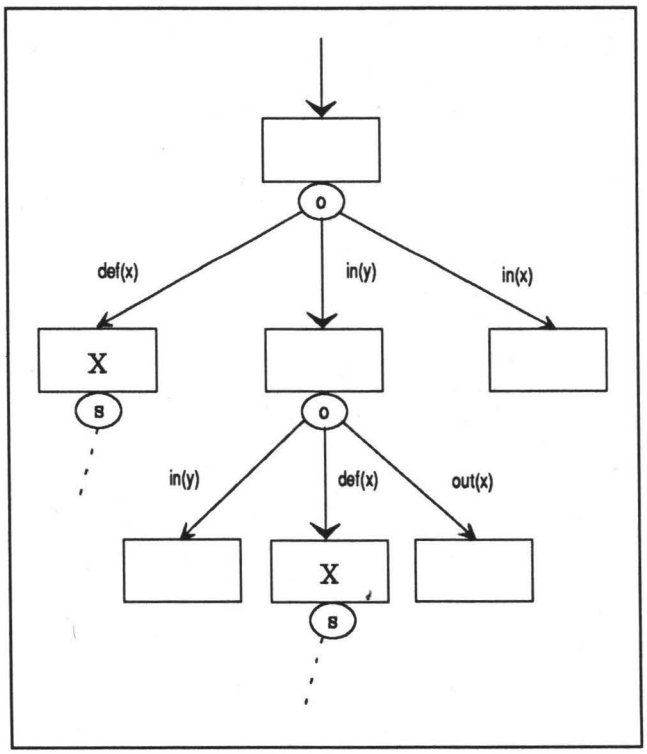


ABB. 3.3.2.2.2. BSP. : DATENFLUßMUSTER BEI BEACHTUNG DES GELTUNGSBEREICHS

2. Kontrolle des Überschreibens von Parametern:

   Taucht im Knoten K_i der Ausgabeparameter x auf ("out" x), so dürfen in den darauffolgenden Knoten K_j (j>i) auftretende Rechte "in" für x nicht nach oben weitergereicht werden, weil K_j seine Eingabe für x von K_i und nicht von außen (Knoten oberhalb von K) erhält, was "in" x ja bedeuten würde (siehe Beispiel).

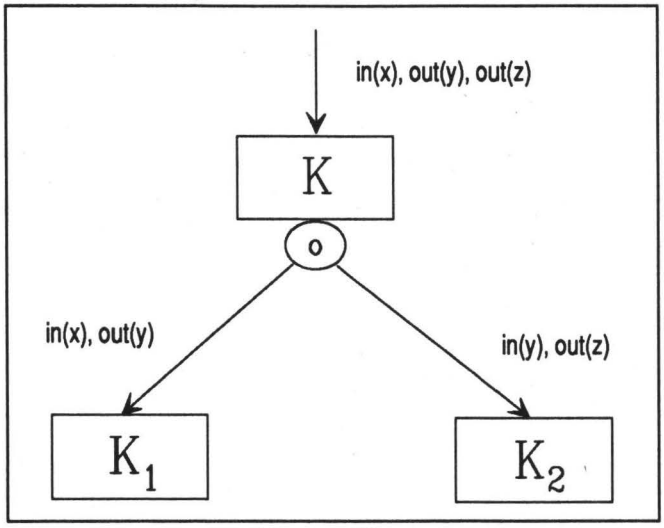


ABB. 3.3.2.2.3. BSP.: DATENFUßMUSTER BEI ÜBERSCHRIEBENEM PARAMETER

Die Vereinigung von Parametern, wobei gemäß obiger Regeln Parameter bzw. Rechte gestrichen werden, soll im folgenden sequentielle Vereinigung genannt werden ($\bigcup_{seq}$).

Def.: sequentielle Vereinigung zweier Parameterlisten

$P = Q \bigcup_{seq} R$ **gdw.**

(1) $H = R \setminus defPar(Q)$ **und**

(2) $P = outdefPar(H) \bigcup (inPar(H) \setminus outPar(Q))$

(1) berücksichtigt Regel 1, durch den Ausdruck inPar(H) \ outPar(Q) werden nur die in-Parameter von H übernommen, die keine out-Parameter von Q sind, gemäß Regel 2. Diese korrigierten in-Parameter werden mit den restlichen ("def" und "out") wieder zur gesamten Parameterliste vereinigt.

Def.: sequentielle Vereinigung mehrerer Parameterlisten $P_1,...,P_n$ $(\bigcup_{seq,\,i=1}^{n} P_i)$

$$\bigcup_{seq,\,i=1}^{0} P_i = []$$

$$\bigcup_{seq,\,i=1}^{1} P_i = P_1$$

$$\bigcup_{seq,\,i=1}^{n} P_i = \bigcup_{seq,\,i=1}^{n-1} P_i \;\bigcup_{seq} P_n$$

Schließlich gilt:

$$\texttt{flow(sequence}_x\texttt{([K}_1\texttt{,K}_2\texttt{,...,K}_n\texttt{]))} := \texttt{inoutPar(}\bigcup_{seq,\,i=1}^{n} \texttt{flow(K}_i\texttt{))}$$

f) Selektionen

Gegeben sei eine Selektion mit n Sohnknoten, also n Alternativen. $S_1,...,S_n$ seien die entsprechenden Selektoren, $A_1,...,A_n$ die Aktionsteile.

Bei der Überlegung, welches Datenflußmuster hier entsteht, ergibt sich das Problem, daß im Gegensatz zu den Sequenzen der Ablauf zum Zeitpunkt der Datenflußanalyse nicht eindeutig bestimmt werden kann.

Folgende Sequenzen von Konstrukten können auftreten:

(1)	S_1A_1,	falls Bedingung aus Selektor S_1 zutrifft
(2)	$S_1S_2A_2$,	falls Bedingung aus Selektor S_2 die erste ist, die zutrifft
	...	...
(n)	$S_1S_2...S_nA_n$,	falls Bedingung aus Selektor S_n die erste ist, die zutrifft

ABB. 3.3.2.2.4. MÖGLICHKEITEN DER AUSWERTUNG EINER SELEKTION

Da die Entscheidung a priori nicht getroffen werden kann, muß die größte mögliche Parameterliste angenommen werden. Sie ergibt sich aus der Vereinigung der Parameterlisten, die die Sequenzen (1) bis (n) erzeugen:

$$\text{flow}(\text{selection}_x([\text{select}(S_1,\_,A_1),\ldots,\text{select}(S_n,\_,A_n)])) :=$$
$$\text{inoutPar}\left(\bigcup_{i=1}^{n}\left(\bigcup_{j=1}^{i}{}_{seq}\ \text{flow}(S_j)\ \bigcup_{seq}\ \text{flow}(A_i)\right)\right)$$

3. Berechnungen für flow für Rekursionen

Rekursionen können deshalb nicht in einem Durchlauf behandelt werden, da sie im Gegensatz zu allen anderen Konstrukten einen Verweis nach "oben", also zu einem Vorfahren V haben. Im Durchlauf 1 ist aber dieser Vorfahre V noch nicht besucht worden. Deshalb liefert $flow_1$ zunächst eine leere Parameterliste, $flow_2$ kopiert die bis dorthin schon Berechnete des Vorfahren.

$$\text{flow}_1(\text{recursion}_x(\_) := []$$
$$\text{flow}_2(\text{recursion}_x(\text{Name}) := \text{inoutPar}(\text{ParList}(\text{Name}))$$
(ParList ist die in Durchlauf 1 schon berechnete Parameterliste)

Es bleibt noch zu zeigen, daß diese Vorgehensweise korrekt ist, d.h. daß die falsche Vorgabe einer leeren Parameterliste in Durchlauf 1 trotzdem zu einer richtigen Liste für den Vorfahren V führt.

Dies macht man sich folgendermaßen deutlich:

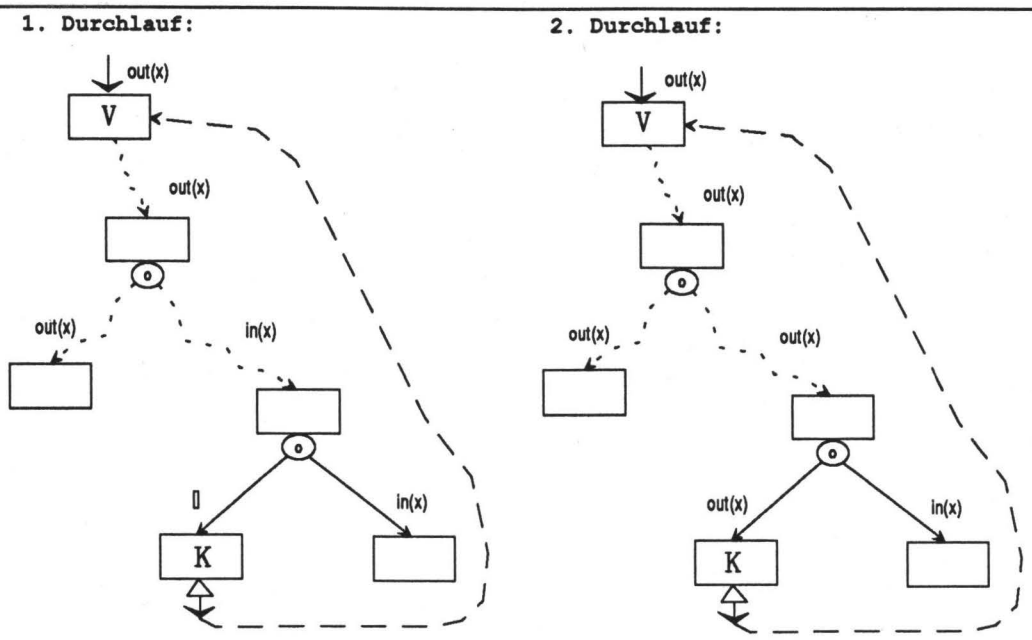
Angenommen, die Vorgehensweise wäre falsch. Das würde bedeuten, daß das Datenflußmuster für V aus Durchlauf 1 ($P_1(V)$ genannt) ein anderes wäre als das von Durchlauf 2 ($P_2(V)$ genannt). Dies könnte nur daraus resultieren, daß sich durch das in K eingesetzte $P_1(V)$ im zweiten Durchlauf andere Resultate auf dem Weg nach oben bis hin zu V ergeben würden, als dies im ersten Durchlauf mit der leeren Liste der Fall war.

Wir betrachten zwei Fälle:

- $P_1(V)$ ist eine Teilmenge von $P_2(V)$, es sind also Rechte oder ganze Parameter hinzugekommen,
- $P_2(V)$ ist eine Teilmenge von $P_1(V)$, es fehlen also Rechte oder ganze Parameter.

Wie entsteht nun eigentlich die Parameterliste von V? Betrachtet man den letzten Abschnitt, so erkennt man, daß sie im wesentlichen durch Vereinigung aller Parameterlisten der Blätter im betrachteten Teilgraphen berechnet wird. Ausnahmen entstehen durch die zwei Streichungsregeln für Sequenzen. Die erste brauchen wir nicht zu betrachten, da keine "def" - Parameter bei K auftreten (Ausblendung durch inoutPar).

Der erste Fall kommt nicht vor, denn beim zweiten Durchlauf kann nie mehr "gefunden" werden als beim ersten, wenn man das Ergebnis des ersten einsetzt, da jeder Teil dieses Ergebnisses nur aus dem Teilgraphen unterhalb von V stammen kann. Das folgende Beispiel soll diesen Sachverhalt etwas klarer werden lassen:



Obwohl zunächst die leere Parameterliste ("[]") angenommen wird, ergibt sich die gleiche Liste für V.

ABB. 3.3.2.2.5. BSP.: DATENFLUßMUSTER BEI REKURSIONEN

Der zweite Fall könnte nur auftreten, wenn durch das Einsetzen von $P_1(V)$ in K im zweiten Durchlauf Streichungsregeln für Sequenzen zum Tragen kämen, $P_1(V)$ also "out" - Parameter enthielte, die das Hochreichen von folgenden "in" - Rechten vermeiden würde (siehe Bsp.). Enthält $P_1(V)$ jedoch solche kritischen Parameter, so müssen sie natürlich aus dem Teilgraphen unterhalb von V stammen. Da "out" - Parameter aber auf jeden Fall hochgereicht werden, wurde die Streichungsregel auch im Durchlauf 1 schon angewendet, wenn auch eventuell erst einige Knoten höher.

Man erkennt dies gut im Beispiel an dem "in" - Recht, das in Durchlauf 2 nicht mehr nach oben gereicht wird (es steht dort nun nur noch das "out" - Recht). Es wurde nur deshalb nicht nach oben gereicht, weil V ein "out" - Recht vom Knoten ganz links erhielt. Eben dieses "out" - Recht vom Knoten ganz links hatte aber schon in Durchlauf 1 das Hochreichen von "in(x)" bis ganz oben zu V verhindert.

Daraus ergibt sich, daß auf dem Weg von V zu K die beiden Durchläufe durchaus verschiedene Ergebnisse bringen können, jedoch nicht für V selbst.

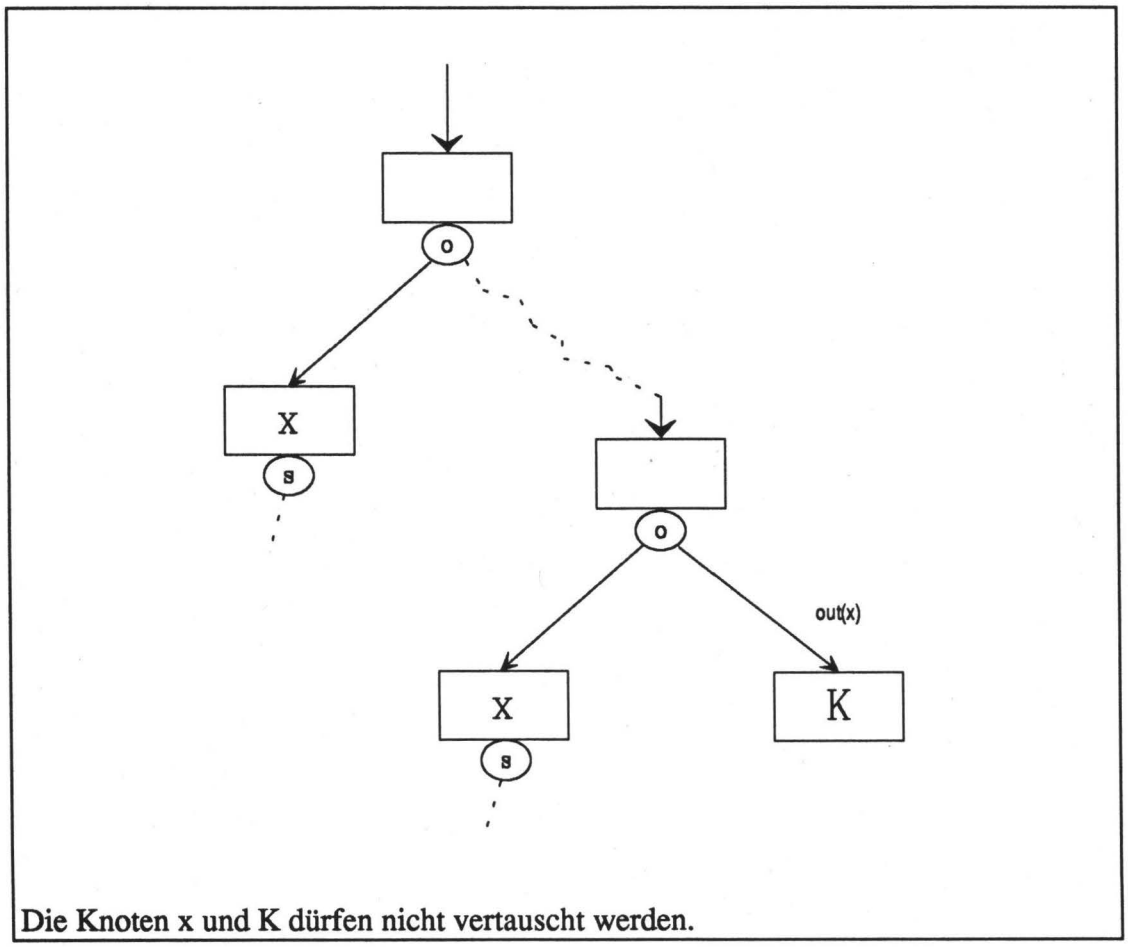
Faßt man die Ergebnisse zusammen, so erkennt man, daß beide Fälle nie auftreten können, $P_1(V)$ und $P_2(V)$ gleich sein müssen, das Verfahren also korrekt ist.

4. Die Berechnung der Folgelisten

In folgenden Fällen verletzt die Vertauschung zweier Sohnknoten den Erhalt der Semantik. Man sollte sie mit den beiden Regeln zur seq. Vereinigung vergleichen, denn dort werden ähnliche Probleme angesprochen.

1. Definition von Namen, Beachtung des Geltungsbereichs

   Ein Name kann erst benutzt werden, nachdem er definiert wurde. Deshalb dürfen Knoten nicht mit benutzenden Knoten ("benutzen" heißt hier: irgend ein Zugriffsrecht auf den Namen haben) vertauscht werden. Sonst fiele der benutzende Knoten in einen anderen Geltungsbereich, in dem der Name nicht bekannt oder der eigentlich zu überdeckende wäre. In beiden Fällen entstünde eine andere Semantik. Dies gilt natürlich auch im umgekehrten Fall. Also: Zwei Knoten K_i und K_j (i<>j) dürfen nicht vertauscht werden, wenn K_i ein "def" - Recht für einen Parameter x hat und K_j den Parameter x besitzt (siehe Beispiel).

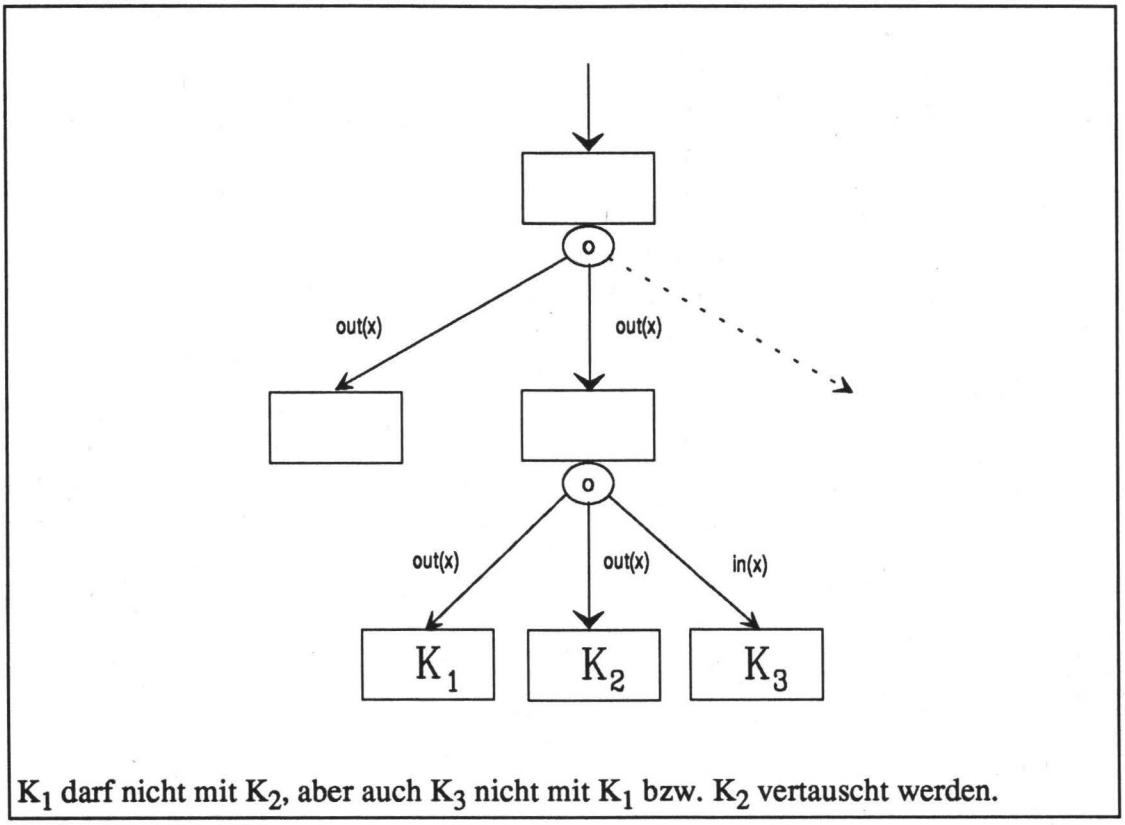


Die Knoten x und K dürfen nicht vertauscht werden.

ABB. 3.3.2.2.6. BSP. ZUR BEACHTUNG DER REIHENFOLGE DER SÖHNE IN SEQUENZEN

2. Überschreiben von Parametern

   Zwei Knoten K_i und K_j (i<>j) dürfen nicht vertauscht werden, wenn K_i ein "out" - Recht für einen Parameter x hat und K_j den Parameter x besitzt. Hat K_j ein "in" - Recht für x, so gilt diese Regel aufgrund der gleichen Überlegung wie bei der seq. Vereinigung. Aber auch "out" - Rechte fallen darunter, da jeweils verschiedene Werte für x weitergereicht werden. "def" - Rechte schließlich sind wegen 1. verboten (siehe Beispiel).



K_1 darf nicht mit K_2, aber auch K_3 nicht mit K_1 bzw. K_2 vertauscht werden.

ABB. 3.3.2.2.7. BSP. ZUR BEACHTUNG DER REIHENFOLGE DER SÖHNE IN SEQUENZEN

Faßt man die beiden Überlegungen zusammen, so erkennt man, daß lediglich dann zwei Knoten K_i und K_j (i<>j) vertauscht werden dürfen, wenn die gemeinsamen Parameter der beiden jeweils nur reine "in" - Rechte besitzen. Dies ist eine starke Einschränkung und illustriert gut den Nutzen der Datenflußanalyse.

Der Aufbau der Folgelisten geschieht folgendermaßen: Für jeden Knoten K_i werden alle ihm folgenden Knoten K_j (j>i) betrachtet, d.h. obige Bedingung geprüft. Dürfen K_i und K_j nicht vertauscht werden, wird j in die Folgeliste für K_i aufgenommen, sonst nicht.

Nachfolgend sei der gesamte Algorithmus noch einmal kurz skizziert.

algorithmus datenflußanalyse(K)

Eingabe: K, *die Wurzel des zu untersuchenden Teilgraphs*
Ausgabe: *den Konstrukten zugeordnete Information:*
- Das Datenflußmuster jedes Knotens des Teilgraphen
- Folgelisten für jede Sequenz innerhalb des Teilgraphens

begin

$flow_1$(K) *Funktionswert wird nicht benutzt*
$flow_2$(K) *Funktionswert wird nicht benutzt*
$flow_3$(K)

Folgende Definitionen für flow gelten für $flow_1$ und $flow_2$!

flow(node(Name,_,FPL,$atom_x$)) := FPL ∪ {"def" Name}
flow(node(Name,_,FPL,Konstrukt)) := flow(Konstrukt) ∪ FPL ∪ {"def" Name}
flow(nil) := []
flow(instance(Name)):= inoutPar(flow(node(Name,_,_,_)))
flow(edge(Name,PEL)):= substPar(flow(node(Name,_,_,_)),PEL)
flow(iteration$_x$(loop(Ein,Anzahl,Aus),V)) := inoutPar(flow(V))
∪ ["in" Ein,
"in" Anzahl,
"in" Aus]

$$flow(sequence_x([K_1,K_2,\ldots,K_n])) := inoutPar(\bigcup_{i=1}^{n}{}_{seq}\ flow(K_i))$$

$$flow(selection_x([select(S_1,\_,A_1),\ldots,select(S_n,\_,A_n)])) :=$$
$$inoutPar(\bigcup_{i=1}^{n}\ (\bigcup_{j=1}^{i}{}_{seq}\ flow(S_j)\ \cup_{seq}\ flow(A_i)))$$

$flow_1$(recursion$_x$(_) := []
$flow_2$(recursion$_x$(Name) := inoutPar(ParList(Name))
(ParList ist die in Durchlauf 1 schon berechnete Parameterliste)
$flow_3$(K) : ... *siehe Beschreibung*
end *datenflußanalyse*

ABB. 3.3.2.2.8. DATENFLUßANALYSE

Nun folgt die Beschreibung der eigentlichen syntaktischen Vergleichsverfahren.

3.3.3. Permutation der Sequenz

3.3.3.1. Einleitung

Dieses Vergleichsverfahren wird auf zwei Sequenzen der gleichen Klasse angewendet.

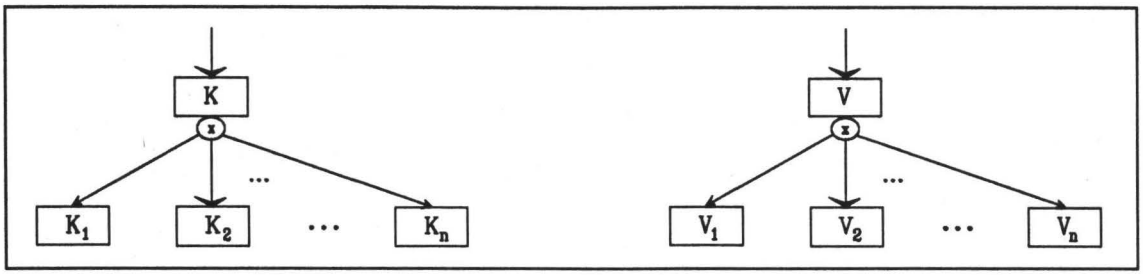


ABB. 3.3.3.1.1. SITUATION ZUR ANWENDG. DES VERFAHRENS "PERM. DER SEQUENZ"

Es wird versucht, den Vergleich der beiden gesamten Sequenzen auf den ihrer Sohnknoten zurückzuführen. Das wesentliche Problem dabei ist, **welches** Paar von Sohnknoten jeweils untersucht werden soll und wie mehrere Einzel- zu einem Gesamtergebnis zu verrechnen sind.

Im folgenden werden die Sequenzen mit K und V, ihre Sohnknoten mit K_1,..., K_n bzw. V_1,..., V_m bezeichnet.

Im **ersten Ansatz** könnte man den Vergleich einfach dadurch bewerkstelligen, daß man die Sohnknoten von links nach rechts paarweise untereinander vergleicht, also K_1 mit V_1, K_2 mit V_2 usw. und die erhaltenen Koeffizienten zu einem Mittelwert verrechnet.

Bei dieser Vorgehensweise würde der Vergleich oft aber schlechter ausfallen als erwartet. Das rührt daher, daß die Reihenfolge der Sohnknoten in Sequenzen mitunter variiert werden kann, ohne die Bedeutung zu ändern, und diese Wahlmöglichkeit müssen wir beim Vergleich berücksichtigen. Änderungen in der Reihenfolge (in anderen Worten: Permutationen) sind dann erlaubt, wenn Paare von Sohnknoten unabhängig voneinander sind in dem Sinn, daß sie in keiner Verbraucher-Erzeuger-Relation stehen bzw. keine Konflikte im Geltungsbereich von Namen auftauchen. Genau diese Verhältnisse sind durch die Datenflußanalyse untersucht worden und die Ergebnisse in den Folgelisten festgehalten worden (siehe 3.3.2). Sie werden an dieser Stelle benötigt.

Seien beispielsweise n = m = 2, K_1 sehr ähnlich zu V_2, K_2 sehr ähnlich zu V_1 und V_1 mit V_2 vertauschbar ohne Semantikänderung, so würde ein hoher Koeffizient durch den Vergleich von K_1 mit V_2 bzw. K_2 mit V_1 entstehen.

Der Fall n<>m muß ebenfalls berücksichtigt werden. Dann kann nämlich davon ausgegangen werden, daß ein Sohnknoten der kürzeren Sequenz einer Sequenz mehrerer Sohnknoten der längeren entspricht oder Sohnknoten der längeren Sequenz anderweitige Teile darstellen, die für die Wiederverwendung ignoriert werden können.

Um einen zufriedenstellenden Koeffizienten zu finden, muß man nach der Kombination der Sohnknotenpaare suchen, die die **größte Übereinstimmung** hervorruft. Dazu müssen zunächst von beiden Seiten alle erlaubten Permutationen zusammengestellt werden. Diese Permutationen werden allen Kombinationsmöglichkeiten unterworfen, es werden quasi neue Sequenzen K´ und V´ gegenübergestellt, die dann nach dem oben angedeuteten ersten Ansatz verglichen werden. Man erhält einen Satz von Koeffizienten. Durch die Suche nach dem **Maximum** wählt man genau die Kombination mit der größten Übereinstimmung aus.

Grob betrachtet könnte man folgendermaßen vorgehen:

funktion	Perm_Seq(K, V)
Eingabe:	Die zu vergleichenden Knoten K und V, die Sequenzkonstrukte einer Klasse enthalten
Ausgabe:	Der Ähnlichkeitskoeffizient des Vergleichs von K und V

Algorithmus:

1. Berechne das Minimum von n und m: s = min(n, m)

2a. Bilde mit Hilfe der Folgelisten von K alle erlaubten Permutationen der Länge s von 1 bis n (entsprechend K_1,..., K_n) und nenne sie P_K.

2b. Bilde mit Hilfe der Folgelisten von V alle erlaubten Permutationen der Länge s von 1 bis m (entsprechend V_1,..., V_m) und nenne sie P_V.

3. Bilde alle Kombinationsmöglichkeiten der Permutationen von P_K und P_V, d.h. bilde das Kreuzprodukt $X = P_K \times P_V$.

4. Richte eine n x m - Tabelle ein, in der die Koeffizienten, die man beim Vergleich der Paare von Sohnknoten untereinander erhält, eingetragen werden sollen. Zunächst sei diese Tabelle jedoch leer.

5. Ordne für jede Kombination aus X von links nach rechts die entsprechenden Paare von Sohnknoten zu und führe für jedes dieser s Paare den Gesamtvergleich durch, falls nicht schon geschehen. Trage die resultierenden Koeffizienten in die Tabelle ein, und zwar den Koeffizienten von K_i und V_j an die Stelle (i, j).

6. Ordne für jede Kombination aus X von links nach rechts die entsprechenden Paare von Sohnknoten zu und bilde aus den dazugehörigen Koeffizienten den Mittelwert. Die Koeffizienten lies aus der Tabelle. So erhält jede Kombination einen Wert, aus diesen Werten bilde das Maximum. Dies ist der endgültige Koeffizient des Vergleichs von K und V.

ABB. 3.3.1.1.2. GROBENTWURF DES ALGORITHMUS ZUR PERMUTATION DER SEQUENZ

Wir wollen das Ganze an einem Beispiel illustrieren:

Sei n = m = 3:

Folgelisten: von K: ([3], [3], []), von V: ([], [3], [])

1. $s = \min(3, 3) = 3$
2a. $P_K = \{123, 213\}$
2b. $P_V = \{123, 213, 231\}$
3. $X = \{(123, 123), (123, 213), (123, 231), (213, 123), (213, 213), (213, 231)\}$
4. Richte leere 3 x 3 - Tabelle ein
5. Komb. (123, 123): Sei $VK(K_1, V_1) = 0.5$, $VK(K_2, V_2) = 0.7$, $VK(K_3, V_3) = 0.9$

   Komb. (123, 213): Sei $VK(K_1, V_2) = 0.4$, $VK(K_2, V_1) = 0.6$, $VK(K_3, V_3)$ schon eingetragen

   Komb. (123, 231): $VK(K_1, V_2)$ schon eingetragen, sei $VK(K_2, V_3) = 0.8$, $VK(K_3, V_1) = 0.2$

   Komb. (213, 123): $VK(K_2, V_1)$, $VK(K_1, V_2)$, $VK(K_3, V_3)$ schon eingetragen

   Komb. (213, 213): $VK(K_1, V_1)$, $VK(K_2, V_2)$, $VK(K_3, V_3)$ schon eingetragen

   Komb. (213, 231): $VK(K_2, V_2)$ schon eingetragen, $VK(K_1, V_3) = 0.1$, $VK(K_3, V_1)$ schon eingetragen

V\K i ->	1	2	3
j = 1	0.5	0.6	0.2
j = 2	0.4	0.7	
j = 3	0.1	0.8	0.9

6. Komb. (123, 123): $\text{Mittelwert} = \frac{0.5 + 0.7 + 0.9}{3} = 0.7$

   Komb. (123, 213): $\text{Mittelwert} = \frac{0.4 + 0.6 + 0.9}{3} = 0.6\overline{3}$

   Komb. (123, 231): $\text{Mittelwert} = \frac{0.4 + 0.8 + 0.2}{3} = 0.4\overline{6}$

   Komb. (213, 123): $\text{Mittelwert} = \frac{0.6 + 0.4 + 0.9}{3} = 0.6\overline{3}$

Komb. (213, 213): $\text{Mittelwert} = \frac{0.7 + 0.5 + 0.9}{3} = 0.7$

Komb. (213, 231): $\text{Mittelwert} = \frac{0.7 + 0.1 + 0.2}{3} = 0.\overline{3}$

Maximum der Mittelwerte = 0.7

Gesamtkoeffizient = 0.7

Wie man sieht, blieb uns durch diese Vorgehensweise der Vergleich $VK(K_1, V_3)$ erspart. Bei größerem n und m ist der Nutzen in der Praxis noch viel größer. Der Algorithmus soll nun näher betrachtet werden.

3.3.3.2. Der Algorithmus

Es werden im folgenden die Schritte des Grobentwurfs erneut aufgelistet, um sie gegebenenfalls zu verfeinern bzw. näher zu erläutern.

1. Berechne das Minimum von n und m: s = min(n, m).

Dieser Schritt ist nötig, damit im folgenden die Länge der Permutationen festgelegt ist. Es sei noch einmal gesagt, daß bei ungleicher Anzahl von Sohnknoten der Vergleich unter der Annahme durchgeführt wird, daß ein Sohnknoten auf der einen Seite mehreren auf der anderen entsprechen kann oder in der längeren Sequenz zu ignorierende Knoten vorkommen. Deshalb müssen aus der längeren Sequenz lediglich s Sohnknoten ausgewählt werden.

2a. Bilde mit Hilfe der Folgelisten von K alle erlaubten Permutationen der Länge s von 1 bis n (entsprechend K_1,..., K_n) und nenne sie P_K.

2b. Bilde mit Hilfe der Folgelisten von V alle erlaubten Permutationen der Länge s von 1 bis m (entsprechend V_1,..., V_m) und nenne sie P_V.

Wir gehen in zwei Schritten vor: Zunächst berechnen wir alle möglichen Permutationen und unterziehen diese dann einem Test mit Hilfe der Folgelisten. Wir erhalten so die übrigbleibenden erlaubten Permutationen.

Auf die Ausformulierung des ersten Schrittes sei hier verzichtet; Algorithmen, die aus den Zahlen 1 bis k eine Folge paarweise verschiedener Zahlen der Länge s auswählen, werden in einführender Informatik-Literatur oft behandelt.

Der zweite Schritt, der Test, ob eine gegebene Permutation zugelassen ist, entspricht folgendem Schema:

Man arbeitet die Permutation $P = (j_1 j_2 ... j_s)$ schrittweise ab. Die schon bearbeiteten Elemente werden in der Menge M gesammelt. Kommt nun irgend ein Element der Folgeli-

ste des aktuellen j_i in dieser Menge M vor, so bedeutet dies, daß der dazugehörige Knoten **vor** dem aktuellen steht und somit kann die gesamte Permutation nicht mehr erlaubt sein.

```
prädikat erlaubt

     Eingabe:   P = (j1j2...js), die zu untersuchende Permutation,
                alle Folgelisten der Sequenz
     Ausgabe:   true,      falls die Permutation erlaubt ist,
                false,     sonst

     begin
          M := {}
          erlaubt := true
          for i := 1 to s do
               if M schnitt Folgeliste(ji) = {}  M und Liste sind disjunkt
                    then  M := M U {ji}
                    else  erlaubt = false
     end erlaubt
```

ABB. 3.3.3.2.1. PERM. DER SEQUENZ: TEST, OB VORGEGEBENE PERM. ERLAUBT IST

3. **Bilde alle Kombinationsmöglichkeiten der Permutationen von P_K und P_V, d.h. bilde das Kreuzprodukt $X = P_K \times P_V$.**

Aus X lassen sich alle dem Vergleich zu unterwerfenden Sequenzen-Paare ablesen, also quasi die neu erzeugten Sequenzen K´ und V´.

4. **Richte eine n x m - Tabelle ein, in der die Koeffizienten, die man beim Vergleich der Paare von Sohnknoten untereinander erhält, eingetragen werden sollen. Zunächst sei diese Tabelle jedoch leer.**

5. **Ordne für jede Kombination aus X von links nach rechts die entsprechenden Paare von Sohnknoten zu und führe für jedes dieser s Paare den Gesamtvergleich durch, falls nicht schon geschehen. Trage die resultierenden Koeffizienten in die Tabelle ein, und zwar den Koeffizienten von K_i und V_j an die Stelle (i, j).**

Die Organisationsform der Tabelle ist mehr eine Implementationssache, sie soll uns hier nicht interessieren. Wichtiger ist Punkt 5, das Füllen der Tabelle. Es wird der Gesamtvergleich VK nur dann aufgerufen, falls der Tabellenplatz noch leer ist. Auf diese Weise wird jedes Knotenpaar nur einmal verglichen.

```
funktion FülleTabelle

Ausgabe:   Tabelle,   die Tabelle der Koeffizienten der Paare von Sohnknoten

begin
      Richte Tabelle ein
      Lösche Tabelle
      for all x aus X do
            begin
                  let x = ( (k1,...,ks), (v1,...,vs) )
                  for i := 1 to s do
                        if Tabelle(ki,vi) empty
                        then Tabelle(ki,vi) := VK(Kki,Vvi)
            end
end
```

ABB. 3.3.3.2.2. PERM. DER SEQUENZ: FÜLLEN DER TABELLE MIT DEN KOEFF.

Die Beschränkung der zu betrachtenden Permutationen war der erste Effizienzgewinn, die Verwaltung der Tabelle ist der zweite. Denn wie man am Beispiel deutlich sieht, wird der Vergleichskoeffizient zweier Sohnknoten mehrmals benötigt, der Vergleich selbst aber nur einmal ausgeführt.

6. Ordne für jede Kombination aus X von links nach rechts die entsprechenden Paare von Sohnknoten zu und bilde aus den dazugehörigen Koeffizienten den Mittelwert. Die Koeffizienten lies aus der Tabelle. So erhält jede Kombination einen Wert, aus diesen Werten bilde das Maximum. Dies ist der endgültige Koeffizient des Vergleichs von K und V.

In Schritt 6 werden die Koeffizienten verrechnet. Je **einem** Paar von Sequenzen wird der **Mittelwert** der Paare ihrer Sohnknoten zugeordnet. Aus diesen Mittelwerten wird das **Maximum** gewählt.

```
funktion Koeff

Eingabe:   X,          das Kreuzprodukt aller erlaubten Permut. von K und V;
           Tabelle,    die Tabelle der Koeffizienten der Paare von Sohnknoten
Ausgabe:   Koeff,      der Gesamtkoeffizient des Vergleichs zwischen K und V


begin
      for all x aus X do
            begin
                  let x = ( (k1,...,ks), (v1,...,vs) )
                  for i := 1 to s do Koeffi := Tabelle(ki,vi);
                  Mittelx := mittel(i aus 1..s)(Koeffi)
            end
      Koeff := max(x aus X)(Mittelx)
end Koeff
```

ABB. 3.3.3.2.3. PERM. DER SEQUENZ: BERECHNUNG DES GESAMTKOEFFIZIENTEN

Punkt 6 arbeitet im Prinzip so, wie wir es in unserem ersten Ansatz vorgeschlagen hatten, jedoch werden anstatt nur dem einen ursprünglichen **alle** Paare von Sequenzen K´ und V´, die semantisch den Originalen entsprechen, geprüft und das ähnlichste gewählt.

3.3.3.3. Zusammenfassung

Das eben vorgestellte Verfahren zeigt stellvertretend für die nun folgenden einige Prinzipien der Entwurfsmethodik auf. Es geht darum, den Vergleich zweier Konstrukte auf den der Sohnknoten (und eventuell anderer Daten) zurückzuführen. Fallen mehrere Teilkoeffizienten an, so ergibt sich i.a. das Problem, wie diese zu verrechnen sind. In diesem Fall wählten wir den Mittelwert bzw. das Maximum als verrechnende Funktionen. Verfeinerungen können grundsätzlich dahingehend gemacht werden, daß man Mittelwerte gewichtet (falls eine Gewichtung aufgrund zusätzlicher Informationen sinnvoll erscheint), das Maximum "sanfter" macht, also in diesem Fall nicht nur eine, sondern auch die anderen Teilkoeffizienten mit berücksichtigt.

In dieser Arbeit werden grundsätzlich einfache Verrechnungen vorgeschlagen, da zunächst die prinzipielle Eignung der Verfahren untersucht werden soll. Erst zu einem späteren Zeitpunkt der Untersuchungen, unter Einbeziehung von Ergebnissen aus der Praxis, erscheinen Verfeinerungen sinnvoll.

3.3.4. Permutation der Selektion

Dieses Vergleichsverfahren wird auf zwei Selektionen der gleichen Klasse angewendet.

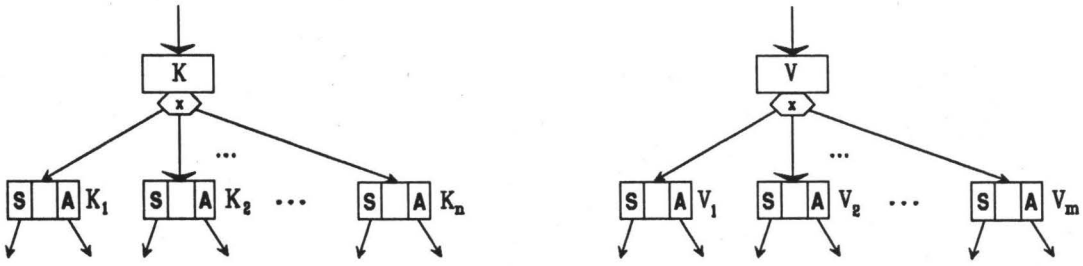


ABB. 3.3.4.1. SITUATION ZUR ANWENDG. DES VERFAHRENS "PERM. DER SELEKTION"

Die Problematik, einen zufriedenstellenden Algorithmus zu finden, ist mit der aus dem vorigen Kapitel verwandt. Denn einerseits ist die Selektion neben der Sequenz das einzige Konstrukt, das **mehr als einen** Sohnknoten haben kann und andererseits entsteht bei der Auswertung einer Selektion eine **Sequenz** von Konstrukten. Diese Sequenz wird aus den folgenden in Abhängigkeit der Auswertungen der Selektoren ausgewählt (S: Selektoren, A: Aktionsteile):

(1)	S_1A_1,	falls Bedingung aus Selektor S_1 zutrifft
(2)	$S_1S_2A_2$,	falls Bedingung aus Selektor S_2 die erste ist, die zutrifft
	...	...
(n)	$S_1S_2...S_nA_n$,	falls Bedingung aus Selektor S_n die erste ist, die zutrifft

ABB. 3.3.4.2. MÖGLICHKEITEN DER AUSWERTUNG EINER SELEKTION

Es kann a priori nicht entschieden werden, welche Sequenz ausgewählt wird, daher müssen, wie auch schon in der Datenflußanalyse, alle berücksichtigt werden.

Für die weiteren Überlegungen sollte man sich noch einmal die Semantik der Selektion und vor allem die Funktion des Selektors und des Aktionsteils klar machen. Die Selektoren sollen in Abhängigkeit von Eingangsparametern Bedingungen auswerten, die darüber entscheiden, ob der dazugehörige Aktionsteil ausgewertet wird oder nicht. Im Aktionsteil selbst sollen, wie der Name schon sagt, die eigentlichen Aktionen ablaufen. Diese Aktionen charakterisieren die Selektion für diese aktuellen Eingabeparameter und sind deshalb für unsere Betrachtungen hier wichtig. Doch auch in den Selektoren können Aktionen im obigen Sinne integriert sein, also nicht nur Berechnungen geschehen, die der bloßen Auswertung der Bedingung dienen. Unabhängig von der Diskussion, wie sinnvoll solche Aktionen in Selektoren sind, müssen wir diese Möglichkeit berücksichtigen. Um den ersten Ansatz eines Vergleichsalgorithmus zu finden, scheint im Augenblick die genaue Betrachtung von Selektions- und Aktionsteil sowie der verschiedenen Permutationen unerläßlich.

In Anlehnung an den Algorithmus aus dem vorigen Kapitel könnte nun folgendermaßen vorgegangen werden: Man bildet zunächst alle möglichen Permutationen der Sohnknoten für beide Seiten, wobei man die Knoten selbst unverändert läßt, also die Einheit von Selktions- und Aktionsteil bewahrt. Danach stellt man wie in Abb. 3.3.4.3 zu jeder Permutation alle dazugehörigen Sequenzen auf und betrachtet sie auch im folgenden als Sequenzkonstrukte (wir erhalten ähnlich wie in 3.3.3 Sequenzen K´ und V´), die linken Sohnknoten sind also Selektoren, der rechte ein Aktionsteil (die Sequenzen haben die Form S...SA). Nun wird mit Hilfe der Folgelisten überprüft, ob diese Sequenzen erlaubt sind. Jedes Paar von (erlaubten) Sequenzen gleicher Länge wird verglichen, indem man sich ähnlich wie in 3.3.3 Knotenpaar für Knotenpaar (von links nach rechts) gegenüberstellt. Auf diese Weise werden jeweils Paare von Selektoren bzw. Aktionsteilen miteinander verglichen. Die erhaltenen Koeffizienten werden geeignet zu einem einzigen verrechnet.

Diese Vorgehensweise läßt noch offen, wie stark die Ergebnisse des Vergleichs von Paaren von Selektoren im Gegensatz zu denen von Aktionsteilen gewichtet werden, wie stark die Ähnlichkeit der gesamten Selektionen von der dieser Konstrukte im einzelnen also beeinflußt wird. Weiterhin fragt sich, wie eine verschiedene Anzahl von Sohnknoten zu behandeln ist (also Selektionen mit unterschiedlicher Anzahl von Fallunterscheidungen) und schließlich ob diese Methode tatsächlich immer einen Koeffizienten liefert, der unserer intuitiven Auffassung von Ähnlichkeit entspricht.

Man sollte sich deshalb beim Entwurf des Algorithmus weniger an der Form der Auswertung orientieren, wie es oben getan wurde, sondern mehr die Gegebenheiten, die die

Ähnlichkeit charakterisieren, berücksichtigen. Es zeigt sich nämlich, daß hier nicht ohne weiteres vorgegangen werden kann wie bisher: Der Gesamtkoeffizient ergab sich im wesentlichen aus einer Verrechnung von Einzelkoeffizienten, die wiederum lediglich aus dem Vergleich der Unterkonstrukte gewonnen wurden, ohne daß die genauere Struktur dieser Unterkonstrukte berücksichtigt wurde. An dieser Stelle gibt es jedoch eben solche Fälle, bei denen eine **genaue** Bewertung auf die **genaue** Untersuchung der Unterkonstrukte hinausläuft.

Einer der Problemkreise, bei dem der obige Ansatz schlechte Ergebnisse liefert, ist z.B. die Möglichkeit der unterschiedlichen Aufspaltung von Fallunterscheidungen, vor allem wenn die Aufspaltungen unterschiedlich groß sind. Zur Illustration betrachte man sich zwei zu vergleichende Selektionen K und V. Beide lösen verschiedene Aktionen in Abhängigkeit des Eingabeparameters x aus, wobei K drei Fälle x<0, x=0 und x>0 unterscheidet, V nur zwei, nämlich x>0 und x<=0. Sind sich K und V sehr ähnlich, so wird man auch eine große Ähnlichkeit zwischen den Aktionsteilen feststellen, **unabhängig** vom Vergleich der Selektoren untereinander. In anderen Fällen wird noch deutlicher, daß die Ähnlichkeit hauptsächlich durch die Aktionsteile charakterisiert wird, weniger durch die Selektoren.

Soweit ist ersichtlich, daß die Gewichtung der Selektoren gering sein sollte. Weiterhin kann man feststellen, daß die oben beschriebene Methode keine ausreichend guten Ergebnisse liefern kann, da sie trotz der Betrachtung der verschiedenen Permutationen stets Selektions- und Aktionsteil als Einheit behandelt. Die Möglichkeit, Fallunterscheidungen unterschiedlich aufzuspalten erfordert darüber hinaus jedoch auch die Untersuchung von Kombinationen nicht zusammengehöriger Selektions- und Aktionsteile.

Nun ist aber eine exakte Berücksichtigung der Selektoren gerade in solchen Fällen überaus schwierig, denn dies würde, wie oben angedeutet, auf eine genaue Untersuchung der Bedingungen selbst hinauslaufen. Es fragt sich, ob dies mit akzeptablem Aufwand und Sicherheit möglich ist oder ob wir uns mit einer Heuristik zufrieden geben können.

Die obigen Überlegungen und die Diskussion einiger anderer charakteristischer Fälle führen zu der (zunächst zu einfach erscheinenden) Idee, die Betrachtung der Selektoren komplett zu unterlassen und lediglich die Aktionsteile zu berücksichtigen, in etwa wie folgt: Man konstruiert aus den Aktionsteilen Sequenzen, auf die man exakt das Verfahren "Permutation der Sequenz" anwendet. Dadurch werden alle möglichen Permutationen berechnet und auch solche kritischen Fälle wie der Vergleich von Selektionen mit unterschiedlicher Anzahl von Sohnknoten berücksichtigt, indem wir die Zuordnung eines Aktionsteils auf der einen Seite zu mehreren auf der anderen mit einbeziehen (siehe obiges Beispiel!). Der Algorithmus lautet deshalb einfach wie folgt:

funktion	Perm_Sel(K, V)
Eingabe:	Die zu vergleichenden Knoten K und V, die Selektionskonstrukte einer Klasse enthalten
Ausgabe:	Der Ähnlichkeitskoeffizient des Vergleichs von K und V
Algorithmus:	Bilde aus den Selektionen K und V Sequenzen K´und V´, indem die Aktionsteile von K zu den Sohnknoten von K´ und die Aktionsteile von V zu den Sohnknoten von V´ werden. Ordne allen Sohnknoten von K´ und V´ leere Folgelisten zu. Vergleiche K´ und V´ mit dem Verfahren "Permutation der Sequenz" und setze: Perm_Sel(K, V) := Perm_Seq(K´, V´).

ABB. 3.3.4.3. PERMUTATION DER SELEKTION

Verbotene Permutationen gibt es selbstverständlich nicht, da Aktionsteile einer Selektion nie hintereinander in Sequenzen ausgewertet werden. Der obige Algorithmus berücksichtigt dies durch die leeren Folgelisten. Leere Folgelisten bedeuten, daß keine Einschränkungen bezüglich der Reihenfolge gemacht werden, also alle möglichen Permutationen erlaubt sind.

Die nähere Untersuchung des Verfahrens zeigt ein durchaus erwünschtes Verhalten: Die Unterschlagung der Selektoren führt dazu, daß die erhaltenen Koeffizienten öfter zu höheren Werten tendieren als gewünscht, jedoch nie zu niedrigeren. Man mache sich unbedingt deutlich, daß die Abweichungen nicht so groß werden, wie man zunächst annehmen könnte. Es entsteht auf den ersten Blick der Eindruck, daß durch die "ziellose" Zuordnung und den Vergleich von Aktionsteilen, die zu völlig verschiedenen Selektoren gehörten, und in diesem Sinn nichts miteinander zu tun haben, der Gesamtkoeffizienz zu stark beeinflußt werden würde. Man sollte dabei aber bedenken, daß solche Koeffizienten durch die abschließende Maximumbildung meistens unberücksichtigt bleiben. Untersucht man Beispiele aus der Praxis, so stellt man fest, daß die Werte in den meisten Fällen sehr gut sind und der Algorithmus zufriedendstellend arbeitet.

3.3.5. Permutation der Iteration

Dieses Vergleichsverfahren wird auf zwei Iterationen der gleichen Klasse angewendet.

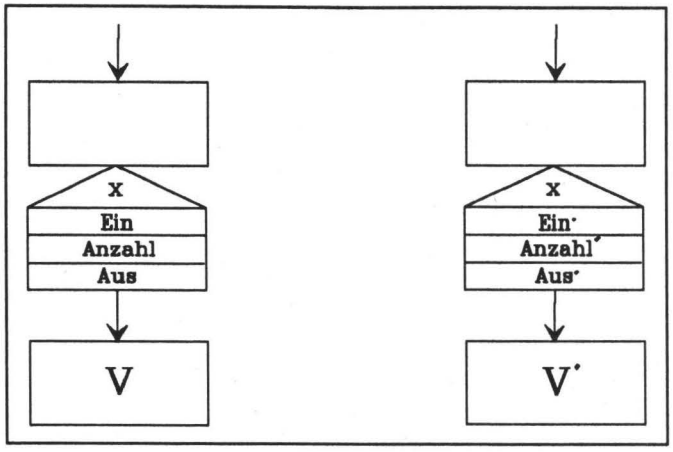


ABB. 3.3.5.1. SITUATION ZUR ANWENDG. DES VERFAHRENS "PERM. DER ITERATION"

Das Iterationskonstrukt ist sehr mächtig, da es durch die Möglichkeit, Eingangs-, Ausgangs- und Zählbedingung gleichzeitig anzugeben, eine vielfältigere Möglichkeit zur Steuerung des Ablaufs der Schleife bietet, als wir es von den meisten herkömmlichen Programmiersprachen her gewohnt sind.

Ein Algorithmus, der einen Vergleichskoeffizienten für zwei Iterationen liefert, wird sowohl den Iterationskörper, als auch die Bedingungen beachten müssen.

Zwei verschiedene Sätze von je drei Bedingungen können den gleichen Effekt haben, ebenso wie z.B. in PASCAL eine **repeat-until**-Schleife die gleiche Wirkung wie eine **while**-Schleife zeigen kann. Ebenso ist es ohne weiteres möglich, eine **for**- bzw. eine **repeat-until**-Schleife in eine äquivalente **while**-Schleife umzuwandeln. Übertragen auf das Iterationskonstrukt bedeutet das, daß man einen Satz von Iterationsbedingungen so umwandeln kann, daß lediglich eine Eingangsbedingung existiert und die Iteration unter Beibehaltung des Körpers die gleiche Semantik hat. Dies führt zu einem ersten Ansatz unseres Verfahrens:

Man normiert zunächst die Bedingungen der zu vergleichenden Iterationen wie oben angedeutet, d.h. man wandelt jeden Satz von drei Bedingungen in eine äquivalente Eingangsbedingung. Danach vergleicht man die erhaltenen Eingangsbedingungen mit Hilfe eines geeigneten Algorithmus. Schließlich vergleicht man noch die Iterationskörper miteinander und verrechnet die so erhaltenen zwei Koeffizienten geeignet zu einem einzelnen.

Bei genauerer Betrachtung stoßen wir hier auf ein Problem, das uns schon im letzten Kapitel begegnete: Um die Bedingungen umzuwandeln, müssen diese genau erkannt werden, d.h. der entsprechende Teilgraph, der sie berechnet, untersucht werden. Weiterhin muß vor dem Vergleich der erhaltenen Eingangsbedingungen eine

prädikatenlogische Normierung vorgenommen werden, d.h. auch dort ganz genau die Struktur der Teilgraphen untersucht werden. Es fragt sich deshalb an dieser Stelle ebenfalls, ob der Aufwand lohnt, bzw. zu sicheren Ergebnissen führt.

Da wir darüber hinaus bei der Berechnung des Gesamtkoeffizienten den Vergleich der Iterationskörper sicher weit stärker gewichten werden als den der Bedingungen, genügt uns die im folgenden beschriebene, einfachere Methode. Sie untersucht die Bedingungen nicht im einzelnen, sondern testet lediglich, ob sie existieren.

funktion Perm_Iter(K, V)

Eingabe: Die zu vergleichenden Knoten K und K´, die Iterationskonstrukte einer Klasse enthalten

Ausgabe: Der Ähnlichkeitskoeffizient des Vergleichs von K und K´

Algorithmus:

1. Berechne den Vergleichskoeffizienten K_B für die Bedingungen wie folgt:
   a) $K_B := 0$
   b) Sind beide Zählbedingungen vorhanden oder fehlen beide, so gilt: $K_B := K_B + \frac{1}{3}$
   c) Sind beide Eingangsbedingungen vorhanden oder fehlen beide, so gilt: $K_B := K_B + \frac{1}{3}$
   d) Sind beide Ausgangsbedingungen vorhanden oder fehlen beide, so gilt: $K_B := K_B + \frac{1}{3}$
   e) Fehlen Ein und Aus´ und existieren Aus und Ein´, so gilt: $K_B := K_B + \frac{1}{3}$
   f) Fehlen Ein´ und Aus und existieren Aus´ und Ein, so gilt: $K_B := K_B + \frac{1}{3}$
2. Berechne den Vergleichskoeffizienten für die Iterationskörper: $K_V :=$ VK(V, V´)
3. Verrechne K_B und K_V geeignet zum Gesamtkoeffizienten, z.B. durch gewichtete Mittelwertbildung.

ABB. 3.3.5.2. PERMUTATION DER ITERATION

Man beachte, daß sich die Fälle 1e) und 1f) gegenseitig ausschließen, ebenso kann 1b) oder 1c) nicht gelten, wenn 1e) oder 1f) zutrifft. Deshalb kann K_B maximal den Wert 1 annehmen.

Die Fälle 1e) bzw. 1f) sollen andeutungsweise berücksichtigen, daß Umwandlungen von Ein- in Ausgangsbedingung möglich sind und deshalb eine größere Ähnlichkeit vermutet wird.

3.3.6. Abstraktion der Hierarchie

3.3.6.1. Einleitung

Die bisher vorgestellten syntaktischen Verfahren berücksichtigen die Tatsache, daß verschiedene Permutationen der Sohnknoten (Perm. der Sequenz, Perm. der Selektion) bzw. der Bedingungen (Perm. der Iteration) zu ähnlichen Beschreibungen führen können. Sie arbeiten auf **einer Ebene**, d.h. Vertauschungen von Knoten finden nur innerhalb einer Ebene, nur zwischen Brüdern statt.

Damit ist jedoch noch nicht das gesamte Spektrum der Möglichkeiten, strukturelle Ähnlichkeiten zwischen Teilgraphen zu finden, abgedeckt. Durch die Definition des Softwareentwurfsgraphen hat der Entwickler weitgehende Freiheiten, wie stark hierarchisch er seinen Entwurf formuliert. Stark hierarchische Beschreibungen führen zu Graphen großer, wenig hierarchische Beschreibungen zu solchen kleiner Höhe.

Um die Gemeinsamkeiten von zwei in ihrer Aussage ähnlichen Teilgraphen, die jedoch sehr verschieden in dem Grad der Hierarchisierung sind, finden zu können, sind **Ebenen-übergreifende** Untersuchungen notwendig.

Dies wird im letzten zu besprechenden syntaktischen Vergleichsverfahren versucht. Vertauschungen (soweit sinnvoll) finden hier auch zwischen Vätern und Söhnen statt. Das geschieht sukzessive, so daß im Verlauf des Gesamtverfahrens die hierarchische Struktur der Teilgraphen vollständig geändert werden kann und so **von der Hierarchie abstrahiert** wird, daher der Name des Verfahrens.

Der Algorithmus geht im wesentlichen in zwei Schritten vor. Zunächst wird versucht, die beiden Teilgraphen in eine Normalform[6] zu bringen, so daß sie von möglichst kleiner Höhe sind. Daraufhin werden alle bisher vorgestellten Vergleichsverfahren auf die umgewandelten Graphen erneut angewendet, d.h. Vergleiche innerhalb jeweils einer Ebene durchgeführt. Diese beiden Schritte werden nicht auf die kompletten Teilgraphen, sondern sukzessive jeweils nur auf deren oberste Ebene durchgeführt, um auch die Zwischenergebnisse bei der Bildung der Normalform zu nutzen. Der genaue Ablauf wird später vorgestellt.

Die einzelnen Regeln, deren fortlaufende Anwendung einen Graphen in seine Normalform überführen, sind so formuliert, daß ein möglichst weitgehender Erhalt der Semantik gewährleistet ist. Auf diese Weise soll eine zu große Verfälschung der Aussage durch die Umwandlung verhindert werden.

Dies ist leider jedoch nicht immer zufriedenstellend möglich, es müssen Vernachlässigungen bzw. unüberprüfte Annahmen in Kauf genommen werden, damit die Umformungen nicht zu komplex werden. Es handelt sich um die gleiche Problematik wie bei

---

[6]Der Begriff Normalform ist in diesem Zusammenhang nicht in zu strengem Sinne zu verstehen. Die im folgenden erläuterte Umwandlung der Graphen kann i.a. nicht eindeutig und vollständig durchgeführt werden, da die einzelnen Auflösungsschritte nur bedingt anwendbar sind.

dem Verfahren "Permutation der Selektion" und wie dort sind auch hier die entstehenden Fehler akzeptabel, wenn man sich vergegenwärtigt, daß die Ähnlichkeit im Sinne von Eignung für Wiederverwendung, weniger im Sinne der Aussage, im Vordergrund steht.

Die Anwendung der Abstraktion der Hierarchie steht im Gesamtverfahren an letzter Stelle. Dies rührt daher, daß die entstehenden Ergebnisse die unzuverlässigsten sind, d.h. das Verfahren im Vergleich zu den anderen das unsicherste ist.

Die Begründung dieses Sachverhalts ist zum einen in der eben erwähnten Formulierung der Umwandlungsregeln, zum anderen aber auch in deren unsensiblen Anwendung zu suchen. Durch die konsequente Bildung der Normalform geht Information verloren, die in der Hierarchisierung steckte. Weiterhin können Hierarchieebenen zum Vergleich gebracht werden, die in diesem Sinne nichts miteinander zu tun haben.

Trotzdem ist die Abstraktion der Hierarchie als letzter Versuch, überhaupt noch eine Ähnlichkeit zu finden, durchaus sinnvoll. In der vorgestellten Kombination mit den anderen Verfahren können brauchbare Ergebnisse entstehen.

3.3.6.2. Der Algorithmus

Wir wenden uns zunächst der Bildung der Normalform zu, um anschließend deren Einbettung in die gesamte Ablaufstruktur zu besprechen.

Die Überführung in eine Normalform geschieht sukzessive durch Anwendung von sog. **Auflösungsregeln**. Sie arbeiten jeweils nur auf der obersten Ebene, d.h. es wird lediglich die Wurzel und deren Sohnknoten betrachtet. Durch wiederholten Aufruf der Algorithmen werden alle Ebenen aufgelöst.

Die Auflösungsregeln verfolgen grob folgende Strategie:

- Der Graph soll in seiner Höhe minimiert werden. Dies erreicht man durch Auflösen von verschachtelten Sequenzen zu einer einzigen Sequenz.
- Die Konstrukte Sequenz, Selektion und Iteration sollen möglichst geordnet werden bezüglich ihrer Stellung im Graphen. Selektionen sollen möglichst weit oben (nahe der Wurzel), Sequenzen möglichst weit unten (nahe der Blätter) stehen. Iterationen sollen zwischen Selektionen und Sequenzen stehen. Man erreicht dies durch Auflösungen aller Kombinationen der erwähnten Konstrukte, die nicht in der gewünschten Reihenfolge stehen. Z.B. wird eine Sequenz, die Selektionen unter ihren Söhnen hat, zu einer Selektion mit Sequenzen als Unterkonstrukte aufgelöst usw.
- Übergänge in Konstruktklassen sollen die Anwendung von Auflösungsregeln verbieten, weil dies sonst eine zu starke Einbuße im Erhalt der Semantik bedeuten würde. Beispielsweise darf eine operationale Sequenz, die eine strukturelle Sequenz als Sohn hat, **nicht** aufgelöst werden[7].

[7]Diese Einschränkung bezieht sich selbstverständlich nur auf **diese** Untersequenz. Hat die Sequenz einen weiteren Sohn, der eine operationale Sequenz ist, so dürfen diese beiden sehr wohl aufgelöst werden.

Wir kommen nun zur Beschreibung der einzelnen Auflösungsregeln. Dazu sollen zunächst einige Vereinbarungen bzgl. der Notation der Regeln getroffen werden und die Vereinigung von Parameterersetzungslisten definiert werden. Wir führen folgende Schreibweisen ein, die die Lesbarkeit verbessern sollen:

- Parameterersetzungslisten von expliziten Knoten erhalten immer den Namen "PEL(*Knotenname*)". Dies erübrigt die Angabe der Parameterersetzungslisten in Konstrukten.
- Parameterersetzungslisten werden bei Anwendung der Regel unverändert übernommen, wenn nicht ausdrücklich eine Vorschrift zu deren Änderung angegeben ist. Die Änderung wird durch "PEL(*Knotenname*) := ..." symbolisiert. Tauchen auf der rechten Seite von ":=" Parameterersetzungslisten auf, so beziehen sie sich auf den Zustand vor Anwendung der Regel.
- Wird ein Knotenname in normaler Schrift innerhalb von Konstrukten angegeben (z.B. "K_1"), so steht er immer für einen expliziten Knoten, d.h. muß durch das Konstrukt "edge("K_1", PEL(K_1))" ersetzt werden.
- Wird ein Knotenname innerhalb von Konstrukten in kursiver Schrift angegeben (z.B. "K_1"), so steht er wahlweise für einen expliziten oder impliziten Knoten. Geht man von einem expliziten Knoten aus, so verfährt man wie oben. Geht man von einem impliziten Knoten aus, so ersetzt man ihn durch ein beliebiges Konstrukt, es sei denn er verweist auf eine Knotenangabe ("node"), dann erstezt man ihn durch das Konstrukt dieses angegebenen Knotens. Außerdem wird bei impliziten Knoten die zugeordnete Parameterersetzungsliste als leer angenommen (Wichtig für eventuelle Vereinigungen der Parameterersetzungslisten).
- Die Namen von Selektoren bzw. Aktionsteilen werden der Einfachheit halber vom Knotennamen, in dem sie stehen, abgeleitet. Heißt z.B. der Knoten V_1, so heißt der zugehörige Selektor VS_1, der Aktionsteil VA_1.

Die Vereinigung von Parameterersetzungslisten geschieht folgendermaßen:

Def.: Vereinigung von Parameterersetzungslisten $P = Q \bigcup R$

Zwei Parameterersetzungslisten Q und R werden zu P vereinigt, indem alle Substitutionen, die in Q oder R vorkommen nach P übernommen werden. Dabei werden Transitivitäten aufgelöst, d.h. alle Teilmengen von P der Form {subst(x, y), subst(y, z)} durch die Menge {subst(x, z)} ersetzt.

Eine Bemerkung zu den Abbildungen: Unterstrukturen sind dann nicht eingezeichnet, wenn sie beliebig sind. Bei den Illustrationen zu Auflösungen muß man also darauf achten, daß in diesem Fall nicht nur der gezeichnete Knoten, sondern auch der gesamte untergeordnete Teilgraph gemeint ist.

Nun die eigentlichen Regeln:

1. verschachtelte Sequenzen

Sei die Wurzel des zu behandelnden Teilgraphen eine Sequenz der Klasse x. Solange einer der Söhne ebenfalls eine Sequenz der Klasse x ist, kann diese Regel angewendet werden.

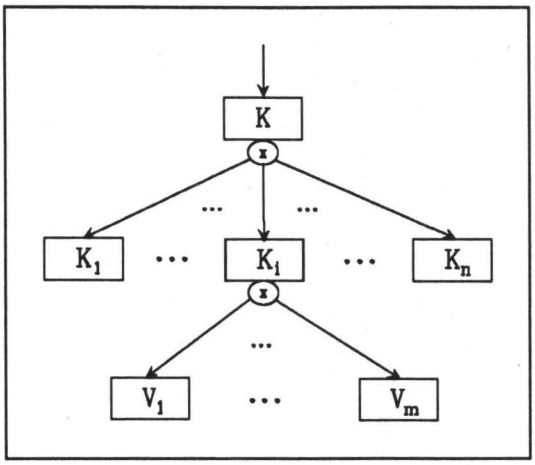


ABB. 3.3.6.2.1. SITUATION ZUR ANWENDUNG DER AUFLÖSUNGSREGEL 1

Ein einfacher Ansatz, die Auflösung zu bewerkstelligen, könnte sein, die Sohnknoten der Untersequenz einfach zu Sohnknoten der eigentlichen Sequenz zu machen, also eine neue Sequenz $K_1,...,K_{i-1}, V_1,...,V_m, K_{i+1},...,K_n$ zu formulieren. Die neuen Parameterersetzungslisten dieser neuen Söhne ergeben sich durch Vereinigung der Parametererersetzungsliste der Untersequenz mit denen ihrer Söhne.

$sequence_x([K_1, \ldots, K_{i-1}, K_i, K_{i+1}, \ldots, K_n])$
$node(K_i, \ldots, sequence_x([V_1, \ldots, V_m]))$

wird aufgelöst zu:

$sequence_x([K_1, \ldots, K_{i-1}, V_1, \ldots, V_m, K_{i+1}, \ldots, K_n])$

mit:

$PEL(V_1) := PEL(K_i) \cup PEL(V_1)$
$PEL(V_2) := PEL(K_i) \cup PEL(V_2)$
$\ldots$
$PEL(V_m) := PEL(K_i) \cup PEL(V_m)$

ABB. 3.3.6.2.2. ERSTER (UNVOLLSTÄNDIGER) ANSATZ DER AUFLÖSUNGSREGEL 1

Hierbei wurde jedoch nicht beachtet, daß Probleme auftreten, wenn der Geltungsbereich von Namen der Sequenz von solchen der Untersequenz überdeckt wird, wenn gleiche Namen auftreten. Zur Illustration sei z.B. Abb. 2.4.1. empfohlen. Sei der gemeinsame Name, wie in der Abbildung, x. Das Problem ergibt sich daraus, daß die Knoten K_{i+1} bis K_n, wenn sie x referenzieren das obere x meinen, nach unserem ersten Ansatz würden sie nach der Auflösung jedoch das untere x ansprechen.

Abhilfe schafft man durch eine eindeutige Umbennenung aller Knoten der Untersequenz. Eindeutig heißt hier, daß die neuen Namen verschieden von denen der Hauptsequenz sind. Dies könnte z.B. durch Anhängen eines Suffixes (etwa eines Apostrophs) geschehen. Wir benennen nicht alle in den Teilgraphen der Untersequenz vorkommenden Knoten um, sondern nur die jeweiligen Wurzeln und simulieren die Umbenenung der restlichen Knoten durch Einfügen einer Substitution in die entsprechenden Parameterersetzungslisten.

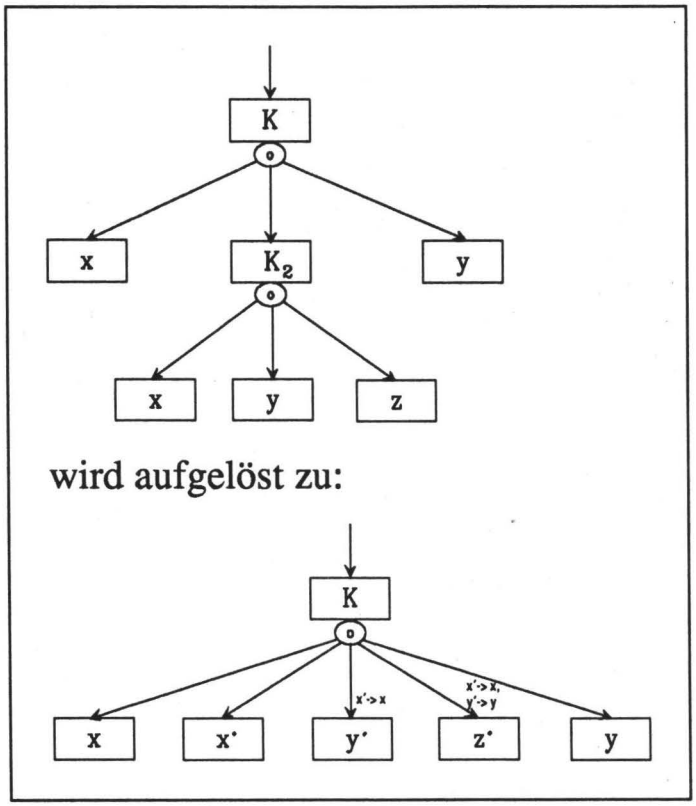


ABB. 3.3.6.2.3. BSP. ZUR AUFLÖSUNG VERSCHACHTELTER SEQUENZEN

Allgemein formuliert sieht die Auflösungsregel so aus (man beachte: $V_1,...,V_m$ müssen explizite Knoten sein, damit in ihre Parameterersetzungslisten die nötigen Substitutionen eingetragen werden können!):

$sequence_x([K_1, \ldots, K_{i-1}, K_i, K_{i+1}, \ldots, K_n])$
$node(K_i, \ldots, sequence_x([V_1, \ldots, V_m]))$

wird aufgelöst zu:

$sequence_x([K_1, \ldots, K_{i-1}, V_1', \ldots, V_m', K_{i+1}, \ldots, K_n])$

mit:

$PEL(V_1') := PEL(K_i) \cup PEL(V_1)$
$PEL(V_2') := PEL(K_i) \cup PEL(V_2) \cup \{subst(V_1', V_1)\}$
$\ldots$
$PEL(V_m') := PEL(K_i) \cup PEL(V_m) \cup \{subst(V_1', V_1), \ldots, subst(V_{m-1}', V_{m-1})$

ABB. 3.3.6.2.4. ZWEITER (UNVOLLSTÄNDIGER) ANSATZ DER AUFLÖSUNGSREGEL 1

Doch auch dieser Ansatz behandelt nicht alle Fälle richtig. Würde etwa im obigen Beispiel der obere Knoten y K_2 referenzieren, so würde diese Referenz nach der Auflösung falsch verlaufen, da K_2 nicht mehr existiert. Die Knoten x, y und z sind nicht mehr als Einheit referenzierbar.

Eine korrekte Behandlung würde auf die Übergabe einer impliziten Sequenz x, y, z als Parameter an das obere y hinauslaufen. Man müßte also in der Parametererersetzungsliste für das obere y eine Substitution einfügen, die K_2 durch diese implizite Sequenz ersetzen würde. Auf diese Weise könnte man garantieren, daß sämtliche Referenzen im Teilgraphen des oberen y richtig behandelt werden würden.

Doch das Konzept der Übergabe von impliziten Konstrukten als Parameter ist im Softwareentwurfsgraph nicht vorgesehen. Wir stellen daher kurz zwei Lösungen vor, die den Erhalt der Semantik zwar verletzen, für unsere Zwecke jedoch ausreichend sind.

Die erste Möglichkeit, den "verlorenen" Parameter K_i[8] wieder zu gewinnen, entsteht dadurch, daß man die Sequenz K nach der Auflösung um einen zusätzlichen Knoten mit dem "verlorenen" Namen K_i erweitert und als Konstrukt dieses Knotens einfach nil wählt. Die günstigste Stelle, an der diese Einfügung stattfinden sollte, ist direkt hinter V_m', weil so die alten Geltungsbereiche für $V_1', \ldots, V_m'$ erhalten bleiben.

Das größte Manko an dieser einfachen Lösung sind die fehlerhaften Datenflüsse, die die Datenflußanalyse später findet. Das nil-Konstrukt ergibt eine leere Parameterliste und dies bedeutet im Zweifelsfall mehr erlaubte Permutationen der Sohnknoten als vorgesehen.

Eine zweite Möglichkeit ist daher, im zusätzlich neu eingeführten Knoten K_i tatsächlich eine Sequenz mit Verweisen auf $V_1', \ldots, V_m'$ zu integrieren und auf K_i von K über eine Instanz zu verweisen. Es entstehen so die richtigen Parameterlisten bei der Datenflußanalyse.

Diese Lösung wird jedoch recht aufwendig, da die wieder eingeführte Sequenz K_i im weiteren Verlauf der Normalformbildung nicht aufgelöst werden darf. Das gilt selbst-

[8]Im folgenden sind die Knotennamen wieder allgemein gewählt, wie im zweiten Ansatz.

verständlich nur für die Sequenz K_i, andere Sequenzen mit den gleichen Sohnknoten V_1',..., V_m' dürfen aufgelöst werden. Der Knoten K_i muß also entsprechend markiert werden und der Auflösungsalgorithmus diese Markierung beachten. Soll die Markierung im Namen selbst geschehen (z.B. Anhängen eines Suffixes "*"), so muß zusätzlich in den Parameterersetzungslisten der Knoten, die K_i referenzieren, eine entsprechende Substitution (etwa subst(K_i*, K_i)) eingefügt werden.

So stellt sich die vollständige Regel in etwa so dar:

$sequence_x([K_1, \ldots, K_{i-1}, K_i, K_{i+1}, \ldots, K_n])$
$node(K_i, \ldots, sequence_x([V_1, \ldots, V_m]))$

wird aufgelöst zu:

a) (erste Lösung)
$sequence_x([K_1, \ldots, K_{i-1}, V_1', \ldots, V_m', K_i, K_{i+1}, \ldots, K_n])$
$node(K_i, \ldots, nil)$ bzw.

b) (zweite Lösung)

$sequence_x([K_1, \ldots, K_{i-1}, V_1', \ldots, V_m', instance(K_i), K_{i+1}, \ldots, K_n])$
$node(K_i, \ldots, sequence_x([V_1', \ldots, V_m']))$

mit:

$PEL(V_1') := PEL(K_i) \cup PEL(V_1)$
$PEL(V_2') := PEL(K_i) \cup PEL(V_2) \cup \{subst(V_1', V_1)\}$
...
$PEL(V_m') := PEL(K_i) \cup PEL(V_m) \cup \{subst(V_1', V_1), \ldots, subst(V_{m-1}', V_{m-1})$

ABB. 3.3.6.2.5. DRITTER ANSATZ DER AUFLÖSUNGSREGEL 1

2. verschachtelte Selektionen

Bei der Konstruktion einer Regel für diesen Fall muß man sich darüber im klaren sein, daß die Selektoren bei dem Verfahren "Perm. der Selektion" nicht beachtet werden.

Erste Konsequenz hieraus ist, daß der Fall, in dem der Selektor der Selektion eine weitere Selektion ist, nicht weiter beachtet werden muß.

Die Regel kann also solange angewendet werden, solange einer der Aktionsteile der Selektion (Klasse x) ebenfalls eine Selektion der Klasse x ist.

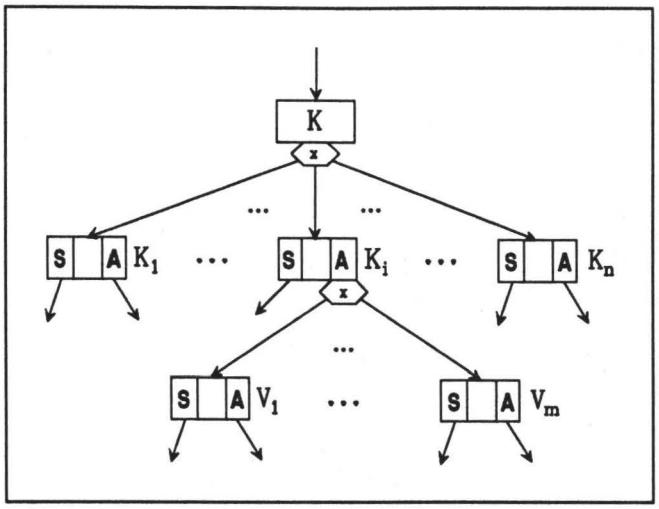


ABB. 3.3.6.2.6. SITUATION ZUR ANWENDUNG DER AUFLÖSUNGSREGEL 2

Die Vorgehensweise ist dann ähnlich wie die bei verschachtelten Sequenzen, d.h. man zieht die Söhne V_1,..., V_m der Unterselektion eine Ebene hoch. Lediglich die Selektoren VS_1,..., VS_m müssen neu bestimmt werden. Die Probleme bzgl. der Überdeckung von Namen treten nicht auf, da Selektoren bzw. Aktionsteile jeweils ihren eigenen Geltungsbereich haben.

Bei korrekter Behandlung würden sich die neuen Selektoren VS_1,..., VS_m durch logische Konjunktion von KS_i und dem jeweiligen VS_j ($1 <= j <= m$) ergeben. Wie schon bei der Perm. der Iteration tritt auch hier das Problem auf, daß die Auswertung bzw. Verrechnung von Bedingungen schwer möglich ist.

Da die neu entstandenen Selektoren VS_1,..., VS_m später jedoch ohnehin nicht beachtet werden, ist eine solch genaue Betrachtung auch gar nicht nötig. Es ist lediglich darauf zu achten, daß bei der Auflösung so vorgegangen wird, daß die Datenflußanalyse später die richtigen Ergebnisse für die gesamte Selektion liefert. Da zur Bildung der Konjunktion beide Selektoren KS_i und VS_j ausgeführt werden müssen, bilden wir den neuen Selektor VS_j als Sequenz der beiden.

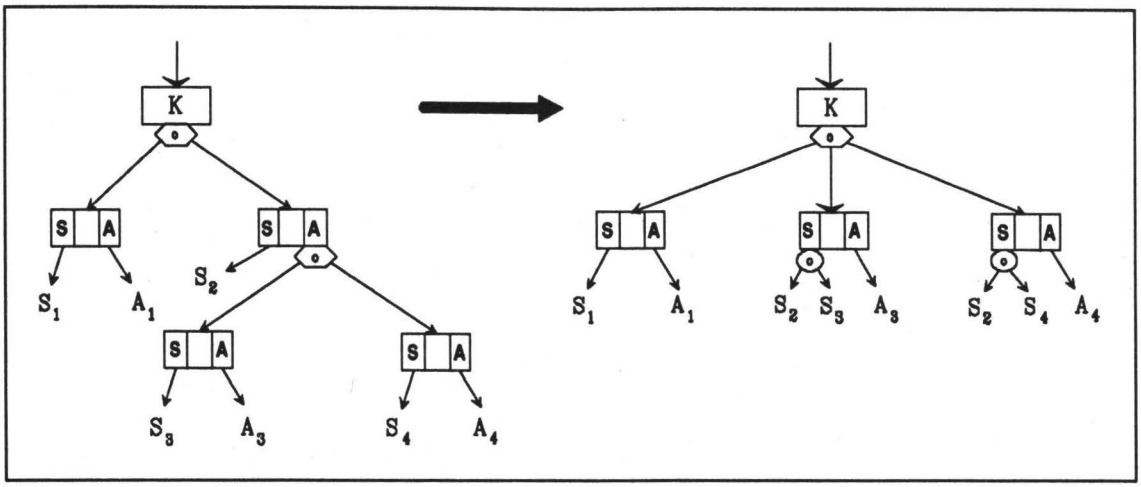


ABB. 3.3.6.2.7. BEISPIEL ZUR ANWENDUNG DER AUFLÖSUNGSREGEL 2

Allgemein kann die Regel in etwa wie folgt formuliert werden:

$selection_x([K_1, \ldots, K_{i-1}, select(KS_i, \_, KA_i), K_{i+1}, \ldots, K_n])$
$node(KA_i, \ldots, selection_x([select(VS_1, \_, VA_1), \ldots, select(VS_m, \_, VA_m)]))$

wird aufgelöst zu:

$selection_x([K_1, \ldots, K_{i-1},$
$select(sequence_x([KS_i, VS_1]), \_, VA_1),$
$select(sequence_x([KS_i, VS_2]), \_, VA_2),$
$\ldots$
$select(sequence_x([KS_i, VS_m]), \_, VA_m),$
$K_{i+1}, \ldots, K_n])$

mit:

$PEL(VS_1) := PEL(KA_i) \cup PEL(VS_1), \quad PEL(VA_1) := PEL(KA_i) \cup PEL(VA_1)$
$PEL(VS_2) := PEL(KA_i) \cup PEL(VS_2), \quad PEL(VA_2) := PEL(KA_i) \cup PEL(VA_2)$
$\ldots$
$PEL(VS_m) := PEL(KA_i) \cup PEL(VS_m), \quad PEL(VA_m) := PEL(KA_i) \cup PEL(VA_m)$

ABB. 3.3.6.2.8. AUFLÖSUNGSREGEL 2

3. Sequenzen, die Selektionen zu Söhnen haben

Sei die Wurzel des zu behandelnden Teilgraphen eine Sequenz der Klasse x. Wenn einer der Söhne eine Selektion der Klasse x ist, kann diese Regel angewendet werden.

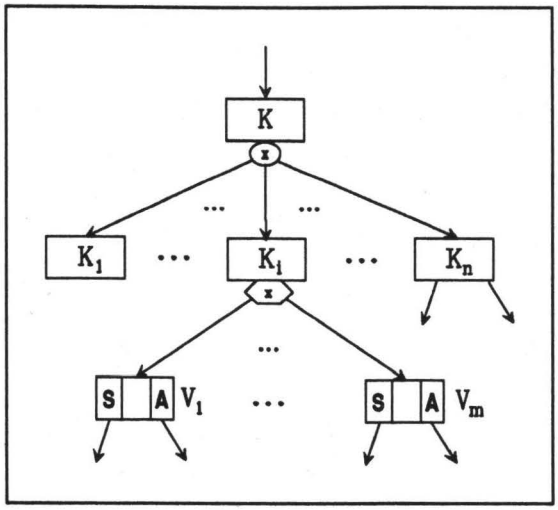


ABB. 3.3.6.2.9. SITUATION ZUR ANWENDUNG DER AUFLÖSUNGSREGEL 3

Die beste Auflösung, bei der der Erhalt der Semantik nicht zu sehr beeinträchtigt wird, ist folgende:

Die Selektion wird zur Wurzel des Teilgraphen gemacht, wobei die Aktionsteile VA_1,..., VA_m durch VA_1',..., VA_m' ersetzt werden. Diese neuen Aktionsteile sind Sequenzen der Knoten K_1,..., K_{i-1}, K_{i+1},..., K_n, wobei zwischen K_{i-1} und K_{i+1} der jeweilige alte Aktionsteil VA_j eingefügt wird.

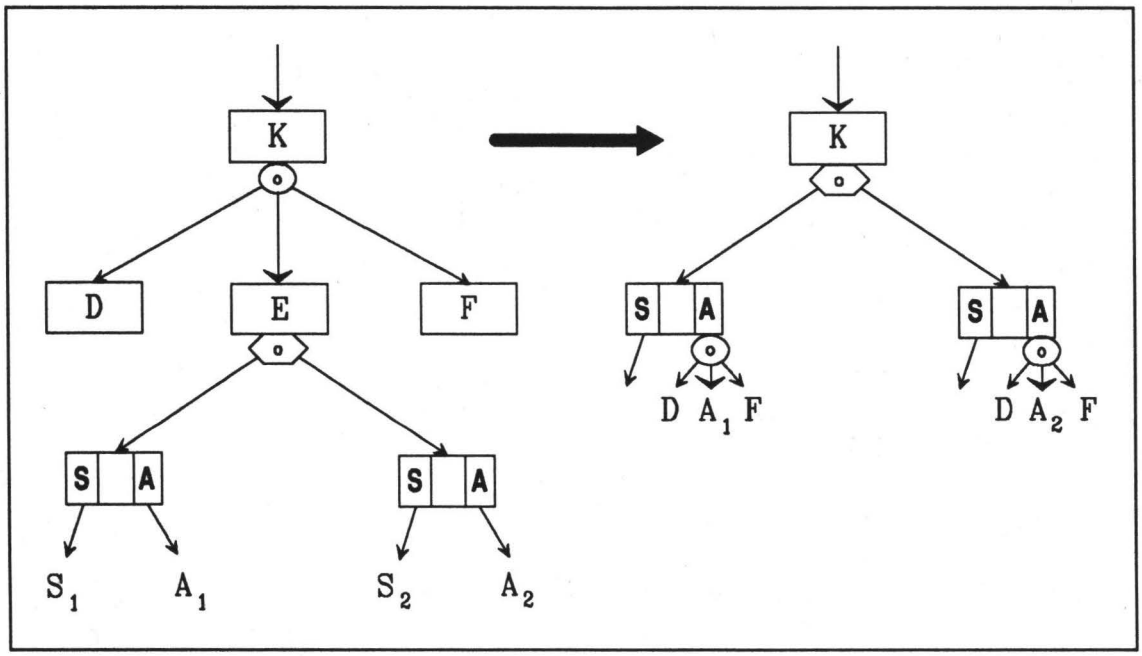


ABB. 3.3.6.2.10. BEISPIEL ZUR ANWENDUNG DER AUFLÖSUNGSREGEL 3

Allgemein kann die Regel in etwa wie folgt formuliert werden:

$sequence_x([K_1, \ldots, K_{i-1}, K_i, K_{i+1}, \ldots, K_n])$
$node(K_i, \ldots, selection_x([select(VS_1, \_, VA_1), \ldots, select(VS_m, \_, VA_m)]))$

wird aufgelöst zu:

$selection_x([select(VS_1, \_, VA_1'), \ldots, select(VS_m, \_, VA_m')])$
$node(VA_1', \ldots, sequence_x([K_1, \ldots, K_{i-1}, VA_1, K_{i+1}, \ldots, K_n]))$
$node(VA_2', \ldots, sequence_x([K_1, \ldots, K_{i-1}, VA_2, K_{i+1}, \ldots, K_n]))$
$\ldots$
$node(VA_m', \ldots, sequence_x([K_1, \ldots, K_{i-1}, VA_m, K_{i+1}, \ldots, K_n]))$

ABB. 3.3.6.2.11. AUFLÖSUNGSREGEL 3

Die Semantik wird dann verletzt, wenn die Berechnung der Selektoren Ergebnisse von $K_1,\ldots, K_{i-1}$ benutzt. Diese Ergebnisse wurden vor der Auflösung nämlich beachtet, danach nicht. Da sie aber wieder im Aktionsteil integriert sind, bleibt die Semantik erhalten, wenn die Auswertung von $VS_1,\ldots, VS_m$ unabhängig von $K_1,\ldots, K_{i-1}$ ist.[9]

4. Sequenzen, die Iterationen zu Söhnen haben

Sei die Wurzel des zu behandelnden Teilgraphen eine Sequenz der Klasse x. Wenn einer der Söhne eine Iteration der Klasse x ist, kann diese Regel angewendet werden.

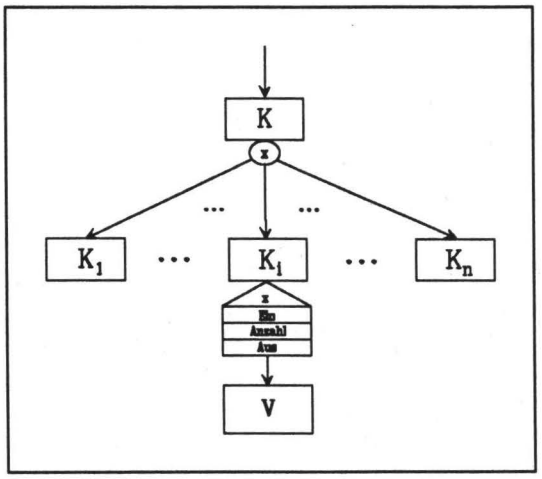


ABB. 3.3.6.2.12. SITUATION ZUR ANWENDUNG DER AUFLÖSUNGSREGEL 4

[9]Integriert man $K_1,\ldots, K_{i-1}$ in den neuen Selektoren anstatt in den Aktionsteilen, so werden ihre Ergebnisse zwar beachtet, wird aber nicht der erste Selektor zu wahr ausgewertet, so werden $K_1,\ldots,K_{i-1}$ **mehrmals** ausgeführt. Diese Lösung erhält die Semantik im genannten Fall also ebenfalls nicht. Da in der Perm. der Selektion nur die Aktionsteile benutzt werden, wählen wir die erste Lösung.

Die beste Auflösung, bei der der Erhalt der Semantik nicht zu sehr beeinträchtigt wird, ist folgende:

Die Iteration wird zur Wurzel des Teilgraphen gemacht, wobei der alte Schleifenkörper V durch die Sequenz ersetzt wird. In diese Sequenz wird die gesamte alte Iteration durch deren Schleifenkörper ersetzt.

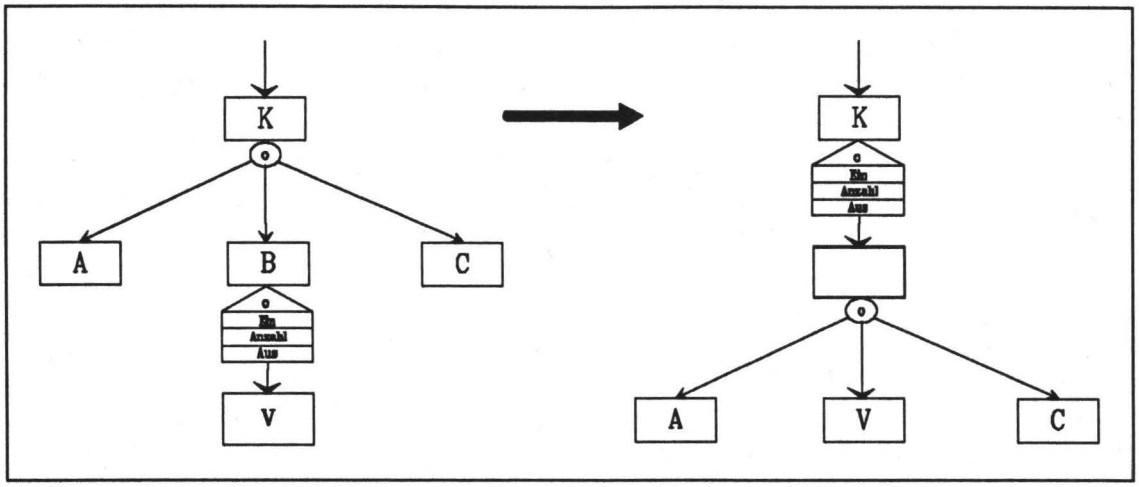


ABB. 3.3.6.2.13. BEISPIEL ZUR ANWENDUNG DER AUFLÖSUNGSREGEL 4

Allgemein kann die Regel in etwa wie folgt formuliert werden:

$sequence_x([K_1, \ldots, K_{i-1}, K_i, K_{i+1}, \ldots, K_n])$
$node(K_i, \ldots, iteration_x(loop(\ldots), V))$

wird aufgelöst zu:

$iteration_x(loop(\ldots), sequence_x([K_1, \ldots, K_{i-1}, V, K_{i+1}, \ldots, K_n]))$

ABB. 3.3.6.2.14. AUFLÖSUNGSREGEL 4

Die Verletzung der Semantik gestaltet sich ähnlich wie bei der alternativen Lösung zu Regel 3 (siehe Fußnote). K_1,..., K_{i-1}, aber auch K_{i+1},..., K_n werden **mehrmals** ausgeführt, wenn die Schleife mehrere Durchläufe hat. Haben jedoch K_1,..., K_n bei mehrmaligem Aufruf den gleichen Effekt, so bleibt die Semantik erhalten.

5. Iterationen, die eine Selektion als Schleifenkörper haben

Sei die Wurzel des zu behandelnden Teilgraphen eine Iteration der Klasse x. Wenn der Schleifenkörper eine Selektion der Klasse x ist, kann diese Regel angewendet werden.

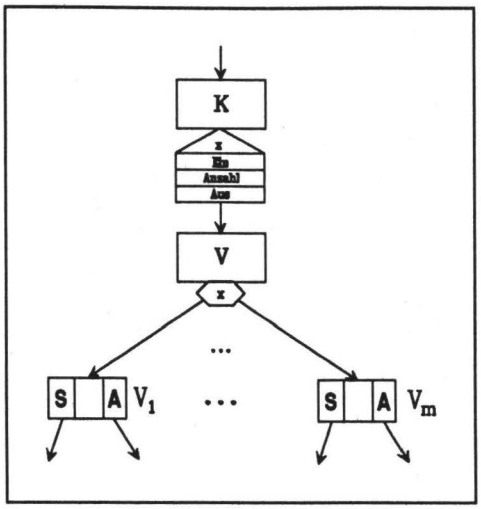


ABB. 3.3.6.2.15. SITUATION ZUR ANWENDUNG DER AUFLÖSUNGSREGEL 5

Die beste Auflösung, bei der der Erhalt nicht zu sehr beeinträchtigt wird, ist folgende:

Die Selektion wird zur Wurzel des Teilgraphen gemacht, wobei die Aktionsteile VA_1,..., VA_m durch VA_1',..., VA_m' ersetzt werden. Diese neuen Aktionsteile sind Iterationen, die als Schleifenkörper die alten VA_j haben.

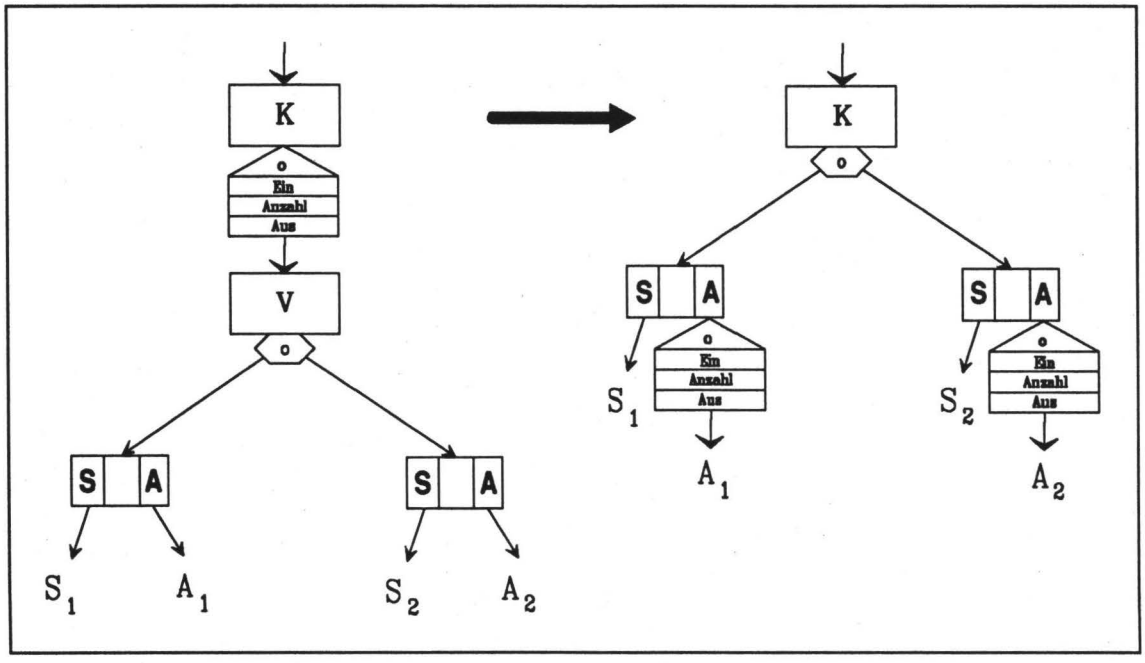


ABB. 3.3.6.2.16. BEISPIEL ZUR ANWENDUNG DER AUFLÖSUNGSREGEL 5

Allgemein kann die Regel in etwa wie folgt formuliert werden:

```
iteration_x(loop(...), V)
node(V,..., selection_x([select(VS_1, _, VA_1),..., select(VS_m, _, VA_m)]))
```

wird aufgelöst zu:

```
selection_x([select(VS_1, _, VA_1'),..., select(VS_m, _, VA_m')])
node(VA_1',..., iteration_x(loop(...), VA_1))
node(VA_2',..., iteration_x(loop(...), VA_2))
    ...
node(VA_m',..., iteration_x(loop(...), VA_m))
```

ABB. 3.3.6.2.17. AUFLÖSUNGSREGEL 5

Die Semantik bleibt dann erhalten, wenn VS_1,..., VS_m unabhängig von den Berechnungen in den Aktionsteilen sind, sonst kann sie verletzt werden.

Soweit ist die Beschreibung der Auflösungsregeln nun erfolgt. Wie sie zusammenwirken und wie sie ins gesamte Verfahren integriert werden, soll im folgenden besprochen werden.

Wir fassen zunächst alle vorgestellten Regeln zu einem Prädikat **auflösen** zusammen. Dabei können Regel 1 und 2 mehrmals hintereinander angewendet werden, aber nur solange bis **eine** Ebene vollständig aufgelöst ist. Bei verschachtelten Sequenzen bedeutet dies z.B., daß die Auflösungsregel 1 bei zwei existierenden Untersequenzen auch zweimal angewendet wird, jedoch nicht ein drittes Mal, auch wenn dies möglich wäre, da sonst damit schon die darunterliegende Ebene aufgelöst werden würde.

Das Prädikat auflösen hat Erfolg, wenn überhaupt eine Auflösungsregel zur Anwendung kam, sonst scheitert es.

Der eigentliche Algorithmus **Abstr_Hierar** wendet auf die zu vergleichenden Knoten das Prädikat **auflösen** an und ruft rekursiv den Gesamtvergleich, um den Koeffizienten zu erhalten, wenn eine Auflösung auf einer der beiden Seiten stattfand. Sonst konnte das Verfahren nicht angewendet worden und es wird "undefiniert" als Koeffizient zurückgeliefert.

```
prädikat auflösen

      Eingabe:    Wurzel des Teilgraphen, dessen erste Ebene aufgelöst werden
                  soll
      Ausgabe:    true,       falls mindestens eine Auflösungsregel zur Anwendung kam
                  false,      sonst.
      Effekt:     Die erste Ebene des Graphen wird, soweit möglich, gemäß der
                  Auflöseregeln aufgelöst

      begin
            Erfolg := false
            while Auflöseregel 1 anwendbar do Auflöseregel 1; Erfolg := true
            while Auflöseregel 2 anwendbar do Auflöseregel 2; Erfolg := true
            if Auflöseregel 3 anwendbar then Auflöseregel 3; Erfolg := true
            if Auflöseregel 4 anwendbar then Auflöseregel 4; Erfolg := true
            if Auflöseregel 5 anwendbar then Auflöseregel 5; Erfolg := true
            auflösen := Erfolg
      end
      Bemerkung:        mit "anwendbar" ist hier gemeint: nur solange eine Ebene
                        behandelt wird.
```

ABB. 3.3.6.2.18. ABSTRAKTION DER HIERARCHIE: PRÄDIKAT AUFLÖSEN

```
algorithmus Abstr_Hierar

      Eingabe:    K, V,       die zu vergleichenden Knoten
      Ausgabe:    Koeff,      der Ähnlichkeitskoeff. des Vergleichs von K und V

      begin
            Erfolg_K := auflösen(K)
            Erfolg_V := auflösen(V)
            Koeff := if Erfolg_K or Erfolg_V  then VK(K, V)
                                              else  undefiniert
      end
```

ABB. 3.3.6.2.19. ABSTRAKTION DER HIERARCHIE

3.4. Das Gesamtverfahren

3.4.1. Einleitung

Nachdem nun alle Einzelverfahren vorgestellt worden sind, soll im folgenden das Gesamtverfahren diskutiert werden.

Dabei wollen wir uns zunächst die Kontrollstruktur betrachten, die die Anwendung der verschiedenen Verfahren steuert.

Wie bereits in der Einleitung 3.1 angedeutet, sind die Verfahren einer sinnvollen Hierarchie zu unterwerfen, die eine Anwendungspriorität definiert. Sie wurde in den entsprechenden Kapiteln jeweils schon bemerkt und sei hier aufgelistet:

1. Textuelle Reduktion
2. Signaturvergleich
3. Synonymvergleich
4. Teilwortvergleich
5. Datenflußanalyse
6. Je nach Kombination der Knotenkonstrukte: Permutation der Sequenz, der Selektion und Iteration bzw. die anderen noch zu besprechenden
7. Abstraktion der Hierarchie

ABB. 3.4.1.1. ANWENDUNGSPRIORITÄT DER EINZEL- IM GESAMTVERFAHREN

Begonnen wird mit dem ersten Verfahren. Das jeweils folgende wird dann angewendet, wenn das vorhergehende scheiterte (1. und 5. scheitern nie, da sie keine Vergleichsverfahren an sich sind). Unter Scheitern verstehen wir in diesem Zusammenhang, daß ein nicht genügend großer Ähnlichkeitskoeffizient erzielt wurde.

Um dies zu modellieren, bietet es sich an, einen **Schwellwert** zu definieren. Erreicht oder überschreitet der Partialkoeffizient des aktuellen Verfahrens diese Grenze, so wird er als Gesamtkoeffizient veranschlagt und das Gesamtverfahren abgebrochen, wird die Grenze unterschritten, so wird das in der Prioritätenliste folgende Verfahren angewendet.

In dieser einfachen Form ist der Ablauf jedoch zu unflexibel. Betrachtet man die einzelnen Arten, wie die Partialkoeffizienten in den Verfahren ermittelt werden, so bemerkt man, daß sie verfahrenübergreifend nicht ohne weiteres verglichen werden können. Ein Partialkoeffizient von 0.2 aus dem Synonymvergleich beispielsweise kann als hohe Ähnlichkeit, der gleiche aus dem Teilwortvergleich als niedrige gewertet werden. Das rührt daher, daß durch die angegebene Formel zur Berechnung des Partialkoeffizienten im Teilwortvergleich auch Zeichenketten mit einer sehr hohen Editier-Distanz

den Wert 0.2 erreichen können, obwohl sie nach unserer intuitiven Auffassung nicht viel miteinander zu tun haben.

Man könnte diese Unterschiede dadurch ausgleichen, daß man nicht einen globalen Schwellwert, sondern einen ganzen Satz, für jedes Verfahren einen, benutzt. Dann mag vielleicht der Kontrollablauf unseren Vorstellungen entsprechen, wird jedoch die Rangfolge der Vorschläge berechnet, wird wieder ein Vergleich von an sich unvergleichbaren Ähnlichkeitskoeffizienten stattfinden und mitunter eine völlig groteske Reihenfolge der Vorschläge entstehen.

Eine bessere Idee ist es deshalb, jeden anfallenden Partialkoeffizienten als Rohwert anzusehen, der Argument einer für jedes Verfahren spezifischen **Anpassungsfunktion** ist. Erst diese Anpassungsfunktion liefert den eigentlichen Ähnlichkeitskoeffizienten des Verfahrens, der mit dem globalen Schwellwert bzw. anderen Koeffizienten verglichen werden kann.

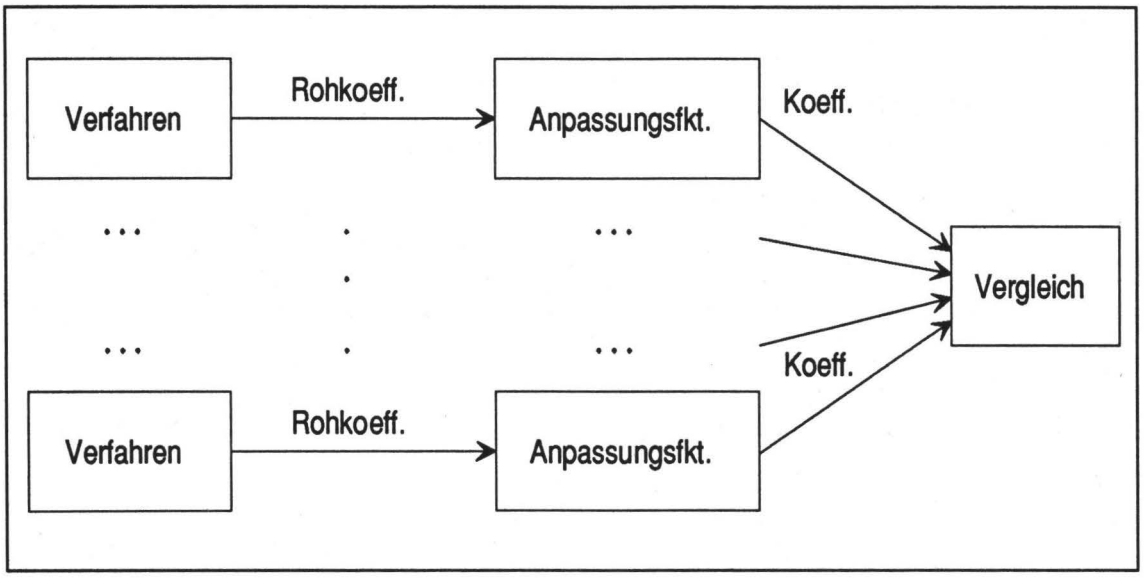


ABB. 3.4.1.2. VERGLEICH VON KOEFF. MIT HILFE VON ANPASSUNGSFUNKTIONEN

Wie diese Anpassungsfunktionen im näheren gestaltet werden können, muß noch im einzelnen untersucht werden (nicht in diesem Bericht). Ein einfacher, sehr wahrscheinlich jedoch ausreichender Ansatz sind lineare Funktionen $a_i x+b_i$, deren Parameter a_i (Steigung) und b_i (Achsenabschnitt) für jedes Verfahren i individuell angepaßt werden. Auftretende Werte kleiner 0 oder größer 1 werden zu 0 bzw. 1 gerundet.

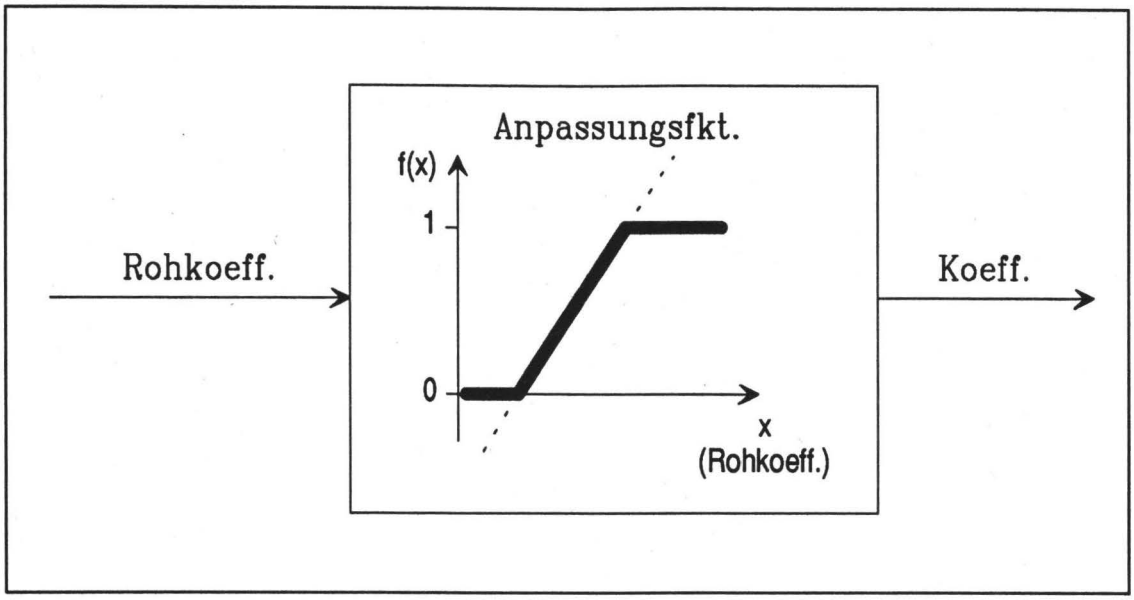


ABB. 3.4.1.3. MÖGLICHE GESTALTUNG DER ANPASSUNGSFUNKTIONEN

Im weiteren Verlauf der Forschungsarbeiten könnten Überlegungen angestellt werden, wie die Parameter a_i und b_i zu behandeln sind. Man kann sie beispielsweise variabel halten und Statistiken über die Leistungsfähigkeit des Wiederverwendungswerkzeuges erheben. Baut man diese Erhebungen als regelndes Element in das Werkzeug selbst ein, erhält man ein lernendes System, daß sich auf Anwender bzw. Anwendungsbereiche anpassen kann.

Ein besonderes Augenmerk ist auf die in obiger Auflistung 6. Gruppe zu richten. Wie in der Einleitung zu den syntaktischen Verfahren bereits erwähnt, führen sie den Vergleich der Teilgraphen rekursiv, beginnend mit den Wurzeln hin zu den Atomen, durch. Dabei wird die Struktur (im Gegensatz zu Punkt 7) der Graphen beibehalten. Die Algorithmen für die noch nicht besprochenen Kombinationen von Konstrukten (z.B. Sequenz - Selektion, Sequenz - Iteration etc.) werden an entsprechender Stelle in diesem Kapitel vorgestellt.

3.4.2. Der Algorithmus

Das Gesamtverfahren wird im folgenden mit $\mathbf{VK}(K_1, K_2)$ bezeichnet, wobei die Argumente K_1 und K_2 die Wurzeln der zu vergleichenden Teilgraphen sind. Die Gesamtheit der semantischen Verfahren formulieren wir gesondert als $\mathbf{V_{sem}}$ (K_1, K_2).

Der global angesetzte Schwellwert, der festlegt, ob ein Verfahren Erfolg hatte oder fehlschlug, wird **Schwelle** genannt.

Die Anpassungsfunktionen werden durch $f_{Anpass,i}$ symbolisiert, wobei i die laufende Nummerierung der Einzelverfahren ist.

Die Vorgehensweise bis einschließlich der semantischen Verfahren dürfte klar sein. Die Information der Inhalte von K_1 und K_2, Info(K_1) bzw. Info(K_2) genannt, werden mit Hilfe der textuellen Reduktion zu $Text_1$ und $Text_2$ normiert. Diese dienen als Eingabe für alle semantischen Verfahren, wobei aufgrund des Vergleichs des sich ergebenden Partialkoeffizientens mit dem Schwellwert entschieden wird, ob das nächste Verfahren aufgerufen wird oder nicht.

Anschließend wird die Datenflußanalyse durchgeführt. Danach werden die Konstrukte von K_1 und K_2 untersucht und der Reihe nach mit den im folgenden aufgeführten Kombinationen verglichen. Trifft eine der Kombinationen zu, so werden die ihr zugeordneten Berechnungen durchgeführt, um zum Gesamtkoeffizienten zu gelangen.

Die Kombinationen der Konstrukte lauten (mit X wird ein beliebiges Konstrukt symbolisiert):

1. **instance(S_1) - instance(S_2)**

Der Vergleich wird auf den zwischen den verwiesenen Knoten S_1 bzw. S_2 reduziert. Damit im semantischen Vergleich alle möglichen Kombinationen von Inhalten gegenübergestellt werden, geht man folgendermaßen vor:

1. $K := V_{sem}(K_1, S_2)$, wenn K >= Schwelle, dann Gesamtkoeffizient := K
2. $K := V_{sem}(S_1, K_2)$, wenn K >= Schwelle, dann Gesamtkoeffizient := K
3. $K := VK(S_1, S_2)$

2. **instance(S_1) - X**

Der Vergleich wird auf den zwischen S_1 und K_2 reduziert:

$K := VK(S_1, K_2)$

3. **nil - X** (und umgekehrt)

Ist einer der Konstrukte nil, so kann keine Ähnlichkeitsaussage getroffen werden. Nil-Konstrukte können im Laufe des Systementwicklungsprozesses nämlich durch jeden beliebigen Graphen ersetzt werden. In diesem Falle wird deshalb auf den mittleren Wert

K := Schwelle

gesetzt.

4. **atom - atom**

Bei Atomen endet der rekursive Abstieg von den Wurzeln zu den Blättern. Sie sind grundsätzlich unvergleichbar. Es wird von einer Verschiedenheit ausgegangen:

K := 0.

5. **atom - X** (und umgekehrt)

Auch andere Strukturen lassen sich nicht mit einem Atom vergleichen. Wir setzen

K := 0.

6. **recursion - recursion**

Rekursionen verweisen immer auf einen Vorgängerknoten. Enthalten beide zu vergleichende Knoten Rekursionskonstrukte, so enthält der eine Teilgraph, auf den verwiesen wird zum Teil den, auf den das andere Rekursionskonstrukt verweist.

Von diesen gemeinsamen Teilgraphen nehmen wir eine grundsätzliche Ähnlichkeit an. Wir setzen:

K := Schwelle.

7. **recursion - X** (und umgekehrt)

Beim Vergleich von Rekursionen mit allen anderen Konstrukten wird

K := 0

gesetzt.

8. **iteration - iteration**

Wir wenden das Verfahren "Permutation der Iteration" an und passen den Koeffizienten an:

$K := f_{Anpass,i}$ (Perm_Iter(K_1, K_2))

9. **sequence - sequence**

Wir wenden das Verfahren "Permutation der Sequenz" an und passen den Koeffizienten an:

$K := f_{Anpass,i}$ (Perm_Seq(K_1, K_2))

10. **selection - selection**

Wir wenden das Verfahren "Permutation der Selektion" an und passen den Koeffizienten an:

$K := f_{Anpass,i}$ (Perm_Sel(K_1, K_2))

11. **sequence - iteration** (und umgekehrt)

Wir reduzieren den Vergleich auf die gesamte Sequenz und den Körper L der Iteration:

$K := f_{Anpass,i}$ (VK(K_1, L))

12. **selection - iteration** (und umgekehrt)

wird genauso behandelt:

$K := f_{Anpass,i}$ (VK(K_1, L))

13. **sequence - selection** (und umgekehrt)

Wir vergleichen zunächst die gesamte Selektion der Reihe nach mit den Söhnen der Sequenz und bilden das Maximum der erhaltenen Koeffizienten. Dann vergleichen wir die gesamte Sequenz der Reihe nach mit allen Aktionsteilen der Selektion und bilden wiederum das Maximum. Die beiden Maxima werden geeignet verrechnet.

Ist der auf diese Weise gewonnene Koeffizient immer noch zu klein, so wird letztendlich das Verfahren "Abstraktion der Hierarchie" angewendet.

Etwas formalistischer notiert hat der Algorithmus folgende Form:

```
function V_sem(K1,K2)

Eingabe:   die zu vergleichenden Knoten
Ausgabe:   der Gesamtkoeffizient des semantischen Vergleichs

     Text1 := Text_Reduktion(Info(K1))
     Text2 := Text_Reduktion(Info(K2))
     K :=     f_Anpass,i (V_signatur(K1,K2,Text1,Text2))
     if K >= Schwelle then V_sem := K; STOP
     K :=     f_Anpass,i (V_synonym(Text1,Text2))
     if K >= Schwelle then V_sem := K; STOP
     V_sem :=  f_Anpass,i (V_teilwort(Text1,Text2))
```

ABB. 3.4.2.1. SEMANTISCHER VERGLEICH

```
function VK(K1,K2)

Eingabe:   die zu vergleichenden Knoten
Ausgabe:   der Ähnlichkeitskoeffizient des gesamten Vergleichs

K := V_sem(K1,K2)
if K >= Schwelle then VK := K; STOP
Kon1 := Konstrukt(K1)
Kon2 := Konstrukt(K2)
case (Kon1,Kon2) of
     (instance(S1),instance(S2)):

          K := V_sem(K1,S2)
          if K >= Schwelle then VK := K; STOP
          K := V_sem(S1,K2)
          if K >= Schwelle then VK := K; STOP
          K := VK(S1,S2)
     (instance(S1),X)):              K := VK(S1,K2)
     (X,instance(S2)):               K := VK(K1,S2)
     (nil,X):                        K := Schwelle
     (X,nil):                        dto.
     (atom,X):                       K := 0
     (X,atom):                       dto.
     (recursion,recursion):          K := Schwelle
     (recu rsion,X):                 K := 0
     (X, recursion):                 dto.
     (iteration,iteration):          K := f_Anpass,i (Perm_Iter(K1,K2))
     (sequence,sequence):            K := f_Anpass,i (Perm_Seq(K1,K2))
     (selection,selection):          K := f_Anpass,i (Perm_Sel(K1,K2))
     (sequence,iteration(_,L)):      K := f_Anpass,i (VK(K1,L))
     (iteration(_,L),sequence):      dto.
     (selection,iteration(_,L)):     K := f_Anpass,i (VK(K1,L))
     (iteration(_,L),selection):     dto.
     (sequence(_,Söhne1), selection(_,Söhne2)):

          a1 := max (S aus Söhne1) (VK(S,K2)
          a2 := max (S aus Aktionsteile(Söhne2)) (VK(K1,S))
          K := f_Anpass,i (g(a1,a2))          g verrechnet a1 und a2,
                                              z.B. g = Mittelwert

     (selection(_,Söhne2),sequence(_,Söhne1)):      dto.
end case
if K >= Schwelle then VK := K; STOP
Ko := Abstr_Hierar(K1,K2)
if Ko undefiniert      then VK := K
                       else VK := Ko
```

ABB. 3.4.2.2. GESAMTVERGLEICH

4. Implementation

In diesem Kapitel sollen die wichtigsten Aspekte der Implementierung angesprochen und eventuelle Unterschiede zu den in Kapitel 3 vorgestellten Algorithmen genannt werden. Weitere Erläuterungen finden sich im dokumentierten Listing selbst.

Als Implementierungssprache wurde, wie auch in den bisherigen Arbeiten innerhalb diese Projektes, **PDC-Prolog** (Version 3.20) gewählt.

Gerade in Prolog fällt die Modellierung des Systementwurfsgraphen komfortabel aus. Desweiteren bietet diese Sprache mit dem Unifizierungskonzept ein mächtiges Instrument, mit dem komplexe Strukturen leicht verarbeitet (in diesem Falle macht sich das vor allem bei den syntaktischen Verfahren bemerkbar) werden können. So ließ sich ein kompaktes und übersichtliches Programm gestalten.

Das Programm hat grob die im folgenden aufgezeigte Struktur.

Zunächst werden, wie in PDC-Prolog gefordert, alle benutzten Prädikate und Domains genannt. Die Prädikate lassen sich in folgende Gruppen zusammenfassen und wie folgt hierarchisch anordnen:

Auf unterer Ebene stehen die **Grundprädikate**, die als Spracherweiterung gesehen werden können und von den Prädikaten, die die Vergleichsverfahren realisieren, benutzt werden. Eine weitere Gruppe, die eine Ebene höher angesiedelt ist, ist die Gruppe der **Verwaltungsprädikate**. Sie erlauben einen Transfer der im Hauptspeicher bzw. auf der Platte gehaltenen Graphen. Die **Vergleichsverfahren** selbst bestehen jeweils aus einer Gruppe von Prädikaten, wie in 4.2.3 beschrieben. Sie sind teilweise einem Prädikat **vSem** untergeordnet, das den semantischen Vergleich realisiert. Das Prädikat des Gesamtverfahren, **vk**, steht an oberster Stelle und benutzt vSem und die Prädikate zu den syntaktischen Vergleichsverfahren.

Im folgenden wird das Listing kurz erläutert.

4.1. Domains

Als wichtige Domains sind **Konstrukt, int_Konstrukt** und **attr_Konstrukt** zu nennen.

Konstrukt ist die Beschreibung der Konstrukte, genau wie sie in [GK-91] vorgestellt wurden. Dieses Format wird lediglich zur externen Speicherung der Graphen in Dateien verwendet.

Auf die Modellierung des Weltgraphen wurde verzichtet. Um trotzdem verschiedene Projektgraphen nebeneinander im Speicher halten zu können, weicht intern die Darstellung der Graphen von der externen in zwei Punkten ab:

- Knoten und Basen erhalten einen zusätzlichen Parameter, die Graph-Nummer. Auf diese Weise ist es möglich, mehrere Graphen parallel im Speicher zu halten. In Prädikaten, die Graphen verarbeiten, ist deshalb die Angabe des Wurzelknotens (i.a. Bezeichner **Name**) und der Graph-Nr. (i.a. Bezeichner **Nr**) erforderlich. Die neuen Funktoren heißen:

  inode(Graph-Nr., Name, Typ, FPL, atrr_Konstrukt) und

  ibase(Graph-Nr., Name, Makro)

- Jedem Konstrukt sind Daten zugeordnet (sie heißen dann attributierte Konstrukte), nämlich die Ergebnisse der Datenflußanalyse. D.h. heißt ein Konstrukt in der Extern-Darstellung K, so ist seine Intern-Darstellung **k(K´,PL,FL)**. PL ist die Parameterliste zu diesem internen Knoten, FL die Folgeliste. K´ ist das veränderte Konstrukt K, bei dem alle Sub-Konstrukte durch attributierte Konstrukte ersetzt worden sind. Daher gibt es neue Funktoren **isequence**, **iiteration** und **iselection** (siehe Listing).

Mit **int_Konstrukt** werden interne Konstrukte bezeichnet, das sind Konstrukte ohne Attribute, wobei jedoch Sub-Konstrukte attributierte Konstrukte sind.

Mit **attr_Konstrukt** werden attributierte Konstrukte bezeichnet.

4.2. Prädikate

4.2.1. Grundprädikate

Zunächst werden einige Grundprädikate wie z.B. member und append definiert. Sie bedürfen keiner näheren Erläuterung.

4.2.2. Verwaltung der Graphen

Danach werden die Verwaltungsprädikate definiert. Dies sind die Prädikate

- **loadGraph** lädt einen Graphen aus einer Datei in die Datenbasis
- **saveGraph** speichert einen Graphen in eine Datei
- **copyGraph** kopiert Graphen innerhalb der Datenbasis

Alle drei Prädikate benötigen die Angabe der Graphennr. (copyGraph benötigt zwei: Quelle und Ziel). loadGraph und saveGraph nehmen selbstverständlich die Transformierung der externen in die internen Formate, und umgekehrt, vor (mit Hilfe von transFormat). Die Attribute PL und FL werden bei loadGraph auf leere Listen gesetzt.

4.2.3. Die Vergleichsverfahren

Die meisten Verfahren berechnen zunächst einen Rohkoeffizienten, der mit Hilfe einer **Koeffizientenfunktion** zum eigentlichen Koeffizienten verrechnet wird. Dabei nimmt die Koeffizientenfunktion die Normierung (nur beim Teilwortvergleich) und die Anpassung (entsprechend den Anpassungsfunktionen) vor.

Alle Verfahren müssen, ähnlich wie Dateien, vor dem ersten Aufruf geöffnet (**open_...**), nach dem letzten wieder geschlossen (**close_....**) werden. In diesen Prädikaten werden Initialisierungsoperationen durchgeführt (z.B. asserts bzw. retracts von Klauseln). Bedient sich ein Verfahren einer Datei (Bsp.: Der Synonymvergleich benutzt ein Wörterbuch), stehen zusätzlich die Prädikate **close_save_...** und **insert_...** zur Verfügung. insert_... fügt neue Daten in die Datei ein, close_save_... ersetzt die alte Datei durch die geänderte (mit save). Die Datei wird im angegebenen Pfad gesucht (CONST pfad).

4.2.2.1. Textuelle Reduktion

Das Prädikat heißt **norm**(Text, Reduzierter_Text). norm benutzt die Datei "satznorm.dbs", die in open_norm konsultiert wird. Sie enthält drei verschiedene Gruppen von Fakten:

- **noinfo** Wörterbuch ignorierbarer Symbole
- **lang** Wörterbuch der Abkürzungen
- **stamm** Stammwörterbuch

4.2.2.2. Signaturvergleich

Das Prädikat heißt **eigenschaft**(Graph-Nr1, Knoten1, Graph-Nr2, Knoten2, Koeff). Für die Darstellung der Eigenschaftstabellen existieren zwei Darstellungen:

- Die Darstellung im Graph
- Die Listendarstellung (zur Verarbeitung)

Die Darstellung im Graphen sieht folgendermaßen aus:

```
node(Name, "etab", [], sequence(info,[atom(info, E1), ... ,atom(info, En)]))
base(E1 ,[W, ... ,W])
     ...
base(En ,[W, ... ,W])
```

Der Wurzelknoten von Eigenschaftstabellen ist vom Typ "etab". Es ist eine informelle Sequenz von informellen Atomknoten, deren Namen die Namen der Eigenschaften

sind. Die Basen zu diesen Eigenschaften enthalten Listen, in denen die Werte (als Strings) stehen.

Anstelle der Atomknoten kann auch das nil-Konstrukt stehen, als Zeichen einer noch nicht näher spezifizierten Eigenschaft. Ebenso kann in den Listen der Basen der leere String ("") als Zeichen eines noch nicht näher bezeichneten Wertes stehen. Sie werden wie in 3.2.3 beschrieben behandelt.

Die Listendarstellung ist folgende:

```
[ez(E1, [W, ... ,W], ... , ez(En, [W, ... ,W])]
```

Das Prädikat getETab(Graph-Nr, Knoten-Name, Tabelle) wandelt die Darstellung im Graph in die Listendarstellung um. Es scheitert, wenn in (Graph-Nr, Knoten-Name) keine Eigenschaftstabelle vorliegt.

4.2.2.3. Synonymvergleich

Das Prädikat heißt **synonym**(Wort_1, Wort_2, Koeffizient). Es entspricht dem ersten Ansatz aus 3.2.4. Es benutzt die Datei "synonym.dbs", die in open_synonym konsultiert wird. Das Faktum in dieser Datei heißt **syn**(Wort_1, Wort_2, Koeffizient). Koeffizient macht eine prozentuale Angabe.

4.2.2.4. Teilwortvergleich

Das Prädikat heißt **teilwort**(Wort_1, Wort_2, Koeffizient). Es benutzt apprMatch, was im wesentlichen dem Algorithmus aus [LV-89] zur Berechnung der Matrix D entspricht, jedoch wird das Minimum in anderer Weise gebildet. (siehe Listing). Daher ist keine Rückverfolgung des Weges notwendig. Dieser an sich einfachere Algorithmus wurde in 3.2.4 jedoch nicht beschrieben, weil seine Korrektheit nicht so leicht zu zeigen ist. Die Matrix wird nicht komplett gespeichert, sondern jeweils nur die aktuelle (**AktZeil**) und die vorhergehende (**VorZeil**) Zeile.

4.2.2.5. Datenflußanalyse

Das Prädikat heißt **flowAnalysis**(Graph-Nr, Wurzel-Name). Es sind Durchlauf 2 und Durchlauf 3 aus 3.3.2 zu einem (pass2) vereinigt worden. Durch den Aufruf von flowAnalysis werden die Attribute PL von allen Konstrukten und FL von den Söhnen der Sequenzen innerhalb des ausgewählten Teilgraphen geändert. flowAnalysis benutzt einige Subprädikate, die vorher definiert wurden. Es sind dies:

- Die Ausblendefunktionen (vergl. 3.3.2)
- Die Mengenoperationen:
  - **unionParlist**(PLaus, PL_1, PL_2), die Vereinigung zweier Parameterlisten (PLaus := PL_1 $\cup$ PL_2),
  - **unionParList**(PLaus, PLs), die Vereinigung mehrerer Parameterlisten (PLs ist eine Liste von Parameterlisten): PLaus := 1. Elem(PLs) $\cup$ ... $\cup$ letztes Elem(PLs),
  - **sequenParList**(PLaus,PLs), die Vereinigung von Parameterlisten unter Berücksichtigung der Streichungsregeln: PLaus := 1. Elem(PLs) $\cup_{seq}$ ... $\cup_{seq}$ letztes Elem(PLs) und
  - **selectParList**(PLaus, Sel, Exe), die Berechnung der Parameterliste für Selektionen, Sel ist die Liste der Parameterlisten der Selektoren, Exe die der Ausführungsteile.
- **substParList**(PLaus, PLin, PEL), entsprechend substPar aus 3.3.2

4.2.2.6. Permutation der Sequenz und Selektion

Die Prädikate heißen **perm_seq** bzw. **perm_sel**. Eingabe sind die Listen der Konstrukte, nicht die Namen der Knoten!

Permutationen werden als Listen von Integers, Mengen von Permutationen als Listen von Listen von Integers dargestellt.

Wichtig zu nennen sind die Subprädikate **validPerms** und **kreuzprod**.

validPerms(Permutationen, Min, FLs) berechnet alle Permutationen der Länge Min mit Hilfe der Folgelisten FL. Der Bereich der Zahlen (1 bis ...) wird dabei aus der Anzahl der Elemente von FL erkannt.

kreuzprod(Koeff, Perms1, Perms2, Ks1, Ks2) entspricht den Schritten 4 bis 6 aus 3.3.3. Die Tabelle wird in der Datenbasis mit Hilfe des Funktors **koeffizient**(Tiefe, Sohnknoten-Nr, Sohnknoten-Nr, Koeffizient) realisiert. Tiefe gibt die Nummer der aufgetretenen Sequenz bzw. Selektion an, damit bei verschachtelten Sequenzen (bzw. Selektionen) und damit entstehenden Rekursionen in kreuzprod getrennte Tabellen gehalten werden.

Die Vergleichsverfahren Perm. der Iteration und Abstr. der Hiersrchie wurden nicht implementiert.

4.2.2.7. Semantischer Vergleich

Das Prädikat heißt **vSem** und entspricht V_{sem} aus 3.4.

4.2.2.8. Gesamtvergleich

Das Prädikat heißt **vergleich**(Schwellwert, Graphen-Nr_1, Graphen-Nr_2, Wurzel-Name_1, Wurzel-Name_2, Gesamtkoeffizient) und entspricht VK aus 3.4. Schwellwert und Graphennummern werden in der Datenbasis gespeichert, so daß die folgenden Aufrufe von **vergleich** diese Angaben nicht mehr benötigen.

4.3. Beispiel für die Anwendung

Es seien zwei zu vergleichende Graphen in den Dateien "Graph1" und "Graph2" gegeben. Ein Programmfragment, das den Gesamtvergleich mit Schwellwert 0.5 durchführt und das Ergebnis druckt, könnte so aussehen:

```
GOAL
     loadGraph(1, "Graph1"),                    % Lade "Graph1" unter Graph-Nr. 1
     loadGraph(2, "Graph2"),                    % Lade "Graph2" unter Graph-Nr. 2
     vergleich(0.5, 1, 2, "root1", "root2", Koeff),      % Vergleich der Graphen
     write("Vergleichskoeffizient: ", Koeff).            % Ausgabe
```

ABB. 4.3.1. BSP.: PROGRAMMFRAGMENT ZUR ANWENDUNG DES GESAMTVERGLEICHS

5. Resümee

In vorliegender Arbeit wurden grundsätzliche Konzepte und Algorithmen zu Vergleichsverfahren für Softwareentwurfsgraphen entwickelt. Die Besonderheiten und Vorteile, die der Softwareentwurfsgraph im Hinblick auf Wiederverwendungssysteme in sich birgt, wurden dabei aufs genaueste berücksichtigt.

Es wurde versucht, dem Leser die Möglichkeit zu geben, die besprochenen Konzepte und Algorithmen in ihrer prinzipiellen Tauglichkeit und ihrem Nutzen selbst zu beurteilen, da in den Beschreibungen großer Wert auf die Darstellung der Zusammenhänge und Hintergründe gelegt und von bloßen Erläuterungen von Vorgehensweisen Abstand genommen wurde.

Durch die recht allgemein gehaltenen Überlegungen und Verbesserungsvorschläge wurden die nötigsten Grundlagen geschaffen, auch abgewandelte oder neue Vergleichs-verfahren zu entwickeln.

Die parallel zur schriftlichen Ausarbeitung angefertigte Implementation kann dabei eine wertvolle Hilfe sein, sie soll aber auch die grundsätzliche Realisierbarkeit der erarbeiteten Konzepte illustrieren.

Beschäftigt man sich über den Rahmen dieser Arbeit hinaus mit der Bedeutung und den Möglichkeiten des Systementwurfsgraphenkonzepts für Softwareentwicklungs-umgebungen, so wird man an mancher Stelle durch die hier aufgezeigten Ideen nützliche Hinweise erhalten. Es sei z.B. nur die Bedeutung der Datenflußanalyse für andere Anwendungen, wie der Parallelisierung von Algorithmen, genannt.

Sehr viel weitergehende Überlegungen zu dem Projekt "Wiederverwendbarkeit von Softwareentwürfen mit Softwareentwurfsgraphen" werden ohne entsprechende Untersuchungen in der Praxis wohl kaum möglich sein. An vielen Stellen in dieser Arbeit wurde deshalb darauf hingewiesen, in welche Richtung solche Untersuchungen gehen können.

Die Beurteilung der Leistungsfähigkeit der Realisation eines Wiederverwendungs-systems mit den vorgestellten Konzpten fällt schwer, nicht zuletzt wegen der mangelden Praxiserfahrung. Wie gut ein solches System arbeitet wird stark von der Feinabstimmung der verrechnenden Funktionen und der Konstruktion des Gesamtverfahrens abhängen. Trotz allem glaube ich, daß es sich als nützliche Hilfe für den Entwickler erweisen wird.

6. Literaturverzeichnis

[Bra-86] IVAN BRATKO

PROLOG: Programming for Artificial Intelligence

Addison-Wesely Workingham (1986)

[Ger-90] RAINER GERTEN

Wiederverwendung von Dokumenten am Beispiel von Software-Entwürfen

internes Papier (1990), Universtät Kaiserslautern, D-6750 Kaiserslautern

[GK-91] RAINER GERTEN, GREGOR KIRSCHALL

Quelltextgenerierung für Software-Entwurfsgraphen

Interner Bericht (1991), Universtät Kaiserslautern, D-6750 Kaiserslautern

[Kel-90] THOMAS KELLER

Systementwurfsgraphgen-Editor

Diplomarbeit (1990), Universtät Kaiserslautern, D-6750 Kaiserslautern

[KMP-77] D. E. KNUTH, H. MORRIS, V. R. PRATT

Fast pattern matching in strings

STAM J. Comput. Vol. 6 (!977), p. 323-350

[LV-89] GAD M. LANDAU, UZI VISHKIN

Fast Parallel and Serial Approximate String Matching

J. of Algorithms Vol. 10 (1989) p. 157-169

[Pdc-90a] PROLOG DEVELOPMENT CENTER

PDC PROLOG version 3.20 User's Guide

H.J. Holst Vej 5A, Copenhagen DK - 2605 Brondby Denmark. (1990)

[Pdc-90b] PROLOG DEVELOPMENT CENTER

PDC PROLOG version 3.20 Reference Guide

H.J. Holst Vej 5A, Copenhagen DK - 2605 Brondby Denmark. (1990)

[Sch-86] PETER SCHNUPP
PROLOG Einführung in die Programmierpraxis
Carl Hanser Verlag München Wien (1986)

[Sel-80] PETER H. SELLERS
The Theory and Computation of Evolutionary Distances: Pattern Recognition
J. of Algorithms Vol. 1 (1980), p. 359-373

[Sun-90] DANIEL M. SUNDAY
A very fast substring search algorithm
Comm. of the ACM Vol. 33, No. 8 (1990), p. 132-142